Pattern specification and application in meta-models in Ecore

Zhang, J.

Award date:
2016

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Software Engineering and Technology Group

Master Thesis

Pattern Specification and Application in meta-models in Ecore

Jia Zhang

Graduation Committee: prof. dr. Mark van den Brand (Supervisor)
ir. Marc Hamilton (Altran)
dr. Natalia Sidorova
Ana-Maria Sutii

Eindhoven, April 2016

Abstract

Design patterns are used pervasively in object-oriented software development. They describe general solutions for recurring problems in software design. In most cases, design patterns are applied at the model level. The popularity of Model-Driven (Software) Engineering has increased the development of meta-models for Domain Specific Languages, even at the meta-model level, design patterns can and are used. For instance, the company Altran has discovered some design patterns in meta-models as well as in associated syntax, and transformation artifacts of their domain specific languages. Unfortunately, the discovered patterns can not be easily (re)used because of the lack of mechanisms to specify and apply design patterns in the existing language engineering workbenches.

This thesis aims at exploring mechanisms for pattern specification and application in meta-models in Ecore. In order to do so, we first analyzed and summarized the state-of-the-art approaches and tooling for specifying and/or applying design patterns. Then, we extended an existing tool, DSL-tao, to assist the development of DSLs in Altran.

Acknowledgments

I have been studying in Software Engineering and Technology (SET) group for a long time. Starting from the courses Generic Language Technology and Software Evolution to the SET Seminar and Capita Selecta, and finally my graduation project, I own many thanks to a lot of people.

My deepest gratitude goes first and foremost to Prof. Mark van den Brand, my supervisor, for providing me the opportunities to work on interesting topics during the Capita Selecta as well as this graduation project and for his insightful guidance, deep discussions and constant encouragement, especially for his understanding and advise during my difficult times.

I would like to express my heartfelt thanks to my tutor ir. Marc Hamilton, an experienced engineer from Altran, for the patient explanation of industry problems and his practical guidance. Special thanks to Ulyana Tikhonova and Marc Hamilton for arranging this external project in Altran. Without their help, I could not get the chance to work in Altran. Thanks to the colleagues in Altran for a nice working environment and relaxed conversations during lunch time.

I am also very grateful to Ana-Maria Sutii, the PhD candidate who gave me detailed advise and feedback through my graduation project. Furthermore, I would like to thank Yaping Luo (Luna Luo) and Zhuoao Li (Luna Li), who sit in the same room with me, for helping me solve problems encountered in this project and for the happiness and comfort they gave to me. Special thanks to the group secretary Margje Mommers-Lenders for arranging weekly meetings and my defense. Besides, I would like to thank my examination committee members: Prof. Mark van den Brand, ir. Marc Hamilton, dr. Natalia Sidorova and Ana-Maria Sutii for reviewing and evaluating my work.

Finally, my thanks would go to my beloved family for their teachings, supports and encouragement, especially from my mother and my grandma. Last but not the least, my sincere thanks go to my boyfriend, for his supports and criticism during the time I have been living in the Netherlands.

Contents

List of Figures

List of Tables

Abbreviation

BPMN	**B**usiness **P**rocess **M**odel and **N**otation
DSL	**D**omain **S**pecific **L**anguage
DSML	**D**omain **S**pecific **M**odeling **L**anguage
EMF	**E**clipse **M**odeling **F**ramework
GPL	**G**eneral **P**urpose **L**anguage
HTML	**H**yper**T**ext **M**arkup **L**anguage
MDE	**M**odel-**D**riven **E**ngineering
MDD	**M**odel-**D**riven **D**eveloping
MDA	**M**odel-**D**riven **A**rchitecture
MOF	**M**eta **O**bject **F**acility
OCL	**O**bject **C**onstraint **L**anguage
OMG	**O**bject **M**anagement **G**roup
QVT	**Q**uery/**V**iew/**T**ransformation
UML	**U**nified **M**odeling **L**anguage
XML	**EX**tensible **M**arkup **L**anguage

Chapter 1

Introduction

This chapter presents the background knowledge and the purpose of this project. Also the structure of this thesis is outlined.

1.1 Motivation

Model-Driven Engineering (MDE) is a promising software development methodology, which employs models as the essential artifacts to raise the level of abstraction in software development. Designing and analyzing models instead of code in MDE brings many advantages. To be more specific, models used in MDE can be transformed into other models via model transformations, or into executable code via code generation or model interpretation, thus accelerating the development process and reducing development costs. Moreover, as models are enforced in MDE, errors can be detected in early stage [32] via model validation, model checking and model-based testing.

Domain Specific Languages (DSLs) is a frequent term used in the context of MDE. DSLs are an important technology that is used to model a variety of domains. As stated in [32], MDE technologies are the combination of Domain Specific Modeling Languages (DSMLs, i.e. DSLs for modeling) and transformation engines as well as generators. Moreover, the development of DSLs follows the MDE methodology. Generally, the abstract syntax of a DSL is defined in meta-models, from which a corresponding implementation can be generated automatically by means of code generators.

Driving the model-driven software development, DSLs are essential but hard to create. Language developers have to communicate extensively with domain experts to understand the relevant domain concept and the problems to be solved, then translate requirements into meta-models, associated syntaxes, transformations and tooling. Because of the specificity - every DSL addresses the needs in a given domain, meta-models that define abstract syntax of DSLs are also specific and have to be created from scratch. This leads to a time-consuming development process and a need for experienced developers to guarantee the quality of a DSL. Moreover, DSLs are becoming more complex and harder to create and maintain with the increasing complexity of the applied domains/systems. This increases the investment in time and money that is consumed in the development of languages to support the model-driven software development.

A promising way to alleviate the current situation is to use design patterns in the design and development of DSLs. “Re-usability is one of the great promises of object-oriented technology” [3]. Up to date, we have seen many successful examples of object-oriented reuse, for example, software libraries are the most commonly reusable code in software development, and design patterns provide reusable common solutions for some recurring problems.

In the context of DSLs’ development, design patterns can be used in the meta-models of a DSL to model the solutions for some recurring problems that the DSL is required to solve. A design pattern is not a finished design like a model fragment. It is an abstraction of a model fragment that can be instantiated into various model fragments in different situations. The flexibility makes

design patterns more appropriate to be reused than model fragments; however, because of the flexibility, design patterns are harder to be applied than model fragments. A model fragment can be copied from one model to another without any modifications if the fragment suits the situation. To apply a design pattern to a specific situation, we have to consider to what extent such pattern can be instantiated. The simplest case is to use a design pattern as is, namely, instantiate the pattern structure without any modifications. But this is not always the case. In many situations, the structure of an instance of a design pattern is different from that of the design pattern itself. In other words, the structure of an instance needs to be adapted for the specific needs of the DSL under development. For example, a meta-class in a design pattern may require specific relations or can be shared across multiple instantiations of patterns.

1.2 Project goal

This project is initiated by Altran[1], a global innovation and engineering consulting firm. The engineers of Altran have developed many DSLs for their clients, and the underlying meta-models were created from scratch every time. The engineers have found some recurring solutions in their languages that can be used as design patterns in the development of DSLs. These design patterns exist not only in the meta-models of DSLs, but also in the associated language engineering artifacts like associated syntax, transformation and constraints. Interestingly, the patterns in the associated artifacts are correlative with the patterns in meta-models. Therefore, if the patterns in meta-models can be reused and the associated artifacts can be reused (semi-)automatically based on the meta-models (e.g. the associated artifacts can be instantiated simultaneously when a pattern is being instantiated), the development of DSLs can be improved significantly.

However, because of the lacking of tool support, the patterns discovered from Altran's DSLs can not be applied directly to their new DSLs. The meta-models have to be created from scratch and the associated artifacts have to be defined manually every time. Meta-models are at the heart of language development, hence applying patterns in meta-models is a good starting point to solve Altran's problems and to develop more advanced tool chains for language development. In this project, we focus on the starting point and formulate Altran's assignment as follows:

> **Altran's assignment:** *we are seeking approaches for pattern specification and application in meta-models in Ecore, and we need tool support to apply our patterns in the development of DSLs.*

The goal of this project is to explore mechanisms for applying patterns in meta-models as well as to provide tool support for pattern application in meta-models. Because Altran mainly uses Ecore as the meta-modeling language, this project focuses on the meta-models defined in Ecore rather than other meta-modeling languages.

As the design patterns proposed by Gang of Four (i.e. GoF) are famous, we use "design patterns" and "patterns" to avoid confusion between the famous GoF design patterns and the other patterns discovered by individuals and/or companies. In the rest of the thesis, "design patterns" refers to the patterns proposed by GoF, and "patterns" refers to the patterns proposed by individuals or companies. Essentially, Altran's patterns are very similar to the design patterns, both of them provide general solutions for recurring problems.

We restrict our research to the meta-models in Ecore, which are popular when developing DSLs. Furthermore, we only focus on pattern specification and application, other applications like pattern discovery, pattern detection and so forth are not considered in this project.

To achieve the goal, the first research question we answer is as follows:

> **RQ1.** *What approaches and tooling are being used to specify and apply design patterns/patterns in meta-models in Ecore and in UML models?*

[1] http://www.altran.com/

Although that our research is limited to the meta-models defined in Ecore, we explore approaches for UML models in this question as well. As UML package *Classes::Kernel* is merged into MOF, and Ecore can be treated as an implementation of MOF, the UML *Classes* package is quite similar to the meta-model of Ecore. Therefore, the approaches for the meta-models defined in Ecore can be learned from that for UML models and design pattern application tools that are suited for UML class diagrams may be applicable to Ecore class diagrams as well.

There are two aspects in this work: one is pattern specification and another is pattern application. Specification of a pattern is a prerequisite for pattern application. First of all, a pattern specification should be understandable to users, otherwise it can not be used appropriately. Secondly, a pattern specification should be processable by machines, otherwise it can not be supported by tooling. Hence, we consider two requirements of a pattern specification in this project:

Req1 *Understandability. A pattern specification should be easy to understand by users.*

Req2 *Processability. A pattern specification should be processable by machines.*

Regarding pattern application in meta-models, the questions are to what extent a pattern can be instantiated and how to instantiate it. In other words, it is necessary to study what kind of modifications can be made to the structure of a pattern in order to create the needed structure in our meta-models and how to implement these modifications. This is a broad topic that is related to pattern variants, and it is not easy to get answers without extensive experiences and experiments. Because of time and resource limitation, we narrow the modifications to addition and deletion of pattern elements (i.e. classes, references and attributes), which means that a pattern element can be instantiated to zero or many instance(s) when a pattern is being applied. Thus, the second research question we answer is as follows:

RQ2. *How to control the occurrences of pattern elements when a pattern is being instantiated and what functions should be provided by tooling to support the control?*

The second research question can be reformulated into two sub-questions:

RQ2-1. *What approaches are being used to control the occurrences of elements of a pattern when the pattern is being applied?*

RQ2-2. *What functionality should be provided by tooling to support the control of the occurrences of pattern elements?*

To answer these research questions, the following activities should be performed:

A1. Study literature on approaches for pattern specification and application in meta-models defined in Ecore and in UML models to answer **RQ1** and **RQ2-1**.

A2. Investigate tool implementations that support pattern specification and application to answer **RQ1** and **RQ2-1**. Then propose a list of potential tools that can be adapted to solve **RQ2-2**.

A3. Modify a potential tool to answer **RQ2-2**. To this end, a pattern should be able ot be specified and applied to meta-models with tool support.

1.3 Thesis outline

The remainder of this thesis is structured as follows.

- Chapter 2 Preliminaries. This chapter introduces the basic terminology that is related to this project. Moreover, a motivating example for this work is described.

- Chapter 3 State of the art. This chapter lists the existing approaches used to specify and to apply patterns in meta-models defined in Ecore and in UML models.
- Chapter 4 Tool exploration. This chapter presents a large collection of related tools. Furthermore, two potential applicable tools are discussed in detail.
- Chapter 5 Extensions of DSL-tao. This chapter first explains our methodology to specify and apply patterns. Then the design and implementation of two extensions of DSL-tao are elaborated with the demonstration of a pattern application using the motivating example.
- Chapter 6 Conclusions and future work. We draw conclusions in this chapter and list some future works.

Chapter 2

Preliminaries

This chapter introduces the basic background knowledge that is related to this project. Additionally, we describe a motivating example illustrating pattern application in meta-models in Ecore at the end of his chapter.

2.1 Model-Driven Engineering

Model-Driven Engineering is a paradigm of software engineering, in which models play a central role. As a software engineering methodology, MDE covers studies and applications on design, development, implementation and maintenance of software in a model-based way. In MDE, models are used as the abstraction of system components, from which code can be automatically generated by means of tools. Also, models can be used as documentation of the system being built. These documents are important for designers and developers to understand and maintain a system. Therefore, MDE can improve productivity and communication in the software engineering processes.

When mentioning MDE, it is important to distinguish it from Model-Driven Development (MDD) and Model-Driven Architecture (MDA). Model-Driven Development is, as the name implies, a model-based paradigm for software development. It can be considered as a subset of MDE that concentrates on software development rather than design and other activities when developing software. In MDD, code can be automatically generated from models and any changes in models can be propagated to code, namely, models and code are always consistent [33].

Model-Driven Architecture, initiated by Object Management Group (OMG), is a particular version of MDD [27]. It defines many standards for software development like UML, MOF, XMI and CWM. The principle of MDA is “everything is a model” [10].

2.2 Eclipse Modeling Framework

The previous section introduces Model-Driven Architecture, which relies on several standards like UML and MOF. MOF is short for OMG’s Meta-Object Facility that defines a four-layer meta-modeling architecture. As seen from Figure 2.1, the top layer M3 provides a meta-metamodel, which defines meta-models at the layer M2. The meta-models at the M2 layer are used to describe models at the M1 layer, and the models can be instantiated at the layer M0 by objects of a real system. An example of the four-layer architecture is illustrated on the right side of Figure 2.1. To model a real world object at the M0 layer, UML models at the model layer (M1) are employed, which is defined by UML meta-models at the M2 layer. UML meta-models are also models, the concepts and relations of which are specified by the meta-metamodel MOF at the M3 layer.

Eclipse Modeling Framework (EMF) is another popular meta-modeling framework, which can be considered as an implementation of the MOF architecture [21]. The core component of EMF is a meta-model named Ecore. Ecore can be put at the M3 layer in the MOF architecture in the case

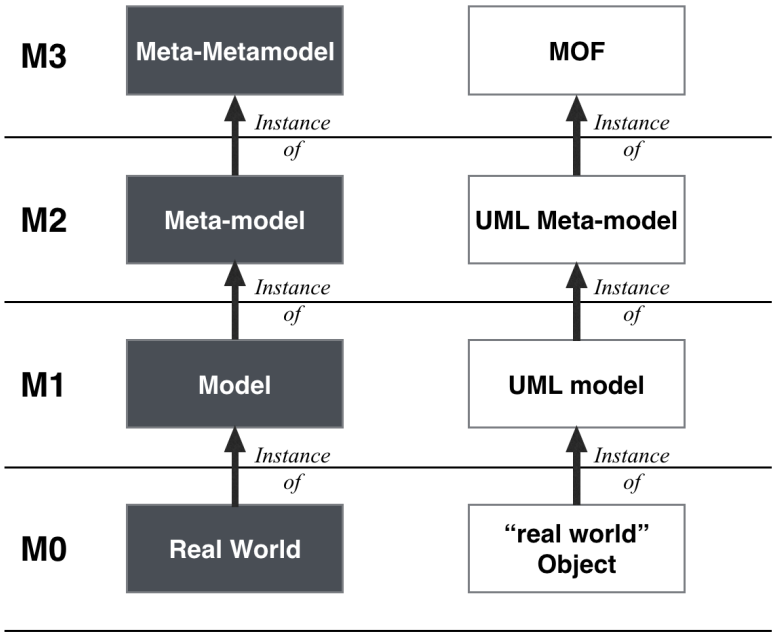


Figure 2.1: The four-layer MOF architecture

that the models that Ecore describes, are meta-models. But this is not always the case. Ecore can also be used to define models at the M1 layer, in which cases Ecore is considered as a meta-model at the M2 layer.

Generally, the meta-model of Ecore language (at the M3 layer) defined in EMF is called *the Ecore meta-model*. The meta-models at the M2 layer described in Ecore language are known as *Ecore models*. In practice, engineers sometimes use Ecore meta-models to refer to the meta-models defined in Ecore to facilitate communication with each others. In order to simplify, we also use "Ecore meta-models" to refer to the meta-models defined in Ecore and use "Ecore models" to refer to the instances (i.e. models) of the meta-models define in Ecore in the rest of this thesis.

Except the modeling language Ecore, EMF provides code generation facility for creating tools and software applications. By means of code generators in EMF, models can be automatically generated to Java code and any changes of models can be automatically updated in code. Hence, "EMF has successfully bridged the gap between modelers and Java programmers" [34].

2.3 Domain Specific Languages

In contrast to General Purpose Languages (GPLs) like Java and C++ that are complex and applicable in many domains, a Domain Specific Language (DSL) is a language tailored to solving problems in a particular domain. DSLs are relatively small and only include features for a target domain, thus they are easy to learn and to understand for engineers and domain experts.

A DSL can be categorized by its form: external or internal [17]. An external DSL, like Cascading Style Sheets (CSS), has its own syntax, and a parser has to be developed to interpret or translate this language, which means this kind of languages is independent of another language (i.e. its host language). On the contrary, an internal DSL relies on a host language. Specifically, it takes advantage of some features of the host language and it is made like a customized language for addressing domain problems.

Another categorization method is based on a DSL's purpose, including Domain Specific Makeup Languages, Domain Specific Modeling Languages (DSMLs) and Domain Specific Programming Languages [39]. For instance, HTML is considered as a domain specific makeup language for web

page development; whereas BPMN is a domain specific modeling language for modeling business processes.

Domain Specific Modeling Languages, as the DSLs for modeling, are important in the MDE world. As described in [31], MDE technologies combine domain-specific modeling languages and transformation engines as well as generators. DSMLs are used to model the structure and behaviors of applications in different domains. Then, model transformation engines and generators are used to analyze models as well as to synthesize artifacts like source code and XML descriptions such that the consistency between models and application implementations remains.

There are some commonly used DSLs in model-driven software development. For example, Object Constraint Language (OCL) is a language for declaring rules of models with assertions. QVT (Query/View/Transformation) defines a set of domain specific model transformation languages including QVT-Operational, QVT-Relations and QVT-Core.

2.4 Design Patterns

In 1994, the book *Design Patterns: Elements of Reusable Object-Oriented Software* was published, and since then design patterns have been popular in software design for decades. A design pattern describes a general solution for a recurring problem in a specific context. Design patterns are considered as reusable good practices in software design.

"Each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems" [20]. The essential elements of each design pattern defined in this book include *pattern name*, *problem*, *solution* and *consequence*. *Pattern name* summarizes a pattern in one or two word(s). *Problem* describes the problem solved by a pattern, namely, in what context the pattern can be applied. *Solution* describes a pattern itself that includes elements, relations, collaborations and constrains of a design. *Consequence* describes the results after applying a pattern.

A design pattern is a general solution instead of a particular concrete design, which has to be adapted in different situations. Design patterns are like templates of problem solutions and need to be customized to fit into particular situations.

Design patterns have different implementations as they are used in various object-oriented languages. In the book, the examples of design pattern implementations are shown in C++ and Smalltalk. This work focuses on patterns in UML and Ecore.

2.5 A Motivating Example

The focus of this project is how to define and apply patterns in Ecore meta-models. From the theoretical perspective, we will present several approaches for this purpose in Chapter 3; and from the practical perspective, we will compare the related tools that can define and/or apply patterns in different models in Chapter 4. Before that, a motivating example is introduced in this section.

In this example, we define a DSL for modeling a workplace. Looking around a real world workplace, it is common that every employee has a fixed office with an assigned table, chair and computer. Also the employees that are in the same position usually have the same working environment (e.g. every manager has an individual small office while six engineers share a big office).

Figure 2.2 shows a model of a company's workplace. This model includes two kinds of `equipments`: desks and computers. There are five equipments described in this model: `regular desk`, `meeting table`, `dining table`, `normal computer` and `engineer computer`. The equipment `regular desk` contains an adjustable element that indicates its height. For the `engineer computer` or the `normal computer`, a registration item is attached to record information about the computer. Every workplace definition is bound with specific equipments and employee's roles (which are described by properties of a workplace definition and they are not shown in Figure 2.2). In this model, an `engineer workplace definition` is defined for software engineers and includes

an `engineer computer` and a `regular desk`. A `manager workplace definition` is defined for top managers and includes a `normal computer` and a `regular desk`. Two instances of these two workplaces are assigned to two employees `Tom` and `Ben` respectively. Tom is a software engineer who has an engineer workplace with a 75-height regular desk and an engineer computer whose serial number is C02PMRLHG8WN. Ben is a top manager who has a manager workplace with a 65-height regular desk and a normal computer whose OS is Windows 10.

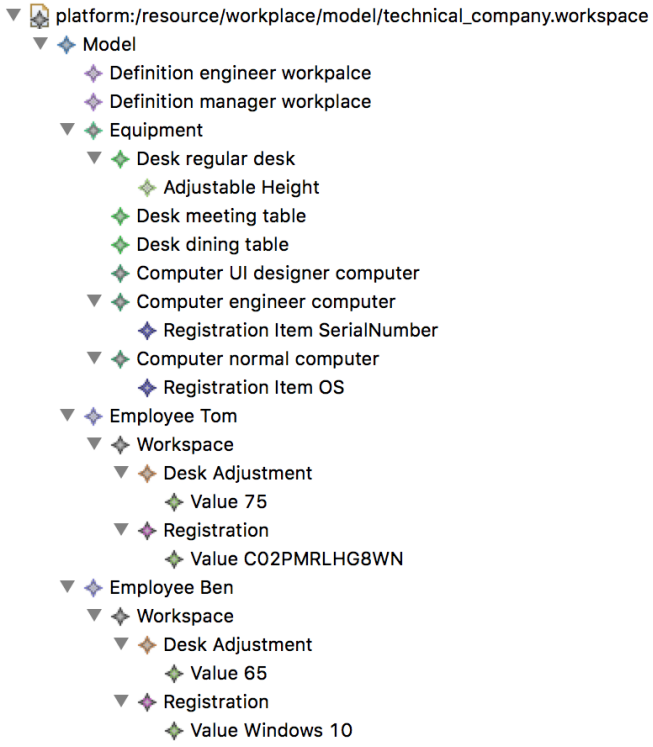


Figure 2.2: A model of a real world workplace

The language used to describe this model is defined using Ecore, by the meta-model in Figure 2.3. The root element of this meta-model is `WorkspaceModel`. The `WorkspaceModel` contains one `Equipment` container and zero to many `WorkspaceDefinition(s)` as well as zero to many `Employee(s)`. An employee plays one or more roles (defined by a data type `Role`) in an organization and is assigned to one or more workspaces based on her role(s). Comparing this meta-model with the model in Figure 2.2, we can see that a `Workspace` (e.g. the workplace of Tom) is defined by a WorkspaceDefinition (e.g. the engineer workplace definition) that is specific for one or more role(s). Every `WorkspaceDefinition` is related to a `Desk` (e.g. the regular desk) as well as a `Computer` (e.g. the engineer computer), and it can be extended by another `WorkspaceDefinition`. Each `Desk` has zero or more `AdjustableItem(s)` (e.g. height of a desk). Each `Computer` has zero or more `RegistrationItem(s)` (e.g. serial number of a computer). These items are specified by `Adjustment` (e.g. 75 cm) and `Registration` (e.g. Windows 10) when a `Workspace` is being created. The values of instances of adjustable items and registration items are `EString` type.

Besides, `Desk` and `Computer` inherit from the abstract classes `EquipmentItem` and `NamedElement`. The `NamedElement` is also inherited by classes `Employee` and `WorkspaceDefinition`.

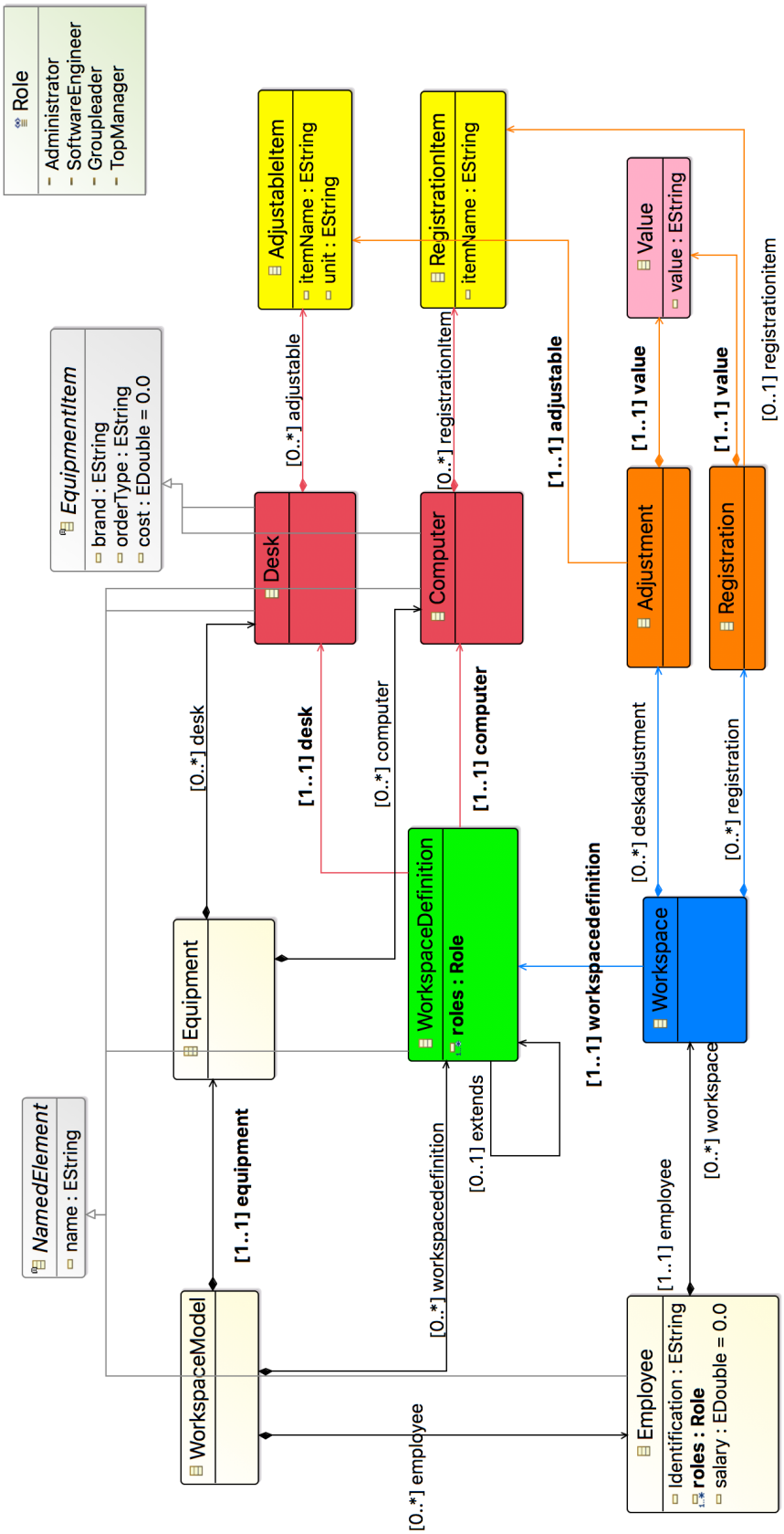


Figure 2.3: The example meta-model of workplace modeling language

The highlighted (colored) elements in the meta-model (shown in Figure 2.3) is shown in Figure 2.5, which indicates an instance of the `Declaration` pattern shown in Figure 2.4. A colored element in the pattern is instantiated by the elements in the same color in the meta-model. For example, the `Container` in the pattern is instantiated by classes `Desk` and `Computer` in the meta-model.

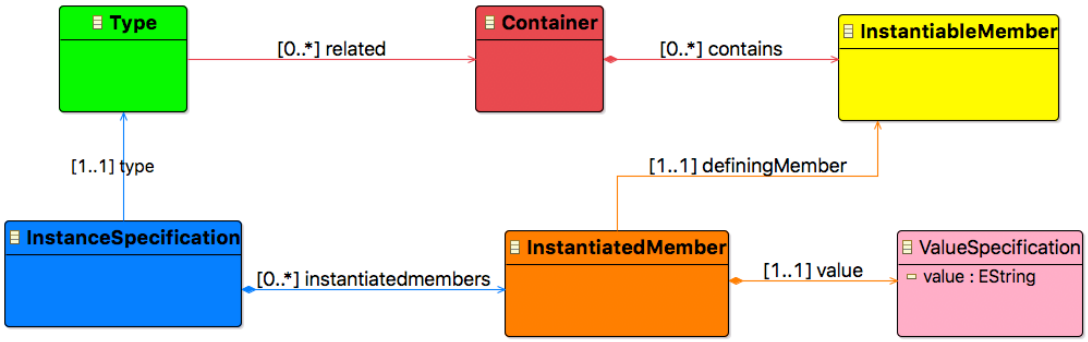


Figure 2.4: The Declaration pattern

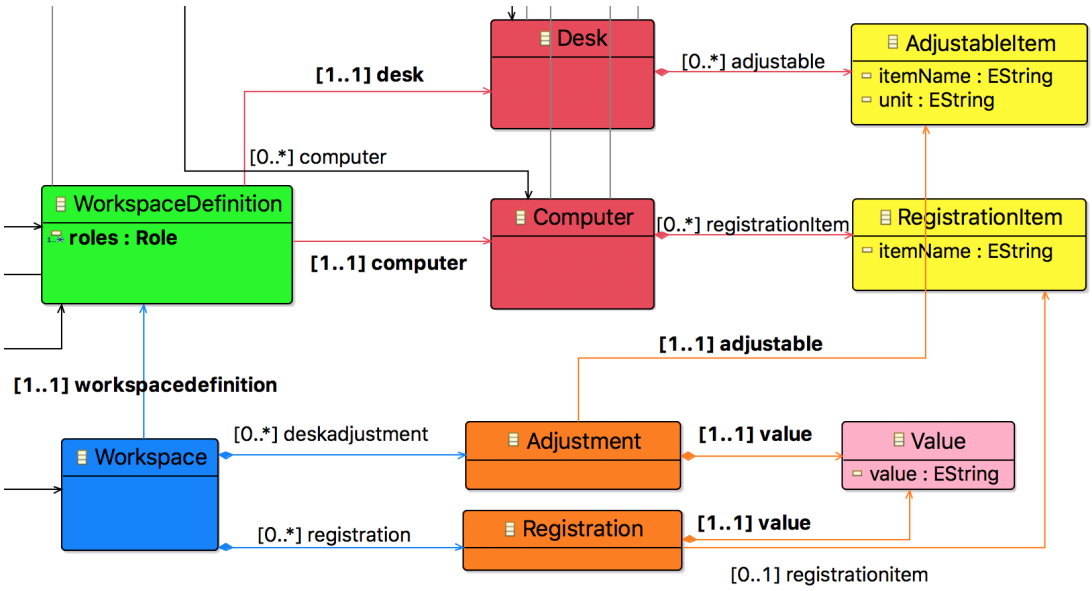


Figure 2.5: The instance of Declaration pattern in Workplace meta-model

An important feature of the example meta-model is that it allows users to specify a form of types of desks or computers, with some variation of properties like adjustment item or registration items. The allocation to an employee via the workspace thus requires specifying the values for the user defined properties (i.e. instantiation of the members of the type), depending on the types of desk or computer associated with the workspace. This feature is implemented by the Declaration pattern, which provides the mechanism to declare instances defined by a specific type. Using this pattern, we can easily create a meta-model of a DSL that contains the mechanism to define a type and to declare instances of this type. This example (in Figure 2.5) is a particular case that the `Type` and `InstanceSpecification` are instantiated (to `WorkspaceDefinition` and `Workspace` respectively) only once. The `Container` is added in this pattern to discriminate between types.

In other variants, the `Container` may be omitted when the type can be directly instantiated.

Chapter 3

State of the Art

In the previous chapters, we detailed the research domain and introduced the general knowledge for this work. In this chapter we aim to investigate the state-of-the-art approaches for specifying and applying patterns (i.e. conduct the activity A1 stated in Section 1.2). To this end, a study on relevant tool support is conducted in order to find potential solutions for the research questions.

3.1 Approaches for pattern specification

In order to be applied in models, a pattern has to be specified in a understandable way to users and be processable by machines. This section aims to describe existing approaches for specifying design patterns/patterns. We categorize these approaches, give a representative example from each category and argue why these approaches do not fully satisfy the requirements stated in Section 1.2.

3.1.1 Literal approaches

As popularized by the book *Design Patterns: Elements of Reusable Object-Oriented Software* [20], design patterns have been commonly used to create models that are used in software development. The design patterns introduced in this book are specified in terms of thirteen aspects: *pattern name and classification*, *intent*, *also know as*, *motivation*, *structure*, *participants*, *collaborations*, *consequences*, *implementation*, *sample code*, *known uses* and *related patterns*. Except for structure (represented in graphical notation) and sample code, other aspects are explained in natural language (e.g. English), which can be understood directly by users.

Such a specification is also used in [30]. Figure 3.1 shows an example pattern specification presented in [30]. Obviously, seven aspects of this pattern are described in English, except a graphical structure. The paper [30] provides a pattern-based development for DSL, in which patterns are specified in English and are applied by their Pattern Application Algorithm.

We call this kind of approaches *literal approaches* because patterns in these approaches are specified in natural languages. Even though these approaches combine graphical structure with literal description, which makes patterns easier to understand by users/readers, they lack a machine-processable mechanism for the subsequent activities like detection, query and verification.

3.1.2 Logic approaches

As natural languages are inaccurate and vague in terms of machine analysis [5], researchers have proposed to use logic-based formalisms to specify, detect and verify patterns.

The formalisms used as basis of these approaches are first order logic, temporal logic, temporal logic of action, process calculus, and Prolog [36]. For instance, Mikkonen [28] proposed a pattern specification method DisCo, the formal basis of which is Temporal Logic of Actions, and Eden [12] proposed a visual language LePUS based on the higher order monadic logic, while Bayley and

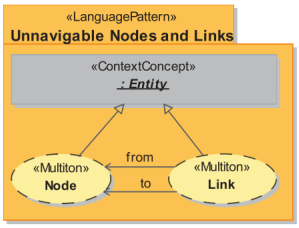
Pattern name:
Unnavigable Nodes and Links

Problem:
Language entities need to be connectable.

Context:
Language entities have already been created, but are still decoupled.

Forces:
Many (graphical) Domain-specific Languages built up on interconnected nodes, with nodes and links being the most general abstraction level.
Navigability between nodes is not an issue, i.e., nodes themselves do not need to know about connected nodes.

Solution:
Existing language entities are specialized into nodes and links. Links reference their source and destination node. Nodes have no references to links.



Rationale:
Specialization of entities into links and nodes clarifies their roles in the language, which can also give hints regarding preferable entity presentations (e.g., circles for nodes and lines for links). Nodes do not need to know about how they are connected. It is sufficient to store this information only within the links.

Related Patterns:
Navigable Nodes and Links

Figure 3.1: An example pattern specification of literal approaches [30]

Zhu [6, 7, 8, 41] focused on specifying design patterns and their variants using first order predicate logic. Additionally, some papers separated the structural aspect and the behavioral aspect of a pattern, and specified the two aspects using First Order Logic and Temporal Logic of Actions respectively [11, 22, 37]. An example of a logic approach is shown in Figure 3.2. The pattern specification *Factory Method* is specified in predicate logic, which includes *Components*, *Static conditions* and *Dynamic conditions*.

Logic approaches are more formal, more precise and less ambiguous than literal approaches. Because of the mathematical basis underlying these approaches, they can be easily processed by machines for detection and verification of patterns and their variants. On the other hand, the mathematical notations makes these approaches hard to apply. First of all, it is likely that pattern users are not familiar with the mathematical knowledge used in logical approaches, and learning such knowledge is difficult as well as time-consuming. Secondly, the logic-based pattern specification can not be used directly on graphical models. In order to apply logic-specified patterns, graphical models have to be transformed into a logical form that is consistent with the logic used for pattern specification. In this case, logic-based specification approaches are inconvenient and time-consuming.

PATTERN NAME: Factory Method (without variants)
COMPONENTS
1. $Creator, Product \in classes$
2. $factoryMethod \in Creator.opers$

STATIC CONDITIONS
1. $factoryMethod.isAbstract$
2. foreach creator subclass there is one product subclass

   $\forall C \in subs(Creator) \cdot \exists! P \in subs(Product)$
3. furthermore, denoting witness P by $f(C)$, then f is a total bijection.

DYNAMIC CONDITIONS
1. for every creator subclass, the factory method creates that unique product subclass:

   $\forall C \in subs(Creator) \cdot isMakerFor(C..factoryMethod, f(C))$

Figure 3.2: An example pattern specification of logic approaches [8]

3.1.3 DSML approaches

Domain Specific Modeling Languages (DSMLs) are languages for modeling in a specific domain. A lot of work have been done on developing DSMLs for different pattern application purposes.

UML has provided a mechanism to specify and apply design patterns using Collaboration and CollaborationUse respectively. Figure 3.3 illustrates an example of a Collaboration [1], which is shown in a dashed eclipse shape. A collaboration includes a name, roles, role types and connectors. A role and its type is depicted in a rectangle, while a connector is depicted as a line. A colon in a rectangle is used to concatenate the role name and its type.

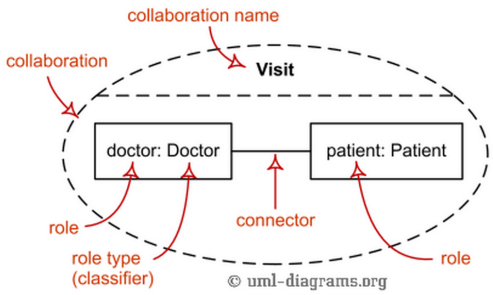


Figure 3.3: An example pattern specification of DSML approaches [1]

Maplesden, Hosking and Grundy [24, 25] proposed a visual language Design Pattern Modeling Language (DPML) for modeling design patterns and their instances in UML models. In this approach, design patterns are specified in Specification Diagrams that show the structure of a pattern, and the instances of design patterns are depicted using Instantiation Diagrams, which indicate binding relations between pattern specification and realization in UML models.

Similarly, Kim et al. [18, 19] presented a Role-Based Modeling Language (RBML) for specifying and applying patterns in UML models. Design patterns in this approach are specified using Static Pattern Specifications that conform to UML class diagrams, and the interactions between pattern participants (i.e. classifiers in a pattern model) are defined using Interaction Pattern Specifications that conform to UML sequence diagrams. Also, this approach provides State Machine Pattern Specifications to specify state-based behaviors of patterns, which conform to UML state machine diagrams.

With the extensive use of model-driven software development approaches and EMF, DSMLs

for modeling patterns are not restricted to UML models.

Elaasar [13, 14, 16] extended EMF with Pattern Modeling Framework (PMF) and introduced an extension of Ecore called Epattern to specify patterns on MOF-compliant modeling languages. In addition, they provided a well-defined iterative specification process for specifying patterns in Epattern language. Recently, a revised work [15] has been published, in which an evolved language Visual Pattern Modeling Language (VPML) is introduced. This language handles not only pattern specifications, but also pattern variants and parameterized patterns. For the purpose of detection, the specifications in VPML are mapped to QVT-Relations (QVTR) transformations that can be used to execute pattern detection.

The other specification approach for querying patterns is proposed by Bergmann et al. [9]. They created a declarative language called IncQuery Graph Pattern Language (IQPL), which is used to specify patterns and pattern queries. This language is specific for querying patterns in EMF models. Figure 3.4 illustrates an example query over an Ecore meta-model. The specification starts with a keyword "pattern" followed by a pattern name. Line 4 to 8 describe a pattern structure, while line 10 to 16 describe queries for this pattern. In this way, it provides an approach to specify patterns, namely, describe pattern structure in IQPL. Although this language is suitable for EMF models, it can only be used to query patterns rather than applying patterns.

```
1  import "http://www.eclipse.org/emf/2002/Ecore"
2
3  pattern SampleQuery(XElement, YElement, Relates, Label1, Label2) = {
4          EClass(XElement);
5          EClass.eStructuralFeatures(XElement,Relates);
6          EReference(Relates);
7          EClass(YElement);
8          ETypedElement.eType(Relates,YElement);
9
10         find EReferenceWithStarMultiplicity(Relates);
11
12         find EClassWithEStringAttribute(XElement, Label1);
13         find EClassWithEStringAttribute(YElement, Label2);
14
15         neg find IsInECore(XElement);
16         neg find IsInECore(YElement);
17 }
```

Figure 3.4: An example pattern query specification in IQPL [9]

In 2015, a new DSL called DSL-tao was proposed for facilitating pattern usage in Ecore meta-models [29]. Different from other DSMLs, every pattern specified in DSL-tao is bound to an Ecore meta-model that represents the structure of a pattern. The language itself is used to model a pattern repository and specify pattern roles in a pattern, namely, every element (i.e. classes, attributes and references) in an Ecore pattern model is described with corresponding pattern role (e.g. ClassRole and AttributeRole) in DSL-tao.

These *DSML approaches* allow to specify patterns directly on MOF-based models, hence avoiding the costs of upfront conversion. Moreover, every DSML can be tailored to achieve specific aims precisely and effectively. For example, IQPL is created for querying patterns, whereas VPML is made for detecting patterns.

However, every coin has two sides. First of all, a DSML is always developed for a specific purpose like pattern detection or pattern application, resulting in a limited usage of patterns in the development chain. To be more specific, the pattern specification defined to apply patterns at the design phase can not be used to detect patterns at the maintenance phase, because such specification can not be recognized and processed as the input of a detection tool. For example, RBML is only used to apply patterns while VPML is used to detect patterns. Hence, to detect a RBML-specified pattern using VPML, the RBML specification would require translation to its equivalent VPML specification because the VPML implementation can only detect VPML-specified patterns, but this translation is not trivial. Secondly, some DSMLs stated above are not tested by a large amount of cases. This means the applicability, efficiency and accuracy can not be

guaranteed when using such DSMLs. Last but not least, compared to literal approaches, DSML-specified patterns are not easy to understand for pattern users (rather than pattern creators).

3.2 Role-binding approach for pattern application

Applying patterns in model-driven software development is important and useful because patterns provide proven common solutions for recurring problems during development. UML, as the most prominent modeling language, has already provided mechanisms for pattern application. In UML, design patterns are specified using Collaborations, while pattern instances are represented using CollaborationUse.

Figure 3.5 illustrates an example of a Collaboration [1] that is shown in a dashed eclipse shape. This pattern `Visit` describes the cooperation between `doctor` and `patient` in order to achieve the given task `Visit`. The two roles `doctor` and `patient` are participants of this pattern, which will be played by instances (described in CollaborationUses) when they are applied. The relationships related to the task are specified in `connector`, while the types of `roles` are specified in `role types` that define all required properties of the instances playing specific roles.

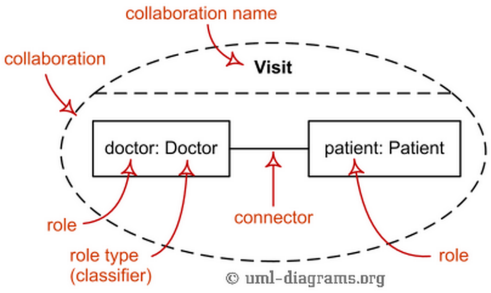


Figure 3.5: An illustration of Collaboration in UML [1]

Figure 3.6 illustrates an example of a CollaborationUse [2] named `childVisit`, which describes an instance of the `Visit` collaboration in the context of children medical treatment. Under this specific context, role `doctor` is played by `Pediatrician`, while `patient` is played by `Infant`. The `connector` between `doctor` and `patient` in Collaboration is instantiated by the relation between `Pediatrician` and `Infant`. By indicating `role bindings` in CollaborationUses, a pattern is applied to an instance in a specific context. In this way, UML provides mechanisms for pattern specification and application.

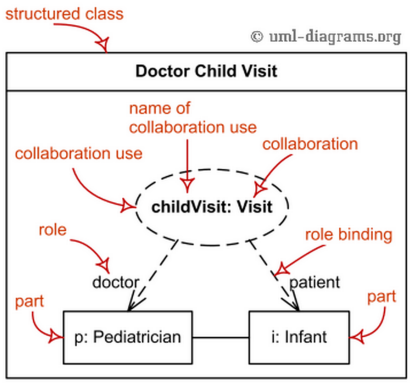


Figure 3.6: An illustration of CollaborationUse in UML [2]

Similar to the approach provided in UML, in [24, 25], patterns are specified in DPML using Specification Diagrams, which are similar to Collaboration in UML, and instances are described in Instantiation Diagrams, which are similar to CollaborationUse in UML.

Not surprisingly, DSMLs for specifying patterns provide similar *role-binding* approaches to describe how a pattern is applied in a specific context [4]. In RBML, patterns are specified in Structural Pattern Specifications while instances of patterns are defined in UML class diagrams. Hence, every element in a pattern has a specific role whose base is a UML meta-class, e.g. *Association* role and *Class* role. Pattern application is implemented by establishing bindings between model elements to pattern roles.

This approach has been adapted to Ecore meta-models. The DSL-tao language is created for pattern usage in Ecore meta-models. It declares `roles` in `patterns` and `roleInstances` in `patternInstances`. The `roleInstances` pointing to `Eobjects` is bound with `roles` pointing to Ecore meta-classes like *EClass* and *EReference*.

Moreover, the role-binding approach has also been propagated to other MOF-based models in [13, 14, 15, 16]. The language Epattern and the revised language VPML are proposed to facilitate the usage of patterns in MOF-based models. For example, in VPML, a `Role` contained in a `Pattern` is played by an instance defined in `PatternUse` via `RoleBinding` (see Figure 3.7).

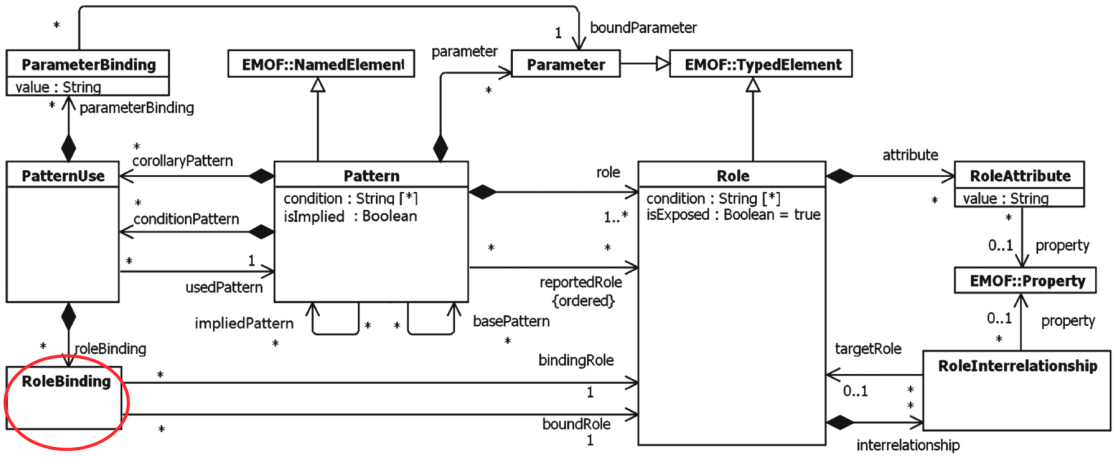


Figure 3.7: Meta-model excerpt of VPML for pattern specification

We only found the role-binding approach for pattern specification in the literature we studied.

3.3 Conclusions

In this chapter, we have shown the results of literature study. Section 3.1 introduced three categories of pattern specification approaches, namely, literal approaches, logic approaches and DSML approaches. Section 3.2 introduced approaches for pattern application. In fact, we found only one application approach - the role-biding approach. In next chapter, the results of tool exploration will be shown by introducing a collection of pattern application tools.

Chapter 4

Tool Exploration

From the practical perspective, we explore the related tool support that can define and/or apply patterns in models/meta-models (i.e. conduct the activity A2 stated in Section 1.2).

In this chapter, we first analyze a large collection of related tools in Section 4.1. Then, in Section 4.2 and 4.3, two tools that are useful for this project are discussed in more detail. These two tools are considered as potential tooling that can be extended for our case. The features of these two tools are introduced by creating the example meta-model using the `Declaration` pattern.

4.1 Pattern application tools

There is a large collection of tools that are considered related to pattern definition and/or application. These tools can be applied to Ecore, UML and other MOF-compliant models. Some of them are open-source tools while others are commercial tools. Below are the details of the investigated tools.

UMLAUT[1] is a tool and framework used for UML models. As introduced in Section 2.4, the official mechanism to define design patterns in UML is to use the *collaboration* construct, which includes the *parameterized collaborations* to define design patterns and the *collaboration usage* to apply design patterns for specific situations. Instead of using parameterized collaborations, UMLAUT provides mechanisms to more precisely define design patterns by means of collaboration and constrains at the meta-level, which is proposed in [22, 23, 35]. This tool is still available but only for an obsolete execution environment (i.e. JDK 1.3) and UML version (i.e. UML 1.3). With the evolution of UML, this tool evolved to a new generation called **UMLAUT NG**[2], which is an extendible tool and framework for model transformation. It enables the application of design patterns in UML by using advanced model transformations [38]. Nowadays, this tool has been replaced by **Kermeta**[3] and its MDKs (Model Development Kits). Kermeta not only provides MDKs to play with UML models but also to play with Ecore, Java5 and other meta-models[4]. However, the MDK for Ecore is not available, so we will not consider any of these tools in the rest of our work.

DPTool is a prototype tool that supports defining and applying design patterns in UML models using Design Pattern Modeling Language (DPML) [26]. This language can be used standalone to define patterns, or it can be used together with UML to represent design pattern instances. Besides, this tool includes features to automatically instantiate design patterns and check consistency between design patterns and their instances. **MaramaDPTool**[5] is the latest version of

[1] http://www.irisa.fr/UMLAUT/
[2] http://raweb.inria.fr/rapportsactivite/RA2006/triskell/uid54.html
[3] http://www.kermeta.org/
[4] For more information: http://raweb.inria.fr/rapportsactivite/RA2006/triskell/uid46.html
[5] https://wiki.auckland.ac.nz/display/csidst/MaramaDPTool

DPTool [25]. Unfortunately, it has been terminated in 2012, so it will not be considered in the following work.

IBM Rational Software Architect[6] **(RSA)** is a commercial software that supports comprehensive design, modeling and development. It is built in the Eclipse framework and it provides multiple functions to address model-driven development (MDD) problems. This tool contains a pattern framework to manipulate design patterns in UML models. For example, a design pattern instance can be dragged to an UML diagram as well as be bound by UML elements when being applied [40]. Although this tool provides a lot of support for pattern application, we can not easily extend it because it is not an open source software.

Epattern is a graphical pattern specification language that is defined in Pattern Modeling Framework (PMF). The meta-model used to define Epattern extends the meta-model of Ecore [14]. This means that the language is an extension of Ecore language, and it is able to define patterns in Ecore meta-models. The implementation of this language provides an editor for Eclipse to specify Ecore patterns in Epattern language. However, this implementation is not available now. As time went on, a revised work [15] came out. In this revised version, a new domain specific language **Visual Pattern Modeling Language (VPML)** and its implementation was designed and developed. The VPML tooling provides an editor in Eclipse for defining and detecting patterns of MOF-based models using VPML. Unfortunately, the VPML tooling cannot be easily accessed either nowadays.

EMF-IncQuery[7] is designed as a declarative domain specific language and an incremental graph query engine to process queries on EMF models (e.g. BPMN and Ecore) [9]. Before processing a query, a graph pattern that will be queried has to be defined in the language EMF-IncQuery. In this way the tool implements an approach to define model patterns, namely, defining model patterns in EMF-IncQuery language.

EMF DiffMerge/Patterns[8] provides supports for pattern applications in modeling. Aiming at improving the productivity and quality of models in MDE, the focuses of this tool are creation, application, as well as management of modeling patterns. It can be customized for a specific modeling language or workbench, hence being integrated into any EMF-based environment. Such customization has been implemented in the modeling environment Capella[9] and UML Designer[10].

DSL-tao[11] is an emerging Eclipse plug-in for rapidly and easily developing DSL meta-models using patterns. This tool includes an editor to create Ecore-like meta-models and features to apply patterns during development [29]. Moreover, this tool contains some commonly-used patterns that are discovered from the public meta-model repository *ATL meta-model zoo*[12]. These predefined patterns are categorized into several domains and can be directly used by users.

Although all tools listed above are relevant to pattern usage, some of them are not available or out of date like VPML and RSA. EMF-IncQuery is easy to use, but it aims at querying patterns instead of applying patterns. The two left tools EMF DiffMerge/Patterns and DSL-tao are considered as potential tools. In the following two sections, we will introduce these two tools in detail.

4.2 EMF DiffMerge/Patterns

One of the potential tools is the EMF DiffMerge/Patterns tool, which was implemented in 2014. This tool is developed based on the EMF DiffMerge engine that was designed to prevent data loss as well as model inconsistency while merging models. The project EMF Diff/Merge[13] including the engine and the DiffMerge/Patterns tool, is part of Eclipse Modeling Framework. The DiffMerge

[6] http://www-03.ibm.com/software/products/en/ratisoftarch
[7] https://www.eclipse.org/incquery/
[8] https://wiki.eclipse.org/EMF_DiffMerge/Patterns
[9] http://www.polarsys.org/capella/
[10] http://www.umldesigner.org/
[11] http://jdelara.github.io/DSL-tao/
[12] http://web.emn.fr/x-info/atlanmod/index.php?title=Zoos
[13] https://www.eclipse.org/diffmerge/

engine is used to compare and merge models in a way that model consistency is enforced. The DiffMerge/Patterns is designed to handle pattern-based activities in software development.

This tool is relatively mature and it supports the creation, application, evolution and management of patterns in modeling process. The aim of this tool is to support the share of design patterns and other user-defined patterns, thus improving productivity and quality in model-based software development.

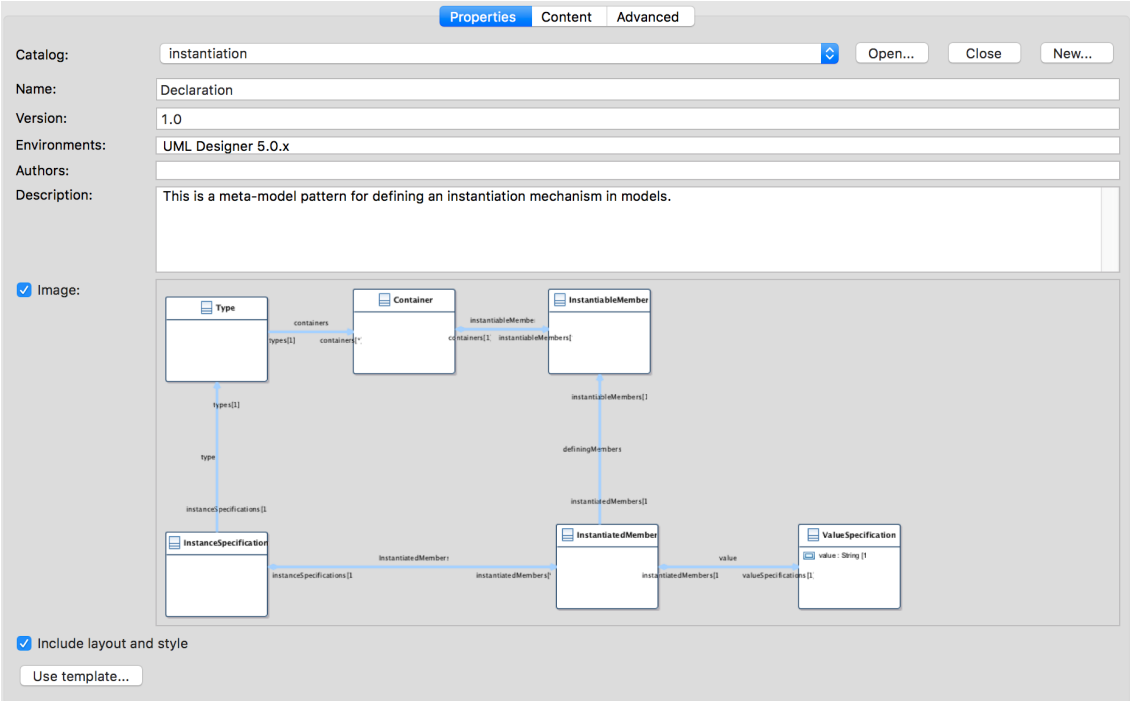


Figure 4.1: Setting of "Properties" tab in the creation wizard

The DiffMerge/Patterns tool provides functions that fulfill the requirements we want. Firstly, a pattern can be created easily. As shown in Figure 4.1, the creation wizard provides an interface to users to specify several aspects (e.g. Name and Version) of a pattern, and shows the structure of the pattern being created. The tool not only stores the information and structure of a pattern, but also optionally stores the layout and style of a pattern.

Secondly, this tool provides a mechanism to control occurrences of instances of every pattern element using the flag "Unique". In the "Context" tab (shown in Figure 4.2), a user can tick the "Unique" flag of a pattern element to set the element unique, namely, the element can only be instantiated once in a pattern instance, or an element can be set to non-unique by un-checking the "Unique" flag. This means the pattern element can be instantiated multiple times in an instance. When applying a pattern, all non-unique elements can be instantiated into multiple instances via the function "Multiplicity per instance". In this example, we set elements *Type*, *InstanceSpecification* and *ValueSpecification* to unique.

This tool uses role-binding pattern application approach, and in the "Context" tab the role of a pattern element can also be modified.

When applying a pattern, application wizards are used to help users to easily apply a pattern. Figure 4.3 shows a wizard to create pattern instances. The roles of all pattern elements are listed and the role-bound elements can be set to be merged with an existing element in your model or to be added in your model. A user can set the instance number of a pattern and also the occurrences of each pattern element (i.e. "Multiplicity per instance" in the Figure 4.3). Coincidentally, in the motivating example, every non-unique element of the Declaration pattern has two instances in the example meta-model, so we set the number of instances of this pattern is 1 and the number

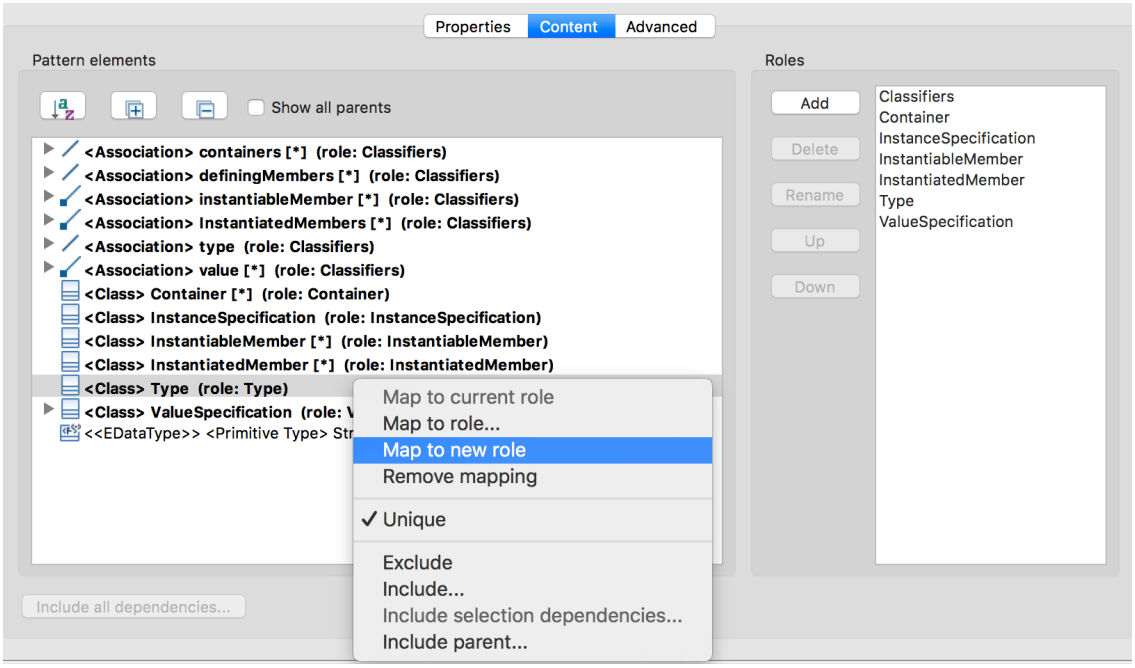


Figure 4.2: Setting of "Content" tab in the creation wizard

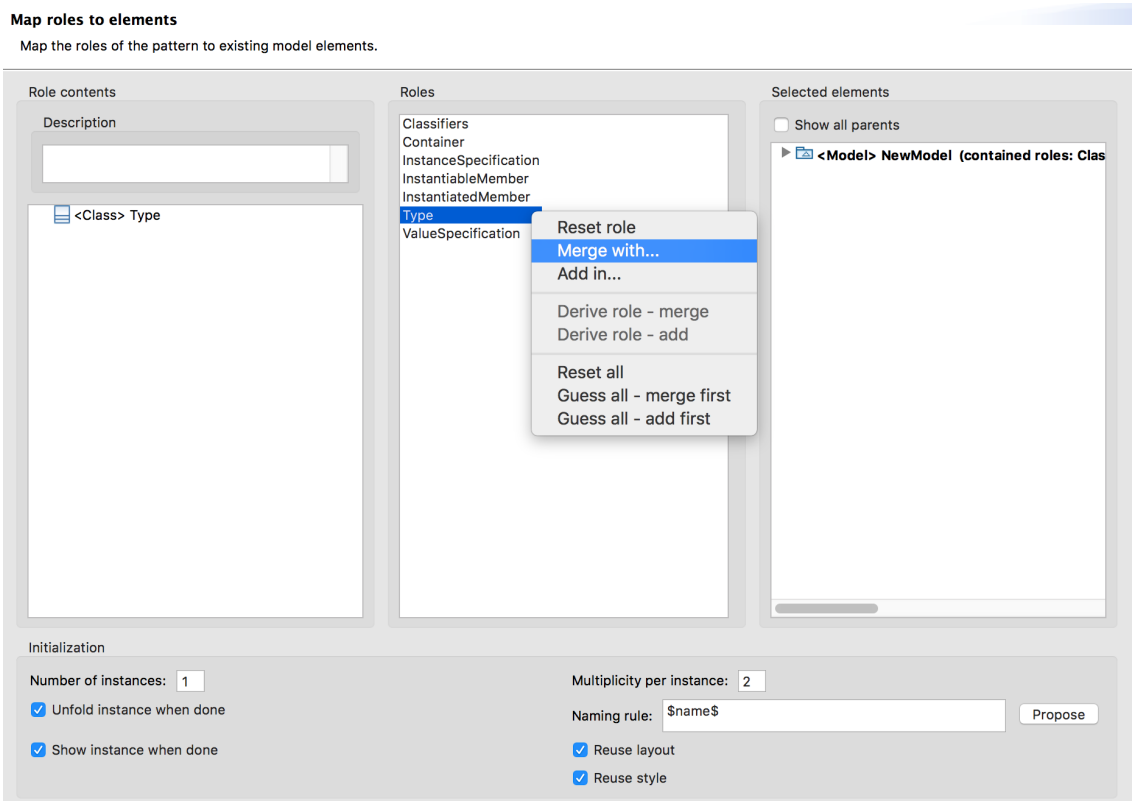


Figure 4.3: Setting of an application wizard

of instances of every non-unique pattern element is 2 (see Figure 4.3). Using this setting we can

create the same structure as that of the highlighted part of the example meta-model by applying the pattern once, which is shown in Figure 4.4.

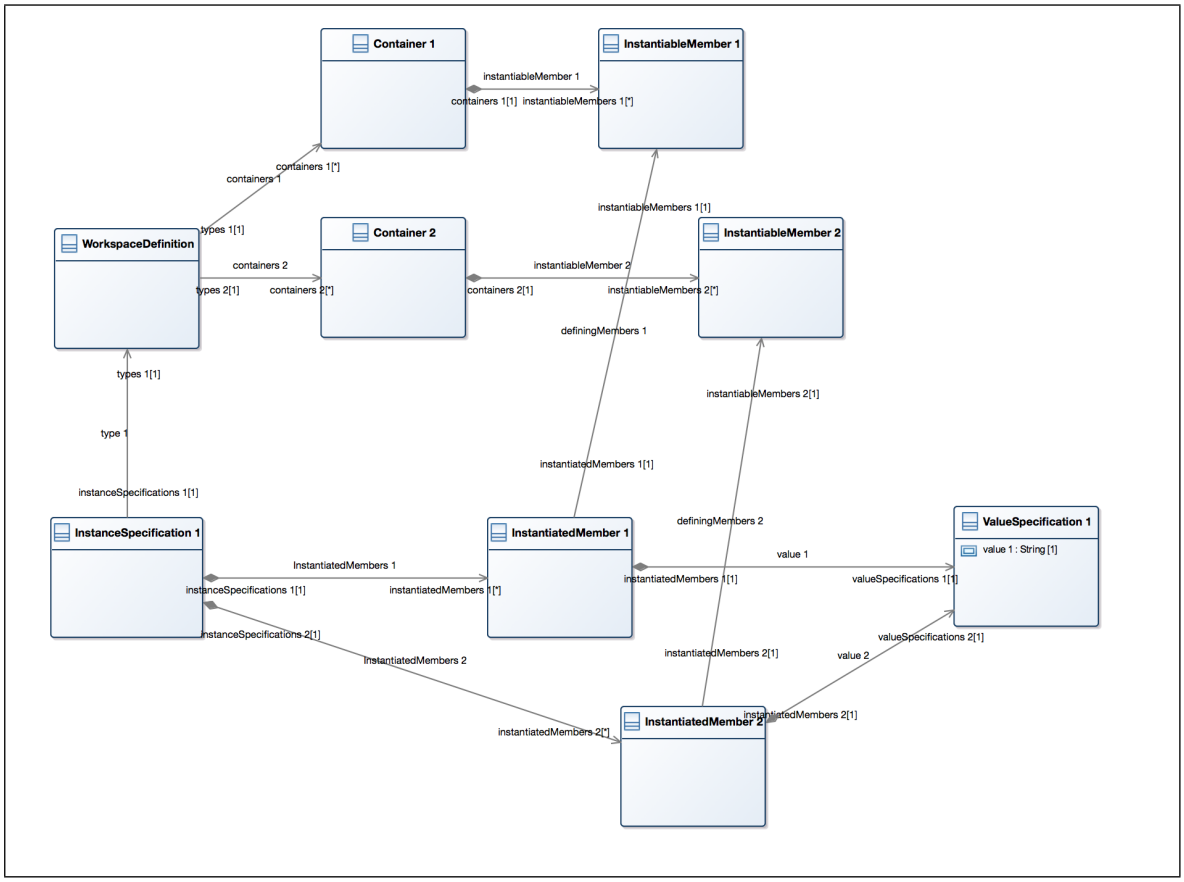


Figure 4.4: The meta-model after the pattern is being applied

The other functions provided by DiffMerge/Patterns are detailed in Appendix A.

Although this tool satisfies the requirements we need, it can only be integrated in UML Designer[14]. This means it can not be used in Ecore meta-models. As stated by the tool developers, the DiffMerge/Patterns technology can be integrated into any EMF-based modeling environment with an appropriated customization. For example, the technology has been integrated into Capella[15], a modeling workbench built based on EMF. However, until now there is not an official released customization for Ecore meta-models. Besides, the functionality to control the occurrences of pattern elements is not flexible. In this example, it is a coincidence that all non-unique elements have the same number of instances. If non-unique elements have different number of instances, the function "Multiplicity per instance" will not be able to create these instances by applying the pattern once.

If we use this tool in this case, we have to migrate this tool from UML Designer to EcoreTools[16] and make a customization for EcoreTools. Unlike UML model elements, however, Ecore meta-model elements do not have unique identifiers. Consequently, the tricky technical work of making a customization is to provide a way to identify elements uniquely, which is not trivial.

[14] http://www.umldesigner.org/
[15] http://www.polarsys.org/capella/
[16] https://www.eclipse.org/ecoretools/

4.3 DSL-tao

Another potential tool is DSL-tao, which was implemented in 2015. This tool was proposed to demonstrate the feasibility of the assisted construction of DSMLs proposed in [29]. In this paper, authors proposed to use patterns to assist the development of DSMLs and provided a taxonomy of patterns in the context of DSMLs.

This tool is created based on EMF, which focuses on Ecore meta-models. In this tool, the structure of a pattern is specified using an Ecore meta-model and its elements (i.e. pattern roles). The meta-model that contains the structure of a pattern is stored in Ecore. The pattern roles are specified using another modeling language called *dslpatterns*. The meta-model of this language is the core of this tool. This language is also used to define the structure of a repository that stores all patterns. Figure 4.5 shows an excerpt of the meta-model of DSL-tao. It is also the meta-model of the dslpatterns language.

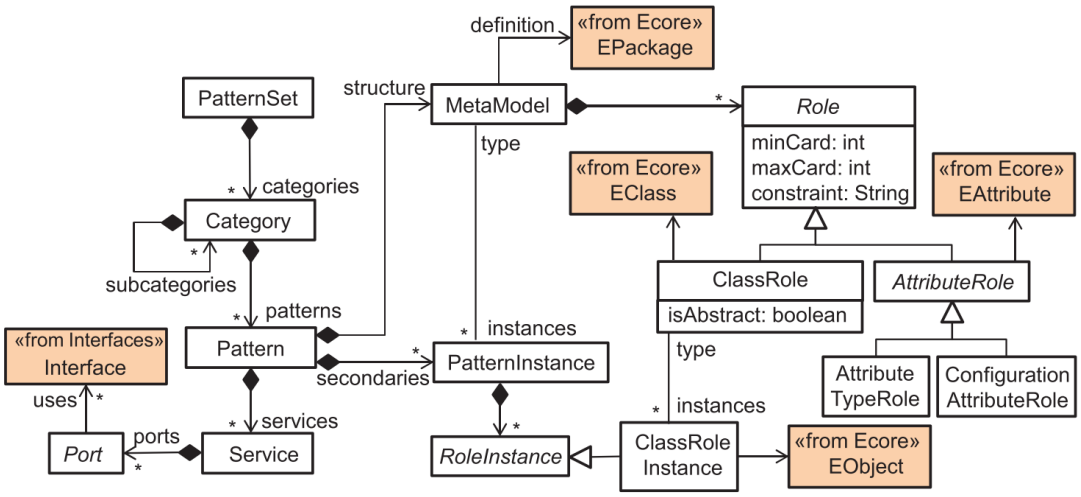


Figure 4.5: Excerpt of the meta-model of DSL-tao/dslpatterns language [29]

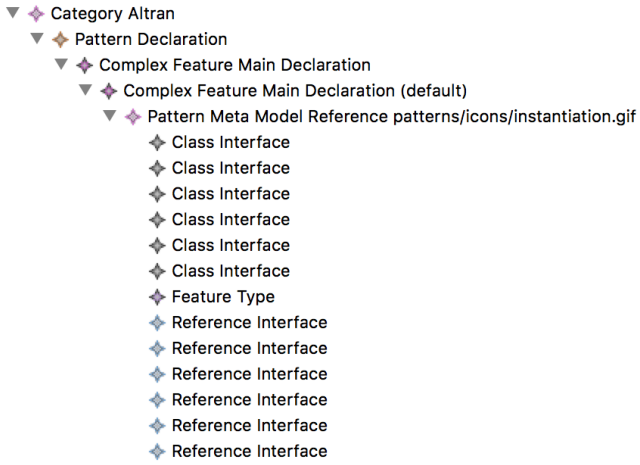


Figure 4.6: The roles of the Declaration pattern defined in the repository

The repository to store patterns is defined in the file `repository.dslpatterns`. When creating a pattern using this tool, the file has to be modified to add pattern roles of the new pattern.

Figure 4.6 shows the roles of the Declaration pattern in the repository. We can see that pattern elements are defined in three pattern roles: `Class Interface`, `Feature Type` and `Reference Interface`.

According to the meta-model of DSL-tao and patterns in the repository file, we summarize the structure of the repository in Figure 4.7. It is clear that a pattern is created under a category in the repository. A pattern must have features that contain the structures of pattern variants. All pattern variants (including the pattern itself) have to be specified by means of binding pattern roles in the repository with the corresponding objects in Ecore meta-models. A pattern variant may be associated with secondary patterns (i.e. instances of other patterns) that define complementary aspects. A secondary pattern can be applied automatically when the associated pattern is being applied. Also, a pattern may have services providing functionality to the environment generated for a DSL.

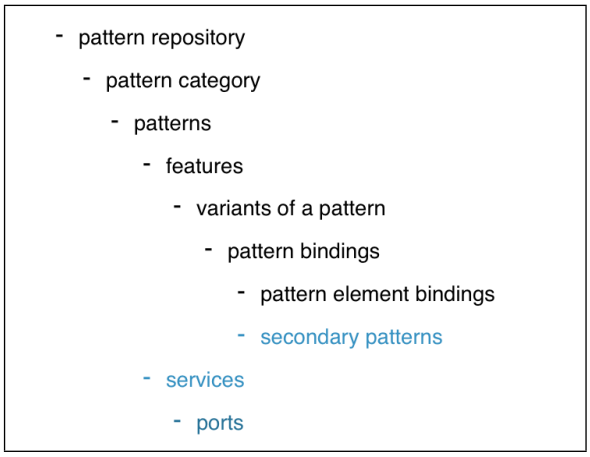


Figure 4.7: The structure of the pattern repository

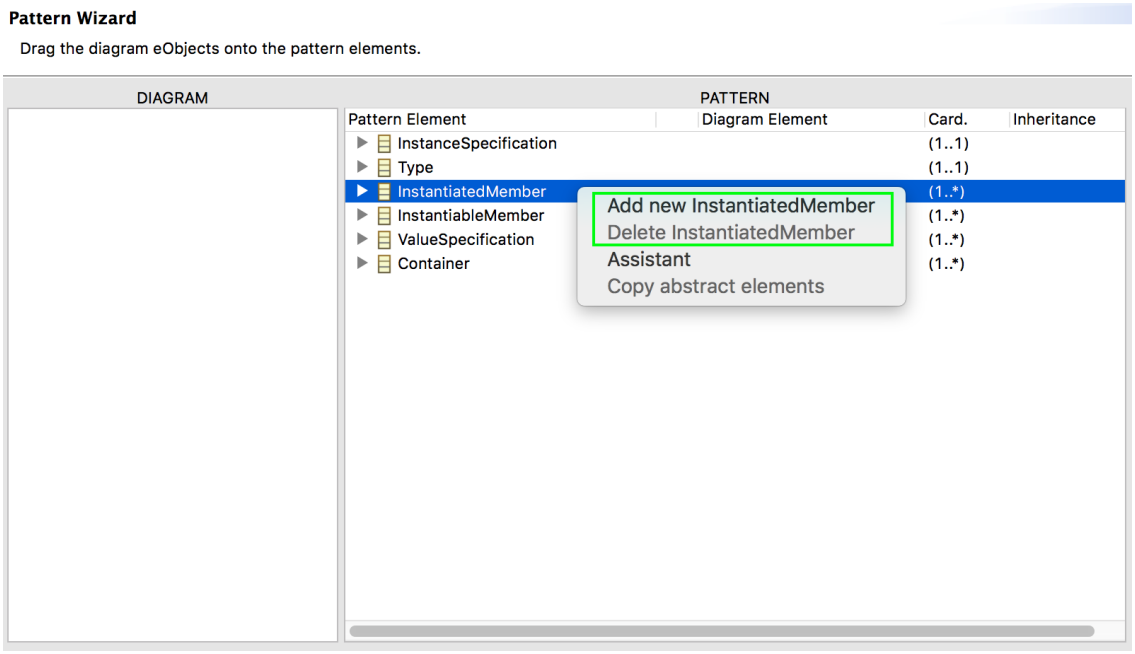


Figure 4.8: The add and delete function in an application wizard

DSL-tao sets max- and min-cardinalities for every pattern element to control the occurrences of instances of pattern elements. When creating a pattern, the cardinalities have to be specified in the repository file. The max- and min-cardinality of a pattern element is similar to the lower and upper bound of a model element in EMF. The max-cardinality is set to -1 means that the element can have multiple instances in an instance of the pattern.

When applying a pattern, the tool also provides application wizards to guide pattern application. Figure 4.8 shows the functions `add` and `delete` provided by DSL-tao to control the occurrences of pattern elements. Using these functions, we can manipulate instances of each single pattern element. Figure 4.9 demonstrates an instance of the Declaration pattern, in which, for example, the class `InstantiableMember` is instantiated three times while the relation `related` is instantiated twice.

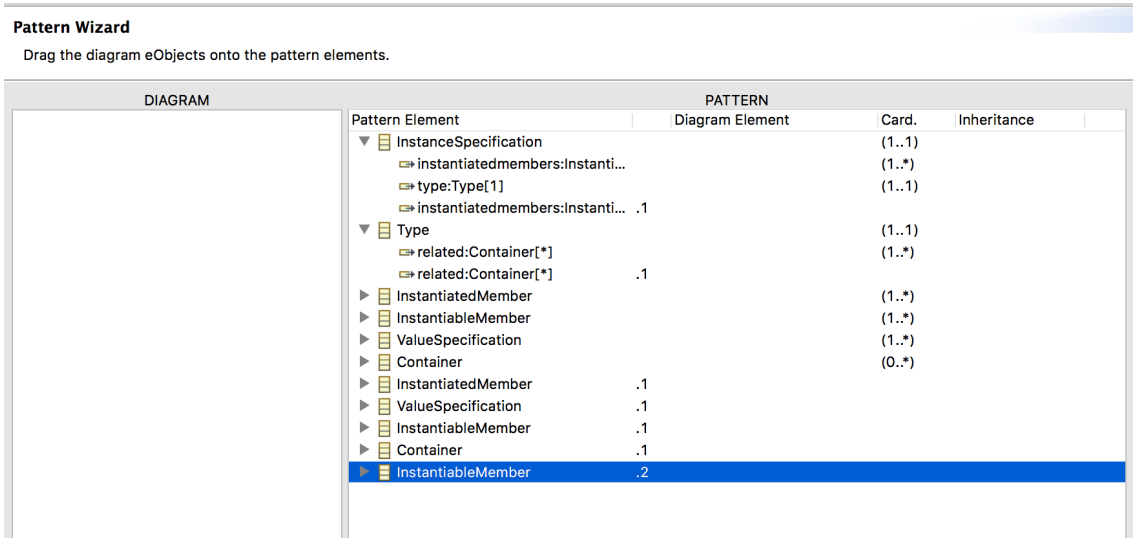


Figure 4.9: The instance of Declaration pattern in a application wizard

Although this tool provides mechanisms to control the occurrences of instances of pattern elements, we can not get the exact number of instances of a pattern element in the motivating example. The red circles in Figure 4.10 indicate the problem we encountered. What we want to create is the partial structure (shown in Figure 2.5) of the example meta-model. However, using the functions provided by DSL-tao, the highlighted part can not be created directly by applying the Declaration pattern once. We get a duplication of class `InstantiableMember` and `Valuespecification`.

The duplication of class `Valuespecification` does not cause a big problem, although it introduces redundancy in the meta-model. One of the duplicated `Valuespecification` can be removed, but if it remains, the meta-model is still correct. On the other hand, the duplication of class `InstantiableMember` may cause more problems. It not only introduces redundancy, but also causes an error in the meta-model. One of the instances of `InstantiableMember` has to be removed and the linked relations has to be changed correspondingly. In the example, for instance, the `InstantiableMember2` can be removed and the end point of the linked relation `definingmember` has to be changed to `InstantiableMember1`. This is another drawback of this tool, namely, modelers have to do extra work to modify pattern instance. When meta-model instances become bigger, making the correction manually would involve even more work, which makes this problem more serious. At the same time, when `InstantiableMember2` is removed and the end point of the linked relation `definingmember` is changed to `InstantiableMember1`, the traceability of the new `definingmember` relation between the pattern and the instance is lost. This means any associated ingredients of a pattern like model transformations can not be applied to the new relation because it is not traceable and it is not an instance of a pattern element any

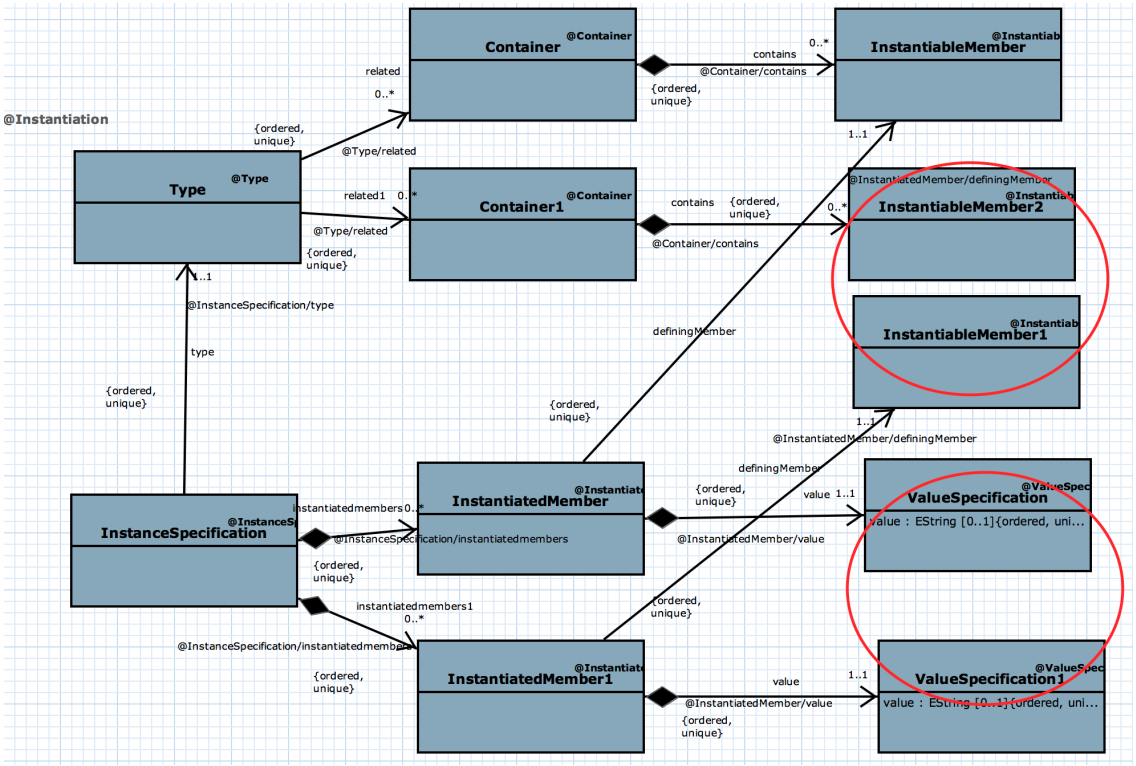


Figure 4.10: The pattern instance in diagram

more.

Apart from this problem, this tool has other drawbacks. Since it is a brand new tool released in the middle of this project, it contains bugs that need to be fixed and there is few guides/tutorial related to this tool.

Yet we decide to extend this tool because it can be directly used for Ecore meta-models and it has much potential to solve Altran's problem. For example, the occurrence control function provided by this tool is more flexible than DiffMerge/Patterns. Furthermore, this tool is a spin-off of pattern-based development research [29], which has proven its practicability. To extend DSL-tao, the first step is to create Altran's patterns using this tool. Then, in order to enrich the pattern application functionality of this tool and to address the problem stated above, we propose a merging function that can merge classes in a pattern instance before it being created. The implementation of the merge function is explained in Section 5.3.2.

4.4 Conclusions

In this chapter, we first summarized the investigated tools that are relevant to pattern application. Then we discussed two potential tools: EMF DiffMerge/Patterns and DSL-tao. Through the discussions, we see that DSL-tao is more applicable in our case. It not only can be used in Ecore meta-models, but also provides a flexible functionality to control the occurrences of pattern elements. However, DSL-tao needs to be extended to solve the problems we encountered. In next chapter, we will explain the design and implementation of two extensions of DSL-tao.

Chapter 5

Extensions of DSL-tao

In the previous chapter, we explain two potential tools DiffMerge/Patterns and DSL-tao in details. This chapter first explains our approaches for pattern specification and application, then describes the design and implementation of two extensions of DSL-tao (i.e. conduct the activity A3 stated in Section 1.2).

5.1 Approach for pattern specification

As stated in Chapter 3, the approaches for specifying patterns are categorized into three categories: literal approaches, logic approaches and DSML approaches. Generally, literal approaches are easy to understand, and it is a good way to use natural language to explain design patterns for the sake of understandability. However, a natural language can not be easily processed by machines, and such specification can not be directly applied to the target models like UML models and Ecore meta-models. Moreover, using a natural language makes pattern specification inaccurate and vague [5], thus inadequate for pattern application.

By contrast, logic approaches are more formal and precise, which provide rigorous and unambiguous specification of design patterns. As these approaches are based on mathematical notations and theories, which can be translated to machine-processable forms, they are good for the following activities such as pattern detection and pattern verification. The drawbacks of these approaches are that the mathematical knowledge is hard to understand, and pattern users have to know the mathematical knowledge before they can specify patterns.

DSML approaches are also formal approaches to specify design patterns. A DSML always focuses on a specific purpose. For example, the IncQuery Graph Pattern Language (IQPL) is designed for querying patterns in graphical models; whereas the Role-Based Modeling Language (RBML) is created for applying patterns in UML models. DSML approaches are good for pattern usage (i.e. various applications of patterns), yet they are not enough to explain a pattern clearly.

In Chapter XI of the book *Design Pattern Formalization Techniques* [36], the authors proposed a specification approach that combines literal specification and logic specification of a pattern. They defined a logic-based language to specify the structure of a pattern and an extension to describe other aspects of a pattern (e.g. intent, applicability, and collaboration).

In this work, we follow this idea to combine the literal specification and the DSML specification. We use the approach provided by DSL-tao to specify patterns, namely, using Ecore to model the structure of a pattern and specifying pattern roles in the DSML provided by DSL-tao. In addition, we provide an extension to describe other aspects of a pattern like problem, solution and consequence.

5.2 Approach for pattern application

Normally, applying a pattern in a model means adding instances of the pattern in a model; in other words, a pattern is instantiated when being applied. When it comes to instantiation in terms of meta-modeling, it is better to clarify the MOF architecture and the layers on which instantiation occurs.

Meta-Object Facility (MOF) defines a four-layer architecture for meta-modeling. In most cases, patterns are specified at the meta-model level and instantiated at the model level. For example, Figure 5.1 shows the conformance between MOF architecture and Epattern specifications. The Language Epattern is defined at the M3 layer as a language to model patterns. The patterns specified in Epattern language are considered at the meta-model level, hence their instances lie at the model layer. This means patterns are applied in the model level instead of in the meta-model level in many cases.

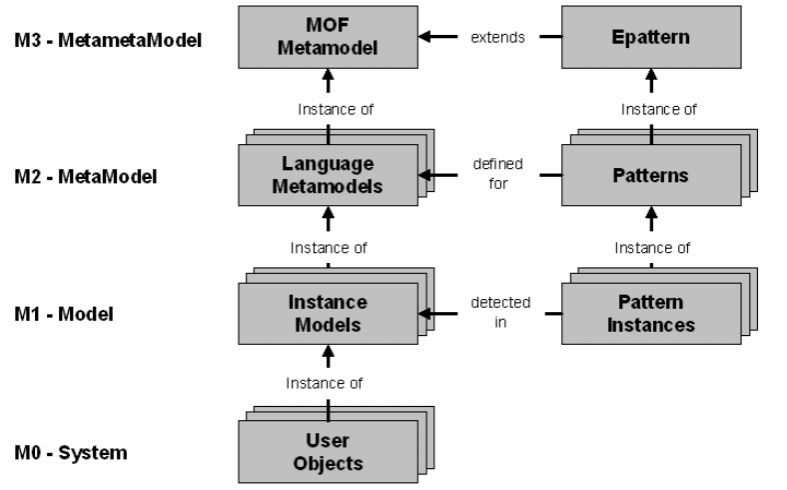


Figure 5.1: Conformance between MOF and Epattern specifications [16]

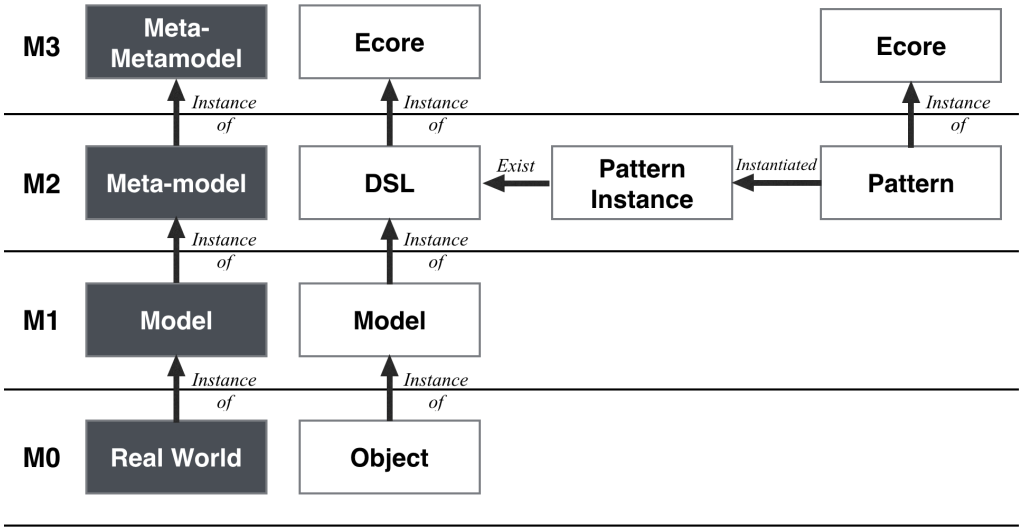


Figure 5.2: Patterns in MOF architecture

In this work, we focus on patterns applied in meta-models (at the M2 layer). In other words,

the instances of a pattern exist in meta-models. In DSL-tao, a pattern is specified using an Ecore meta-model and its pattern roles, hence we consider the specification also at the M2 layer. Put it simply, both patterns and their instances are at the meta-model layer. This is illustrated in Figure 5.2.

5.3 Tool support

Previous chapter introduces two potential tools: EMF DiffMerge/Patterns and DSL-tao. As analyzed in Chapter 4, we decided to extend DSL-tao in this work as the tool support for pattern specification and application for Ecore meta-models. The first step is to create our patterns in DSL-tao, but as DSL-tao mainly focuses on demonstrating the predefined patterns and their applications, pattern creation is missing from the tool and user guide. We had to figure out how to add our patterns as predefined patterns, for what we contacted the authors and studied their papers. The main steps are as follows:

1. Model a pattern structure in Ecore.
2. Create a figure to show the structure of the pattern and the cardinality of every pattern element.
3. Add the pattern to the repository.
4. Re-run DSL-tao.

The details about how to create a pattern in DSL-tao is explained in more detail in Section 4.3 and Appendix B.

Then we design two extensions to adapt DSL-tao for our case.

Extension 1. Add literal specification.

This is an extension for pattern specification. DSL-tao specifies patterns using Ecore meta-models and pattern roles (specified in the dslpatterns DSML), we modify the meta-model of DSL-tao to provide literal specification of a pattern such that other aspects of a pattern can be specified along with pattern roles.

Extension 2. Add a function to control occurrences of instances of pattern elements.

This is an extension for pattern application. DSL-tao uses min- and max-cardinality to set an interval to limit the possible number of instance(s) of a pattern element. Also it provides functions `add` and `delete` to duplicate and remove an instance of a pattern element. However, these functions are not enough to control the occurrences (see details in Section 4.3). To address this problem, we design an `merge` function to merge classes before an instance of patterns being created in a meta-model as the second extension.

5.3.1 Extension 1

This extension is designed to provide literal specifications in DSL-tao. When creating a pattern in DSL-tao, the structure of a pattern is specified in an Ecore meta-model and pattern roles are specified in the language dslpatterns. Hence we extend the meta-model of dslpatterns such that the other aspects of a pattern can also be specified in dslpatterns as well.

Figure 5.3 shows the extended part (excluding class `Pattern`) of the meta-model of DSL-tao (i.e. the meta-model of dslpatterns). The `Pattern` class is extended with zero or one `Explanation`, which may contain `Problem`, `Solution`, `Consequence`, `Version`, `Date` as well as `Author`. All these aspects inherit from the class `Aspects`. The aspects `problem`, `solution` and `consequence` are added because they are the essential aspects of design patterns [20] (another essential aspects

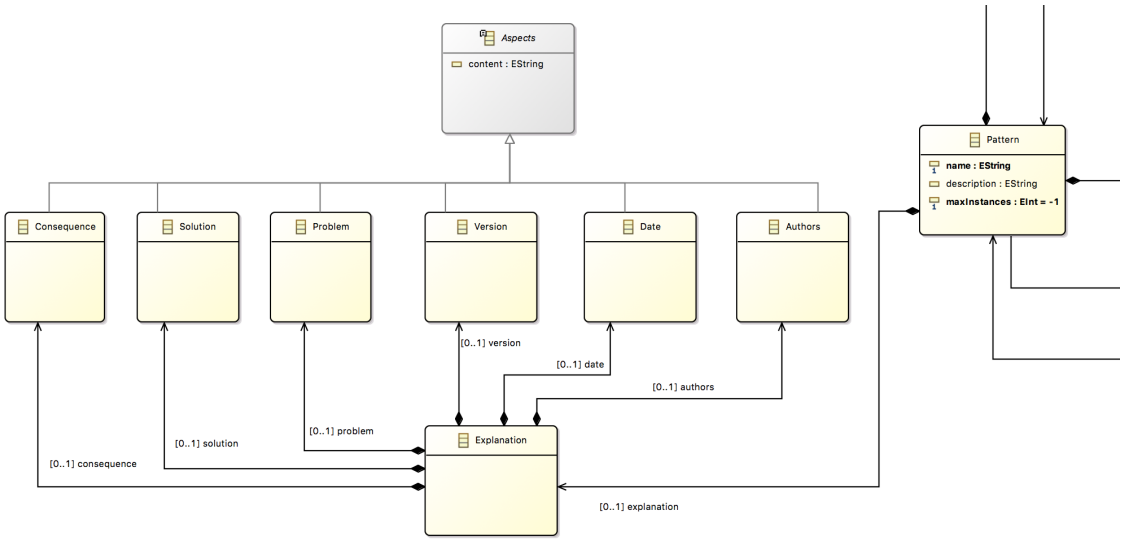


Figure 5.3: Excerpt of the extended meta-model of DSL-tao

`Name` have already defined in the original meta-model). The aspects `version`, `date` and `author` are added for the sake of software maintenance and evolution. In this way, a pattern can have not only a structure, but also an explanation.

As DSL-tao does not provide any user-friendly interface (e.g. wizards and menus) to create a pattern, we just follow their method to add literal specification, namely, create a pattern meta-model and add it to the repository file. When adding a pattern in the repository file, we also specify the aspects of a pattern if needed. Figure 5.4 shows the Declaration pattern specification with the extended explanation in the repository.

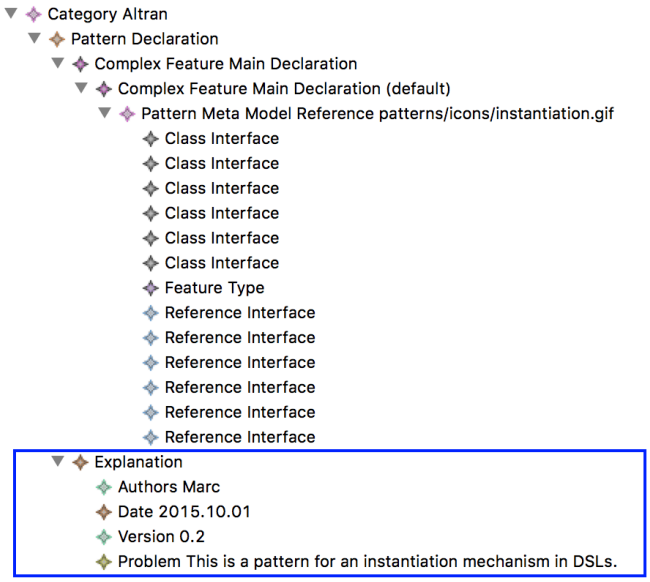


Figure 5.4: The Declaration pattern in the repository

In order to show the aspects of a pattern when a pattern is being applied, we also extend the application wizard that shows the structure of a pattern in DSL-tao. When applying a pattern, the aspects of a pattern will be shown in the application wizard if they are specified. Figure 5.5 shows

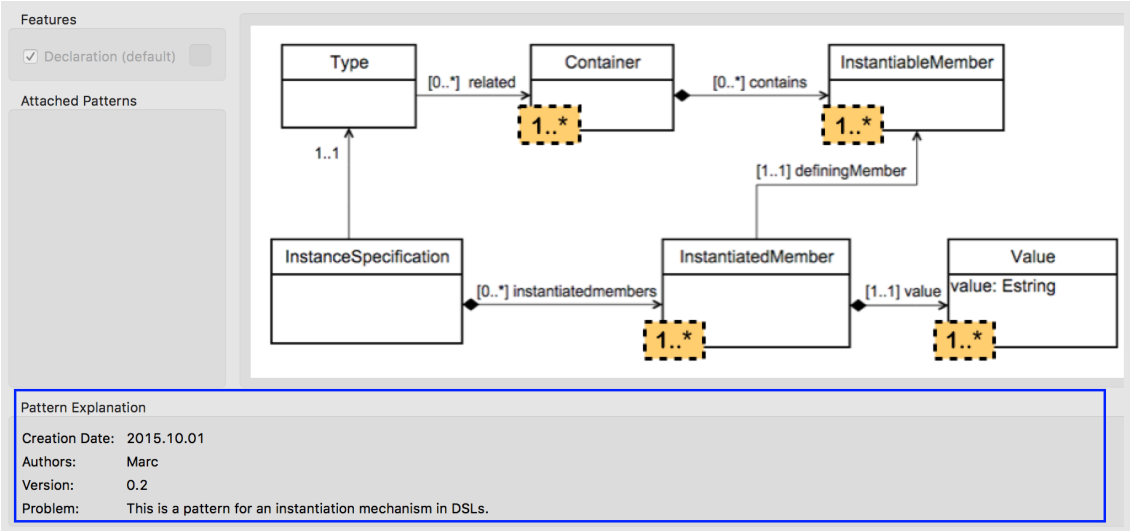


Figure 5.5: An example of extension 1

an application wizard that presents the aspects of the Declaration pattern specified in DSL-tao.

5.3.2 Extension 2 - Problem statement

In DSL-tao, every pattern element has a max- and min-cardinality that specify a lower bound and an upper bound to control the occurrences of instances of pattern elements in a pattern instance. Table 5.1 indicates the cardinalities of elements in the Declaration pattern.

Table 5.1: Cardinalities of elements in Declaration pattern

	Element Name	Min-cardinality	Max-cardinality
Class	Type	1	1
	Container	1	-1
	InstanceSpecification	1	1
	InstantiatedMember	1	-1
	InstantiableMember	1	-1
	ValueSpecification	1	-1
Reference	type: Type	1	1
	instantiatedmember: InstantiatedMember	1	-1
	value: ValueSpecification	1	-1
	related: Container	1	-1
	definingMembers: InstantiableMember	1	-1
	contains: InstantiableMember	1	-1
Feature	value: Estring	1	-1

The value -1 represents multiple instances. In the Declaration pattern, class `InstanceSpecification`, `Type` and reference `type` can only occur once in a pattern instance, other elements can have one or multiple instance(s) in a pattern instance.

When applying a pattern, DSL-tao provides two functions `add` and `delete` to duplicate or remove instances of a pattern element (see Figure 5.6). The effects of these two functions are listed in Table 5.2. When deleting a class, an error always occurs. This is because the delete class function is made to delete a class and its (directly and indirectly) linked elements, but the incoming references of the deleted class will not be deleted. Hence, if a class is deleted, its incoming references will point to an non-existing element, which is illegal in a meta-model.

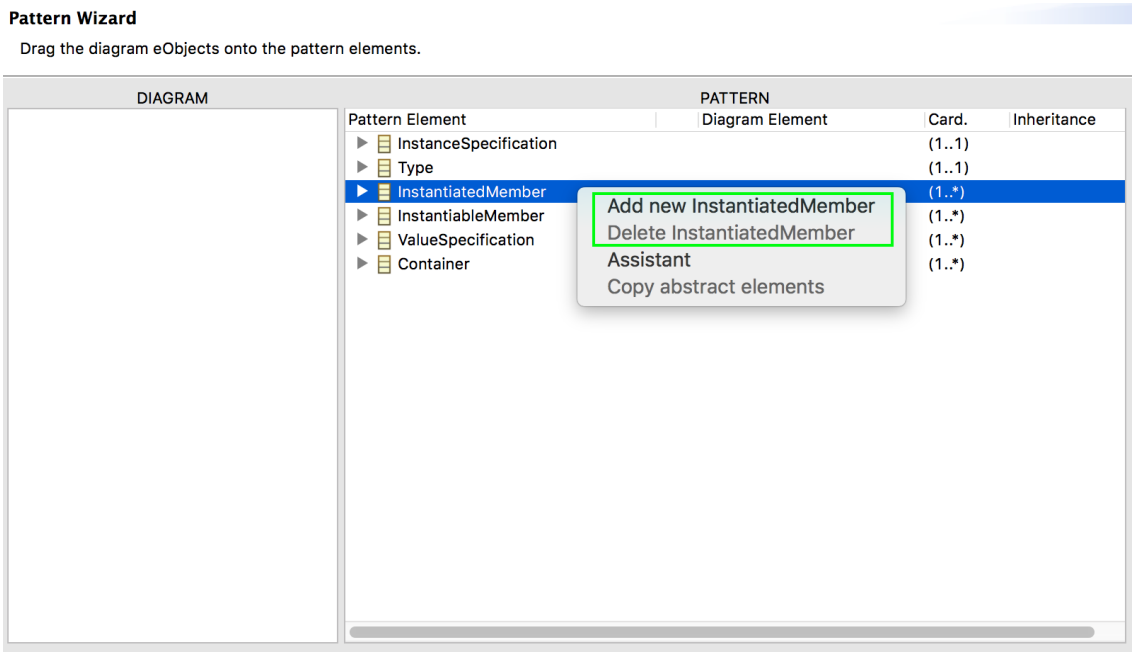


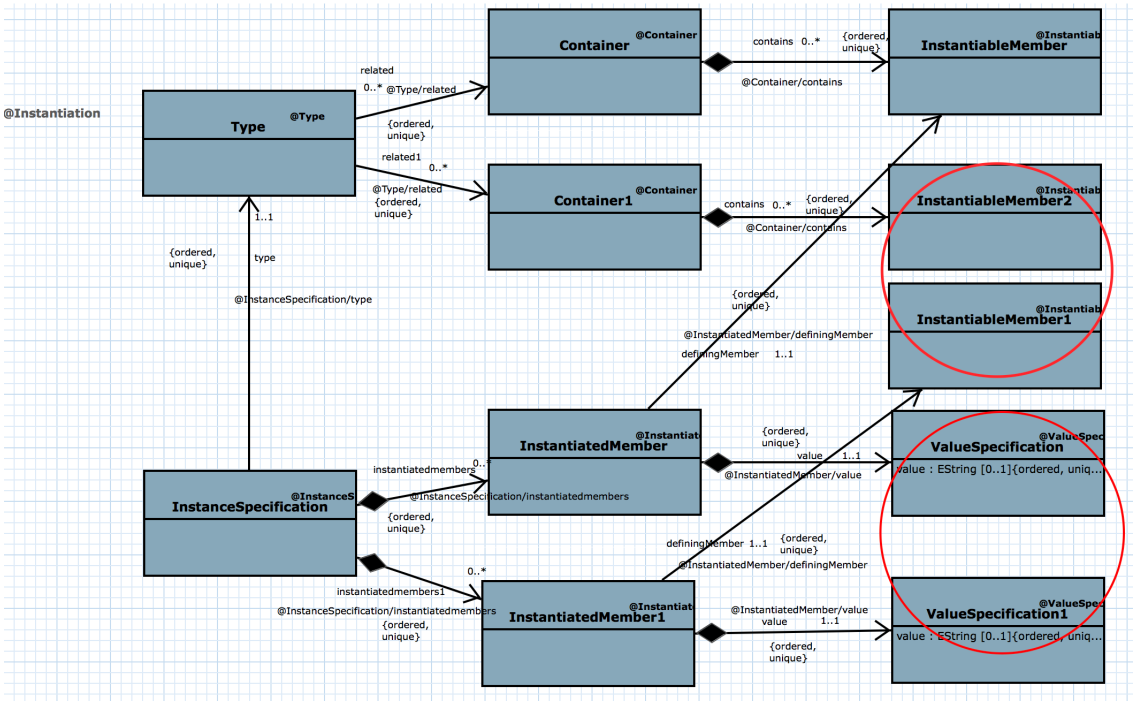
Figure 5.6: The functions provided in DSL-tao to modify instances of a pattern element

Table 5.2: The effects of `add` and `delete` functions

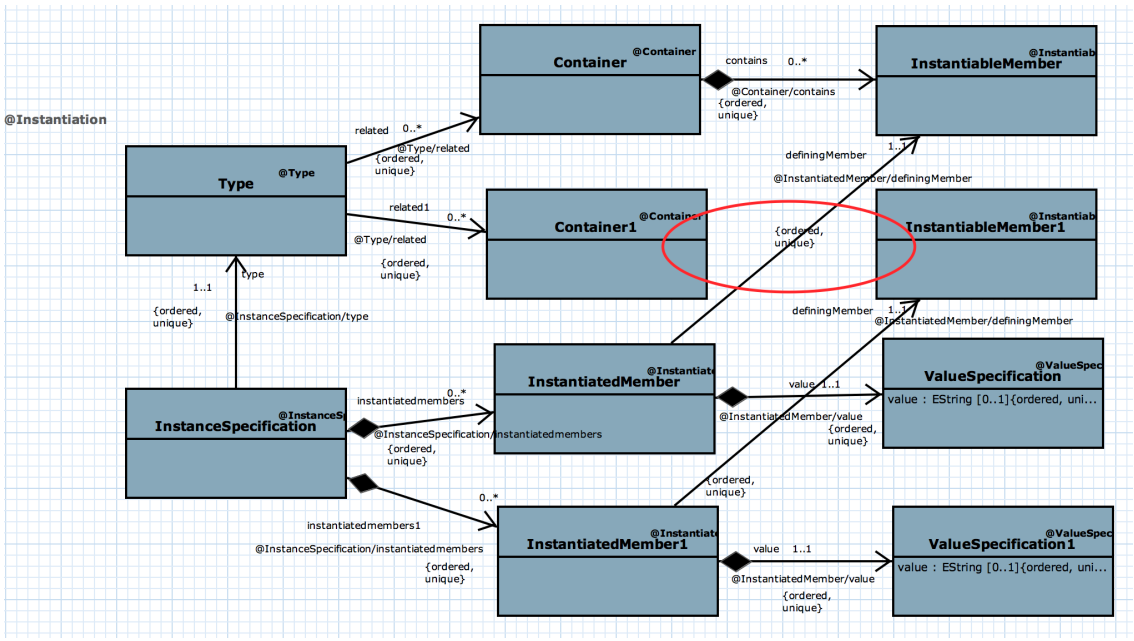
functions	classifiers	effects
add	Class	add a class
	Attribute	add an attribute
	Reference	add a reference, the linked Class A and all the following elements that are (directly or indirectly) linked to Class A
delete	Class	error function
	Attribute	delete an attribute
	Reference	delete a reference

In DSL-tao, it is impossible to directly create the highlighted structure (shown in Figure 2.5) in the example meta-model by applying the Declaration pattern once. Figure 5.7 shows two possible instances we created using DSL-tao when applying the Declaration pattern once. To create the highlighted part of the example meta-model, the first steps are to duplicate the references `related` and `instantiatedmember`. The result of the duplications of these two references is shown in Figure 5.7a. We can see that when duplicating the `related` reference, the `add` function creates a new `related` reference and its following elements (i.e. the class `Container`, `InstantableMember` and the reference `contains`). Similarly, when duplicating reference `instantiatedmember`, the class `InstantiatedMember1`, `ValueSpecification1`, `InstantiableMember1` and the linked references `value`, `definingMember` and `instantablemember` are created. As indicated in the red circle in Figure 5.7a, there are two duplicated `InstantiableMember` classes, which should be merged into one to get the highlighted structure in the example meta-model.

Indeed, we can use the `delete` function to remove redundant elements. It is impossible to delete one of the duplicated `InstantiableMember` classes because the "delete class" function causes an error, so we have to delete a reference. For example, Figure 5.7b shows the result of deleting reference `contains` between `Container1` and `InstantiableMember2`. The `delete` function removes not only the reference `contains`, but also the linked class `InstantiableMember2`. However, comparing to the highlighted part in the example meta-model, we lost the reference `contains` between `Contianer1` and `InstantiableMember1` (indicated by the red circle).



(a) Instance 1



(b) Instance 2

Figure 5.7: Two possible instances of the Declaration pattern

There are three possible ways to address the problems stated above. The most simplest way is to modify the meta-model after a pattern is being applied. For example, we can directly add a reference from `Containers1` to `InstantiableMember1` in Figure 5.7b to get the needed structure. If we do so, the added reference will not be considered as an instance of the pattern element, so the traceability between this element and the corresponding pattern elements will not be maintained.

Also it is possible to apply a pattern multiple times to get the needed structure. For example, we can apply the Declaration pattern again (total twice) to the meta-model in Figure 5.7b to get the highlighted part in the example meta-model. However, how many times a pattern has to be applied to create a needed structure is unknown and many redundant traces between pattern elements and their instances will be introduced in this way.

The third way to deal with the problems encountered is to provide more flexible tooling to support this feature in development. Thus we propose to add a `merge` function to complement the `add` and `delete` function, the implementation of which is introduced in next section.

5.3.3 Extension 2 - The merge function

In this section, we introduce the design and implementation of the second extension - the `merge` function. Before implementing this function, we analyze the requirements of this function based on three classifiers: attribute, reference, class.

- **Merge attributes.** Now attributes can be add or delete (if the `delete` function works well) in DSL-tao, and normally there is no need to merge two attributes in development, so we do not consider merging attributes separately.
- **Merge references.** Regarding the merging of two references, a user has to consider how to deal with the linked classes of these two references, namely, whether the linked classes need to be merged or not. In fact, it boils down to merge classes.
- **Merge class.** Merging classes is a commonly needed operation when applying a pattern. The prerequisite of merging two class is that the classes are derived from the same pattern element, such that they should have the same attributes and references (with same name but different name suffix). It is meaningless to merge any two different classes. The principle of merging two classes is simple: keep all references that connected with these two classes and delete one class with its attributes.

We conclude that the requirement of the `merge` function is to merge classes rather than attributes and references. Hence we enable the `merge` function only when two classes that have the same name are selected in pattern application. In other words, if two non-class elements are selected when applying a pattern, the `merge` function will not be enabled. To implement it, we make a new drop-down menu for the `merge` function. When any two elements are selected in the Pattern Wizard, the `merge` function menu will appear (see Figure 5.8), but the function is enabled only when the selected elements are two classes that have the same name. The original `add` and `delete` function menu will appear (see Figure 5.6) when only one pattern element is selected.

The idea behind the `merge` function is shown in Algorithm 1. The input of this function are the `instanceDiagram` that includes all the elements in an instance and two classes `class1` and `class2` that need to be merged. Before starting to merge two classes, the attributes number and the creation order (i.e. the suffix number of a class) of the classes are compared. The principle is that if the number of their attributes are different, the class who has more attributes will remain; if the number of their attributes are the same, the class who has the smaller order number (i.e. who is created first) will remain. This is shown from line 2 to line 16 in Algorithm 1. After that, the real merging starts. The aim is to remove `class2` and change all references connected to `class2` to `class1`. First, we change the end point of all incoming references of `class2` to `class1`, which is indicated from line 18 to 23 in Algorithm 1. Secondly, we change the parent of all outgoing references of `class2` to `class1` and add these outgoing references to the child list of `class1`, which is indicated from line 25 to 32. Finally, we remove all the attributes of `class2` and the class itself from line 34 to 40.

Now you may wonder how the number of attributes of two classes that are derived from the same pattern element can be different. This is because a user can use `add`/`delete` function to duplicate/remove an attribute of a class in some cases. No matter the number of attributes of two classes are the same or not, the attributes of the class that has less attributes are covered by

Algorithm 1 The `merge` function

```
1: function MergeFunction(instanceDiagram, class1, class2)
2:     attrNum1 ← getAtrributeNum(instanceDiagram, class1)
3:     attrNum2 ← getAtrributeNum(instanceDiagram, class2)
4:     if attrNum2 > attrNum1 then
5:         class ← class1
6:         class1 ← class2
7:         class2 ← class1
8:     else
9:         if attrNum2 = attrNum1 then
10:            if class1.order > class2.order then
11:                class ← class1
12:                class1 ← class2
13:                class2 ← class1
14:            end if
15:        end if
16:    end if
17:
18:    incomeRefList ← getIncomeRefs(instanceDiagram, class2)
19:    if incomeRefList ≠ NIL then
20:        for each ref in incomeRefList do
21:            ref.orderPointer ← class1.order
22:        end for
23:    end if
24:
25:    refChildrenList1 ←getRefChildren(instanceDiagram, class1)
26:    refChildrenList2 ←getRefChildren(instanceDiagram, class2)
27:    if refChildrenList2 ≠ NIL then
28:        for each child in refChildrenList2 do
29:            refChildrenList1 ← refChildrenList1 + child
30:            child.parent ← class1
31:        end for
32:    end if
33:
34:    attrChildrenList ←getAttrChildren(instanceDiagram, class2)
35:    if attrChildrenList ≠ NIL then
36:        for each achild in attrChildrenList do
37:            deleteAttribute(instanceDiagram, achild.index)
38:        end for
39:    end if
40:    remove(instanceDiagram, class2)
41:
42: end function
```

the attributes of the class that has more attributes. Therefore, we keep the class that has more attributes in the merging function.

In Algorithm 1, there are some auxiliary functions used to assist the merging function, which are listed in Table 5.3.

Table 5.3: The auxiliary functions in the merging function

Name	Parameters	Function
getAttributeNum	instanceDiagram, class	calculate the number of attributes of a class
getIncomeRefs	instanceDiagram, class	return a list of all incoming references of a class
getRefChildren	instanceDiagram, class	return a list of all outgoing references of a class
getAttrChildren	instanceDiagram, class	return a list of all attributes of a class
deleteAttribute	instanceDiagram, attribute	delete an attribute from the instance diagram
remove	instanceDiagram, class	delete a class from the instance diagram

We have made several examples to test this function, the correct test results give confidence in the correctness of the `merge` function.

Using this function, we can merge the `InstantiableMember1` and `InstantiableMember2` in Figure 5.7a before creating the instance. As shown in Figure 5.8, we have already duplicated the two references and get the same structure in Figure 5.7a. Then we chose the duplicated `InstantiableMember` classes to merge them. To get the needed structure, we also merged the classes `ValueSpecification` and `ValueSpecification1`. The result (i.e. a pattern instance) we get is shown in Figure 5.9, the structure of which is the same as that of the highlighted part in the example meta-model. To this end, we can create the need structure using DSL-tao.

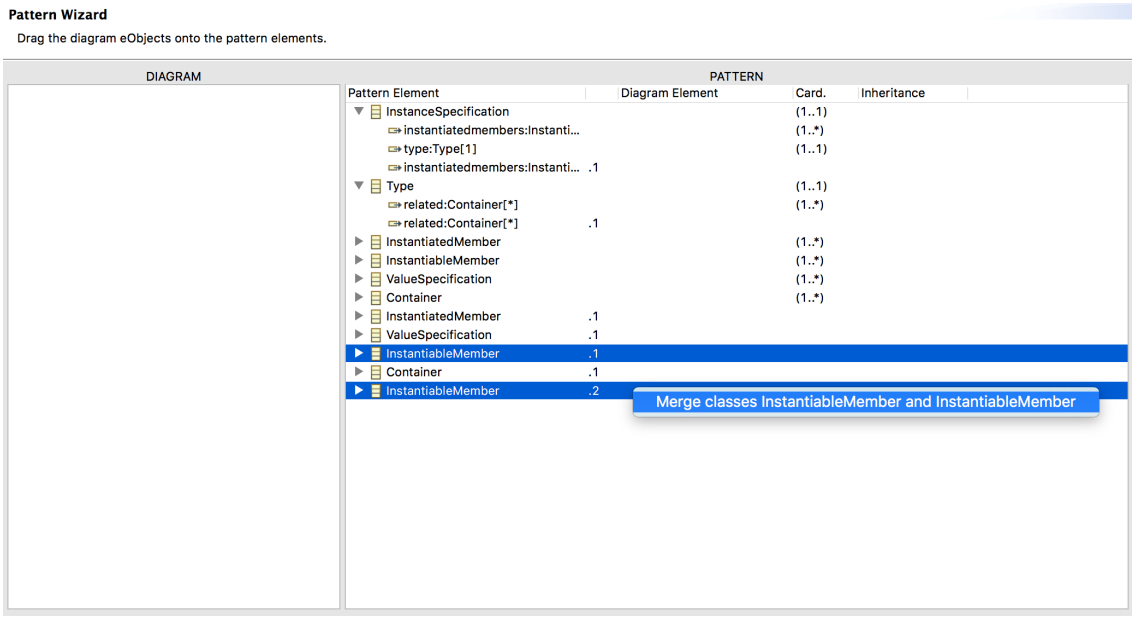


Figure 5.8: The merge function in DSL-tao

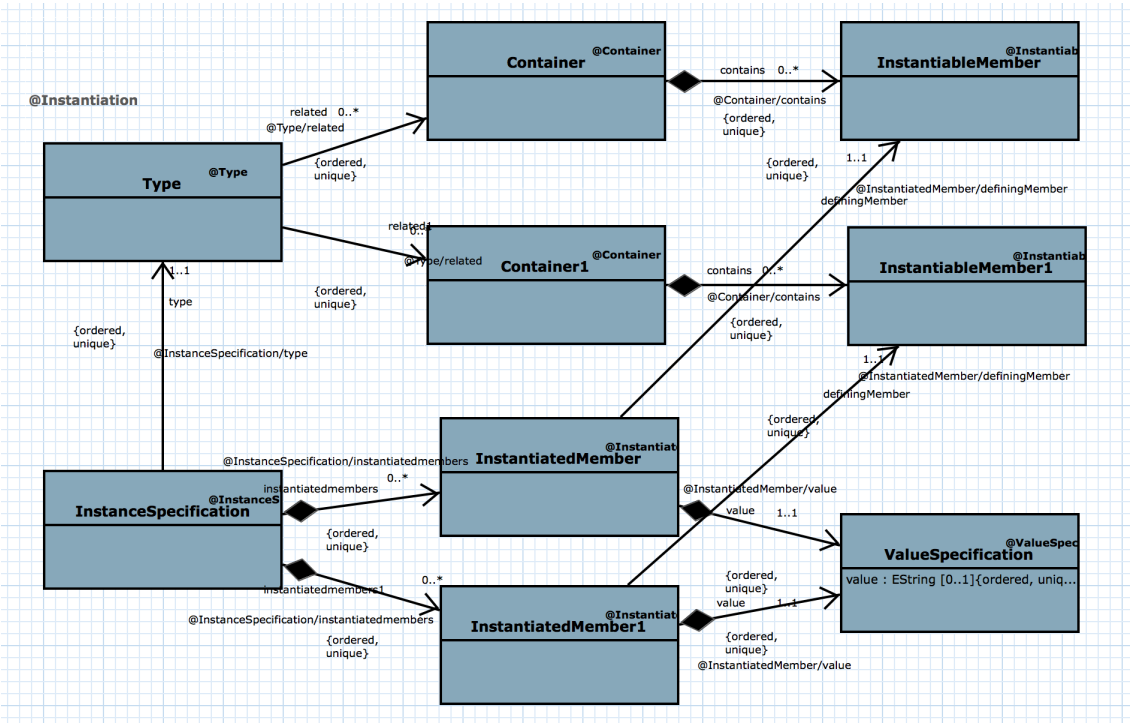


Figure 5.9: The result instance after using merge function

5.4 Conclusions

This chapter explains the design and implementations of two extension of DSL-tao. The Extension 1 is designed to enrich the pattern specification approach provided by DSL-tao. After adding the literal specification, a user can record information of a pattern for better understanding of a pattern and for its maintenance in the future.

The Extension 2 is implemented to complement the functions provided by DSL-tao to control the occurrences of instances of pattern elements. Using this extension, we can create the example meta-model by applying the Declaration pattern once. These two extensions are made based on the results of literature study and tool exploration, and they are useful for solving research questions and accomplishing Altran's assignment. In next chapter, we will draw conclusions for this project and describe possible work in the future.

Chapter 6

Conclusions and Future Work

In the previous chapter, we explained the design and implementation of two extensions of DSL-tao. The aim of the extensions is to provide better tool support for pattern specification and application in meta-models. In this chapter, we draw conclusions and state the possibilities of future work.

6.1 Conclusions

In this section, we draw conclusions by answering the research questions and summarizing the other chapters of this thesis. At the beginning of this work, we have proposed two research questions that needed to be solved.

> **RQ1.** *What approaches and tooling are being used to specify and apply design patterns in meta-models in Ecore and UML?*

> **RQ2.** *How to control the occurrences of structure elements of a pattern when the pattern is being instantiated and what functions should be provided by tooling to support the control method?*

To answer **RQ1**, we have studied literature and tools that are relevant to pattern specification and application in meta-models. Chapter 3 summarized three categories of pattern specification approaches - the literal approaches, logic approaches and DSL approaches used in meta-models in Ecore and/or in UML models to specify patterns, and analyzed suitability of each category approaches by matching their cons and pros to our requirements. Furthermore, we explained the role-based pattern application approach and its application in UML models in Chapter 3. The existing tools that are used for pattern application are summarized in Chapter 4. We first briefly introduced a collection of relevant tools, then elaborated two potential tools that are useful in our case by discussing their functionality and possible extensions.

The second research question **RQ2** can be divided to two subquestions.

> **RQ2-1.** *What approaches are being used to control the occurrences of elements of a pattern when the pattern is being applied?*

> **RQ2-2.** *What functionality should be provided by tooling to support the control the occurrences of elements of a pattern?*

The question **RQ2-1** has been answered in Chapter 4. The EMF DiffMerge/Patterns tool uses a boolean value named Unique to control the occurrences of a pattern element in an instance. If the Unique value is true, a pattern element can only occur once in a pattern instance. While if the value is false, an element can occur multiple times in an instance. The tool DSL-tao sets an

interval value (i.e. cardinalities) to control the occurrences of a pattern element. The min- and max-cardinality indicate the lower and upper bound of the interval. The min-cardinality limits the minimal number of instances that a pattern element can have, while the max-cardinality restricts the maximal number of instances of a pattern element.

The question **RQ2-2** was answered in Chapter 4 and 5. The EMF DiffMerge/Patterns tool provides functionality to specify how many instances of all "non-Unique" pattern elements should be created when applying a pattern. However, this function is not flexible enough because it can not process pattern elements individually. The tool DSL-tao provides two functions `add` and `delete` to handle the occurrences of pattern elements. But still we encountered problems when instantiating a pattern. In Chapter 5, we extended this tool with a `merge` function to complement the functions in DSL-tao. Now we consider that at least the `add`, `delete` and `merge` functions should be provided to handle the occurrences of pattern elements.

To this end, we consider that the research questions are answered.

Also, at the beginning of this project, Altran's assignment is formulated as follows:

> **Altran's assignment:** *we are seeking approaches for pattern specification and application in Ecore meta-models, and we need tool support to apply our patterns in the development of DSLs.*

We have summarized many approaches for pattern specification and application in Ecore meta-models and provided a list of tools for applying patterns. The DSL-tao tool with the extensions makes it possible to use Altran's patterns in software development. Hence, we consider that we have accomplished the Altran's assignment.

6.2 Future work

In this section, we discuss the possible research directions in the future.

6.2.1 Validation of the extensions

We have made two extensions to complement the functions in DSL-tao to support pattern specification and application. The extensions work well in the motivating example, which proves that the extensions are necessary. However, the extensions have not been validated in the real world. This can be done by applying the extensions in the development of DSLs in Altran. As the validation work takes more time, we formulate it as future work.

6.2.2 Validation of Altran's patterns

In this project, we use the Declaration pattern created in Altran as a motivating example to illustrate the functions of tools and the extensions. Besides the Declaration pattern, there are also some other patterns created in Altran. Although these patterns are abstracted from many DSLs, the correctness and practicability are not validated. In the future, these pattern can be validated by being applied to new meta-models of DSLs.

6.2.3 Further extension for pattern application

A pattern is an abstraction of model fragments, so when applying/instantiating it, the structure of a pattern has to be modified in order to create the needed instances. In this work, we limit the modification to addition and deletion of pattern element, but in the real-world application, the modifications are more complex. For example, a normal class element in a pattern may need to be instantiated to an abstract class; in addition, a relation element in a pattern may need to be instantiated to a class with multiple relations in an instance. Hence, the pattern application tool like DSL-tao has to be improved to provide more advanced functions to manipulate pattern elements before an instance is created.

On the other hand, it is possible to modify pattern specification approaches to make the application more flexible. For example, OCL constrains can be added to a pattern specification to limit the the instance number of a pattern element. Indeed, new pattern specification approaches can be developed for the sake of flexible pattern application.

6.2.4 Pattern variants

As stated in Section 6.2.3, a pattern is an abstraction of model fragments. Therefore, the structure of a pattern is not always the structure needed in meta-models. This is a reason why we need to study how to apply patterns in meta-models. One way to solve the "near-match" problem between a pattern and its instance is to provide tool support to modify patterns before being applied. Another way is to make pattern variants for different application situation. Through extensive study and analysis of meta-model zoo, we may find more variants of a pattern. However, what kinds of modifications for a pattern should be defined in a pattern variant and what kinds of modifications should be supported by tooling is unclear.

6.2.5 Blueprint of an advanced language workbench

In the development of a DSL tool chain, a meta-model is an important starting point: other contributions to the tool chain like representation options, constraints or queries, transformation units etc. depend heavily on the developed meta-models. It is important to capture the experience (i.e. the good- and bad-practices) and be able to reuse good practices in new developments. Design patterns capture good practices and provide proven solutions to the recurring language problems. Therefore, the first step is to apply design patterns in meta-models. This is exactly what we did in this work. In the future, other contributions like notation rules, model transformations and constraints in the development of DSL should be considered.

To make an advanced modeling workbench that supports pattern application, the first requirement is that the workbench has to provide mechanisms to specify and apply patterns. Secondary, associated contributions in the further development of the tool chain can be automated based on the application of a pattern. This requires the pattern and its application to be traceable throughout the development of the tool chain. The ultimate goal is to make a language workbench that allows for adding markers (e.g. properties or keywords) to meta-models of DSLs such that the associated contributions can be derived automatically using the markers. The actual meta-model with the applied patterns would then only be an intermediate (derived) artifact in the development process.

Bibliography

[1] UML Collaboration. http://www.uml-diagrams.org/collaboration-diagrams/collaboration.html.

[2] UML CollaborationUse. http://www.uml-diagrams.org/collaboration-diagrams/collaboration-use.html.

[3] Scott Ambler. A Realistic Look at Object-Oriented Reuse. January 1, 1998. http://www.drdobbs.com/a-realistic-look-at-object-oriented-reus/184415594.

[4] Mira Balaban, Azzam Maraee, Arnon Sturm, and Pavel Jelnov. A pattern-based approach for improving model quality. *Software and System Modeling*, 14(4):1527–1555, 2015.

[5] Aline Lúcia Baroni, Yann-Gal Guëhéneuc, and Hervé Albin-Amiot. Design patterns formalization. Technical report, Ecole Nationale Supérieure des Techniques Industrielles et des Mines de Nantes, June 2003. Technical Report 03/3/INFO.

[6] Ian Bayley and Hong Zhu. Formalising design patterns in predicate logic. In *5th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007), 10-14 September 2007, London, England, UK*, pages 25–36, 2007.

[7] Ian Bayley and Hong Zhu. Specifying behavioural features of design patterns in first order logic. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2008, 28 July - 1 August 2008, Turku, Finland*, pages 203–210, 2008.

[8] Ian Bayley and Hong Zhu. Formal specification of the variants and behavioural features of design patterns. *Journal of Systems and Software*, 83(2):209–221, 2010.

[9] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A graph query language for EMF models. In *Theory and Practice of Model Transformations - 4th International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings*, pages 167–182, 2011.

[10] Jean Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[11] Jing Dong, Paulo S. C. Alencar, and Donald D. Cowan. Ensuring structure and behavior correctness in design composition. In *7th IEEE International Symposium on Engineering of Computer-Based Systems (ECBS 2000), 3-7 April 2000, Edinburgh, Scotland, UK*, page 279, 2000.

[12] Amnon H. Eden, Yoram Hirshfeld, and Amiram Yehudai. Lepus - a declarative pattern specification language. Technical report, 1998.

[13] Maged Elaasar. Computer method and system for pattern specification using meta-model of a target domain. Patent. US 20080127049A1. May 29, 2006.

[14] Maged Elaasar, Lionel C. Briand, and Yvan Labiche. A metamodeling approach to pattern specification and detection. In *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, pages 484–498, 2006.

[15] Maged Elaasar, Lionel C. Briand, and Yvan Labiche. VPML: an approach to detect design patterns of mof-based modeling languages. *Software and System Modeling*, 14(2):735–764, 2015.

[16] Maged Elaasar, Lionel C. Briand, and Yvan Labiche. A metamodeling approach to pattern specification and detection. Technical report, Carleton University, March 2006. Technical Report SCE-06-068.

[17] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[18] Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song. A uml-based metamodeling language to specify design patterns. In *WiSME*, 2003.

[19] Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song. A uml-based pattern specification technique. volume 30, pages 193–206, 2004.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[21] Abel Gómez and Isidro Ramos. Automatic tool support for cardinality-based feature modeling with model constraints for information systems development. In *Information Systems Development, Business Systems and Services: Modeling and Development [Proceedings of ISD 2010, Charles University in Prague, Czech Republic, August 25-27, 2010]*, pages 271–284, 2010.

[22] Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel. Precise modeling of design patterns. In *≪UML≫ 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000. Proceedings*, pages 482–496, 2000.

[23] Wai-Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. UMLAUT: an extendible UML transformation framework. In *The 14th IEEE International Conference on Automated Software Engineering, ASE 1999, Cocoa Beach, Florida, USA, 12-15 October 1999*, pages 275–278, 1999.

[24] David Mapelsden, John Hosking, and John Grundy. Design pattern modelling and instantiation using dpml. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, CRPIT '02, pages 3–11, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[25] David Maplesden, John G. Hosking, and John C. Grundy. A visual language for design pattern modelling and instantiation. In *2002 IEEE CS International Symposium on Human-Centric Computing Languages and Environments (HCC 2001), September 5-7, 2001, Stresa, Italy*, pages 338–339, 2001.

[26] David Maplesden, John G. Hosking, and John C. Grundy. Design Pattern Modelling and Instantiation using DPML. *Proceedings of the 40th International Conference on Tools Pacific: Objects for internet, mobile and embedded applications (CRPIT '02), February 18-21, 2002, Sydney, Australia*, 10:3–11, 2002.

[27] Jean marie Favre. Towards a basic theory to model model driven engineering. In *Proceedings of the UML2004 Int. Workshop on Software Model Engineering*, 2004.

[28] Tommi Mikkonen. Formalizing design patterns. In *Forging New Links, Proceedings of the 1998 International Conference on Software Engineering, ICSE 98, Kyoto, Japan, April 19-25, 1998.*, pages 115–124, 1998.

[29] Ana Pescador, Antonio Garmendia, Esther Guerra, Jesús Sánchez Cuadrado, and Juan de Lara. Pattern-based development of domain-specific modelling languages. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, Canada, September 30 - October 2, 2015*, pages 166–175, 2015.

[30] Christian Schäfer, Thomas Kuhn, and Mario Trapp. A pattern-based approach to DSL development. In *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '11, Proceedings of the compilation of the co-located workshops, DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, and VMIL'11, Portland, OR, USA, October 22 - 27, 2011*, pages 39–46, 2011.

[31] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[32] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[33] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[34] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[35] Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel. Design Patterns Application in UML. In *ECOOP 2000 - Object-Oriented Programming, 14th European Conference, Sophia Antipolis and Cannes, France, June 12-16, 2000, Proceedings*, pages 44–62, 2000.

[36] Toufik Taibi. *Design Pattern Formalization Techniques*. IGI Global, Hershey, PA, USA, 2007.

[37] Toufik Taibi and David Ngo Chek Ling. Formal specification of design patterns - A balanced approach. *Journal of Object Technology*, 2(4):127–140, 2003.

[38] Didier Vojtisek and Jean-Marc Jézéquel. MTL and Umlaut NG - Engine and Framework for Model Transformation. *ERCIM News 58*, 58, 2004.

[39] Wikipedia. Domain-specific language. `https://en.wikipedia.org/wiki/Domain-specific_language`.

[40] Colin Yu. Model-driven and pattern-based development using rational software architect: Part 2. model-driven development tooling support in ibm rational software architect. `http://www.ibm.com/developerworks/rational/library/07/0116_yu/`.

[41] Hong Zhu, Ian Bayley, Lijun Shan, and Richard Amphlett. Tool support for design pattern recognition at model level. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2009, Seattle, Washington, USA, July 20-24, 2009. Volume 1*, pages 228–233, 2009.

Appendix A

EMF DiffMerge/Patterns Guide

This appendix demonstrates how the DiffMerge/Patterns tool works in UML Designer step by step by using the Declaration pattern to create the motivating example meta-model. After UML Designer and DiffMerge/Patterns are being installed, a "Patterns" item will be added to the pop-up menu when you right click on a diagram (see Figure A.1). Now we start to create the example meta-model.

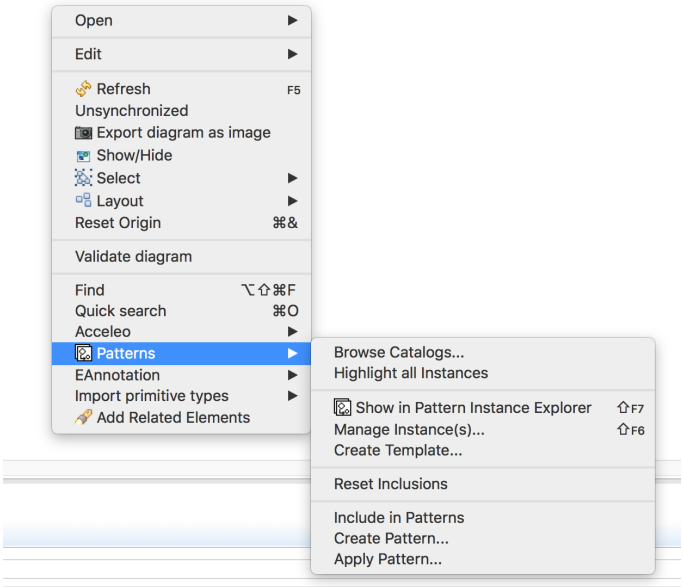


Figure A.1: DiffMerge/Patterns menu

Step1: create the example pattern.

(a) First, we model the Declaration pattern in UML (shown in Figure A.2, which is exactly the same as the example pattern).

(b) Then, select all elements in the UML pattern model, right-click and select `Patterns -> Create Pattern...`to create a pattern. The creation wizard (see Figure A.3) will pop up.

(c) The DiffMerge/Patterns tool stores patterns in "catalogs" files whose names end with `.patterns`. In the "Properties" tab, we create a catalog using the"New" button. Also,

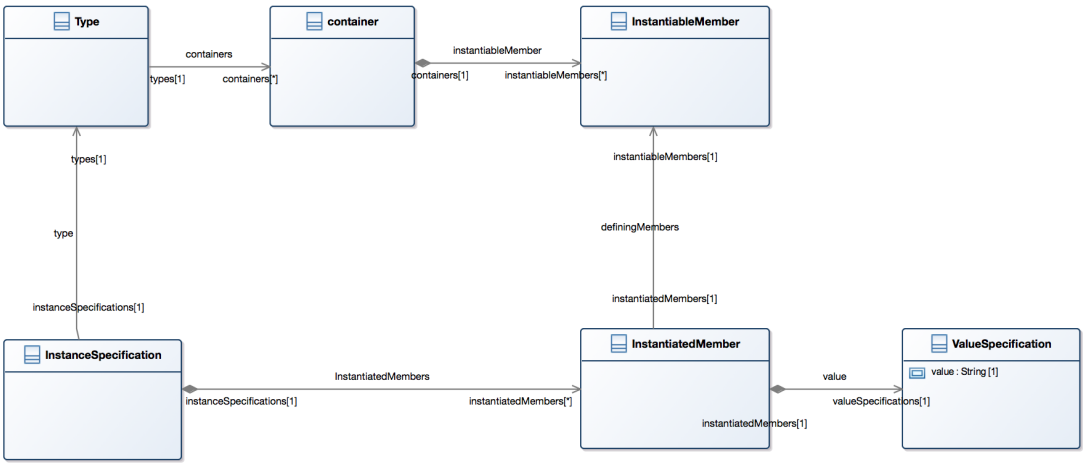


Figure A.2: The Declaration pattern in UML

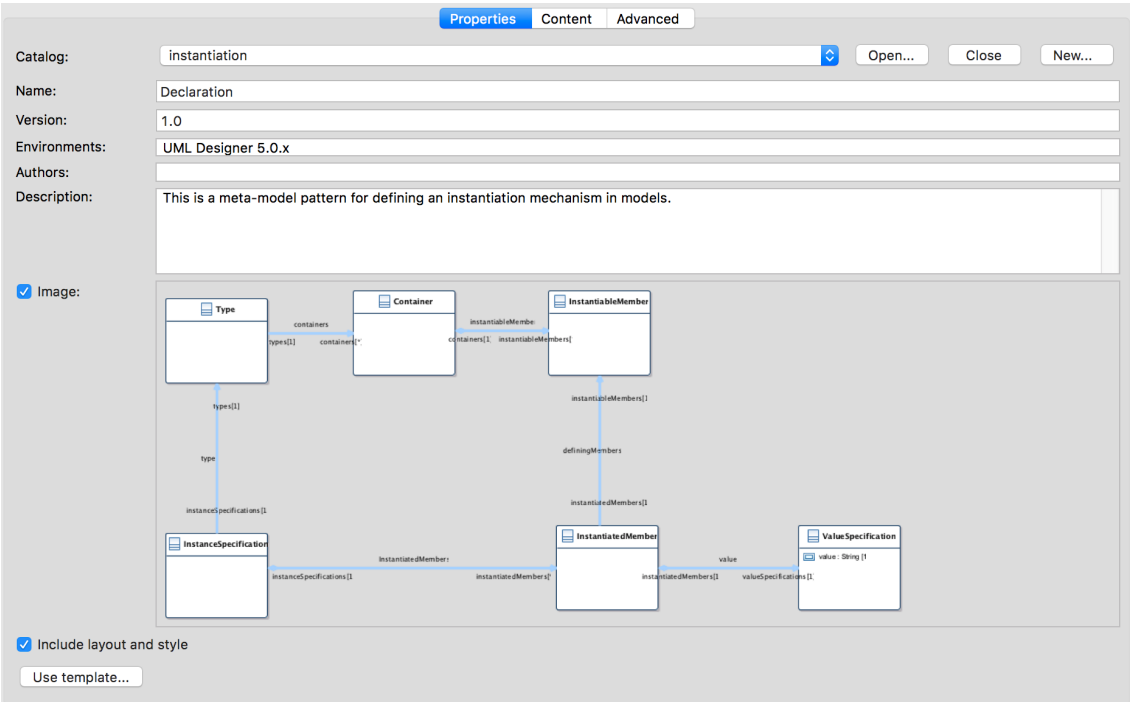


Figure A.3: Setting of “Properties” tab

we specify the information related to this pattern (e.g. name, author and description) in this tab. As shown in Figure A.3, the layout and style of this pattern will also be stored as we tick the “Include layout and style” box.

(d) In the “Content” tab (shown in Figure A.4), all elements of a pattern and their roles are listed. Now we need to add dependencies to the pattern using button “Include all dependencies...” in the lower left corner. In this case, the dependencies of this pattern is EString data type defined in UML.

(e) Now we add roles for pattern elements. The default role of all elements is “Classifiers”. In this case, we create new roles for every class object in the pattern. The name of

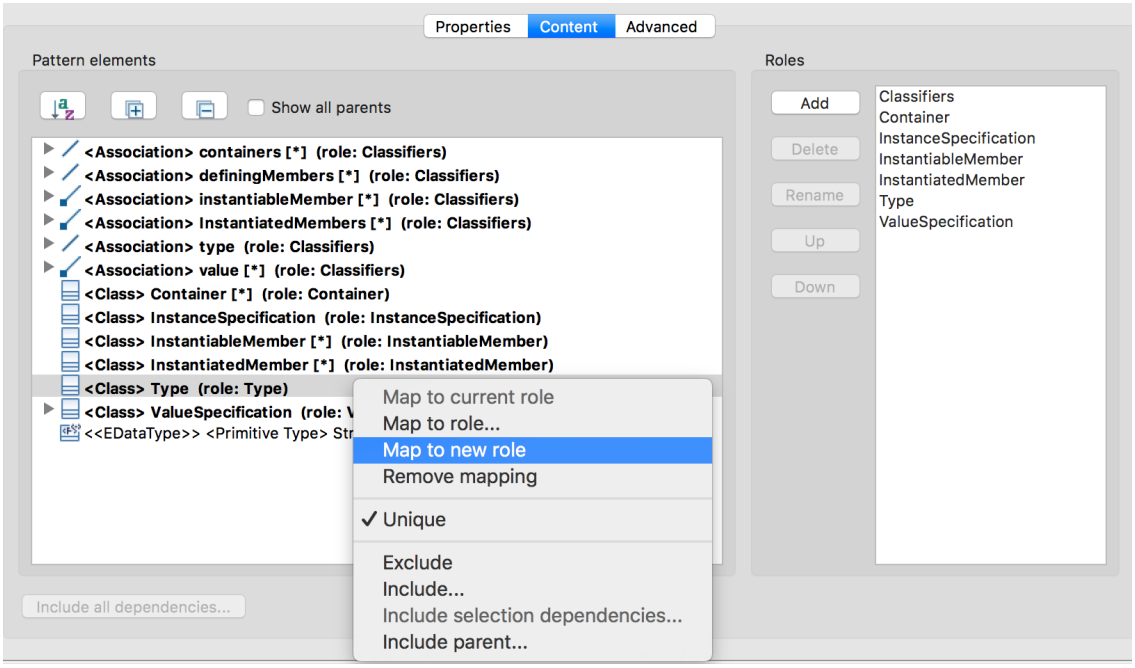


Figure A.4: Set the multiplicity and roles of pattern elements

each role is the same as that of the related class. Other elements' roles keep the default setting.

(f) The DiffMerge/Patterns tool uses "Unique" flag to control the number of occurrences of pattern elements. If the "Unique" label of a element is marked, this element can only occur once in a pattern instance. Otherwise, it can appear multiple times. The default setting of all element is unique. In this case, we set class objects "Type", "InstanceSpecification" as well as "ValueSpecification" and association object "type" as unique, other objects are not unique. The final setting of "Content" tab is shown in Figure A.5. The elements with a [*] label are not unique.

(g) In the "Advanced" tab, the related OCL constraints can be added. In this case, there is no constraints to be added.

Step 2: apply the example pattern to a meta-model diagram.

(a) In order to demonstrate how to bind pattern elements with existing model elements, we first create a diagram with one class object named "WorkspaceDefinition". Figure A.6 indicates the original meta-model.

(b) Then we apply the instantiation pattern to this meta-model to create new elements. Right-click and select `Patterns -> Apply Pattern...` to initiate the application wizards.

(c) In the "Select a pattern" wizard, we choose "instantiation" catalog. The information of the pattern stored in this catalog are automatically loaded (see Figure A.7). The click `Next`.

(d) In the "Map roles to elements" wizard, we first map the "Type" role to the existing class WorkspaceDefinition. Right-click "Type" role and select `Merge with...`. Then another window of all existing model elements will pop up. Choose the class WorkspaceDefinition to finish this mapping. When the pattern applied, other elements in this pattern will be added to the diagram except class Type, which will be replace by class WorkspaceDefinition.

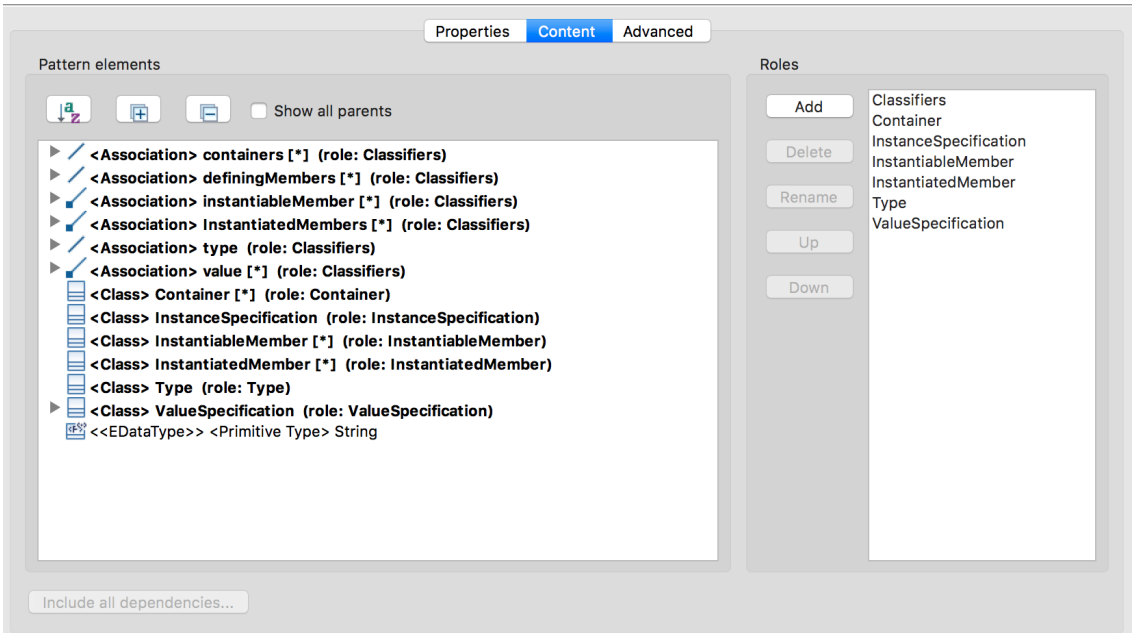


Figure A.5: Setting of "Content" tab

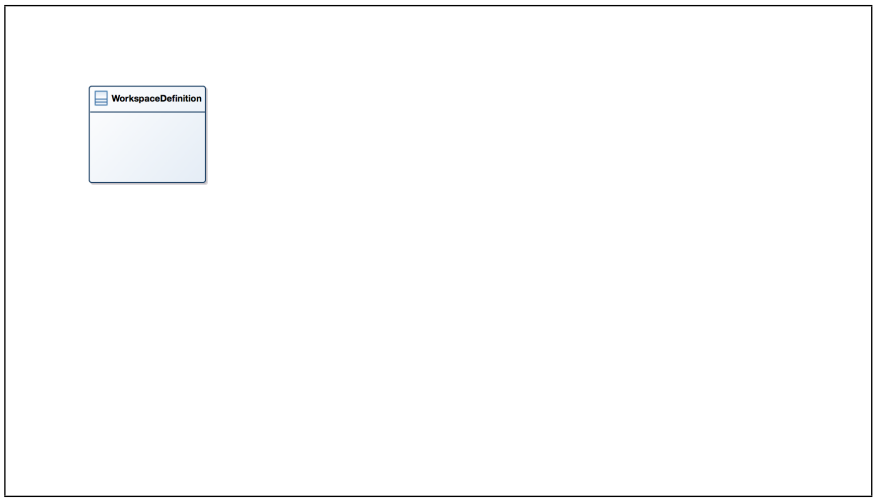


Figure A.6: The original meta-model

(e) Also, in this wizard, we set "Number of instances" to 1, which means there is one instance of this pattern will be applied. The "Multiplicity per instance" is set to 2. This means all the non-unique elements will occur twice in the instance. In other word, two model objects will be created for each non-unique element in a pattern.

(f) As we reuse layout and style of the pattern (see Figure A.8 ticked boxes), we get the same layout and style of the pattern in my instance. It is clear that the layout in Figure A.9 is exactly same as the layout in Figure A.2.

(g) Now we unfold the instance to get the full structure of this instance. Indeed, all non-unique elements in the pattern have two instances in this pattern instance.

Step 3: finish the example DSL meta-model. The last step is to change the names of elements in this instance and add other objects according to the example DSL meta-model

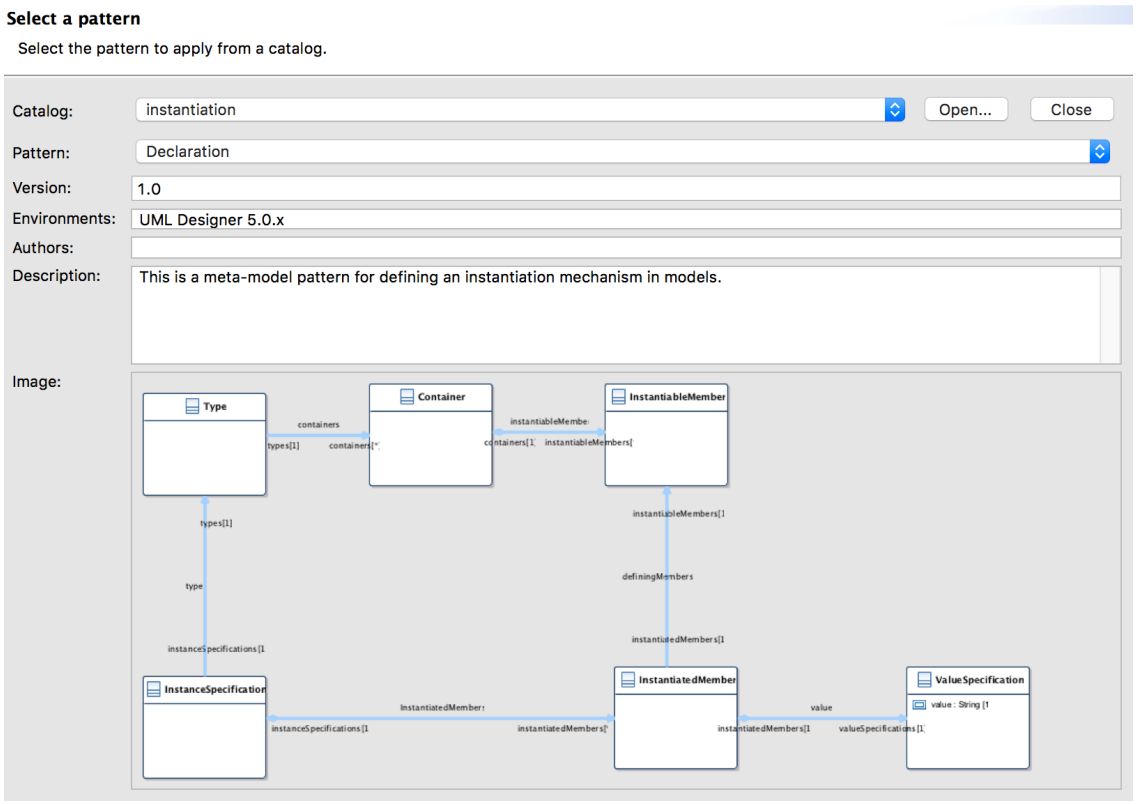


Figure A.7: Select a pattern

to finish this diagram. The finial meta-model is shown in Figure A.11, which is same as the example meta-model.

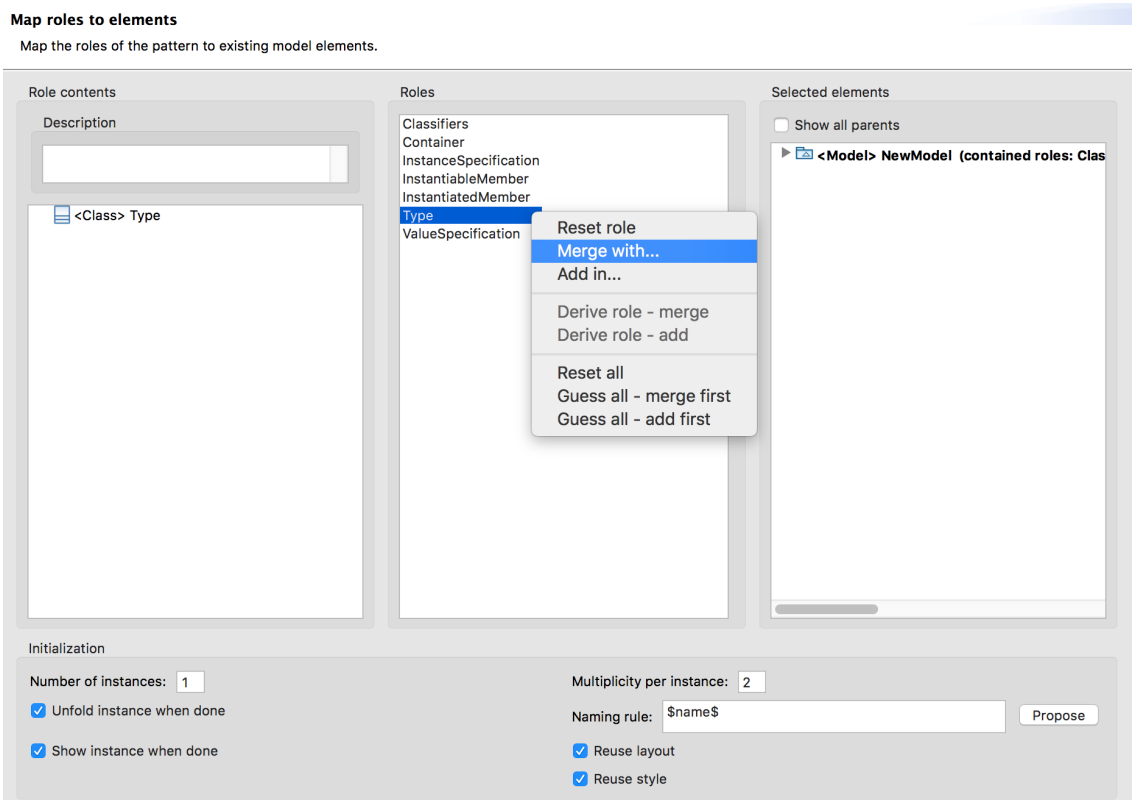


Figure A.8: Map roles to elements

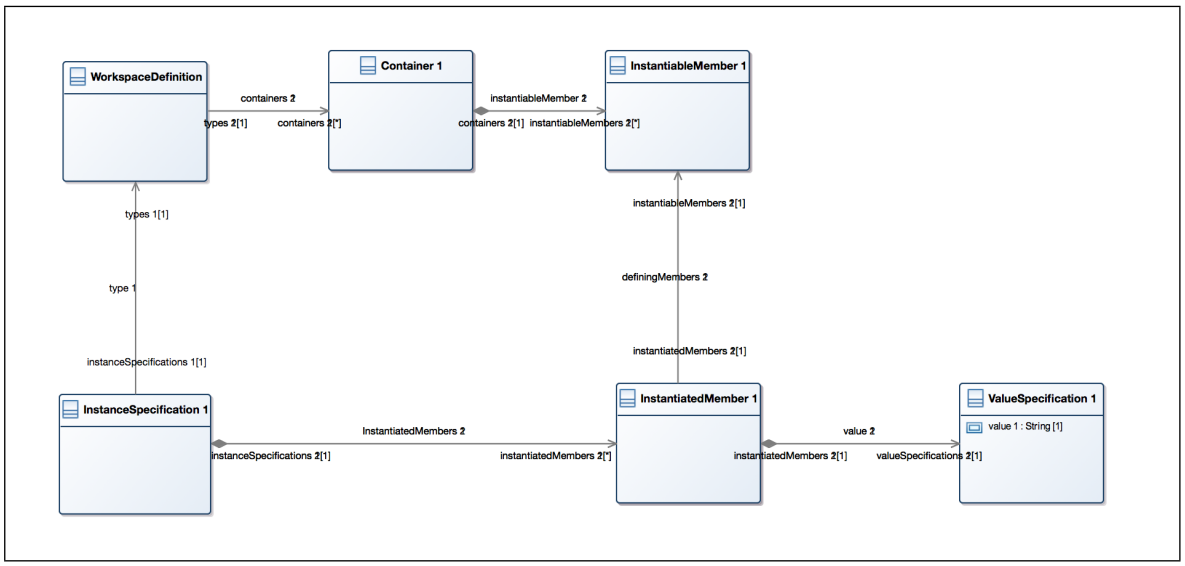


Figure A.9: The layout of the pattern instance

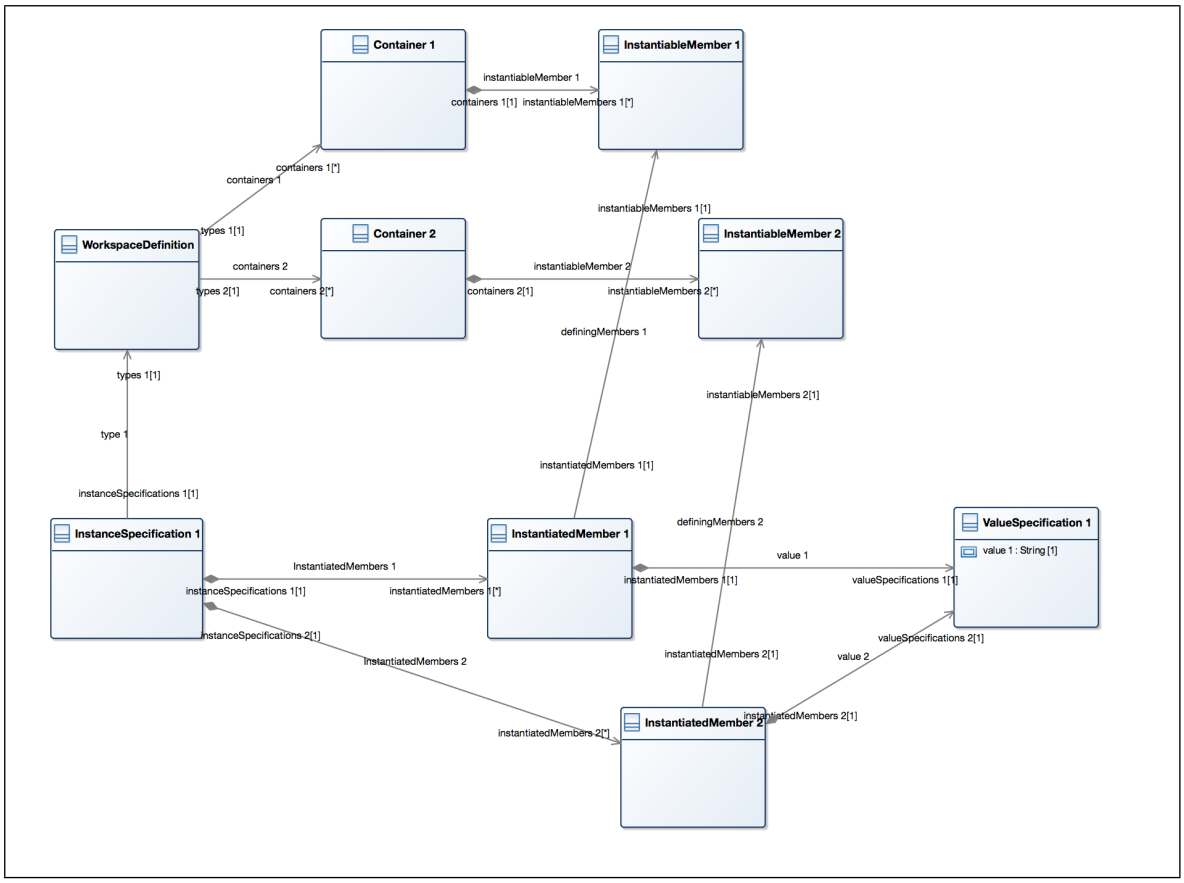


Figure A.10: The meta-model after the pattern is being applied

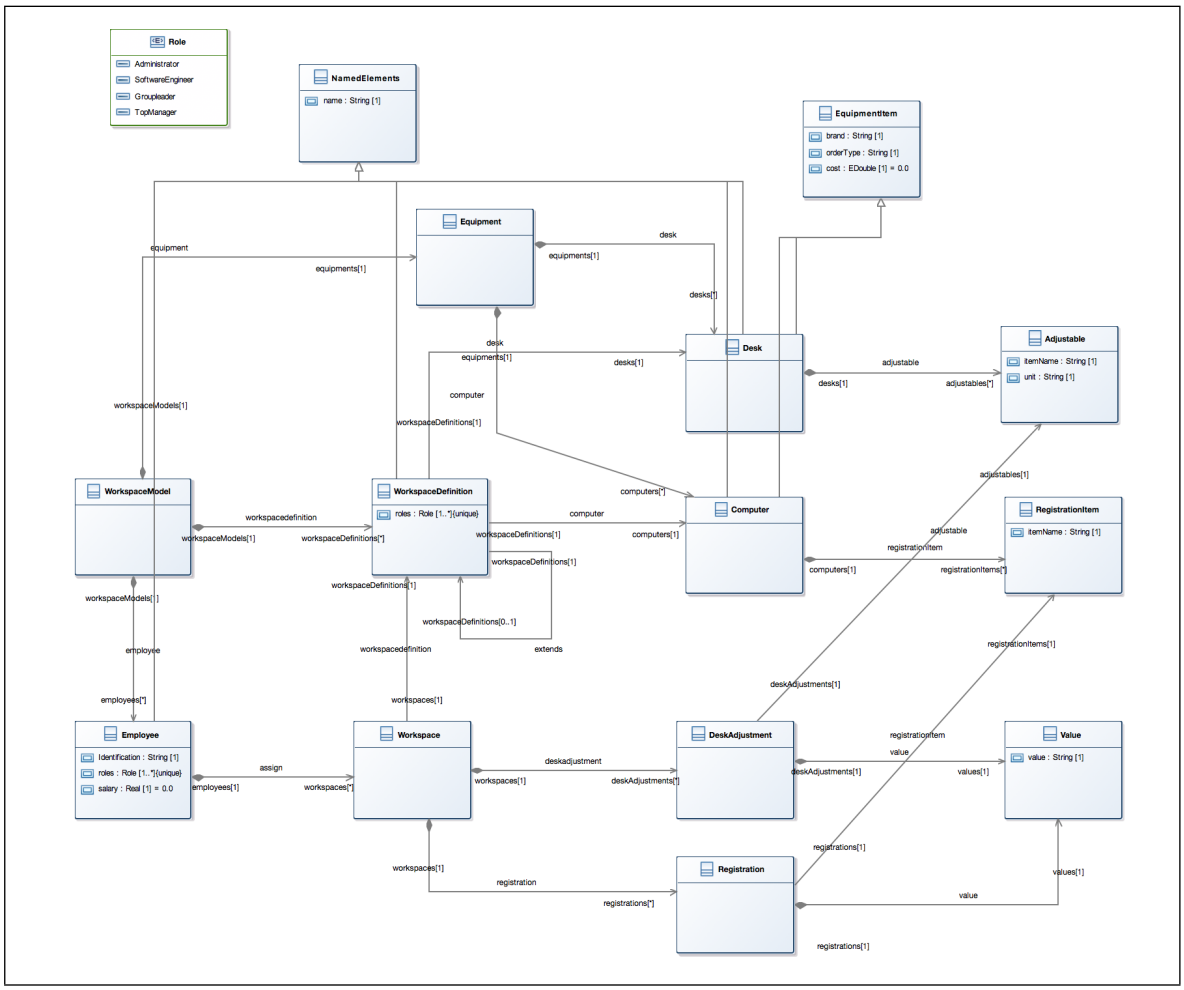


Figure A.11: The final meta-model

Appendix B

DSL-tao Guide

Similar to Appendix A, we introduce another tool, DSL-tao in this appendix by means of applying the Declaration pattern to create the example meta-model. Assume Eclipse and DSL-tao have been downloaded and installed, now let us follow these steps to create the example meta-model.

Step1: create the example pattern.

(a) Before creating a pattern in DSL-tao, we have to model the pattern in Ecore. The Ecore meta-model should be the same as that in Figure 2.4.

(b) Then we create a new DSL-tao project and add a new DSL-tao design diagram (see Figure B.1).

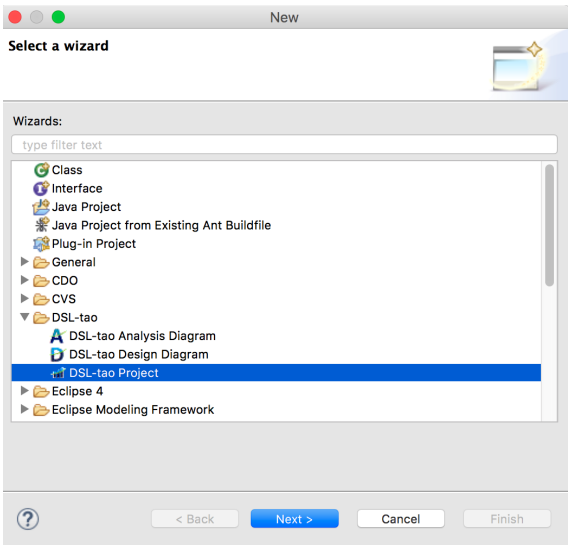


Figure B.1: New a DSL-tao project and design diagram.

(c) In this case, we name the project and the diagram "example". Under the `example` project, we can see (in Figure B.2) a folder named patterns, which includes all the files that are used to define and apply patterns. The structure of the predefined patterns are modeled in Ecore and stored in the `patterns` folder. This folder also includes a management file named `repository.dslpatterns`. The management file models the structure of a pattern repository provided by DSL-tao. In addition, every predefined pattern has a corresponding `.gif` file stored in the `icons` folder to show its structure and cardinalities of every element when being applied. Therefore, to create our own patterns, three

things have to be done: add the pattern Ecore file, add a `.gif` file for the pattern and add the pattern into repository, namely, modify the `repository.dslpatterns` file.

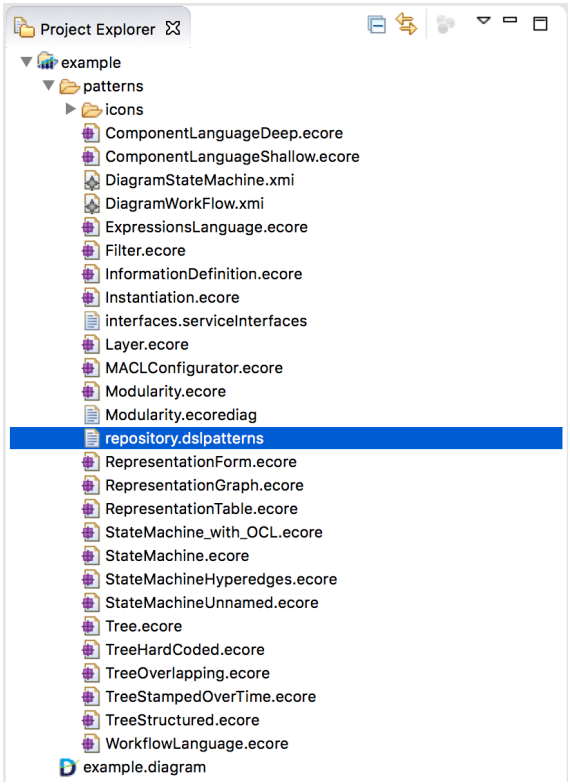


Figure B.2: The pre-defined patterns.

(d) To create the example pattern in DSL-tao, we first put the pattern Ecore file (i.e. "instantiation.ecore" in this case) to the pattern folder under the `example` project.

(e) Secondly, the `repository.dslpatterns` file need to be modified to add this new pattern. The structure of the "repository" is shown in Figure B.3.

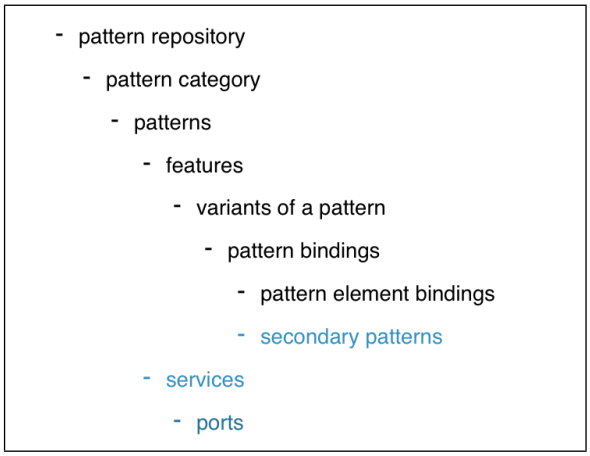


Figure B.3: The structure of the pattern repository

(f) Hence, we add a pattern category `Altran` and a pattern `Declaration`. The structure of this pattern is shown in Figure B.4.

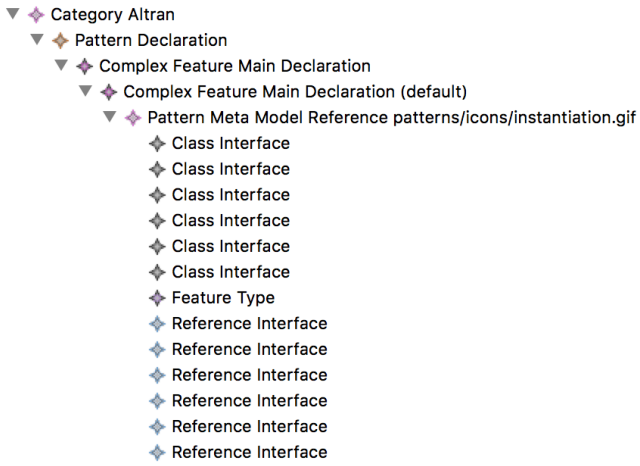


Figure B.4: The structure of Declaration pattern in the repository

Step2: apply the example pattern to a meta-model digram.

(a) To apply the Declaration pattern, open the empty"example" diagram and find the pattern in `Patterns View` under `Altran` folder. Then follow wizards to apply this pattern after double click it.

(b) In the first wizard, we set the pattern name as "Declaration".

(c) In the second wizard, the pattern variant and secondary pattern to be applied are selected. In this case, we only have a default pattern, no variants and secondary pattern can be selected.

(d) In the last wizard, a tree view of the original pattern instance is shown at right side. To create the example DSL meta-model, we add a new `instantiatedmembers` relation under class `InstanceSpecification`and a new `related` relation under class `Type`. The finial tree view of the instance we will apply is shown in Figure B.5.

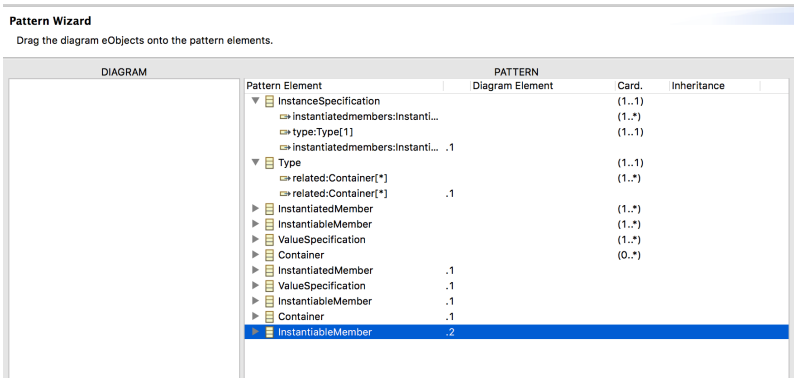


Figure B.5: The finial tree view of the instance of Declaration pattern

(e) Click `finish`, the pattern instance will be added to the empty diagram. As see from Figure B.6, the structure of this instance is not the same as the fragment in example DSL meta-model (see the parts highlighted in red circle). Unlike DiffMerge/Patterns, DSL-tao can not create an instance with the structure we want to model.

Step3: finish the example DSL meta-model. According to the example DSL meta-model in Figure 2.3, we can modify the diagram to create a meta-model with the same

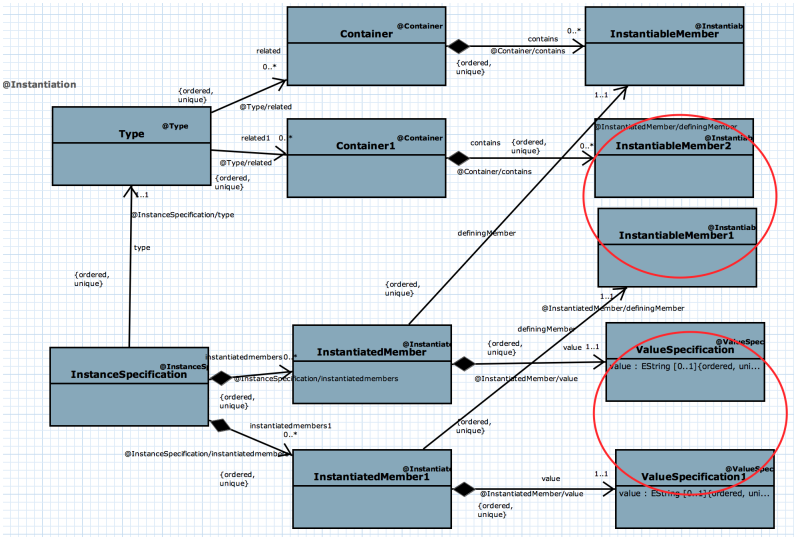


Figure B.6: The pattern instance in diagram

structure of meta-model in Figure 2.3.