Final Thesis

Meta-Model Guided Error Correction for UML Models

by

Fredrik Bäckström and Anders Ivarsson

LITH-IDA-EX--06/079--SE

2006-12-13

Final Thesis

Meta-Model Guided Error Correction for UML Models

by

Fredrik Bäckström and Anders Ivarsson

LITH-IDA-EX--06/079--SE

Supervisor: **Åsa Detterfelt**
Attentec AB

Examiner: **Peter Bunus**
Dept. of Computer and Information Science
at Linköping University

Abstract

Modeling is a complex process which is quite hard to do in a structured and controlled way. Many companies provide a set of guidelines for model structure, naming conventions and other modeling rules. Using meta-models to describe these guidelines makes it possible to check whether an UML model follows the guidelines or not. Providing this error checking of UML models is only one step on the way to making modeling software an even more valuable and powerful tool.

Moreover, by providing correction suggestions and automatic correction of these errors, we try to give the modeler as much help as possible in creating correct UML models.

Since the area of model correction based on meta-models has not been researched earlier, we have taken an explorative approach. The aim of the project is to create an extension of the program MetaModelAgent, by Objektfabriken, which is a meta-modeling plug-in for IBM Rational Software Architect.

The thesis shows that error correction of UML models based on meta-models is a possible way to provide automatic checking of modeling guidelines. The developed prototype is able to give correction suggestions and automatic correction for many types of errors that can occur in a model.

The results imply that meta-model guided error correction techniques should be further researched and developed to enhance the functionality of existing modeling software.

Keywords: modeling, meta-modeling, refactoring, error correction, validation, UML

Acknowledgements

We would like to thank Thomas Wiman at Objektfabriken for giving us much appreciated help, many hours of discussing our thoughts back and forth and for the rewarding collaboration we have had during the simultaneous development of MetaModelAgent for Eclipse and our extension of the very same program.

We also would like to thank our supervisor Åsa Detterfelt at Attentec AB and our examiner Peter Bunus for the assistance in getting started with the project and for helping us during the writing of this thesis report.

Contents

List of Figures x

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Purpose . . . 2
- 1.3 Objective . . . 3
- 1.4 Method . . . 3
  - 1.4.1 Research . . . 3
  - 1.4.2 Prototyping . . . 4
- 1.5 Limitations . . . 4
- 1.6 Thesis Outline . . . 5

I Theoretical Background 7

2 Modeling and Meta-Modeling 9
- 2.1 Modeling Languages . . . 10
- 2.2 Modeling Layers . . . 11
- 2.3 Meta-Models Defining Modeling Guidelines . . . 12

3 The Modeling Environment 13
- 3.1 Eclipse . . . 13
- 3.2 Eclipse Platform . . . 13
- 3.3 Eclipse Modeling Framework and UML2 . . . 15
- 3.4 IBM Rational Software Architect . . . 15
- 3.5 MetaModelAgent . . . 16

4 Modeling and Meta-Modeling by Example 17
- 4.1 Model . . . 17
- 4.2 Meta-Model . . . 19
  - 4.2.1 Elements . . . 19

4.2.2 Relations 22

5 Definition of Model and Meta-Model Problems 29

5.1 Model Problems 29

5.2 Meta-Model Problems 31

II Correction of Models 33

6 Correction Suggestions 35

6.1 Different Approaches to Correction Suggestions 35

6.1.1 The Simple and Generic Approach 35

6.1.2 The Rule-Based Approach 36

6.1.3 A Combination of Approaches 37

6.2 Our Approach to Correction Suggestions 38

6.2.1 Finding Correction Suggestions 38

6.2.2 Calculating Suggestion Probability 40

6.2.3 Suggestions for Different Problem Kinds 40

6.2.4 Meta-Model Errors 47

6.2.5 Applying Suggestions Using Refactoring 47

7 Meta-Model Guided Model Refactoring 49

7.1 Model Refactoring 49

7.2 Using the Meta-Model as Guidance 50

8 Implementation 53

9 Related Modeling Environments 57

9.1 The Generic Modeling Environment 57

9.1.1 Meta-Modeling in GME 57

9.1.2 The Modeling Environment 58

9.1.3 GME Compared to MetaModelAgent 58

9.2 A Tool for Multi-Formalism Meta-Modelling 59

9.2.1 Meta-Modeling in AToM3 59

9.2.2 The Modeling Environment 60

9.2.3 AToM3 Compared to MetaModelAgent 61

9.2.4 AToM3 Compared to GME 61

10 Related Work 63

III Result 65

11 Benefits of Meta-Modeling 67
11.1 The Benefits of MetaModelAgent 67
11.2 Correction of Model Errors 68
11.3 Guided Model Refactoring 69

12 Conclusion and Future Work 71
12.1 Conclusion 71
12.2 Future Work 71

Glossary 75

Bibliography 77

List of Figures

2.1 Four-layer architecture. 11

3.1 Eclipse Platform components. Figure from IBM [16] 14
3.2 Eclipse Platform architecture. Figure from IBM [16] 15

4.1 Model elements representing a Java class and an actor. 18
4.2 Meta-class representing an actor element. 20
4.3 Two meta-classes describing two different types of actors. . . . 21
4.4 Meta-class with operation. 21
4.5 Meta-class using enumeration. 22
4.6 Relation between two meta-classes. 23
4.7 Inheritance in a meta-model. 24
4.8 Meta-classes for a class and operations. 25
4.9 Meta-classes for a diagram and classes. 25
4.10 Meta-model allowing inheritance between classes. 26
4.11 Inheritance in a model. 27
4.12 Instantiation of a meta-model. 27

8.1 Implementation of MetaModelAgent on the Eclipse Platform. . 54
8.2 Implementation of the extension to MetaModelAgent. 55

Chapter 1

Introduction

1.1 Background

The use of modeling as a process in the development of complex systems is becoming more widespread. The need to abstract a system and its behavior has been evident for a long time, but no standard has existed. There is also a need for computer-aided systems that can check the model against the system guidelines. To model a complex system is not an easy task, as such system can contain elements, aspects, constraints and properties that are not possible to model by using only one single modeling language.[35]

One solution to this is to create a modeling language that can describe every part of the system. In many cases this is neither possible nor meaningful. There exist formalisms that span multiple domains such as Unified Modeling Language (UML) and Entity Relationship-diagrams, although none that can describe all types of domains. One of the drawbacks when using generic modeling languages is that no domain-specific knowledge may be used to enhance functionality or to handle domain-specific cases. On the other hand one could use different modeling languages for different aspects of the system, and choose the most practical language for each part. The problem with this approach is that the modeler has to be skilled in all modeling languages and it will be hard to integrate the models and use them together. What is needed is something in between, a way of describing the domain specific knowledge of the system, while still keeping the advantages of a general modeling language capable of modeling many different aspects of a system.

A system will typically have guidelines (often domain specific) that should be followed by every user of the system. To create correct models the domain has to constrain the models according to these guidelines. What is needed is a domain that can describe the systems model and put constraints on it.

Guidelines for modeling are often used to ensure that certain name standards are followed or that the models use the same structure. This could be compared to the use of templates in word processing programs to get the same look and structure on all documents produced, or perhaps to the use of a company style guide that ensure that all produced documents and products uses the company logo.

In this thesis we present an extension for MetaModelAgent, a tool that is used for creating domain-specific modeling environments for software systems. MetaModelAgent has a meta-model layer that serves as system guidelines for modeling. These guidelines describe what elements are valid in the domain and what constraints need to be fulfilled. The tool validates models against a meta-model to check for structural conformance and detect behavioral differences. UML is used both in the modeling environment and to describe the meta-model. Since both model and meta-model is described by UML, users modeling with MetaModelAgent will only need to learn and use one modeling formalism.

Up until now MetaModelAgent would only find errors in the model by validating it and present these problems to the user. This thesis presents an extension to MetaModelAgent that combines meta-model guided refactoring and techniques for finding the most appropriate correction to a problem in the model. This transforms MetaModelAgent from a domain specific modeling environment to a fully type-based modeling environment capable of performing model refactoring and presenting correction suggestions for user-created errors.

1.2 Purpose

The purpose of this project is to give users of the modeling environment IBM Rational Software Architect (RSA) with the meta-modeling plug-in MetaModelAgent more help in producing correct UML-models or validating already existing UML-models. The thesis examines if it is possible to develop a system for finding possible error corrections for UML-models based on a validation against their meta-models.

The thesis answers the question of whether it is possible to develop a good algorithm for model correction by finding the most relevant or in some way "best" solution to an error in the model. It also answers the question of what the best approach for this algorithm could be.

1.3 Objective

The goal of the project is to develop a working prototype as an extension to MetaModelAgent that provides the user with error corrections for models that are found to have errors when validated against their meta-model. The objective with this prototype is to explore the possibility of producing correction suggestions based on the meta-model, rather than trying to provide a fully functional system.

The aim is also to include in the prototype means of presentation of correction suggestions. Corrections suggestions must be easy to use in MetaModelAgent and one important part in this is to present them in a useful way for the user. The project should result in one or many presentation techniques that integrate nicely with RSA and MetaModelAgent while at the same time making it easy for the user to get a clear image of the changes that are being suggested.

The thesis also includes a theoretical part (see chapter 6.1) concerning the correction algorithm. The theoretical part is based on previous work published in the area of meta-modeling, model validation and UML in general. The theoretical part motivates our choice of approach for the algorithm, together with other possible approaches and their pros and cons.

1.4 Method

This project was performed at Attentec AB, a company specialized in consultancy in Linköping. The project is done in cooperation with Objektfabriken in Stockholm, using software produced by Objektfabriken and maintaining a close contact with them throughout the project.

The nature of the problem this thesis targets is such that there exist a need for both research and development. Because of this we had a clear division of two major tasks from the beginning; research and prototyping. The tasks were divided into several other parts that we will describe in this section.

1.4.1 Research

The research part of this thesis was divided into literature study and examination of already existing tools. Interestingly enough there seems to be little previous research about the idea of finding correction suggestions for modeling errors based on meta-models. The topics that have been researched included, but were not limited to, modeling, meta-modeling and model cor-

rection based on other theories, for example by cross-checking different models describing the same system against each other to find inconsistencies (see Chapter 10 Related Work).

1.4.2 Prototyping

The results of this project are presented by the implementation of a prototype that demonstrates that the results are applicable and usable.

This project implements a model parser to gain more knowledge about what kinds of problems might occur in a model, even though a parser from Objektfabriken (MetaModelAgent) already exists. This fine-granulation of the problem space is used to generate more exact correction suggestions.

Using these definitions of possible problems in a model the project examines different approaches for a system that generate correction suggestions and calculate the probability that a given suggestion would be useful for the user of the system. The set of generated suggestions is developed by examining the problem definitions and by evaluating what kind of suggestions each problem could be solved by. For each problem this results in a set of possible suggestions. The project developed a probability generator for correction suggestions that calculates the probability for each suggestion that this correction suggestion is the suggestion preferred by the user of the system.

In the final stage of the project a prototype has been developed as an extension to MetaModelAgent that incorporates all the functionality mentioned above and presents the results to the user.

1.5 Limitations

Since this project has been limited in time and the area of modeling, metamodeling and error correction is such a large area, some limitations on the project had to be done.

One limitation is that only a prototype of the proposed solution for error correction in models will be implemented, thus leaving parts of implementation and research to later projects.

Another limitation is that the project will not perform any thorough evaluation of how the implemented prototype will be used, which implies that statistics for performance improvement, measures of usability, etc. will not be presented.

1.6 Thesis Outline

This thesis is divided into three distinct parts, each consisting of a few chapters. The first part gives the reader the needed theoretical background, focusing mostly on modeling and meta-modeling. The second part describes the project closer, including our proposed solution to finding correction suggestions and technical implementation details. The third part contains the results of the project and the conclusions drawn from them.

This thesis is intended for anyone interested in the software modeling. The reader may be a developer already using modeling in hers or his daily work, but it might as well be a person working in or studying technical or computer related areas.

Part I

Theoretical Background

Chapter 2

Modeling and Meta-Modeling

Computer software is sometimes said to be one of the most complex creations of humans so far. Regardless of whether this is true or not, it is still clear that software can be very complex. A large software system can reach millions of lines of code, which might be compared to an aircraft with millions of parts or even a space shuttle with tens of millions of parts [3]. The complexity of software systems often make them hard to understand and it is a problem to get a good view of the whole system, how the system works and what parts of the system interact with each other.

Models are one approach to making large systems (not necessarily software systems) more understandable. A model tries to capture a specific aspect of a system by presenting only the information of the system that is necessary to describe the current aspect of the system. A model might also be interpreted as a collection and ordering of the information we have about a system, a notion which is easily compared to the human mind and its' way to build a mental representation of an item.

Models can also be used as a way of performing experiments on a system without changing the actual system. By using a model of the system, changes and alterations of the system can be tested without having to change the real system.

In software engineering, models are often used to describe a system. A model of a system can be developed before the actual system is constructed to help plan the development and to find possible errors in architecture before doing the implementation [8]. The model can also be developed and used after the system has been constructed, to help keeping the system understandable.

When talking about models and modeling there may sometimes be some confusion because of the different abstraction levels of different models. A number of models describing the same system may be entirely different from

each other and describe the system from different views and abstraction levels. For example a software system could be described by two models. One model may be very detailed and show the interoperations between different classes and their methods, while the other model describe large parts of the system and their communication or even use-cases for the system. This is the strength of models, to be able to view a specific aspect of a system, but can also lead to confusion - especially if used in an unclear way.[23]

A model is the first level of abstraction above the actual implementation of a system. It describes the system and keeps information about the system. One level of abstraction higher we find the meta-model, i.e. a model describing the model. The meta-model can be a way to keep the model understandable, to capture specific aspects of the model and to ensure that different models follow the same guidelines or use the same structure.

In much the same way as models can be developed before the system they describe is developed, meta-models can be developed before the model they are describing to help in the creation of the model. They can also be developed afterwards to make the model more understandable.

One common way of using meta-models is to use them as guidelines for creating the model. The same meta-model can be used for many different models, thus assuring that they follow the same guidelines. These guidelines might include rules for structure of the model, naming conventions for elements in the model or limitations in the way different elements are related to each other. We will return to this subject in Chapter 4 Modeling and Meta-Modeling by Example.

2.1 Modeling Languages

There exist many standards and languages for modeling. Most modeling languages use some form of graphical notion for modeling to enhance the use of the models. A graphical notion often gives the user a better view of how the system really looks and works in contrast to a text-based notion. There are many standards and languages that are domain-specific. A domain-specific modeling language is ER-models that are used for modeling Entity-Relationship-systems, often used for modeling databases [9]. Other domain-specific languages (or formalisms) are for example Petri Nets, Bond Graphs, Differential-Algebraic Equations (DAEs) and many more [37].

There are also some general-purpose modeling languages that are not domain-specific. UML from the Object Management Group (OMG) is the most widely-known of these languages. UML has become the industry de facto-standard for modeling software systems, but can also be used for busi-

ness process modeling, system engineering modeling or representing organizational structures.

2.2 Modeling Layers

Since a model describes the actual implementation of a system, the system can be seen as an instance of the model. In the same way, the model of a system is said to be the instance of the meta-model. This definition is recursive, in the meaning that any level of abstraction can be instanced into the level which it describes, as well as be abstracted further by adding another meta-level.

OMG defines the four-layer meta-model architecture [31], containing four levels of abstraction as can be seen in Figure 2.1. These levels are called M0 to M3. The M0-level in the bottom is the "real world", i.e. the objects created during execution of the software. The objects are an instance of the M1-level, the classes and the UML-model describing the system. The M2-level contains the meta-model, i.e. the model of which the UML-model in M1 is an instance. This meta-model could be the UML meta-model which describes the UML specification, or it could contain more project specific guidelines for the model. The highest level of abstraction as described by OMG is M3, which is the meta-meta-model level. This level is meta-circular, meaning it is an instance of itself and defines the language for specifying meta-models.

Layer	Description
meta-metamodel	The infrastructure for a metamodeling architecture. Defines the language for specifying metamodels.
metamodel	An instance of a meta-metamodel. Defines the language for specifying a model.
model	An instance of a metamodel. Defines a language to describe an information domain.
user objects (user data)	An instance of a model. Defines a specific information domain.

Figure 2.1: Four-layer architecture.

Even though OMG defines four layers of abstraction, it is common to have a smaller number of abstraction levels (e.g. model and implementation

only) for a given system or project. It also possible to define any number of meta-levels, for example adding one to the four-layers architecture by having both the UML meta-model describing the possibilities of UML and a meta-model describing constraints or guidelines that are specific to the problem domain, project or organization.

2.3 Meta-Models Defining Modeling Guidelines

Meta-models often define the language and the possibilities for the model, i.e. how a correct model should look and work, what elements are allowed and how elements can relate to each other. These are often given by a third-party organization, like the UML meta-model defining the language UML and its' semantics. However, there is often a need for companies to include their own modeling guidelines in a meta-model. These meta-models could contain information about model structure, naming conventions and many other modeling guidelines. Many processes used in the industry, e.g. quality processes like Capability Maturity Model (CMM) and Software Process Improvement and Capability dEterminiation (SPICE) or project processes like Rational Unified Process (RUP) also supply guidelines for the processes that also reflects in the models of the system. Such guidelines can also be described in user defined meta-models.

Different tools for modeling use different approaches to user defined meta-models. Some let their users create their own meta-models in the same modeling language as the model itself is expressed in; while others provide a separate meta-modeling language to let their users express their guidelines for the models. There is also a difference in how these tools use the user created meta-models, while some use the meta-models to only let their users create models that are allowed by the meta-model, others use the meta-model for validation of the model afterwards. In Chapter 9 Related Modeling Environments we will describe some of these tools more thoroughly and explain how meta-models are used for expressing modeling guidelines.

Chapter 3

The Modeling Environment

This chapter will describe the modeling environment in which the project have been performed and implemented. This serves as a background for our solution as well as a view of these tools and environments.

3.1 Eclipse

Eclipse is the name of an open-source community that aims at providing a developing platform and application framework for building software, although the name Eclipse is often used for other products made by the Eclipse community. Even though the Eclipse project is an open-source community, much of the contributions to the different projects come from personnel at IBM.[17]

The main project of the Eclipse community is the Eclipse Platform which is a set of frameworks for integrating different products with other Eclipse products.[16]

The Eclipse SDK is a Java development tool built on top of the Eclipse Platform, including among other features a java compiler plug-in and a java debug plug-in.[11]

3.2 Eclipse Platform

The Eclipse platform is a collaboration of numerous components that together form the platform. The entire platform can be divided into two major levels, as seen in Figure 3.1, where the Eclipse Rich Client Platform (Eclipse RCP) is a subset of the Eclipse Platform. The Eclipse Platform is the top layer and contains extended features that are applicable for developing IDE

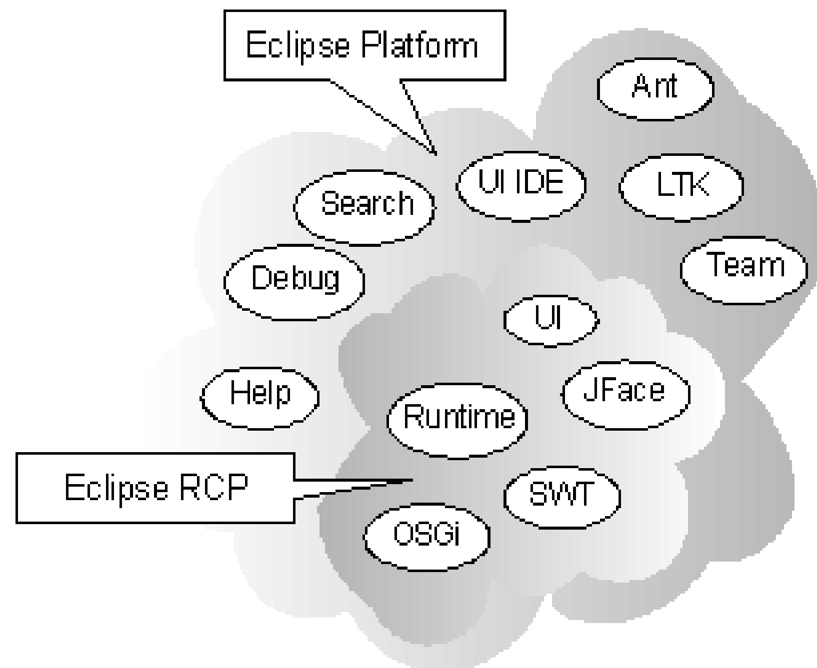


Figure 3.1: Eclipse Platform components. Figure from IBM [16]

environments. The Eclipse RCP contains features for building arbitrary applications that does not necessarily have to be software engineering applications.

To make all the components work together during runtime they are developed as plug-ins to the core of the platform called Eclipse Runtime Core, which detects them during start up. A plug-in must declare what extensions it will make to the platform so that the core can decide what it needs to load before it can start the plug-in. Most of the functionality of the Eclipse Platform resides in plug-ins that need to be loaded before it can be executed.

It would take a lot of time if all of them were to be loaded at start up. To solve this, Eclipse uses a proxy between the real plug-in and the program, which enables Eclipse to load the plug-in when it is needed, but still present adequate information about the plug-in by only loading the plug-ins definition file at start up. Once loaded, the plug-in will not be disposed until the program is closed. Since Eclipse is made up of plug-ins that declare extension points, there are many possible places to extend the platform as seen in Figure 3.2.

A plug-in is written in Java and compiled into a .jar file along with an XML-file named Plugin.xml that Eclipse Runtime Core loads at start up.

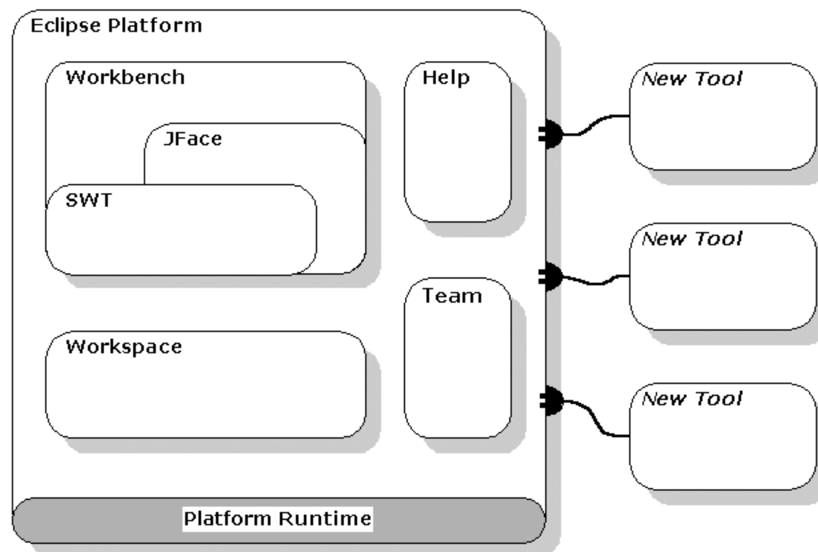


Figure 3.2: Eclipse Platform architecture. Figure from IBM [16]

The Plugin.xml file declares where the plug-in will extend Eclipse and optionally how other plug-ins can extend the plug-in.[16]

3.3 Eclipse Modeling Framework and UML2

Eclipse Modeling Framework (EMF) is a modeling framework for building tools and other applications based on a structured data model [6]. EMF provides the foundation for interoperability with other EMF-based tools and applications. EMF uses XML Metadata Interchange (XMI) as its' way to save and handle models. This makes it easy to move models between EMF and many other modeling tools, for example Rational Rose that can both import and export their models to XMI [14].

UML2 is an EMF-based implementation of the UML 2.x standard [15]. This gives developers an API for working with UML-models and EMF [25].

3.4 IBM Rational Software Architect

IBM Rational Software Architect (RSA) is a design and development tool built on top of the Eclipse SDK. RSA is intended for design and development

of entire software projects. The main extension of the Eclipse SDK is the ability to design models using RSAs UML-modeling feature. This is not the only extension but the most relevant for this thesis. RSA was released in 2004 and was the first release that was entirely built on top of the Eclipse SDK and Framework.

RSA is a total remake of the program Rational Rose, which was a standalone program used for modeling and design. This program was created by the company Rational. IBM acquired Rational in 2003 and changed the name of the program to IBM Rational Rose.[7]

3.5 MetaModelAgent

MetaModelAgent works as a plug-in for RSA. MetaModelAgent provide developers with an automatic tool for verifying their models against predefined guidelines in the form of meta-models. Models and meta-models are created in UML, using the UML model development features of RSA.

MetaModelAgent mainly uses the UML2 API, which makes it generic enough to be easily extended to any modeling tool built on the Eclipse Platform that uses their UML2 API [15]. See Chapter 8 Implementation for more details about MetaModelAgent and the extension that this project have implemented.

The first task is to create guidelines in the form of a meta-model that can be instantiated. MetaModelAgent then let users create new models by instantiating them from the meta-models. A model that is instantiated from a meta-model should conform to it. MetaModelAgent performs a validation of the model against the instantiated meta-model to check for conformance. If the validation find problems in the model, it does not totally conform to the meta-model. MetaModelAgent will notify the user of the lack of conformance, and describe the problems found during validation.

MetaModelAgent was at first a plug-in to Rational Rose, but will during this project be rewritten as a plug-in to RSA by Objektfabriken. This project is only examining MetaModelAgent for RSA and the results is implemented as an extension for MetaModelAgent for RSA.[29, 30]

Chapter 4

Modeling and Meta-Modeling by Example

This chapter shows how to use modeling and meta-modeling by using examples. It clarifies the different parts that make up a meta-model, how they relate to models and how meta-models can control models.

Meta-modeling is the process in which a new modeling domain is developed. A meta-model describes the valid model elements in a domain and what constraints to put on each element and relations among them.

To put constraints on a model domain is the sole purpose of meta-modeling. The purpose of a meta-model is to take a general modeling language and narrow it down so that it will only describe the domain it will be used for and nothing else. By doing this it can be ensured that the modeled system will be correct according to a set of guidelines.

This project has been implemented as an extension to MetaModelAgent, which is a plug-in to RSA. Read more about the implementation of the system in Chapter 8 Implementation. MetaModelAgent is built on top of RSA and therefore uses the UML standard.[32]

4.1 Model

This section explains what a model element is, the structure of a model and the model elements and how a model element can be controlled and visualized.

Model elements can be of any UML type [32]. Apart from the type of the element there are a number of properties that can be set on each element. What properties can be set for a model element is different depending on the type of the element. Most elements in UML have properties for name and

stereotype that can be set. The name is used to identify a model element among elements of the same type. The stereotype is used to further specify a UML type. When the stereotype is set, the original type of the element can be said to be a basic template for the element and the stereotype a further specification. E.g. the stereotype Java Class can be set on a model element of the type UML Class to make the further specification that this is a Class but more specifically a Java Class. In what way a Java Class is different from the base Class is up to the user to define.

Some types of model elements can have other model elements as child elements, e.g. Packages can have any other model element as a child element or a Class can have Operations or Properties as child elements. A child element is contained in the model element that is the parent to it.

Some types of model elements cannot have child elements of their own but can contain references to other elements, e.g. the Diagram model element. Diagrams are only used to visually present other model elements and the reference is a way to refer to the model elements that are being showed in the Diagram.

The type of a model element defines how it should be visualized. There is no exact standard for how UML elements should be visualized, although the main look is decided. The details are however left to the modeling tool to decide. This has created a variety of styles between different UML modeling tool vendors. Despite the differences, they are similar enough that there should be no problem for an experienced modeler to know what element it is. Typically a visualization of a model element should present the most important features of an instantiated type. Besides from a visual presentation of the type in some way, the important features of an element that should be presented are the name and stereotype properties.

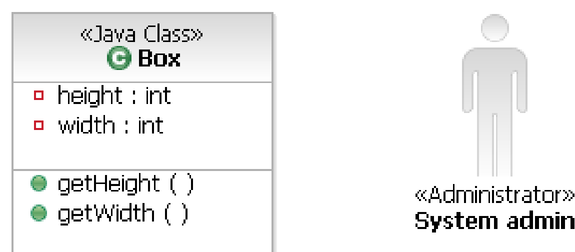


Figure 4.1: Model elements representing a Java class and an actor.

In Figure 4.1 two examples of model visualizations as they are done in RSA are presented. The left one is a Class model element and the right one is an Actor model element. The Class is visualized by a box that displays the

important properties that the class has. At the top is the stereotype «Java Class» and beneath the stereotype is the name of the model element. To the left of the name is an icon that shows that this is a Class element. The icon is needed to display the type of the model element because there are other elements that are visualized in the same way, e.g. Interfaces.

The Class element also shows the four child elements of the model element, the two Properties height and width and the two Operations getHeight() and getWidth().

The Actor element has its own visualization, but as can be seen the stereotype and name is shown.

4.2 Meta-Model

Meta-models are models too [37]. The meta-model is a layer that lies above the model layer; see Chapter 2 Modeling and Meta-Modeling. In the same way as the model tells the developer how to write his code, the meta-model tells the modeler how to develop his model.

The concept of creating the meta-model in the same environment and the same modeling language as the model itself is a powerful concept. The modeler does not need to know any other formal language besides UML (in the RSA and MetaModelAgent case - however, it could be any other general modeling languages as well) to create and interpret meta-models. Since it is done in a visual modeling language the process of working with meta-models will be easier.

In this section we will present the different elements that make up a meta-model in RSA and MetaModelAgent, how to structure them and how to express modeling guidelines with them. It should be noticed that the described method of meta-modeling is specific for MetaModelAgent.

4.2.1 Elements

Meta-models in MetaModelAgent are described using a subset to UML Class Diagrams. Classes, Attributes, Operations, Enumerations and Packages make up the entire structure of a meta-model [28].

An element in the meta-model is referred to as a meta-model element to differentiate it from model elements, i.e. elements in the model. Classes in the meta-model are referred to as meta-classes. A meta-class represents an UML type in the modeling domain. The stereotype of the meta-class describes what type of model element that the meta-class represents. As can be seen in Figure 4.2 the stereotype is set to Actor to make the meta-class

represent an UML Actor. The model element Administrator (with UML type Actor) will be an instance of the Actor meta-class, and vice versa the meta-class Actor will be the classifier of the model element.

Figure 4.2: Meta-class representing an actor element.

Attributes can be added to a meta-class to put constraints on model elements that are instances of the meta-class. The attribute has to reflect something in the model element, that is, the property of the model element it is constraining must exist in the model element. E.g. the name can be constrained to a constant or a regular expression. An attribute in a meta-class has a stereotype, a name, a type and a value. The text notion for this in MetaModelAgent is "«stereotype» name : type = value".

The stereotype of the attribute can be set to key, rule, rec (short for recommendation) and info. If it is set to key it implies that for a model element to be classified as an instance of this meta-class, the property of the model element with the same name as the name of the attribute that has key as stereotype has to be correct (according to the value of the attribute) for the model element. If the stereotype rule or rec is used on a property it signals the importance of the constraint, where constraints marked with rule are mandatory for the model to be valid and rec is recommended for the model. When the stereotype is set to info this will be seen as a piece of information, for example a rule that will not be enforced - only informed about.

The name of the property has to be a valid property of the UML element that it is representing. Setting a constraint on a property that is not present in the UML type will result in an incorrect meta-model.

The values that is possible to set for an attribute in a meta-class depends on the type of the attribute. If the type is String, the value could be any static string or it could be a regular expression. If the type is Boolean, the values could be true or false. It is also possible to use enumerations for expressing a set of allowed attribute values, which will be described in detail later on in this section.

In the process of finding a classifying meta-class for a model element, only the stereotype of the meta-class and property constraints marked with stereotype key is considered. Neither rule nor rec property constraints are evaluated in this process since they do not contain any information that would change whether the meta-class is a classifier for the model element or not.

For example we could have two meta-classes with the same stereotype (thus classifying the same type of model elements), but with different key property constraints on the property name. In Figure 4.3 two meta-classes with the stereotype Actor uses the key property constraint on the property name to ensure that the meta-model is unambiguous.

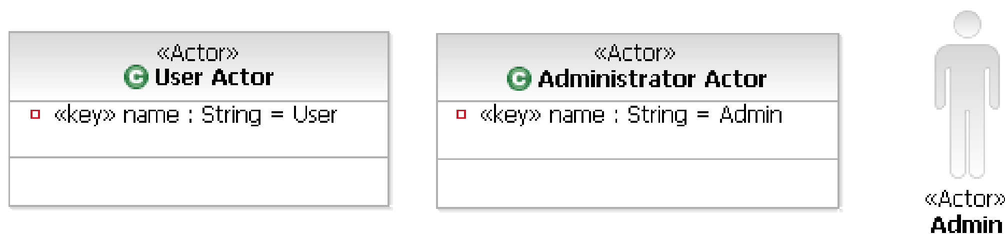


Figure 4.3: Two meta-classes describing two different types of actors.

When a classifying meta-class has been found for a model element, the rule and rec constraints are evaluated for the model element. If the model element does not follow the guidelines expressed as property constraints in the meta-model, the model is a non-valid model.

Operations in the meta-model are used to specify behavioral features of a meta-class. See Figure 4.4 for an example of a meta-class with an operation.

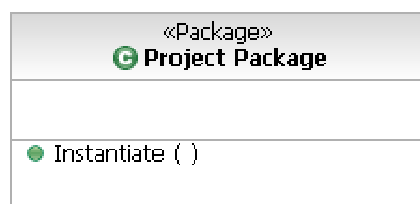


Figure 4.4: Meta-class with operation.

The operation Instantiate() tells the parser that models can be instantiated from this meta-class, i.e. that the meta-class with the operation is a classifier for the top element in the model. This operation is only allowed for meta-classes with the stereotype Package, since only packages can be top elements. To permit a model element to have any type of children the operation permitAll() is included in the classifying meta-class of the model element. This will stop the parser at this model element, thus allowing any kind of underlying constructions. The operation External() expresses that a model element is not allowed in the model but is allowed to be referenced by the model element. This can be used to show things in a diagram that is not a model element, e.g. showing resources in a diagram from the RSA project that are outside the model.

Enumerations are used to create a set of valid values for the attribute. In a meta-class a constraint on the name could allow an enumeration of values, i.e. give a set of names that would be allowed. Enumerations could also be used to express that all values except for those listed in the enumeration is allowed. See Figure 4.5 for an example where enumerations are used to create a meta-class which allows the name of an Actor to be either "Administrator", "Admin" or "SysAdmin". By changing the value "YES" to "NO" the name could have any value except for those three names.

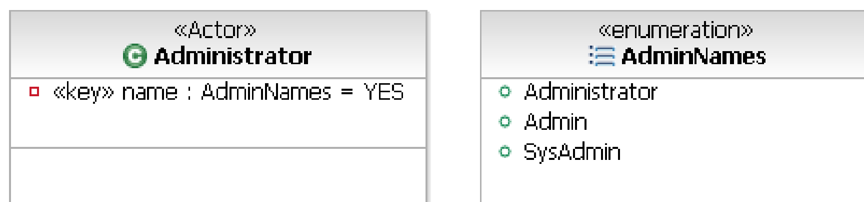


Figure 4.5: Meta-class using enumeration.

Packages are used to structure the meta-model. By using packages a meta-model can be divided into sub models. Each sub model might be a meta-model for a specific part of a system.

4.2.2 Relations

The possible relations in a meta-model are Association, Dependency and Inheritance relations. Relations describes the structure and the constraints in the meta-model, defining which model elements can be placed at each level

of the model and also the multiplicity (i.e. the allowed number of elements) of model elements.

In the same way as any other model element, a relation can have a stereotype. In a meta-model this is on Associations and Dependencies used to express the severity of the relation, i.e. if it is a rule or a recommendation. Setting the stereotype on an Inheritance relation does not have any semantic meaning for the meta-model.

By setting the stereotype of the relation to «rule» we let the parser know that it should generate an error if the model does not contain the model elements that the meta-model indicates it should. If the stereotype is set to «rec»the parser only generates a warning, which for example could be used to express that each package in the model should have a diagram in it, but that it is not necessary for the model to be valid. If the stereotype is not set for a relation, this is interpreted as a rule.

A relation between two elements also has an attribute named multiplicity. Multiplicity cannot be set for inheritance relations (see further down for explanation of inheritance). The multiplicity attribute allows the user to set a minimum and a maximum number for the relation. In a meta-model this is used to set an upper and a lower limit on how many model elements that may be related through the relation.

See Figure 4.6 for an example of a relation using the multiplicity to express that there should be at least one Java class (1..* is read as 1 to any number, i.e. more than one) in the class package and using the stereotype to express that it is a rule.

Figure 4.6: Relation between two meta-classes.

Inheritance in meta-modeling is used in the same way as inheritance is used in object-oriented languages. A meta-class can inherit from another meta-class, inheriting all the parents' attributes, the relations of the parent and any operations such as Instantiate() or PermitAll(). When inheriting from a meta-class the child has to have something that uniquely identifies it (e.g. a key-attribute), otherwise the meta-model will be ambiguous. This is especially true if more than one meta-class inherits from the same meta-class.

If the meta-model is ambiguous, MetaModelAgent will not be able to find the correct instantiation for some model elements.

As seen in Figure 4.7 the two packages Actor Package and Use-Case Package inherit from an abstract meta-class Package (i.e. a meta-class that a model element cannot be instantiated from - much like abstract classes in object-oriented languages). The Package meta-class contains one attribute that both the inheriting meta-classes will inherit. To make the two packages unique, each of them has a key-attribute on the stereotype property (named keywords when used in the meta-class).

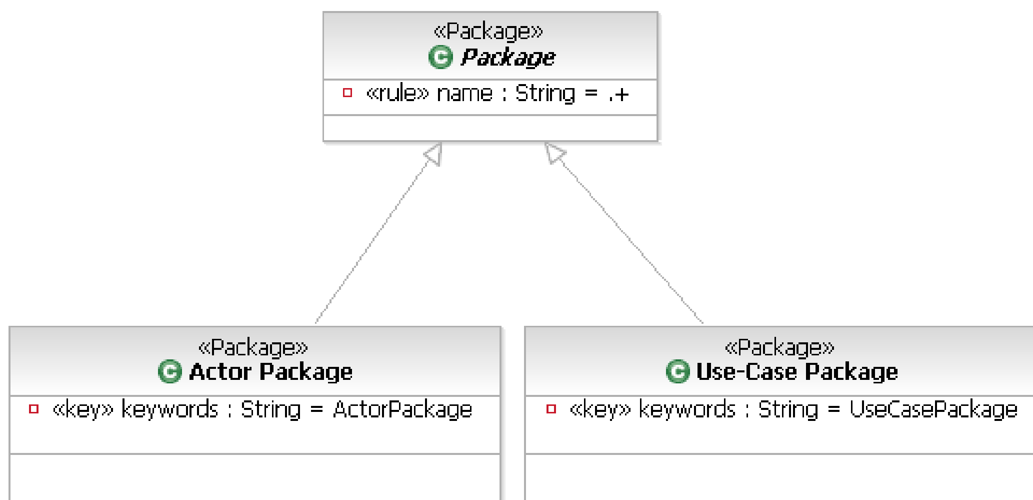


Figure 4.7: Inheritance in a meta-model.

Relations of the type Association can be of different types. The types that are used in MetaModelAgent are composition, aggregation and directed association.

A composition association has two ends, one owner and one target. In each end there will be a meta-class. The owner side will have a filled diamond to mark this side. The meaning of the composition association in a meta-model is that the meta-class that owns the composition will be able to have the target meta-class as child element. This means that a model element that is an instance of the owner meta-class will be able to have model elements that are instances of the target meta-class as child elements.

Multiplicity on a composition will put a constraint on how many model elements of the owned meta-class type the parent model element can have. A multiplicity constraint can be set to an exact integer, any interval of integers (e.g. 2..15) or any number of elements (expressed by *). It is also possible

to only set a lower limit by writing 3..*, thus expressing that there should be at least 3 elements. In Figure 4.8 an example where a class can have any number of operations is depicted.

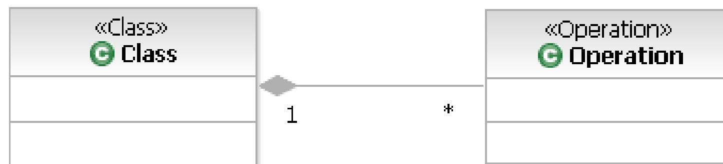


Figure 4.8: Meta-classes for a class and operations.

Aggregation associations are used in the same way as compositions, the difference is that an aggregation association means that the owning meta-class can have a reference to the target meta-class. This means that a model element that is an instance of the owning meta-class type will be able to have a reference to one or more model element children of the target meta-class type. As mentioned earlier, the only time references are being used in models is in diagrams, when the diagram has a reference to each model element that is visible in the diagram. An example of the meta-classes for a diagram that should view at least one class can be seen in Figure 4.9. The type FreeformDiagram is a diagram that can show any type of elements in RSA.

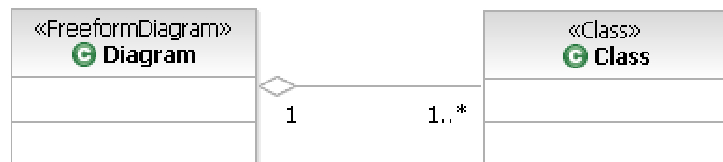


Figure 4.9: Meta-classes for a diagram and classes.

Directed associations represents an "is a"-relation. The best way to define the meaning of the directed association is by an example. As seen in Figure 4.10 the example shows two meta-classes. The first meta-class is named Class and describes model elements of type Class and the second meta-class is named Class Inheritance and describes model elements of type Generalization. The generalization meta-class is a meta-class for model elements of the type generalization, which is an inheritance relation between two model

elements (note that the relations themselves are model elements as well - thus allowing them to be described by meta-classes in the meta-model).

This structure makes up a rule that a model element of type Class can inherit from at most one other model element of the type Class (i.e. it can inherit, but it does not have to). There is a composition association owned by the meta-class Class that allows a model element of type Class to have at most one child of the type Generalization. The Generalization meta-class has a directed relation to the meta-class Class. This tells us that a Generalization element has a property called target that is of the type Class. The meaning of the target property is that a model element of type Generalization can point at a model element of type Class. In the model this structure will mean that a model element of type Class can own a Generalization association that points to another model element of type Class, as shown in Figure 4.11.

Figure 4.10: Meta-model allowing inheritance between classes.

Dependency relations are only used in the meta-model to specify that a meta-model is dependent on a meta-meta-model in the same way as declaring that a model is dependent on a meta-model. This is done by creating a dependency relation between the model and the meta-model (or the meta-model and the meta-meta-model) and giving the relation the stereotype instanceOf. In Figure 4.12 we show how to connect the use case model "Order and Storage System" to the meta-model "Use-Case Model Guidelines".

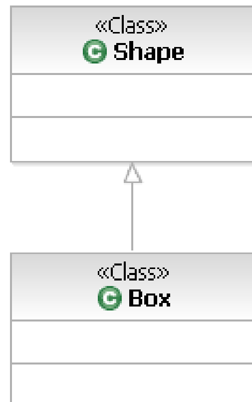


Figure 4.11: Inheritance in a model.

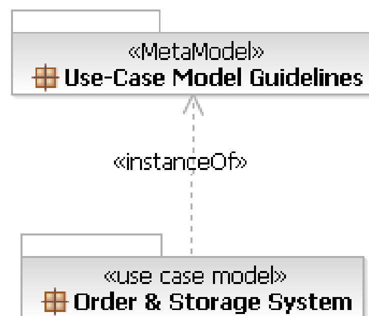


Figure 4.12: Instantiation of a meta-model.

Chapter 5

Definition of Model and Meta-Model Problems

This chapter presents the definitions that we use to categorize the problems that may be found in a model or a meta-model. The categorization has been made after examining what errors may occur in a model.

5.1 Model Problems

Most of the problems that can be found during validation are problems occurring in the model, due to an error in the model when it is validated against its' meta-model.

Child Overflow

A child overflow problem occurs when a model element has too many children of a given type according to the meta-model. If the meta-model specifies that a model element can only have one child of a certain type and the model includes an element with more than one child of the given type a child overflow problem occurs.

Child Underflow

A child underflow problem occurs when a model element has too few children of a given type according to the meta-model. If the meta-model specifies that an element should have one or more children of a given type and the model element does not contain any elements of the specified type, a child underflow problem occurs.

Reference Overflow

A reference overflow problem occurs when a model element has too many references to other elements that are instantiations of the same meta-class. If the meta-model specifies that a model element can only have one reference to a certain type and the model includes an element with more than one reference to the given type a reference overflow problem occurs.

References are only used in diagrams where a reference to a model element indicates that this model element is shown in the diagram. A reference underflow will only occur when a diagram does not have enough references to a certain type of model elements.

Reference Underflow

A reference underflow problem occurs when a model element has too few references to other elements of a given type according to the meta-model. If the meta-model specifies that an element should have one or more references to an element of a given type and the model does not contain any references to elements of the specified type, a reference underflow problem occurs.

References are only used in diagrams where a reference to a model element indicates that this model element is shown in the diagram. A reference underflow will only occur when a diagram does not have enough references to a certain type of model elements.

Invalid Child

An invalid child problem occurs when a model element has a child that is a valid UML item and is allowed by the meta-model, but not as a child to the given model element. This means that the child element is a correct element according to the meta-model, but that it is placed in the wrong place.

Invalid Reference

The illegal reference problem is the same as the illegal child problem, but instead of a child to the element it is a reference to an invalid element that causes the problem. It is still an element that is connected to a parent that is not allowed to have it. But a reference does not have the child directly under it and is therefore not the owner of the element. This changes the types of correction suggestions that are applicable to the problem. Since we are not the owner of the element we should not give as a suggestion that we should alter the element in some way, this will be taken care of if the element is also an invalid child.

Unknown Item

An unknown item problem occurs when a model element is found during validation that is not allowed at all by the meta-model. This means that there are no meta-classes in the meta-model that classifies the model element.

Invalid Property Value

An invalid property value problem occurs when the value assigned to a property of a model element breaks the constraint in its meta-model element.

5.2 Meta-Model Problems

There are some errors that occur in the meta-model. These problems do not need a validation of the model to be found - indeed it is not necessary to have a model that is an instance of the meta-model to find them.

Unknown Property Definition

An unknown property definition is found when a meta-model element that represents an UML element has a constraint on a property that the element cannot have. For example if the meta-model element has a constraint on the property abstract declaring it to be true and the element that it represents is a package, this will fail since a package cannot be abstract. To resolve this problem we should either remove the property rule or change the property rule to a valid property for the element.

Illegal Property Expression

All properties in a construct have a type. The type can be boolean, string, numeric or of some other sort. If the meta-model states in the same example with the property abstract that the value of the abstract property must be a numeric value, the meta-model states something about the model that cannot be true since the property abstract is a boolean. The constraint on abstract to be numerical is therefore an illegal property expression.

Unknown Meta-Class Definition

An unknown meta-class definition is when the meta-model has an element in it that cannot be resolved to a valid UML construct. The most likely source to this error is a misprint when typing the stereotype of the element. If the creator of the meta-model misses an "s" in "class" (thus writing "clas")

it will not be possible to resolve the element to a valid construct. In this case the only correct suggestion would be to prompt the user to change the stereotype of the element so that it becomes a valid UML construct.

Meta-Model Ambiguity

The meta-model will contain an ambiguity if for one model element there is more than one possible meta-element that can classify it.

Part II

Correction of Models

Chapter 6

Correction Suggestions

This chapter describes different approaches to correction suggestions for erroneous models in general and explain the proposed approach to correction suggestions in detail.

6.1 Different Approaches to Correction Suggestions

The main challenge when suggesting corrections is to keep the algorithm simple and generic for all sorts of UML diagrams, while at the same time using as much information about the specific problem as possible.

6.1.1 The Simple and Generic Approach

One way to keep the algorithm simple and generic is to search for problems in the model. For each problem in an element we handle it without caring about what kind of element it is and without using any other information about the element that might be gained from the model or the meta-model. The correction suggestions can be found by running a search algorithm (e.g. breadth-first or depth-first search) on the possible operations for changing a model to find a sequence of these operations that would solve the problem.

For example, the algorithm could have the operations "add attribute", "delete attribute" and "change attribute" together with a few others, and by searching these operations it could find a sequence of operations that would produce correct the problem found during model validation. This approach would basically be a brute-force approach where computer power and search algorithms solves the problem of finding a correct model which is close to the erroneous model. The algorithm would not be intelligent in any way and

would not use extra information about common errors, common solutions to these, semantic meaning of the diagram, etc.

The advantage with this approach is that the algorithm is simple and generic. The approach automatically works for any type of UML diagrams and probably also for later extensions to the specification of the language. The difficulties will only be in determining which changes could be made to a model element and then to apply them to the model. This problem is of course non-trivial, since an operation "change attribute" could be very complex if the algorithm wants to give suggestions for possible changes to the name of an element for example. If it would like to present suggestions for changing the stereotype of an element, the new stereotype really could be any string or any string that is a known stereotype from the meta-model if the algorithm was a bit more complex.

Another advantage with this approach is that even though it rarely will happen, the algorithm will be able to find new solutions to a problem that no-one has thought about before or a solution that is very specific for a given problem. This is loosely related to other types of problems where different search algorithms have provided optimal solutions to a problem or finding completely new approaches to an old problem.

The disadvantage with this approach is that the algorithm will have a high complexity, since the search tree would have a high branching factor and the search depth would get quite high. There are other drawbacks that could be even more serious, e.g. the algorithm will mostly produce suggestions that have no impact on the model correctness at all. Since the algorithm doesn't use any additional knowledge that it may have about solutions to different problems, the algorithm will always try operations like "change attribute" even though the error might be completely unrelated.

The algorithm will probably also find strange solutions that it might know would never be what the user intended, e.g. deleting all elements in the model and thus assuring that no inconsistencies exist. The fact that the algorithm is able to find new solutions is in itself an advantage, but most of these new solutions are so unlikely that the user would want to commit that it may in fact prove to be a disadvantage of the algorithm.

6.1.2 The Rule-Based Approach

On the other extreme end of possible approaches to an error correction algorithm, an algorithm which uses as much information and semantics from the model as possible could be produced. This approach would typically take an error and analyze and categorize it. This could be seen as some kind of rule-based system which has specially written rules for every kind of error

that could exist in a model.

This approach would be very suitable for a simple model validation algorithm which also provides some kind of basic hints about what could be done to correct the errors found. E.g. if the validation algorithm finds an inconsistency between an element and its meta-element based on a rule in the meta-element, the typical correction suggestion could be to "change the attribute so it conforms to this rule". These kinds of somewhat fuzzy correction suggestions can be useful in many cases, but they are in reality nothing more than a textual presentation of the error.

To take this algorithm to a higher level of error correction where the suggestions are useful and corrects the problem found, it would take a lot of time and too many special rules to make this approach feasible. The algorithm would quickly degrade to a "what if"-scenario where each possible case has to be solved separately.

The advantage with this approach is that the quality of the given correction suggestions could be very high, given that a good-enough problem classification could be achieved. In this case the algorithm would produce very precise solutions to each specific error. Another advantage for this approach is that the algorithm would be efficient and have a low complexity since it is just a mapping from the problem to a predefined solution that will not need any time-consuming search.

The disadvantage with this method of solving the problem is that it may prove impossible to categorize errors with such a precision that a correction suggestion based on these categorizations is good enough. Even if it was possible to get the required precision on error categorization, there would be problems with implementing this algorithm, since it would take a great amount of rules for each type of error that might occur in a model. The algorithm will also be very sensitive to change in the UML specification, where each change has to be included in the algorithm and every addition to the language would need new rules to handle the additions.

6.1.3 A Combination of Approaches

The best thing would be a combination of the two approaches above to get the advantages from both, while at the same time getting rid of the drawbacks. There are many ways that this could be done, and all have different advantages of their own. The general combination approach would include a search to make it possible to find new solutions and to get a more flexible and robust solution. It would also incorporate knowledge that the algorithm has about models and common problems, and use the semantics of the model to guide the search for an error correction suggestion. The search

would thus be a directed search or a search using a heuristic function.

One way is to categorize the errors that are found during validation of the model and from this categorization perform a directed search for different solution. E.g. a problem where an attribute of an element is incorrect. Examining the element more closely tells the algorithm that this is a element of the UML-type Class and that it is the attribute name that does not conform to the meta-model. With knowledge about common errors the algorithm suggests that the best way to solve this problem is to change the name, but a search also suggests that it might be possible to change the stereotype of the element to have it match to another meta-element. It might also suggest that moving the element to another part of the diagram (e.g. changing an association) might solve the problem as well.

The main problem in finding a good algorithm that is a combination of the two extremes is to find a good level of abstraction where as much information and semantics as possible is used without getting swamped in different rules and possibilities for each type of error.

6.2 Our Approach to Correction Suggestions

Our approach to finding correction suggestions to problems found in a model is a combination of the two extreme approaches explained above, as outlined in sec 6.1.3 A combination of Approaches. The algorithm finds a good balance between using as much information from the model and the meta-model as possible, while at the same time keeping the algorithm quite simple and generic.

The extension to MetaModelAgent that have been developed is a system divided into three parts. The first part uses the list of problems found by the validation in MetaModelAgent. This part generates all the correction suggestions for each problem. The second part of the system takes each correction suggestion and calculates the probability that the suggestions is the "best" suggestion, i.e. how likely it is that this suggestion is the one the user would like to commit. The third part connects each suggestion to a refactoring action, which allows the user to commit a suggestion, i.e. to let the system automatically solve the problem according to the selected suggestion.

6.2.1 Finding Correction Suggestions

The first step of the system is to generate all possible suggestions to a problem. The system takes a problem from the validation algorithm and ana-

lyzes this problem. From the problem the algorithm gets information about what kind of problem it is (see Chapter 5 Definition of Model and Meta-Model Problems for definitions of the problems) and what elements are affected by it. Depending on what kind of problem it is, the algorithm has a predefined set of suggestions types that will be generated. Suggestions are divided into the following groups: add, delete, change and move suggestions.

Suggestions of the type add is used for suggestions where the algorithm adds a model element to the model. This may be used to solve problems where the model does not have the required number of child elements of a certain kind.

Suggestions of the type delete are used for suggestions where the algorithm deletes a model element to solve the problem. This may be the case when it has more model elements in the model than the allowed number of elements of the given type at this place in the model.

Suggestions of the type change represent changes to a model element, usually by changing a property of the element. This could for example represent a change in the name of the element, the stereotype of the element or perhaps some other attribute. These suggestions may solve problems where there is an illegal attribute value, but may also be used to change the model element so the type of the element changes, i.e. so that it is described by another meta-model element than before. In this case the suggestion may solve multiplicity problems as well as many other problem kinds.

Suggestions of the type move represent solutions to a problem where the algorithm moves a model element to a different part of the model. This may solve multiplicity problems as well as other kinds of problems.

The algorithm will try to find suggestions of different types depending on the problem. Some kinds of problems will be tested for a valid suggestion of the type add and change, while other problem kinds lead to a test of suggestions with type delete and change.

For a given problem, the algorithm may produce several suggestions of the same type. For example the system may produce multiple suggestions of the type add, where each suggestion represents the adding of a a new and different model element. The difference between the suggestions may be the type of the element to be added for example. In the same way it may produce several suggestions of the type change, where each suggestion represents a change of different attributes of the model element or changes in different model elements.

6.2.2 Calculating Suggestion Probability

To be able to present the error correction suggestions for the user in a user-friendly way, to enhance the usefulness of the system and to avoid invalid correction suggestions to be presented the system calculates the probability for each suggestion that this suggestion is a valid and plausible correction suggestion. For each problem the suggestions are ordered based on this probability. There is also a threshold level for probabilities, ensuring that suggestions that have too low probability are not shown. This makes it possible to filter suggestions that are either too unlikely or that does not solve the problem.

It is important that the user will see the suggestion that he or she is most likely to apply when the suggestions are presented in the list. It is also important that the user is not flooded with too many suggestions. Even though the search for suggestions is exhaustive the user must not get all possible solutions, only the ones that are relevant.

To calculate the probability for a suggestion the system once again examines the problem. Based on the kind of the problem and the type of the suggestion, the system has different algorithms for calculating the probability of the given suggestion. For example a suggestion of the type change for an overflow problem may look at how many attributes has to be changed to make it match against another meta-model element. It will also take into account whether the model element to be changed has child elements of its' own and how they would be affected by this change. If they are affected negatively (e.g. not being allowed after the change), the suggestion is more unlikely and gets a lower probability than if the model element did not have any child elements.

The algorithm for calculating probabilities for each suggestion is based on constants given from a preference file. After the system has done the analysis of each suggestion it uses those constants to calculate the probability for the given suggestion. This makes it easy to change which types of suggestions are presented, in what way they are prioritized, what factors should be considered during analysis and what threshold levels should be used.

An extension to our suggested system could be to let the system automatically change these constants depending on how it is used. See Chapter 12.2 Future Work for more details about this.

6.2.3 Suggestions for Different Problem Kinds

As previously stated the algorithm finds suggestions of different types depending on the problem kind of the given problem. This section will describe

which types of suggestions the algorithm finds for each problem kind and how they may solve the problem.

Note that this section describes the suggestions for each problem kind, even though many suggestions are quite similar between different problem kinds. A remove suggestion for a child overflow problem does not differ much from a remove suggestion for an invalid child problem, but since there almost always is some difference this section describes them completely for each problem kind. This may lead to some parts of the text being identical.

Child Overflow

To correct a child overflow problem the algorithm examines the following suggestions.

Remove one of the child elements

To correct a child overflow the algorithm tries to find a solution where it removes one of the child elements that are indicated by the child overflow problem. This includes finding all model elements that are instantiations of the meta-class that is being indicated as the child overflow meta-class. For each of these model elements the algorithm produces a remove suggestion, i.e. a suggestion to remove the model element.

When calculating the probability for each remove suggestion the algorithm take into account whether the model element to be removed has child elements and whether it is being referred to by other model elements in the model. In each of these cases the probability that the user wants to delete the model element is significantly lowered.

Change one of the child elements

Another suggestion being tested by the algorithm is to change the type or a key property of one of the child elements so that it will be an instantiation of another meta-class. For each child element of the overflowing type, the algorithm looks at all meta-classes which are allowed at the same place in the model as the current meta-class and for each for these meta-classes creates a change suggestion, i.e. a suggestion to change the given child element to match the new meta-class.

By committing a change suggestion the number of model elements of the overflow type is reduced by one, but at the same time the number of model elements that is an instantiation of the meta-class that the element is being changed to is increased by one. Because of this it is important that the algorithm check the multiplicity of the meta-class that it changes the model element to, so that it doesn't create a new child overflow problem.

When calculating the probability for each change suggestion the algorithm take into account how many properties of the model element it has

to change and also which properties it has to change. This means that a change suggestion which has to change the name, stereotype and perhaps another key property of the model element as well is much more unlikely than a suggestion to change only the name of the model element.

The system also use the probability calculation to filter away otherwise occurring correction suggestions. The change suggestions that are filtered for are changes of the type of the model element that makes no sense. To ensure this the algorithm uses the following set of rules for type changes:

- Packages cannot change their type, i.e. a package is always a package since there is no sense in talking about changing a package to another type of model element.
- A diagram can only be changed into other types of diagrams.
- A relation can only be changed into other types of relations, i.e. it might be okay to change an aggregation to a composition, but not to change an inheritance to an interface.
- All other types of model elements can be changed into all other types of model elements.

These rules give the algorithm a crude way of filtering away type changes that are too unlikely and that are close to impossible to commit automatically. These rules could however be developed further to allow changes between other types of element, but by classifying them as more or less likely. For example it would be more likely to change a model element used in Class Diagrams to other types used in Class Diagrams rather than in other types of UML-diagrams. This extension has not been researched within the scope of this project, since in most cases the meta-model won't allow these changes anyway.

In the current implementation of the system, the algorithm does not check any child elements to the model element being changed. This would be an important extension to the system, since the current implementation is running a risk of introducing new problems. When changing key properties of the model element the algorithm is changing which meta-class the model element is instantiated from and this also changes whether the child elements of the model element would be allowed or not.

Move a child

The third type of suggestion that the algorithm finds for the child overflow problem is move suggestions. The correction the algorithm tries to find is to move one of the child elements of the model element which has the child

overflow problem to some other model element in the model. This would reduce the number of child elements with one.

To find a valid move suggestion, the system takes the child elements and finds all other places in the model where one of them may be placed.

More specifically this is done by starting to find all meta-classes in the meta-model that could classify the given child element. This is done for each of the child elements that are indicated by the child overflow problem. For each of the meta-classes the system finds, the algorithm examines all meta-classes in the meta-model that could own instances of the meta-classes that could classify the child element. I.e. the algorithm find all the parents to all the meta-classes in the meta-model that could classify the given child element. For each parent the algorithm finds instantiations of the parent in the model, i.e. it finds model elements that are instantiated by the parent. These model elements are then plausible parents to the child element that the algorithm wants to move and thus the algorithm generates a suggestion to move the child element to the found parent model element.

To ensure that the algorithm does not present move suggestions that are not valid, it checks that the move suggestion does not create a new child overflow problem in the model element that it moves the child element to. It also checks that the new parent that is being suggested is not the same as the current parent, since it will not solve any problem to suggest to move a child element from its' current position to the same position again.

When calculating the probability for a move suggestion the algorithm does not only filter for the above-mentioned cases, it also looks at whether the child element has children of its' own and whether it is being referred to by other elements. Both these factors make it less likely that the user intend to apply that suggestion.

Child Underflow

To correct a child underflow problem the algorithm only examines the following suggestion.

Add a child element

To correct a child underflow problem the only correction suggestion that makes sense is to add a new model element of the type that is indicated by the child underflow problem. A new element is created with default-values or by user-interaction to get the values for each property in the new model element. Read more in section 6.2.5 Applying Suggestions Using Refactoring. about committing suggestions and user-interaction.

Reference Overflow

To correct a reference overflow problem the algorithm examines the following suggestion.

Remove one of the references

To correct a reference overflow problem the only type of suggestions the algorithm will try to find is remove suggestions. For each of the references that are indicated by the reference overflow problem it generates a remove suggestion to remove the reference.

Unfortunately there is no good way of giving a probability on the reference remove suggestion since there is nothing that helps us tell each reference apart. This means that all remove reference suggestions generated for one reference overflow problem will have the same probability.

Since references are only used in diagrams, this suggestion translates to taking one element away from a diagram.

Reference Underflow

To correct a reference underflow problem the algorithm examines the following suggestion.

Add a reference

The only plausible suggestion that would solve a reference underflow problem is to add a reference to a model element that is an instantiation of the meta-class given by the reference underflow problem. This suggestion will need some user-interaction to let the user indicate which model element should be referred to (choosing from all the matching model elements). Read more about committing suggestions and user-interaction in section 6.2.5 Applying Suggestions Using Refactoring.

Since references are only used in diagrams, this suggestion translates to adding a model element in a diagram.

Invalid Child

To correct an invalid child problem the algorithm examines the following suggestions.

Remove the invalid child element

To correct an invalid child problem the algorithm generates a suggestion to remove the child element that is causing the invalid child problem.

When calculating the probability for the remove suggestion the algorithm takes into account whether the model element to be removed has child elements and whether it is being referred to by other model elements in the

model. In each of these cases the probability that the user wants to delete the model element is significantly lowered.

Change the invalid child element

Another suggestion being tested by the algorithm is to change the type or key property of the child element so that it will be an instantiation of another meta-class. The algorithm looks at all meta-classes which are allowed at the same place in the model as the current meta-class and for each for these meta-classes creates a change suggestion, i.e. a suggestion to change the child element to match the new meta-class.

By committing a change suggestion the algorithm reduces the number of model elements of that type by one, but at the same time increase the number of model elements that is an instantiation of the meta-class that the element is being changed to. Because of this it is important that the algorithm checks the multiplicity of the meta-class that it changes the model element to, so that it does not create a child overflow problem.

When calculating the probability for each change suggestion the algorithm takes into account how many properties of the model element it has to change and also which properties it has to change. This means that a change suggestion which has to change the name, stereotype and perhaps another key property of the model element as well is much more unlikely than a suggestion to change only the name of the model element.

The algorithm also uses the probability calculation to filter away otherwise occurring correction suggestions. The change suggestions it filters for is changes of the type of the model element that makes no sense. To ensure this it uses the following set of rules for type changes:

- Packages cannot change their type, i.e. a package is always a package since there is no sense in talking about changing a package to another type of model element.
- A diagram can only be changed into other types of diagrams.
- A relation can only be changed into other types of relations, i.e. it might be okay to change an aggregation to a composition, but not to change an inheritance to an interface.
- All other types of model elements can be changed into all other types of model elements.

These rules give the system a crude way of filtering away type changes that are too unlikely to occur and that are also close to impossible to commit automatically. These rules could be developed even further to allow changes

between other types of element, but by classifying them as more or less likely. For example it would be more likely to change a model element used in Class Diagrams (see UML-specification section) to other types used in Class Diagrams rather than in other types of UML-diagrams. This extension has not been researched within the scope of this project, since in most cases the meta-model won't allow these changes anyway.

In the current implementation of the system, the algorithm does not check any child elements to the model element being changed. This would be an important extension to the system, since the current implementation is running a risk of introducing new problems. When changing key properties of the model element the algorithm are changing which meta-class the model element is instantiated from and this also changes whether the child elements of the model element would be allowed or not. This is not something the algorithm is examining today, but it would make a securer system if it were to be implemented.

Move the invalid child element

The third type of suggestions that the algorithm finds for the invalid child problem is move suggestions. The correction it might find is to move the child element of the model element which has the invalid child problem to some other model element in the model where the child element would be allowed.

To find a valid move suggestion, the system takes the child elements and finds all other places in the model where one of them may be placed.

More specifically this is done by starting to find all meta-classes in the meta-model that could classify the given child element. For each of the meta-classes the system finds, it examines all meta-classes in the meta-model that could own instances of the meta-classes that could classify the child element. I.e. it examines all the parents to all the meta-classes in the meta-model that could classify the given child element. For each parent it finds instantiations of it in the model, i.e. it finds model elements that are instantiated by the parent. These model elements are plausible parents to the child element that the algorithm is trying to move and thus it generates a suggestion to move the child element to the found parent model element.

To ensure that the algorithm does not present move suggestions that are not valid, it checks that the move suggestion does not create a child overflow problem in the model element that it moves the child element to.

When calculating the probability for a move suggestion the algorithm does not only filter for the above-mentioned cases, it also examines whether the child element has children of its' own and whether it is being referred to by other elements. Both these factors make it less likely that the user intend to use that suggestion.

Invalid Reference

To correct an invalid reference problem the algorithm examines the following suggestion.

Remove reference

The only suggestion that is generated for an invalid reference problem is to remove the reference. Since references are only used in diagrams, this suggestion translates to removing a model element in the diagram.

Invalid Property Value

To correct an invalid property value problem the algorithm examines the following suggestion.

Change property value

The only suggestion that is generated for an invalid property value problem is to change the property value.

In some cases this correction suggestion is committable without any user-interaction. This is true when the guidelines given by the meta-model implies that there is only one option of which value the property should have. More specifically this may happen in the following cases:

- The property is a string and the meta-model specifies what the string should be (in comparison to a regular expression where it is not possible to generate a valid input without user interaction)
- The property is a boolean, which means the algorithm only inverts the current value, i.e. true becomes false and false becomes true.

6.2.4 Meta-Model Errors

In the current implementation of the system it does not create any correction suggestions for problems that occur due to meta-model errors. This is a feature that might be an interesting area for further research and work, but it is not within the scope of this project. Automatic correction of the meta-model is not an option that should be available to any user of the system, but rather to a specific person or role in the project.

6.2.5 Applying Suggestions Using Refactoring

The third part of the system allows the user to commit or apply a certain correction suggestion. To enable this functionality a suggestion is constructed of a set of refactoring actions that should be committed to apply the whole

suggestion. Each refactoring action is an atomic action, i.e. a single change to the model.

The different refactoring actions that the system uses is add element, move element, change attribute and delete element.

Some of these refactoring actions may need user interaction when being committed. This depends on the type of refactoring action, but also on what elements they work on. A delete action will never need user interaction, except if the system wants to warn the user that deletion of the element also will delete child elements or to warn about other side effects of the refactoring. An add action may however need more interaction with the user. When the user commits an add action, the type of the element that should be created is known. However, not all attributes of the element may be known. The attributes that are controlled by the meta-model might be easy to set to the value required by the meta-model, but attributes that are not controlled by the meta-model could have any value. If the meta-model has a regular expression to express what kinds of values are correct, e.g. for the name attribute, it may be impossible to set a value without user-interaction. The same is true for change actions when handling attributes that are not explicitly specified in the meta-model what values they should contain.

One way of handling this is to create a new element or change the element to an element with default values, but this approach may lead to new problems in the model. The default value of a new element might for example break a rule of the meta-model, a problem that is hard to check and correct before committing the action.

Chapter 7

Meta-Model Guided Model Refactoring

Refactoring is a technique for restructuring existing code by changing the internal structure without changing the behavior of the code. The restructuring is done by changing a part of the code, for example by moving it, but at the same time changing all references to this part of code so that they reflect the change. The change made to the code could be the renaming of a variable or a method, to move a method from one class to another or to move classes between packages. Each change that is done through refactoring is often quite small and have little impact on the total structure of the code, but a whole set of refactoring steps can restructure the code totally and make it easier to use, more transparent and perhaps ensure that if follows a design pattern.

Refactoring [22] is a technique that has become common in Integrated Development Environments (IDE) such as Microsoft Visual Studio, Eclipse and many others.[4]

7.1 Model Refactoring

The same concept of making changes to structure without changing the semantics and behavior can be applied to models. This is called model refactoring. Each refactoring step can be a change in one model element or to move an element from one place in the model to another.

One can argue that moving an element or changing its' name will alter the semantics of the model, but if one sees the model as a view of the system, a change in the model that describes the system will not change the functionality of the system, but the way in which it is structured and presented

in the model. This means that changing the name of a model element or relocating an element will not change the semantics of the model.

Model refactoring is provided in some modeling tools as help for the user of the program to make changes to the model. In RSA, the user can use model refactoring to move an element or to change the name.

Using model refactoring in RSA is however not as helpful as it could be. When choosing to move an element through refactoring, the user will not gain any additional help from the program as to the new placement for the model element. The modeling tool does not have any knowledge about semantics of the model and because of this cannot decide where the most suitable place for a model element would be, or even where it is allowed and not.

7.2 Using the Meta-Model as Guidance

By using the extra information that is gained from the meta-model about the semantics in the model and the structure that the model must follow, the system can give additional help to the user in the refactoring steps provided by this project. This project has investigated a couple of different refactoring actions and how the information from the meta-model can be used to give additional help to the user.

The four model refactoring actions that have been investigated are add element, change element, remove element and move element.

- Add element

  The action of adding an element to the model is the same as creating a new model element in a given place of the model. If the tool does not consider the meta-model or if there is no existing meta-model for the model, the element that is to be created could be any UML-element and have any attribute values allowed by the UML-standard.[32]

  By using the information in the meta-model about which element types that are allowed in a specific place in the model, the choice can often be reduced to a couple of different element types. E.g. in a package for Java classes the user will only be able to add classes, interfaces and packages.

  Furthermore the meta-model could be used to restrict the user to only choose valid attribute values for the created element, e.g. by only allowing correct names if there is a rule for the name of the element, or by only allowing the correct value of a Boolean attribute if there is a rule in the meta-model for this attribute.

- Change element

  If the tool does not consider the meta-model the changes to a model element could be any possible change that would be allowed by the UML-standard for this kind of element.

  The tool can restrict the user to only perform allowed changes to an element by using the meta-model. This have to be done carefully, since a change in an element could be restricted by the current meta-class that the model element is connected to, but after the change is made the model element should be connected to another meta-class in the meta-model which might allow the change. This is the case if the changed attribute is a key attribute in either the existing meta-class or the potentially new meta-class for the model element.

  By performing a change to a model element so that it is classified by another meta-class the system has altered the semantics of the model, which might be seen as breaking the definition of refactoring. However the need to be able to perform such changes to correct errors in a model is quite evident, see Chapter 6 Correction Suggestions for more reasoning about this.

- Remove element

  When removing a model element the information from the meta-model could be used to alert the user if she or he is about to introduce an error into the model. This should happen if by removing the element from the model, the number of instances of the same meta-class as the model element decreases below the minimum number of instances specified by the meta-model. For example the meta-model might state that there must be at least one element of a specific type. If the user tries to remove a model element of this type and there are no other elements of the same type, the user should receive a warning regarding this.

- Move element

  Moving an element in the model without considering the meta-model is a refactoring action that lets the user move the element to any place in the model. Since the tool does not have any knowledge about the semantics of the model or the rules for the structure, any element is allowed anywhere in the model.

  By using the information about semantics and structure from the meta-model, the tool can restrict the user to only moving an element to a

place where it would be allowed. This will often reduce the number of possible moves greatly, e.g. if the user wants to move an element of the type Class and we only have one package that is allowed to contain classes, this will be the only option for the user.

Chapter 8

Implementation

The objective of this project is to develop a prototype for error correction in UML-models. This chapter gives a summary of the implementation of the system.

The system developed during this project is a prototype that serves as a proof of concept for our proposed solution to error correction in UML-models (see Chapter 6 Correction Suggestions for details about the proposed solution). Since the system produced is a prototype and only parts of a full-fledged system for error correction have been developed, the resulting design and architecture uses a modular approach to make it possible to add parts of functionality as they are developed. The system that is developed during this project is an extension to an existing program, MetaModelAgent, and have used MetaModelAgent as a library of functionalities - sometimes limiting what have been possible to do during development.

The fact that MetaModelAgent for Eclipse (see Section 3.5 MetaModelAgent for more information about MetaModelAgent) have not been completely finished during this project, but have evolved and been changed throughout the development of the extension, have had impact on the evolution of the extension as well. During the project there has been a close collaboration with Objektfabriken throughout the entire development process with discussions about how to design the system and what functionality that are needed for our extension.

MetaModelAgent is implemented on top of the frameworks Eclipse Platform, EMF and UML2 (refer to Chapter 3 The Modeling Environment for details about these frameworks) can be seen in Figure 8.1. EMF and UML2 are large frameworks containing much functionality for modeling. Much of this functionality is not needed in MetaModelAgent, e.g. a model element in EMF and UML2 saves much information about the elements that are irrelevant to MetaModelAgent. To separate MetaModelAgent from this irrelevant

information, an abstraction layer is created between MetaModelAgent and the underlying platforms. This abstraction level consists of two major parts, Model Access and Meta-Model Access, which give the functionality for accessing, manipulating and working with models and meta-models. These parts work as a façade layer that presents abstract methods that are available for model manipulation. The third major part of MetaModelAgent is the parser which uses the model access and meta-model access methods to parse the model against its' meta-model to find errors in it.

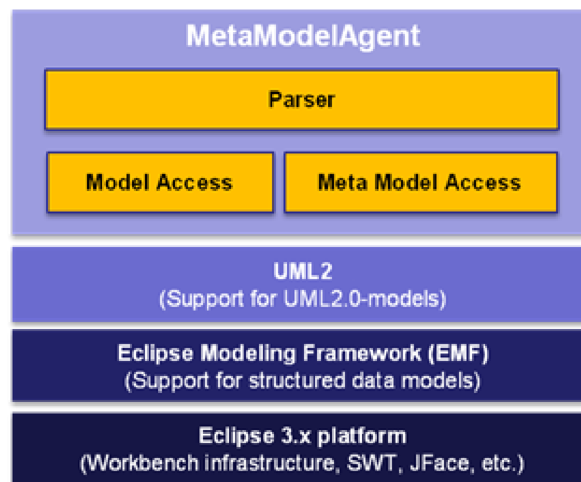


Figure 8.1: Implementation of MetaModelAgent on the Eclipse Platform.

The system developed in this project address two new features for MetaModelAgent, model refactoring and error correction of UML-models, both guided by the meta-model. Both these features have been developed as an extension to MetaModelAgent, using MetaModelAgent as provider of the errors in the model and of functionality for accessing and manipulating models and meta-models.

The extension to MetaModelAgent that have been implemented is planned to be integrated completely into MetaModelAgent. Figure 8.2 shows how this integration will be made. Note that the extension is integrated on the same level as the parser.

The core of the developed extension for MetaModelAgent is the model refactoring part. This part provides functionality for generating model refactoring actions and to commit a refactoring action. The different refactoring actions that are available are to add, remove, change and move a model element. See Chapter 7 Meta-Model Guided Model Refactoring for more details

about this part. The other major part of the developed extension is the error correction. This part uses the modeling refactoring functionality to express the correction suggestions to errors in the model as refactoring actions. The error correction feature takes problems or model errors from the parser and finds correction suggestions to these. See Chapter 6 Correction Suggestions for more information about how this is done. Each correction suggestion is described in terms of refactoring actions. After creating the set of correction suggestions for a specific problem in the model, the model correction uses a probability generator to calculate how likely it is that the user wants to apply each specific correction suggestion to the model.

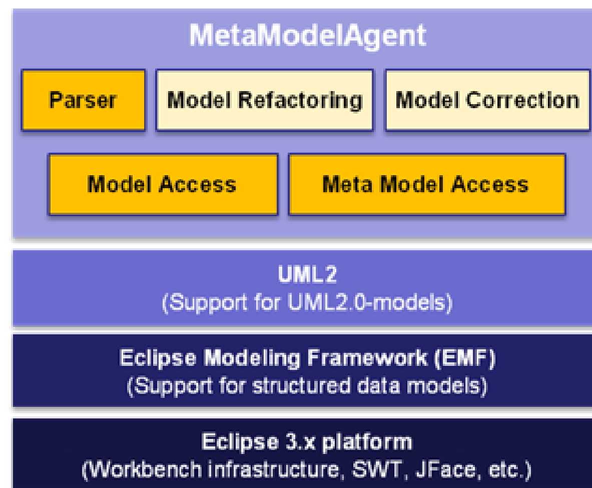


Figure 8.2: Implementation of the extension to MetaModelAgent.

Chapter 9

Related Modeling Environments

This chapter describes some of the other tools and products for modeling and meta-modeling available in the market today. It compares these products to MetaModelAgent to find differences between the tools and functionalities that are shared among the different systems.

9.1 The Generic Modeling Environment

The Generic Modeling Environment (GME) [19] is a modeling tool in which it is possible to create model domains. The tool allows the user to create their own meta-models that describes the domains of the models. When users create a new model, they choose a meta-model to use as the domain for the model. The meta-model gives the domain the functionality for the model and configures the tool to allow only modeling within that domain and its' specified rules.

9.1.1 Meta-Modeling in GME

The meta-model in GME describes entities, their relations and their constraints. An entity in GME is the same as a model element in MetaModelAgent. An entity could be any object that the user might want to model in the domain, e.g. an Actor to an OR-gate. The meta-model in GME is referred to as the model domain. By letting the user describe their domain, the tool is able to provide a domain-specific tool that is configurable for many different domains.

A superset to UML class diagrams make up the modeling language for domains in GME. The addition that is made to class diagrams is the functionality to change the way an element is being visualized and presented,

which helps the tool present the model in a way that suits the user. Each entity that is an available entity in the model is described in the domain by an UML class element. As in MetaModelAgent, the stereotype of the entity is used to describe which meta-model element it is.

The basic constraints between elements are set by using multiplicity on entity relations. To express more complex constraints GME uses Object Constraint Language (OCL). OCL can constrain a single entity or the entire domain. An OCL constraint on a single entity could for instance be to control the value of a property. An OCL constraint on the whole domain expresses relations that the entire domain should conform to, e.g. constraining the occurrence ratio between two entities in the model or the containment hierarchy of an entity.

9.1.2 The Modeling Environment

GME is able to create meta-models for any type of model domain. A domain could for example be anything from UML to circuit diagrams to business processes. When GME is in modeling mode a domain is set to control what actions are possible and how entities are presented. Only actions that are valid actions according to the domain will be available in the tool. This means that it is not possible for a user to add an entity to the model that is not in the domain or to put a relationship between two model entities that is not allowed to have a relationship by the domain. Since the domain has total control over what actions the user can take, the numbers of errors a user can create in the model are very limited.

9.1.3 GME Compared to MetaModelAgent

MetaModelAgent is similar to GME in numerous ways. Both tools have identified the value of modeling new model domains in the tool and then utilize the information about the domain to verify models. They both use UML class diagrams to specify the meta-model and they both put constraints on entities or elements.

Both MetaModelAgent and GME can express simple relation constraints and property constraints. However, MetaModelAgent cannot express as complex constraints as GME since it does not use OCL today. MetaModelAgent will therefore not be able to put constraints on the entire model or constraints on an element that would need information that is not present in the property that is constrained.

GME is also a general modeling tool letting the user do modeling for as good as any domain, which MetaModelAgent is not. MetaModelAgent is

limited to what it can and cannot model by the environment it is used in. Since MetaModelAgent is built upon UML2 and EMF, it is able to control any domain that is implemented in EMF. In its current state MetaModelAgent is only available for RSA and can therefore only build meta-models for UML.

The important difference between MetaModelAgent and GME is that while it is possible for the user in MetaModelAgent to add entities to the model that are not described by the meta-model and to set values that are not correct according to the meta-model, it is not possible in GME. Since the user cannot create errors while modeling in GME in the same way as the user can in MetaModelAgent, the model correction feature developed for MetaModelAgent would be of little or no use in GME.

One way to describe the difference between the meaning of the meta-models in MetaModelAgent and GME is that the meta-model in MetaModelAgent lets the user define guidelines for how their model should look and work and to define rules that must be followed by the model. In GME on the other hand, the meta-model is a definition of the modeling domain, describing the domain and what shall be possible to model in it. In GME the environment and the tool changes to reflect the domain in use.[20]

9.2 A Tool for Multi-Formalism Meta-Modelling

A Tool for Multi-Formalism Meta-Modelling (AToM3) is another tool for creating domain specific environments. It relies on a formal basis of graphs. The approach of AToM3 is to consider all models and meta-models as graphs. To validate a model against a meta-model, graph rewriting is performed to see if the model matches the meta-model. AToM3 is developed in collaboration between the Modelling, Simulation and Design lab at McGill University, Quebec Canada and Universidad Autónoma de Madrid.[2]

9.2.1 Meta-Modeling in AToM3

To create a meta-model in AToM3 the user first has to load it with a specific meta-meta-model. The meta-meta-model is to the meta-model the same as the meta-model is to the model, i.e. a set of entities and constraints, controlling what can and cannot be modeled in the meta-model. The meta-modeling language in AToM3 is Entity Relation Diagrams (ER-diagrams) and thus the meta-meta-model describes the ER-formalism.

AToM3 lets the user create meta-models in a visual browser. ER-diagrams are used to visually represent data objects and their relations and are often used to describe databases [26]. The ER formalism is made up of entities,

relations and cardinality, making it general enough to describe any entity in a model, and simple constraints.

Entities have attributes that describe aspects of the entity. E.g. a meta-model for a UML Class is an entity with attributes common to a Class element connected to it. The attributes can for example be name, stereotype or abstract. AToM3 can model the appearance for an entity in the meta-model. This way AToM3 becomes a very general modeling environment, capable of modeling any domain and presenting it in any way possible.

Constraints are divided into two groups in AToM3. The first group is the semantic constraints and the second is graphical constraints. Semantic constraints describe the entities attributes and relations. The simple semantic relations are described by multiplicity constraints between entities in the meta-model. To create more complex constraints that are not possible to express by means of ER-diagrams, AToM3 uses OCL [33]. The use of OCL makes it possible to create constraints like for example a model entity's name should be unique. Sometimes it could be desirable to change an entity's appearance when some event has occurred, AToM3 handles this by letting the user put graphical constraints on entities in the meta-model. This way we could model one entity for both Interface and Class since their behavior is similar, and let the appearance change depending on whether or not an interface attribute is true or not.

As mentioned earlier, AToM3 relies on the graph formalism. AToM3 treats all models as graphs regardless of how they are presented or what data they contain. The way AToM3 performs its' model validation is by graph rewriting. Graph rewriting has been used in many other areas to perform e.g. program optimization [1]. AToM3 say that to their knowledge, no one has ever applied it to formalism transformation [12]. The basic idea behind graph rewriting systems is to have a general graph, the meta-model, and take a model graph and try to rewrite the general graph into the model graph [13]. This is performed by a set of rules and action in each step. The rewriting is complete when there are no more available rules to apply to the graph rewriting.

9.2.2 The Modeling Environment

The environment that exists today for AToM3 is sparse. It has the functionality it needs, but no fancy interface, since the tool is still under development. The environment changes depending on what type of meta-model is loaded. Hence the user can only create entities that are valid in the domain described by the meta-model.

9.2.3 AToM3 Compared to MetaModelAgent

The similarities between MetaModelAgent and AToM3 are mostly on an abstract level. If we dig deeper into their concrete implementations and functionalities, not many things are the same. MetaModelAgent relies on a parser that takes a model and a meta-model and then iterates over the model to check for meta-model conformance. Any problem discovered along the way is stored and then presented at the end. AToM3 will, as recently described, use graph rewriting to accomplish the same thing.

At an abstract level, they have similarities. Both MetaModelAgent and AToM3 model the meta-model in a modeling language, and although AToM3 uses ER-diagrams, it could use UML if a meta-meta-model for UML were created. AToM3 is a more general modeling environment than MetaModelAgent, able to model any domain and presenting entities in a way that suits the user, while MetaModelAgent is today limited to UML-diagrams.

AToM3 will let the meta-model control the modeling environment and only allow actions and entities from the meta-model to be executed and created, much like GME. AToM3 can handle OCL-constraints, making it a more expressive meta-modeling environment than MetaModelAgent.

9.2.4 AToM3 Compared to GME

AToM3 has more in common with GME than it has with MetaModelAgent. Both GME and AToM3 have the same functionality and both aims at being a general modeling environment that is controlled by a meta-model. The difference is that AToM3 can create new meta-meta-models, making it possible to create new meta-modeling environments. At the core AToM3 and GME works in totally different ways, but what they achieve is the same. GME is a finished product, which one cannot say about AToM3. While AToM3 is still a tool under development and research, GME is a stable environment ready for use in the industry [13].

Chapter 10

Related Work

This chapter will describe some of the related work which we have come in contact with during our work with this thesis. Most of the research in this area has been focused on validation of UML models and consistency checking between different UML diagrams describing the same system. These topics are closely related to our own, but do still not provide an easy solution to our problem. However, much of the research is highly interesting and gave us many new insights and starting points for our problem.

When changing an UML model it is generally hard to know what side effects might occur and which model elements of the UML model that might have to be changed accordingly. Briand et al. [27] proposes a solution to impact analysis and change management on UML models. Using formally defined impact analysis rules written in Object Constraint Language, OCL, and a measure of distance between model elements to prioritize results based on these OCL rules, they present a way to estimate the impacts a particular change in the model would have.

Working with different versions of UML models is a problem addressed by Ohst et al. [21] Their approach to detect differences between two models include classification of changes between models. To visualize these differences they use the concept of unifying two UML models into one, thus showing both of the models at the same time while highlighting the differences between them.

Validation of UML models is a common problem and there are many approaches to different solutions. Shen et al. [10, 36] proposes a toolset based on the semantic model of UML using Abstract State Machines, ASMs (read more about ASM in [5]). By using ASMs and the ASM Model Checker they are able to prove that the UML validation is correct with respect to the semantic model used.

Finding inconsistencies among a set of UML models is a common prob-

lem and there are many approaches to this problem. Egyed [18] proposes a method for doing instant consistency checking while the developer is working with the model, through scope-based consistency checking for a number of consistency rules.

UML is naturally not the only modeling language which have been meta-modeled. Paige and Ostroff [34] builds a meta-model for the object-oriented modeling language Business Object Notation, BON, using both the language BON itself and the PVS specification language. Using the PVS theorem prover they perform conformance checking of models against the meta-model.

Part III

Result

Chapter 11

Benefits of Meta-Modeling

As stated earlier, modeling as a process is becoming more and more used as a process in the development of software systems. The need to have blueprints for computer programs as well as visualizing a system with models is evident, not only as a mean for programmers to know what to code, but also to discover faults before any code has been written [8]. Finding faults in the design of a system before the programming phase can save a lot of time and money since it is always easier changing the design before any real implementation is done. If large changes are done to the design after implementation has begun, there is a much greater risk that much of the code will have to be rewritten to meet the new demands of the changed design.

To get the most out of modeling and finding design faults as early as possible in the development process it is important that there are no errors in the model itself. To make the process of modeling as valuable as possible to a company developing software systems, no new type of faults should be able to be introduced by the model. If the modeling process introduces new types of faults that can be made, the company has not gained as much value from modeling as possible. If the model contains errors, it is likely because the system being modeled is so complex that the possibility of the modeler making a mistake is substantial. This is where the potential of MetaModelAgent and the extension to it provided by this project can be really beneficial.

11.1 The Benefits of MetaModelAgent

MetaModelAgent introduces a way of controlling the modeling environment in a non-intrusive way. The modeling environment is still able to create any correct UML constructions, but the system will alert when and where faults

that are not allowed by the meta-model are being made. The fact that a modeler will be notified when a fault has been made will make the modeling process more valuable since any faults made by the modeler can be corrected by the modeler at the time of creation, instead of during a time consuming model review process.

Using meta-models will help to ensure that a model is correct from the beginning, thus saving time and money by being able to skip or cut down on time spent on model review processes. Using MetaModelAgent as a part of a software process will be most beneficial in large development processes rather than small scale software projects, since these are often more complex, the models might be created by more than one person and they might follow a set of different guidelines that come from processes being used in the development process.

MetaModelAgent can also be a valuable asset for companies that strive to follow quality assurance models, e.g. CMM. Quality might be measured by how reproducible, verifiable and documented each step of the development process is and using MetaModelAgent to verify models against guidelines will aid a company in following a quality assurance model.

11.2 Correction of Model Errors

The model correction extension to MetaModelAgent that has been developed will greatly improve the speed at which a modeler can model a correct model that conforms to a set of guidelines. When a modeler creates a fault in the model, the modeler needs a way of correcting that fault in a quick manner. The model correction feature that has been developed during this project will aid the modeler in finding the solution to the problem and then to automatically solve it.

This can be compared to how programming environment like Microsoft Visual Studio or Eclipse can be more efficient to use compared to a simple text editor. Many programming environments will mark the faults made in the code once a fault has been made and most often also suggest solutions to the problem, in Visual Studio this is called IntelliSense [24]. In a simple text editor the developer has to compile the code in a separate compiler and find the problem in the text once compilation is completed. In the same way as many programming environments, MetaModelAgent with the model correction feature will find the fault once it is created and mark it as a fault and also suggest solutions to the fault. The model correction feature is to MetaModelAgent what the IntelliSense is to Visual Studio.

11.3 Guided Model Refactoring

In a perfect environment a model would not need to be changed after it is finished, but in real-world use a model changes during the lifetime of a software development process. The need to have functionality for changing a model in a safe way and in a way that will not violate the meta-model is clear. The refactoring functionality that this project provides aid the modeler in changing models without breaking the modeling guidelines given by the meta-model and will be a valuable addition to MetaModelAgent. For a modeler this will save valuable time when the need to change a model appears.

Chapter 12

Conclusion and Future Work

12.1 Conclusion

This project has resulted in a prototype presenting ways to use meta-models to correct UML models. The prototype supplies refactoring guided by the meta-model, giving the user more help during model refactoring and allowing only refactoring that would be allowed by the guidelines expressed in the meta-model. The prototype also produces correction suggestions to errors found in the UML model when validating it against its' meta-model. These correction suggestion are based on the meta-model and can be automatically committed using the meta-model guided refactoring that this project have developed.

The project is mainly based on a classification of model errors that can be found during validation of a UML model. These classifications have been refined throughout this project and is the foundation for the meta-model guided error correction that the prototype implements.

These results shows us that using meta-modeling of the modeling domain to provide the user of a modeling environment with helpful tools such as meta-model guided refactoring and error correction is possible. The results also imply that the classification of errors is a sufficient base for providing useful help for correcting errors in an UML model.

12.2 Future Work

The area of meta-modeling is a complex, yet interesting area with possibilities and opportunities for more research. We divide our thoughts about future research in three areas: research into meta-modeling and error correction in general, further development of MetaModelAgent and new applications and

development of the prototype that this project resulted in.

Meta-modeling is an area of research which does not seem to be as explored as it could or should be. There are lots of interesting opportunities where meta-modeling can be used for different purposes. This project has examined using meta-models to guide error correction of UML models and have successfully implemented a prototype showing this functionality. We perceive that this is an area that could be further researched, i.e. correction of UML models using meta-models. It would also be interesting to see if these ideas and notions would be applicable to other modeling domains and languages. Another aspect that would contribute to this area of research is a comparison between the notion of using meta-models to correct a model made in a more generic modeling environment, such as done in this project, and the notion of using meta-models to describe the domain and restricting the modeling environment to only allow such constructs, as done in GME and other similar modeling tools.

Regarding MetaModelAgent as a program, there are a few areas that would be interesting to explore further to enhance the use of the tool even more. The single most important of these would be to add support for OCL which would allow for things such as rules and guidelines that apply to all elements in the model and for context-dependent rules, e.g. stating that the value of an attribute in an element must be the same as the value of the same attribute for the parent element. There are other extensions to MetaModelAgent that could be interesting as well, for example a functionality to export and express a meta-model as guidelines in text or perhaps functionality for using several meta-models for the same model.

The prototype that this project have resulted in would need some more work before it can be considered a part of a product ready for the modeling market. There are also a number of features that could be added to our extension of MetaModelAgent that would increase the gained value even more.

Today the meta-model is seen as a static model that describes the modeling framework and which guidelines should be followed. Another perspective is to view the meta-model as a dynamic model that is allowed to change. This could in particular be helpful when developing the meta-model. The first step could be to let the prototype correct meta-model errors as well, or to change the meta-model to allow for errors found in the model. One way to interact with the program with this feature could be to create an UML model and then use the model to create the meta-model to allow for the constructions and structure in the model. This is an area which would need much more research before it is possible to implement, but it is at the same time an interesting area.

Another feature that could extend the prototype is to let the correction suggestions adapt to a specific user of the modeling environment. This could allow the tool to suggest solutions that the user often chooses as the first choice suggestion, i.e. giving priority to the suggestions that have been chosen as solutions to a certain kind of error before. One of the reasons that this have not been implemented in the current version of the prototype is that the typical user will only use the error correction feature a few times for a specific kind of error, thus not creating enough material for the tool to adapt to. However it is an interesting notion that should be explored if the prototype were to be further developed.

This is of course only a few of the areas where interesting research could be made in the domain of meta-modeling, but it is those areas that the authors feel is most important and that would be the most relevant continuation of the research done in the span of this project.

Glossary

Notation	Description	
CMM	Capability Maturity Model	11
Eclipse RCP	Eclipse Rich Client Platform	49
EMF	Eclipse Modeling Framework	51
ER-diagrams	Entity Relation Diagrams	59
GME	The Generic Modeling Environment	57
IDE	Integrated Development Environments	45
OCL	Object Constraint Language	58
OMG	the Object Management Group	10
RSA	IBM Rational Software Architect	2
RUP	Rational Unified Process	12
SPICE	Software Process Improvement and Capability dEterminiation	11
UML	Unified Modeling Language	1
XMI	XML Metadata Interchange	51

Bibliography

[1] U. Assmann. *Graph rewrite systems for program optimization*, University of Karlsruhe, 2000. ACM Transactions on Programming Languages and Systems (TOPLAS).

[2] *AToM3 Home Page*, Modelling, Simulation and Design Lab, McGill University. http://atom3.cs.mcgill.ca/, accessed on 2006-11-24.

[3] *747 Celebrating the Past, Building the Future*, The Boeing Company, 2006.
http://www.boeing.com/news/feature/747evolution/747facts.html, accessed on 2006-11-21.

[4] D. Bradley. *Practical Analysis for Refactoring*, University of Illinois, 1999. http://st-www.cs.uiuc.edu/ droberts/thesis.pdf, accessed on 2006-11-23.

[5] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*, Springer-Verlag, 2003.

[6] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose. *Eclipse Modeling Framework*, Addison Wesley Professional, 2004.

[7] G. Cernosek. *Next-generation model-driven development*, IBM Corporation, IBM Software Group, 2004.
ftp://ftp.software.ibm.com/software/rational/web/whitepapers/rsa-cernosek-wp.pdf, accessed on
2006-11-23.

[8] G. Cernosek and E. Naiburg. *The Value of Modeling*, IBM Software Group, 2004. ftp://ftp.software.ibm.com/software/rational/ web/whitepapers/ValueOfModeling.pdf, accessed on 2006-11-21.

[9] P. Chen. *The Entity-Relationship Model - Toward a Unified View of Data*, 1976.

[10] K. Compton, Y. Gurevich, J. Huggins, and W. Shen. *An Automatic Verification Tool for UML*, 2000.

[11] J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy. *The Java Developer's Guide to Eclipse, 2nd Edition*, Addison Wesley Professional, 2005.

[12] J. de Lara and H. Vangheluwe. *AToM3: A Tool for Multi-Formalism and Meta-Modelling*, Modelling, Simulation and Design Lab, McGill University, 2002.

[13] J. de Lara and H. Vangheluwe. *Using AToM3 as a Meta-CASE Tool*, Modelling, Simulation and Design Lab, McGill University, 2002.

[14] *The Eclipse Modeling Framework Overview*, 2005.

[15] *Eclipse Modeling - MDT*, 2006. http://www.eclipse.org/uml2/, accessed on 2006-11-23.

[16] *Eclipse Platform Technical Overview*, IBM corporation, 2006. http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf, accessed on 2006-11-23.

[17] *Eclipse.org*, 2006. http://www.eclipse.org/, accessed on 2006-11-23.

[18] A. Egyed. *Instant Consistency Checking for the UML*, Teknowledge Corporation, 2006.

[19] A. Ledeczi et al. *Composing domain-specific design environments*, IEEE Computer, pp. 44-51, November., 2001.

[20] A. Ledeczi et al. *The Generic Modeling Environment*, Nashville, Vanderbilt University, 2001.

[21] D. Ohst et al. *Differences between Versions of UML Diagrams*, Universitaet Siegen, 2003.

[22] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series, 2000.

[23] H-E. Eriksson et al. *UML 2 Toolkit*, Wiley Publishing, Inc., Indianapolis, Indiana, 2004.

[24] K. Getz. *An Overview of Visual Basic 2005*, MCW Technologies, LLC, 2005. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs05/html/vb2005verview.asp, accessed on 2006-11-24.

[25] K. Hussey. *Getting Started with UML2*, International Business Machines Corp., 2006. http://www.eclipse.org/modeling/mdt/uml2-uml/docs/articles/GettingtartedithML2/article.html, accessed on 2006-11-23.

[26] *Data Modeling: Entity-Relationship Model*, 2004. http://www.utexas.edu/its/windows/database/datamodeling/dm/erintro.html, accessed on 2006-11-24.

[27] L. O'Sullivan L. Briand, Y. Labiche. *Impact Analysis and Change Management of UML Models*, Carleton University, Ottawa, 2003.

[28] *MetaModelAgent - Meta-modeling of modeling guidelines*, Objektfabriken, 2004. Technical documentation for MetaModelAgent supplied by Objektfabriken for this thesis.

[29] *MetaModelAgent - Modeling Agent for IBM Rational Rose*, Objektfabriken AB, 2006. http://www.objektfabriken.se/mma/metamodelagentroductsheetng.pdf, accessed on 2006-11-23.

[30] *Objektfabriken AB*, 2006. http://www.objektfabriken.se/english.shtml, accessed on 2006-11-23.

[31] *UML 2.0 Infrastructure Specification*, The Object Management Group, 2003. http://www.omg.org/docs/ptc/03-09-15.pdf, accessed on 2006-11-21.

[32] *Unified Modeling Language: Superstructure - version 2.0*, The Object Management Group, 2005. http://www.omg.org/docs/formal/05-07-04.pdf, accessed on 2006-11-21.

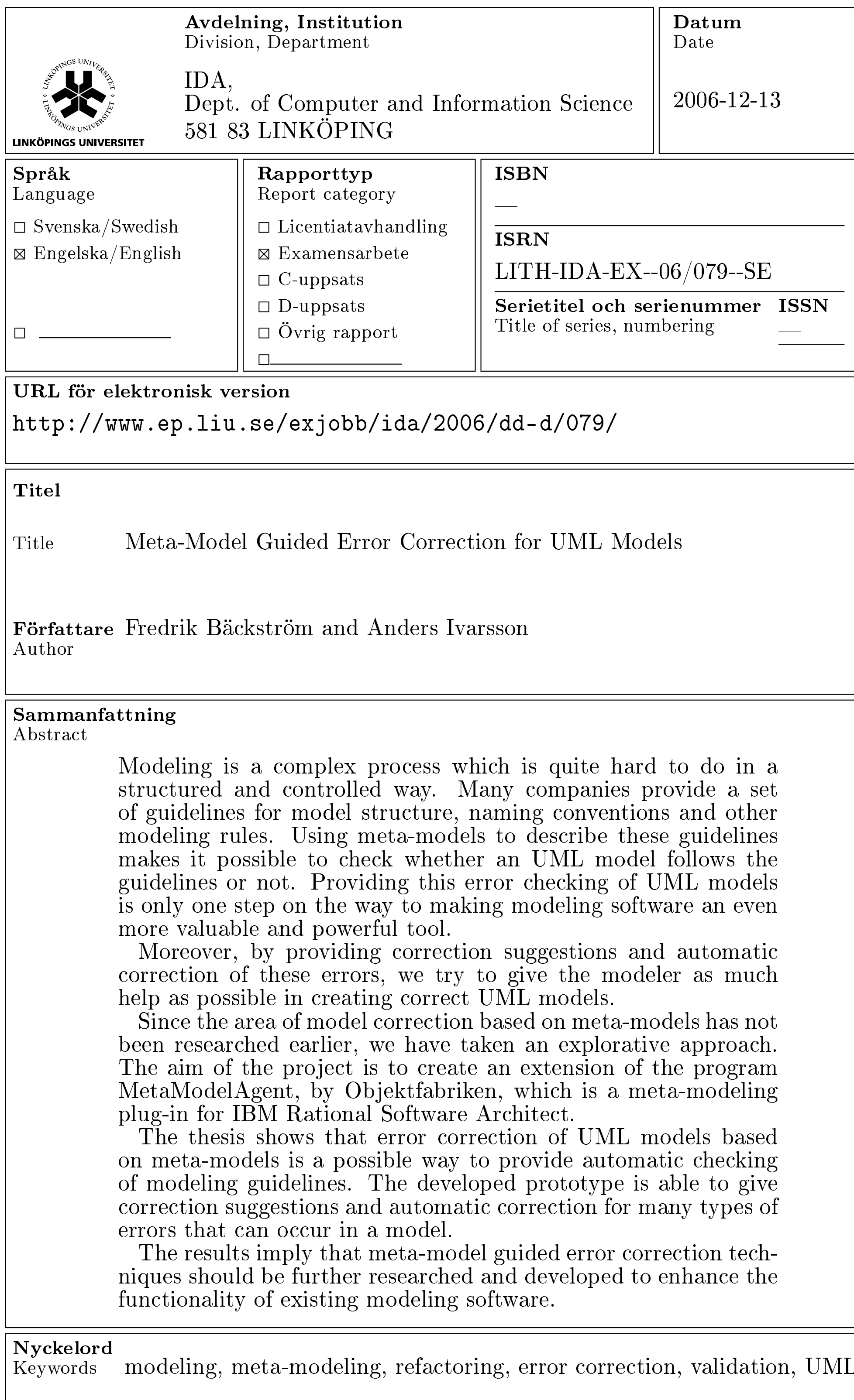
[33] *Object Management Group*, 2006. http://www.omg.org/, accessed on 2006-11-24.

[34] R. Paige and J. Ostroff. *Metamodelling and Conformance Checking with PVS*, York University, 2001.

[35] D. Schmidt. *Model-Driven Engineering*, Vanderbilt University, 2006. http://www.cs.wustl.edu/s̃chmidt/PDF/GEI.pdf, accessed on 2006-11-21.

[36] W. Shen, K. Compton, and J. Huggins. *A Toolset for Supporting UML Static and Dynamic Model Checking*, Springer-Verlag, 2003.

[37] H. Vangheluwe and J. de Lara. *Meta-Models Are Models Too*, Proceedings of the 2002 Winter Simulation Conference, 2002.



Avdelning, Institution
Division, Department

IDA,
Dept. of Computer and Information Science
581 83 LINKÖPING

LINKÖPINGS UNIVERSITET

Datum
Date

2006-12-13

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ ______

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ ______

ISBN
—

ISRN
LITH-IDA-EX--06/079--SE

Serietitel och serienummer
Title of series, numbering

ISSN
—

URL för elektronisk version
http://www.ep.liu.se/exjobb/ida/2006/dd-d/079/

Titel

Title Meta-Model Guided Error Correction for UML Models

Författare Fredrik Bäckström and Anders Ivarsson
Author

Sammanfattning
Abstract

Modeling is a complex process which is quite hard to do in a structured and controlled way. Many companies provide a set of guidelines for model structure, naming conventions and other modeling rules. Using meta-models to describe these guidelines makes it possible to check whether an UML model follows the guidelines or not. Providing this error checking of UML models is only one step on the way to making modeling software an even more valuable and powerful tool.

Moreover, by providing correction suggestions and automatic correction of these errors, we try to give the modeler as much help as possible in creating correct UML models.

Since the area of model correction based on meta-models has not been researched earlier, we have taken an explorative approach. The aim of the project is to create an extension of the program MetaModelAgent, by Objektfabriken, which is a meta-modeling plug-in for IBM Rational Software Architect.

The thesis shows that error correction of UML models based on meta-models is a possible way to provide automatic checking of modeling guidelines. The developed prototype is able to give correction suggestions and automatic correction for many types of errors that can occur in a model.

The results imply that meta-model guided error correction techniques should be further researched and developed to enhance the functionality of existing modeling software.

Nyckelord
Keywords modeling, meta-modeling, refactoring, error correction, validation, UML

LINKÖPING UNIVERSITY
ELECTRONIC PRESS

LINKÖPINGS UNIVERSITET

Copyright

Svenska

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om *Linköping University Electronic Press* se förlagets hemsida `http://www.ep.liu.se/`

English

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the *Linköping University Electronic Press* and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: `http://www.ep.liu.se/`

© Fredrik Bäckström and Anders Ivarsson
Linköping, December 21, 2006