FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Constraints and Models@Runtime Support for EMF Profiles

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Christian Wiesenhofer, BSc
Matrikelnummer 0926066

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Ass.Prof. Mag. Dr. Manuel Wimmer

Wien, 19. April 2017

Christian Wiesenhofer	Manuel Wimmer

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Constraints and Models@Runtime Support for EMF Profiles

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Christian Wiesenhofer, BSc
Registration Number 0926066

to the Faculty of Informatics

at the TU Wien

Advisor: Ass.Prof. Mag. Dr. Manuel Wimmer

Vienna, 19th April, 2017

Christian Wiesenhofer	Manuel Wimmer

Erklärung zur Verfassung der Arbeit

Christian Wiesenhofer, BSc
Sandgrubengasse 22, 3251 Purgstall

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. April 2017

Christian Wiesenhofer

Danksagung

Zuallererst möchte ich meinem Betreuer Manuel Wimmer danken. Vor allem, dass er mir überhaupt erst ermöglicht hat, meine Arbeit zu diesem Themengebiet zu verfassen. Weiters möchte ich mich auch für die Betreuung und die hilfreichen Wegweiser während dieser Zeit bedanken.

Natürlich auch recht herzlichen Dank an meine Eltern für die Unterstützung und den Beistand, nicht nur während meiner Studienzeit.

Allen Freunden und Bekannten während des Studiums sei an dieser Stelle auch gedankt, einerseits für die lustige Zeit und andererseits auch für die tolle Zusammenarbeit und Hilfsbereitschaft.

Zu guter Letzt möchte ich natürlich auch Magdalena von ganzem Herzen für ihre Geduld und Unterstützung danken.

Kurzfassung

Modellierungssprachen spielen in der Software-Entwicklung eine große Rolle. Zurzeit wird hierfür hauptsächlich UML verwendet, sogenannte domänenspezifische Modellierungssprachen (DSMLs) finden aber immer größeren Anklang. Ihr Hauptvorteil ist ein höherer Abstraktionsgrad, wodurch die automatische Generierung von Quelltext erleichtert wird. Solche DSMLs zu erstellen ist jedoch mit großem Zeitaufwand verbunden. Um dieses Problem zu lösen wurde das EMF Profiles Projekt gestartet. Es ermöglicht, ähnlich wie UML-Profile, die Verwendung von leichtgewichtigen Profilen, um ein bestehendes Modell zu erweitern. Dadurch, dass mithilfe dieser Profile eine Änderung der Meta-Modelle wegfällt und Bestandteile wiederverwendet werden können, verringert sich die benötigte Entwicklungszeit.

Im Vergleich zu reinen Meta-Modell-basierten Sprachen, bestehen gewisse Einschränkungen in EMF Profiles. So existiert zurzeit keine Möglichkeit starke Nutzungsbeschränkungen oder Laufzeitverhalten in Profile einzubinden. Dadurch ist zum Beispiel ein typischer Anwendungsfall - die gleichzeitige Verwendung mehrerer Sprachen - nicht möglich. Daher stellte sich die Frage wie diese Funktionalitäten realisiert werden können.

In dieser Arbeit werden zwei Erweiterungen von EMF Profiles vorgestellt und prototypisch implementiert. Zur Evaluierung dieses Prototypen wurde eine Fallstudie durchgeführt. Die Lösung der Problemstellungen erfolgte durch einen OCL-Mechanismus für Beschränkungen bei der Anwendung von Stereotypen, sowie durch einen Generator, der AspectJ Code-Fragmente zu Profilen hinzufügt, die dann das Laufzeitverhalten eines Elements ändern. Die Fallstudie beinhaltet eine Basis Petri-Netz Sprache, der drei Petri-Netz Erweiterungen hinzugefügt wurden. Diese Erweiterungen sind als EMF Profile umgesetzt, wobei all deren Spezifikationen vollständig umgesetzt werden konnten. Weitere Metriken wurden erhoben, um die Vorgangsweise und den Prototypen bewertbar und vergleichbar zu machen.

Abstract

Modeling languages play an essential part in the software engineering process. Currently, mostly UML is used for that purpose, but domain-specific modeling languages (DSMLs) get more and more attention. Their main benefit is a higher abstraction-level, which eases generating code from such models. One major drawback of DSMLs, is their time-consuming development. To tackle this problem the EMF Profiles project was founded. It provides a lightweight extension mechanism, just as UML profiles, to be used for DSMLs. This way models can be altered without modifying their whole metamodel and domain properties can be reused, thus reducing the required development time.

In comparison to pure metamodel-based languages there are certain limitations in EMF Profiles. There is no way to model constraints regarding the restricted use of stereotypes or to include runtime behavior. A typical use case is for example to use multiple languages at once. However, considering these shortcomings, such an attempt is not possible. Thus the question emerged, how these features can be realized.

In this thesis two extensions to EMF Profiles are presented and implemented as prototype, which is then evaluated using a case study. The research problems were solved by introducing an OCL constraint mechanism, which manages the stereotype application. Furthermore a generator was implemented to add AspectJ-based code fragments to profiles, so they can influence the runtime behavior of a model element. The case study was conducted by creating a base Petri net language and adding three Petri net extensions, implemented as EMF profiles, to it. All of their specifications could be fully implemented. Further metrics about the approach and the prototype were collected, in order to ensure it is assessable and comparable.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Modeling languages play an essential part in the software engineering process. They are used in the design stage to help visualize the software project with all its components and behavior. The created models are important software artifacts, which not only ease the design and development phases, but also maintenance and extension of software. While UML is currently the primarily used language for this purpose, others such as Domain-Specific Modeling Languages (DSMLs) [GNT+07] are catching up. They get more and more attention, due to their benefits in certain areas over UML, as described later. Several conferences and workshops cover topics out of the DSML domain, most notably the annual Model Driven Engineering Languages and Systems (MoDELS) conference or the Object-Oriented Programming, Systems, Languages & Applications (OOPSLA) Workshop on Domain-Specific Modeling. Also noteworthy is the book called "Domain-Specific Modeling: Enabling Full Code Generation" by Steven Kelly and Juha-Pekka Tolvanen [KT08], where they propose to use DSMLs to generate source code. Both Kelly and Tolvanen were major contributors to the field of DSML.

UML is designed to be a general-purpose language as described by the OMG UML Standard up to version 2.4.1 (cf. bibliography entry [staf]). That means the language design is held more generic to allow the modeling of a broad spectrum of domains. In contrary, DSMLs are designed to fit to only one or a few domains. This way they provide a higher abstraction-level than UML, since its abstraction-level can only rise up to the common denominator of all supported domains. The concept of UML Profiles allows a more domain-specific modeling with UML too. It uses profiles, a collection of stereotypes and other mechanisms, to extend standard UML semantics with domain-specific semantics. This approach however also does not reach the abstraction level of DSMLs. They can only add semantics but not remove them, also the possibilities to add semantics are restricted.

Due to the fact that DSMLs can be tailored to the needs of one specific domain, developers using these languages spend less time on general and low-level modeling parts and therefore have more time to model the actual important domain-specific parts. These precise specifications are also a prerequisite for code generation, where source code is automatically generated based on a model. But not only source code can be generated from models, depending on the framework even documentation, configuration and test cases can automatically be generated. Code generation is not restricted to DSMLs. It is also possible to generate code from UML models. The main difference between the two approaches however, is the degree to which code can be generated. As addressed before, the more general UML models only allow a broader code generation with less detail.

In software engineering the use of code generation marks the first major raise of abstraction, since the move from assembler to languages such as object-oriented programming languages [KT08]. This abstraction leads to a significant raise in productivity. According to Kelly and Tolvanen case studies in companies showed a five to ten times faster development, when using DSMLs and code generation, compared to traditional software development.

The possibility to generate code from models is one of the main reasons to use DSMLs. This feature brings a lot of benefits to software projects. The first and most important benefit is the reduced programming effort. Most of the low-level code like getters and setters can easily be generated, whereas it otherwise would have to be manually written and, in case of changes, rewritten. Depending on the size of the software project, code generation of a model can easily result in some thousand lines of code. These lines can be worth a lot of money, since even simple lines of code do cost time and therefore money to develop.

Also when it comes to security concerns code generation can be of much help. In case a security flaw has been found in the code, the leak only has to be resolved once in the generator and every generated program just has to be rebuilt with the new generator to resolve the issue. Furthermore the generator ideally produces best practice code for a given function and is therefore not prone to human errors such as lack of knowledge or just pure mistake.

In conventional software projects the development is done by programmers. With code generation however, the larger part could be done by modelers. Modelers, who at its best don't have to write code at all and, due to validations and restrictions, have nearly zero risk of producing errors. These benefits suggest a bright future for DSMLs.

1.2 Problem Statement

One major drawback of DSMLs however is the time-consuming development of each of those specific modeling languages. Over the time DSML tools were developed to ease and reduce the effort required to create a new language. The three most evolved and used tools

are the Eclipse Modeling Framework (cf. bibliography entry [Ecl]), MetaEdit+[1] and DSL Tools for Microsoft Visual Studio[2]. While such tools provide the needed functionalities to create new languages and slightly reduce the effort to do so, the main problem - namely time consuming development - is still an issue.

One reason as to why the language creation is so tedious is the reduced flexibility to change the metamodel [LWWC12]. To tackle this problem the EMF Profiles project was founded. EMF Profiles is a project developed by the Business Informatics Group department at the TU Wien[3]. The project first started as a prototype for the paper "From UML Profiles to EMF Profiles and Beyond" published in 2011 [LWWC11]. The goal of EMF Profiles was to bring the benefits of the UML profile design to the DSML area. The purpose of UML profiles is described in the Object Management Group UML specification [stag] as follows: "The intention of Profiles is to give a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain, platform, or method." With the EMF Profiles project EMF too got a lightweight extension mechanism to extend models without redefining their metamodels.

In EMF Profiles this was achieved, as in UML Profiles, through the use of so called stereotypes. Stereotypes are indirect instances of EClass, as defined in the Ecore meta-metamodel, that can be applied onto a model. They provide specific properties for a domain and can, through profiles, be reused in different models, therefore minimizing the overall development time.

With the problem of time consuming development mitigated another one emerged, which limits the further usage of EMF Profiles and therefore DSMLs. One thing to keep in mind is that developers of DSMLs are most of the time not the same ones that use these languages afterwards. So a language should stand for itself and include all necessary restrictions to create a valid model. Currently there is no possibility to apply stereotypes only if certain constraints are valid. Modelers may therefore not be able to create certain languages that require strong constraints. Even if a language with weaker constraints can be created, it will result in tedious work to manually check every applied stereotype and may even be unmanageable for larger models. With such an error-prone approach the outcome will often be an invalid model. The severity of this problem becomes apparent when keeping in mind that - as stated in [TK05] - the main reason to use DSMLs, besides speeding up the development, is to reduce the number of errors in a model.

Another deficiency in the current EMF Profiles project is that there is no support to alter the state of entities at runtime. Stereotypes can only have structural features within their tagged values to decorate model objects with. This prohibits the use of EMF Profiles for altering dynamic behavior. Developers can currently only use other products to model such behavior, without the benefits of the profile concept and therefore probably increased development time. When creating languages through a metamodel in EMF it

[1] MetaEdit+ website: `http://www.metacase.com/mwb/`
[2] Microsoft Visual Studio website: `https://www.visualstudio.com/vs/`
[3] Business Informatics Group website: `http://www.big.tuwien.ac.at/`

generates executable DSMLs (xDSML), an extension to DSMLs which include runtime behavior. Such behavior cannot yet be stereotyped. It should however be possible to stereotype every aspect of a model. This way profiles will be powerful and reusable tools to change models without redefining the metamodel.

This thesis aims to tackle the last two mentioned problems: the stereotype constraints and the runtime behavior support. Solving these issues will increase the project's area of application, the speed of development, reduce possible errors and subsequently help to spread DSMLs with all their benefits and prospects. DSML-modelers therefore gain a powerful tool to create all or nearly all their languages with, in a time-saving and error-preventing manner.

1.3 Aim of the Work

The aim of this work is to create a prototype of a generic framework as extension of EMF Profiles to add constraint as well as runtime support. This should allow the definition of various constraints for stereotypes, which will prohibit the application of the corresponding stereotype to a model object, if a constraint is not met. Furthermore the extension should be able to process runtime behavior and it should then be possible to automatically change the behavior of certain events if a stereotype is applied.

With this framework it should be possible to answer the two research questions, "How can EMF Profiles be expanded to support the creation of modeling languages, which by definition have strong modeling constraints?" and "How can EMF Profiles be expanded to support the adaptation of runtime behavior?". To be able to assess these results it is also necessary to create and evaluate a case study. In the end, there should be a comprehensive case study to evaluate each of the implemented features and therefore deliver answers to the research questions. The case study includes standardized languages, which are built according to their specifications using EMF Profiles. The language specifications have to include constraints and runtime behavior.

The evaluation should also result in an extensive list of strengths, weaknesses and metrics of the approach. These gathered metrics serve as numeric criteria to compare and rate different approaches as answer to the research questions. Metrics in this study may for example be, how many classes needed to be changed and how many functions or constraints could not be implemented as defined in the standard.

The prototype that is developed during this thesis should allow a comprehensive profiles mechanism for EMF. This means a fully functional stereotyping feature, where every language feature can be packed into a profile to use and reuse it, without any metamodel or model changes. Reusable modeling parts - in this case profiles - reduce the development time, the overall development amount and the error rate. The prototype itself is provided as open source code, so it can be further extended and improved.

A more universal goal of this thesis is to increase usage of DSMLs and code generation. With the increased power of EMF through the performance enhancing and lightweight

use of EMF Profiles, more developers may choose this approach to create their programs. This way development projects can be completed quicker and require less effort and costs, while still holding up an equal or even increased level of quality.

1.4 Methodological Approach

The methodological approach to develop these implementations and afterwards evaluate their outcome is based on the general steps of design science. It consists of the four main stages research, design, implementation and evaluation as follows:

1. Literature Research

   An important first step towards a proper solution for the two research questions, is to get an overview of published papers in the field. The probably best way to do so is a systematic literature review (SLR), as proposed by Biolchini et al. [dABMN+07]. This SLR process is a standardized, structured and efficient approach to find papers and articles relevant to a specific research area. Using this systematic literature review, papers regarding the general topics of DSMLs, Models@Runtime, the existence of other tools for similar problems and best-practice approaches were reviewed. This research was conducted as first step of the thesis, but systematic and ad-hoc literature reviews were also conducted while working on the approach definition, for any upcoming questions or in case of requiring new concepts and technologies. For example the decision between Java or fUML (cf. bibliography entry [stae]) as programming language, was the first decision which required a review.

2. Designing the Approach

   After the required information had been gathered, the main step of this thesis followed. This is where the actual solution approach is proposed. One main requirement the approach has to fulfill, is that the finished framework should still offer a lightweight and independent profile mechanism. This means the support of constraints and runtime should be usable without changing the metamodel or model in any way other than applying the stereotype. Since there are two research questions, this step is also divided into two subsections:

   a. Constraint Support for EMF Profiles

   The first part of the approach deals with a solution to the missing constraint support, as this seemed to be the lesser problem of the two. The key goal of this task was to find a standardized language that can define constraints. These constraints have to be able to handle class structure. It should be possible for instance to write constraints based on the value of an objects attribute. The approach definition also lists the necessary changes in EMF Profiles to support the additional notation of constraints for stereotypes, their

evaluation at the time of stereotype application to a model object as well as further required features.

b. Runtime Support for EMF Profiles

The approach-part for the implementation of runtime behavior was developed next. A few concepts had to be researched and analyzed, regarding if they could be used to achieve changing the runtime behavior of a model. The new behavior should be based on a definition in the stereotype, but it should thus only execute on an object that has the same stereotype applied to it. Other objects, also those of the same class, should not have this new behavior, if they do not also have the stereotype application. Furthermore the approach also has to support the application of multiple stereotypes on the same model object.

3. Realization

After the approach had been finalized it was implemented accordingly. In this part the designed approach had to prove its suitability. If, at any point of the implementation process, the approach would have appeared to not lead to a proper solution or to not work at all, it would have been reconsidered and adjusted based on the findings. The goal of this phase was to receive a working prototype, which can afterwards be evaluated.

4. Evaluation

As a final stage in the research process it is necessary to evaluate the produced prototype. Therefore a case study using the newly developed framework was conducted. In this study standardized modeling languages were selected and built according to their specifications, using the prototype. The chosen modeling languages had to have a few constraints and runtime behavior in their specifications. The evaluation was split into three parts, one for each research question plus another one for a more general evaluation.

a. Evaluating Constraint Support

To evaluate the constraint support feature, language specifications regarding restrictions on the usage of these languages were assessed and evaluated. This tasks' purpose was to show how many constraints could be implemented and how many could not.

b. Evaluating Runtime Support

Additionally all functional specifications were evaluated, whether or not they could be achieved using the prototypes stereotype features.

c. Evaluating Further Metrics

Finally additional metrics of the stereotype were collected. This was done to get a comprehensive look on the prototype and make it comparable with other approaches in every possible way. Therefore not only numerical but also

qualitative metrics were assessed. Metrics in this study are for example the amount of changed classes.

1.5 Structure of the Work

The following six chapters document the research regarding the problem statement, decisions taken in the process of finding a solution to the problems, implementation of the prototype based on the beforehand defined methodology and finally an evaluation of the prototype. All occurring issues, criteria, options and decisions are documented so the final approach is fully comprehensible. With the information in this thesis it should of course be possible to come to the same conclusions.

- **Chapter 2 - Prerequisites** deals with the base on which this thesis starts. All used or required prerequisites are listed and explained. This is necessary to get an overview on the presetting and restrictions, which affect the further approach definition and solutions.
- **Chapter 3 - Methodology** analyzes all possible frameworks, technologies and other components. Based on the findings an approach to develop a solution was designed. The chapter is divided into three sections, the first one - Overview - offers a quick introduction into the further process in this chapter. In the following two sections the approach for each of the two research problems is defined.
- **Chapter 4 - Realization** covers the implementation of the combined approach defined in chapter three. First, prerequisites to implement the approach are documented, followed by the actual implementation of the constraint support and secondly the runtime behavior support. The resulting prototypes' source code is available on GitHub [Pro]. Important parts of the source code are also documented in this chapter.
- **Chapter 5 - Evaluation** includes the language selection for the case study, the study itself and the resulting metrics. This chapter also includes a critical reflection of the results, a summary of open issues and an assessment of possible threats to the validity of the gathered results.
- **Chapter 6 - Related Work** covers other publications and mechanisms that deal with similar problems or are relevant to the thesis in any other way. This chapter also includes a short documentation on the systematic literature review that was used to gather information and retrieve relevant papers for this chapter, but also for the rest of the thesis.
- **Chapter 7 - Conclusion and Future Work** finally again lists the initial problems and a short overview of the proposed solution. Furthermore shortcomings and resulting open items of the prototype are discussed. The last part of this chapter

then proposes future work to be done in this context. This chapter concludes the thesis.

All referenced papers, websites and other materials are accessible in the bibliography.

CHAPTER 2

Prerequisites

In this chapter the given or selected frameworks and technologies are presented, these represent the environment in which to solve the research questions. The chapter also provides an overview on what base within the EMF Profiles project - which is the main component in this work - the development of the thesis starts. Before the explanation of what EMF Profiles is and does, its foundation and the concepts on which it operates are explained. EMF Profiles itself is explanation later in this chapter. The first section describes the Eclipse Modeling Framework (EMF), the base DSML tool. After this section the used technologies within Eclipse are explained. These are on the one hand the underlying Equinox plugin system of Eclipse, and on the other hand the Graphical Modeling Framework (GMF), which provides a generator for graphical editors. Afterwards the UML Profiles concept is explained, which served as blueprint for EMF Profiles. The last section offers a detailed description of the EMF Profiles project including its technological composition. Throughout these sections, the main plugins and features these frameworks use are also documented.

2.1 Eclipse Modeling Framework

EMF Profiles is an extension of EMF, the Eclipse Modeling Framework. EMF itself is a tool which provides all required functionalities to create and use DSMLs. According to the projects website the aim of the EMF project is to provide a modeling framework with a code generation facility, in order to create tools and applications based on a structured data model [Ecl]. The structured data model in the case of EMF Profiles is an Ecore model as described later. The code generation facility can currently only generate Java code and projects. Once generated, a fully functional executable DSML (xDSML) is ready to use. The difference between common DSMLs and xDSML is that the latter are runnable.

In its core EMF is just another configuration of the Eclipse platform, so the basic Eclipse IDE platform with a specific set of additional pre-installed plugins. The most relevant plugins in this package are Ecore tools for the diagram editor, OCL as constraint language in Ecore diagrams, GMF for the graphical editor as well as the core EMF plugin including the code generator. EMF can be downloaded on the Eclipse project website (eclipse.org) under the package-name Eclipse Modeling Tools.

History The Eclipse platform was initially developed by Borland, IBM, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft and Webgain, starting from November 2001, according to the websites history section. In 2004 it was later released as open-source to further accelerate its development. Since then the newly founded Eclipse Foundation is responsible for the development of all the Eclipse projects, most importantly the Eclipse platform, also known as Eclipse IDE. One of those projects is EMF, which already was an independent project since 2002.

EMF is the defacto standard tool for development in the area of Model Driven Engineering (MDE) [FNM+12]. A comprehensive book to learn and understand every aspect about EMF was written by the main contributors to EMF, and is named "EMF: Eclipse Modeling Framework" [SBPM09]. The first edition was released in 2004, but there is also a newer edition to cover the numerous changes and newly developed parts of the framework.

Composition EMF comprises three core-components, each with its own separate purpose. The first one is the EMF core framework. It provides the main functionalities to create and use Ecore, such as serialization into XMI format for persistence and a change notification system based on the observer pattern to allow runtime behavior. According to its project description this component also contains a reflective API to alter EMF object in a generic fashion. This part includes very powerful functionalities to generically access every aspect of the class hierarchy, such as operations or parent objects. The XMI format was first developed and standardized by the OMG in 2005 and stands for XML Metadata Interchange [stah]. It is a convenient way to serialize and store models as files.

The second framework is EMF.Edit. It provides generic classes used to create editors for the Ecore models. These classes are primarily for common functionalities that do not have to be specialized, such as providers, property sources and generic commands. The third and last framework is EMF.Codegen, which, as the name already suggests, provides code generators to generate the needed classes and parts for an EMF model editor. The specific process to start and create models and editors for it are described in detail in the paragraph "Using EMF".

Meta-Object Facility EMF is based on the Meta-Object Facility (MOF) as standardized by OMG [stac] and also by ISO as standard ISO/IEC 19508:2014 [staa], which are both publicly available. MOF defines a hierarchical metamodel architecture. This architecture defines four separate levels called M0 up to M3, as depicted in Figure 2.1.

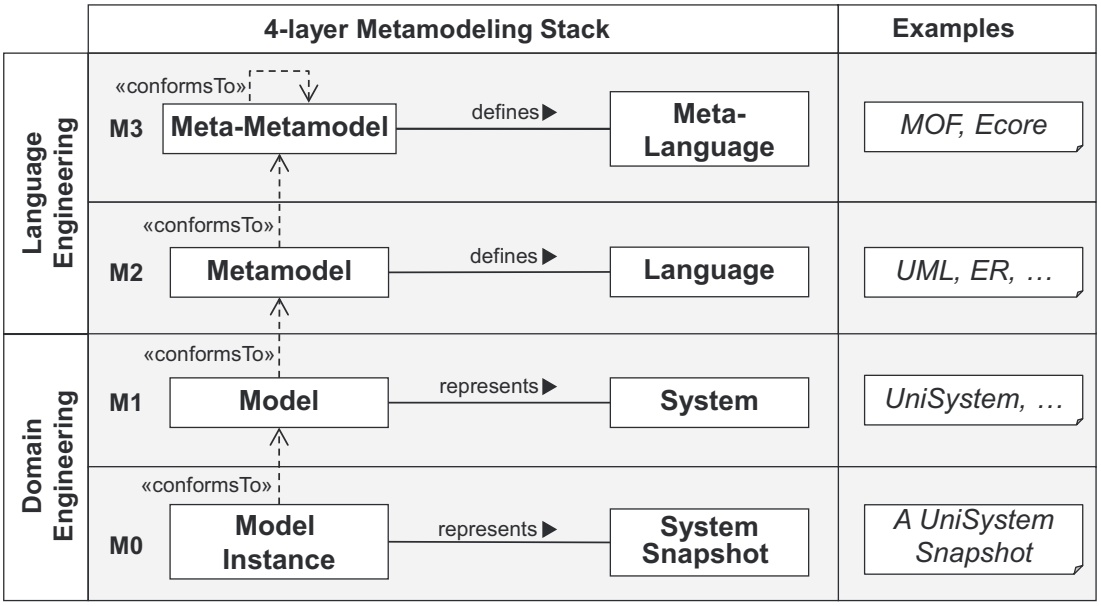

Figure 2.1: Meta-Object Facility architecture (taken from [BCW12]).

The start is at the highest, most abstract level M3, which is the meta-metamodel or MOF level. The next level below is M2, the metamodel level and after that M1, the model level. At last the run-time level M0 is the concrete object or data as instance of the model in M1 [Tan09].

The purpose of the MOF architecture level is to define other metamodel languages. In EMF this concept was adopted into Ecore, which is the base meta-metamodel language for the conception of EMF metamodels. Ecore is essentially a subset of MOF, equally to "Essential MOF" (EMOF). Although it is basically EMOF, it is called Ecore, apparently to avoid confusion as stated by Ed Merks and Sridhar Iyengar at EclipseCon 2004. The Ecore metamodels are the core part of any EMF project. They are used to generate the actual source code.

Using EMF So far what EMF is and how it is composed was described. Now details on how EMF is used follow. This is relevant because the core of EMF Profiles is also a DSML generated by EMF.

The start of every EMF project, after being created through the right-click menu under "New > Project..." as "Ecore Modeling Project" in the folder "Eclipse Modeling Framework", is a metamodel. The metamodel is the base of the whole project, it defines the main classes and references to each other. Metamodels can be built in five different ways. The most common and also the one used by EMF Profiles, is an Ecore model. Since this is the approach used by EMF Profiles, it is described in detail later on. Next to Ecore there are also options to describe the metamodel with Java Annotations, a

UML model, XML Schema or XMI and later convert them into Ecore. For this Ecore metamodel EMF will later generate an editor to create models. These models can then be edited, serialized as XMI and validated.

Usually the Ecore file is already created automatically. In case of another project type or if additional Ecore metamodels are used, they can be created in the right-click menu as before under the folder "Eclipse Modeling Framework". The next task is to fill the Ecore model with classes, references, attributes and the likes according to the specifications of the project at hand. Ecore files can be opened with a tree based or graphical editor, such as the "Ecore Editor" or the "Sample Ecore Model Editor". Once opened, the editor will show only one item, which is a package, or EPackage to be more specific, serving as the root node. Just as in object-oriented languages it is used to differentiate the underlying classes from those in other packages. Within an EPackage there are five different possible child elements:

- **EAnnotation:** Can be used to generate annotations prior to the operation definition. The required information for the generation are in child elements called Details Entry. EAnnotations can also be created in each of the sibling elements.
- **EClass:** This element represents an actual java class. Just as java classes it can have annotations (EAnnotation), methods (EOperation), variables (EAttribute) and relationships to other classes (EReference).
- **EData Type:** Is needed to include and use a data type that is not part of the Ecore model.
- **EEnum:** It will generate an enumeration class based on the Literal elements it contains.
- **EPackage:** Just as the EPackage one level above, the sub-package serves as further differentiation of its child elements.

After the Ecore model is finished, it can be used to generate code. The generation is done through the genmodel file. The genmodel file is usually created in the same folder as the Ecore. In contrast to the Ecore file, whose purpose is to represent the class structure in the domain, the genmodel file is purely for options regarding the generation process. It therefore includes the referenced Ecore model as basis and decorates it with additional properties, such as naming and editing options.

By right-clicking the root package there are five options available on what to generate from the genmodel definition. By selecting to generate either Model-, Edit-, Editor-, Test-Code or all of them, EMF automatically generates the corresponding eclipse projects. These are generated into separate projects, except the main model code, which will be generated into the current project where the genmodel is located. Within the model project the generator will create three packages. The first and main package includes

interfaces and a factory as in the factory pattern [GHJV95] to create new class instances. Furthermore there are two sub-packages generated. The first one .impl includes the concrete implementations for each of the interfaces in the main package. The second one, .util, includes utility classes such as an adapter factory.

After making changes to the metamodel or genmodel the generated code can simply be regenerated. In nearly every domain full code generation is not too simple, there are certain aspects that cannot just be automatically generated. These can be parts where Ecore is just not specific enough to model them or custom method bodies. In the case of missing information to create a body for a method, the generated code will let the code throw an exception if the method is called. For such cases the missing code can be manually added to the class files. To check if the method should be overwritten with the standard code upon regenerating, the code generator considers the Java annotation "@generated". This annotation is generated for every method by default and every method marked with it will be overwritten upon regeneration. This can be avoided by changing it to "@generated NOT" or delete the annotation entirely. Best practice however, is to add the "NOT".

2.2 Eclipse Equinox & OSGi

Eclipse Equinox is one of the main sub-projects of Eclipse. It is a fully standalone implementation of the OSGi (Open Service Gateway Interface) framework specification and therefore not dependent on the Eclipse runtime, as it could also be used within the command line using the Java runtime. But it is the runtime environment for the Eclipse IDE and RCP, and thus an inevitable component for Eclipse. Eclipse Equinox is currently handled as the reference implementation of OSGi [MVA10].

OSGi defines a component and service model according to the eclipse documentation. These main features are provided by the packages in the org.eclipse.osgi plugin. The component framework consists of all the numerous plugins that are described in the next paragraph. Each of these plugins can also extend other plugins or provide its own API-hook for others to extend. This concept is explained in the last paragraph of this section.

Plugins The whole Eclipse platform is completely based on plugins, also called bundles, which provide functionality for a specific area. Plugins are encapsulated and modular components - jar files containing Java, class and configuration files - that define their own dependencies and offered API; the latter is done by exporting its Java packages [Vog15]. An application within this framework can then consist of one or, in the usual case, a set of bundles. The EMF Profiles project for example is too comprised of plenty such Equinox plugins. A benefit of this plugin structure is to be able to easily reuse not only the whole application but already some single packages or classes out of a plugin.

A plugin has to have a manifest file (META-INF/MANIFEST.MF) in which all relevant OSGi meta information is declared. Within this file all exported packages are defined,

as are required dependencies to other plugins or packages. This file also includes the bundle-version and bundle-name which uniquely identify the plugin. All in the manifest file entered dependencies are automatically added to the plugins classpath. The purpose of exporting packages is that only those declared ones are available to other plugins, if they have the exporting plugin in their dependencies. Other packages are not visible. OSGi checks if all dependencies are loaded before starting a plugin. It reads all dependencies defined in the manifest and resolves them or, if it is necessary, activates plugins if they are not already active. If a dependency is not defined the classes are not available to the plugin and any call to such classes would result in a class-not-found error. This ensures all dependent plugins are completely resolved before the desired plugin starts.

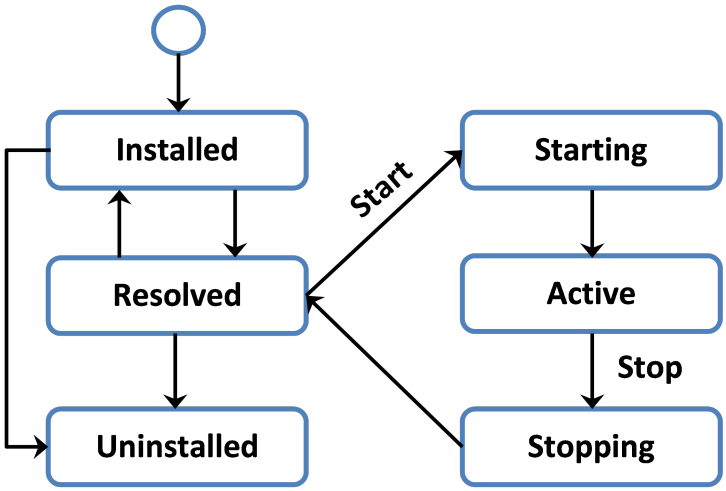

Figure 2.2: Life-cycle of a plugin (Based on [Vog15]).

The complete life-cycle of a plugin is depicted in Figure 2.2. It shows two main sequences. Within the system a plugin has to be installed first to start its life-cycle. From there on it can either be uninstalled again or resolved by the OSGi framework. To be in the resolved state all its dependencies are resolved first. It can then be started where it will have the starting state. Once the start is successfully completed it will have the active state. This is the most important state where it can actively run its code and offer its API for other plugins to be used. Once the plugin is stopped it will go into the stopping state and after completion again back into the resolved state.

To debug or control these states the framework also provides an OSGi console within Eclipse. It offers OSGi actions to manually start, stop, un- and install a plugin. To start the console the standard Console view has to be opened. Using the option “Open Console”, the Host OSGi Console can be selected. By typing the command “ss” all

plugins are listed and also the state they are in. This is an easy way to look up in what life-cycle state a certain plugin currently is.

Extensions & Extension Points Another important feature of Eclipse Equinox is the extension mechanism. It provides a flexible and extensible way for collaboration between applications. A plugin can define its extension point, which is a unique id. Each other plugin can then provide their own extensions, by declaring an extension for the extension point id. Both these declarations, extension points and extensions, are defined within the plugin.xml file.

The plugin offering the extension point usually includes a singleton class, which is responsible for processing the supplied extensions to this extension point. It therefore also provides an XML schema definition for its extension point within the plugin.xml, serving as contract according to which the supplied extensions have to be designed. This feature therefore assures the extension can be successfully processed by the API.

2.3 Graphical Modeling Framework

Another important Eclipse project is the Graphical Modeling Framework (GMF). It provides, just as EMF does for models and model editors, a generator framework to create graphical editors. To do so it utilizes the Ecore metamodel provided through EMF. Hence GMF is dependent on EMF. It is also dependent on three own metamodels to generate the appropriate source code. They provide additional informations regarding the user interface and its connection to the model as described in detail later. The generated output is created as Eclipse plugin project, ready to be deployed.

The generated source code for the graphical editor is based on GMF Runtime, which is again made up of EMF and GEF (Graphical Editing Framework) components. It is therefore an extension to the EMF generator mechanism to further generate parts of the program. All is implemented as a textbook MDE approach, where a model is the core from which all further development is started. The resulting source code for the graphical editor is then also generated into an Eclipse plugin project, to be deployed and used as OSGi plugin.

GMF provides the following main benefits:

- **Reuse:** Components for the generation can be reused for multiple domains, for instance the graphical definitions of entities. The shape of a class entity may well be reused within several domains. Also the strict separation of the GMF metamodels allows to unlink them from each other and reuse them somewhere else without dependencies.
- **Use of Patterns:** Throughout the generated source code classes multiple software design patterns are used. This way GMF provides an efficient, extensible and state-of-the-art code base.

- **Fast Development:** After a steep learning curve GMF allows a quick creation of editors.

Development To aid developers through the process of creating and configuring the metamodels, GMF provides a view called GMF Dashboard, as depicted in Figure 4.1. It gives a quick overview on how the development process with GMF works and also on the current progress of this development. Also further information such as the currently configured and linked files are shown. This is very helpful since it can be quite confusing where to start, using which metamodel and in what order. Within the dashboard however every derive, combine or transform step is graphically displayed with connecting arrows.

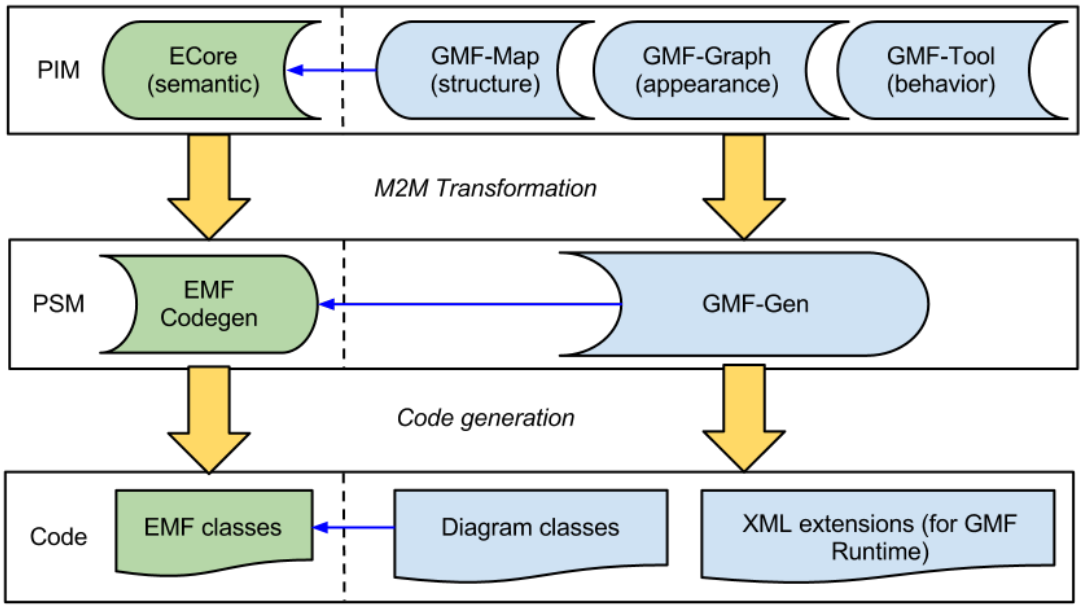

Figure 2.3: Overview of the involved entities in the generation process of GMF (Taken from GMF-Tooling project page).

All in the generation process involved entities can be seen in Figure 2.3, as depicted on the GMF project page[1]. There are five important metamodels used for configuring the editors generation. One of them, the domain model, is part of EMF the others are purely GMF models. They are described in detail in the following listing:

- **Domain Model:** The Ecore metamodel is the main file used for generation. It is the starting point, as it includes every class and relation that is defined for the model.

[1]GMF-Tooling project page: `http://www.eclipse.org/gmf-tooling/`

- **Tooling Definition Model:** Within the tooling definition the complete tool palette is defined, but also menus, toolbars and similar. The palette is a separate box next to the diagram area, which includes every item that can be placed into the diagram. The available tools can be ordered into groups, which are then put together within the group and visually separated from other groups. The tools themselves are displayed by a name label and a small icon next to it.

- **Graphical Definition Model:** As the name already states in this metamodel all graphical components are defined. These are figures, shapes, links and so on of various entities that should later be displayed within the diagram.

- **Mapping Model:** Within the mapping model the three already configured metamodels will be combined. This includes also defining how the elements of each of the metamodels are related to each other. For example which tool in the palette will create which class out of the domain model, and what figure out of the graphical model will it use to be displayed in the diagram area.

- **Diagram Editor Generator Model:** Out of the beforehand finalized mapping model this generator model will be transformed. It can be customized to further modify the outcome of the generation. Using this model the generation process can be invoked, which will create all necessary Java files and packages for a ready-to-use graphical editor.

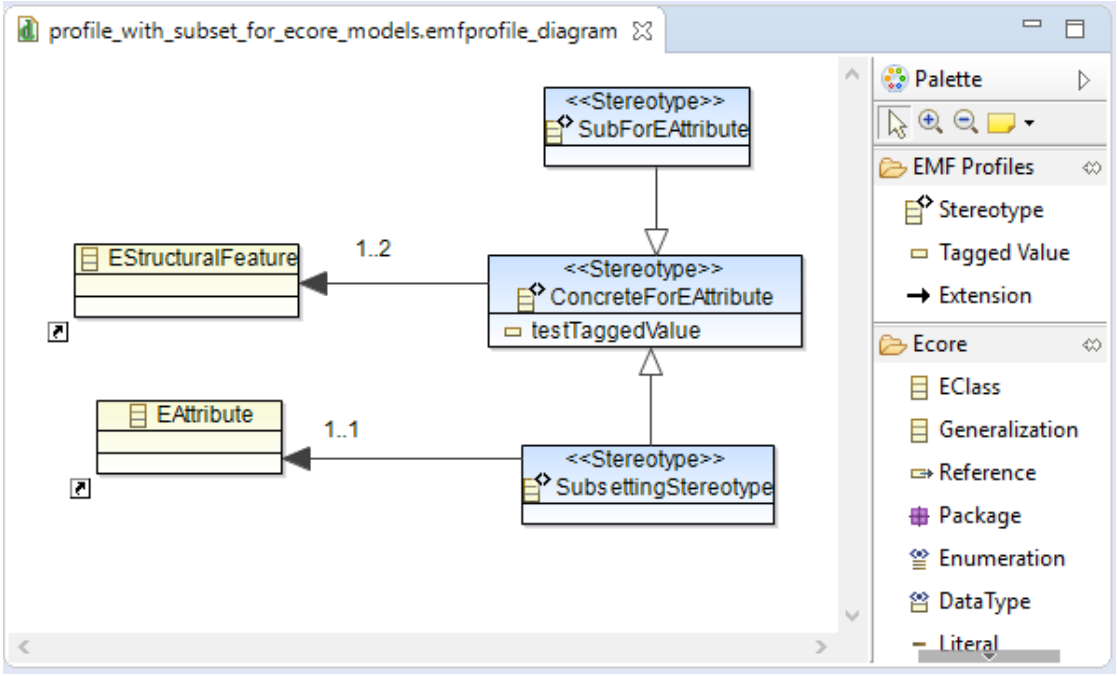

Figure 2.4: Example graphical editor, generated by GMF.

After a complete configuration of all metamodels and the generation process the final editor will look like shown in Figure 2.4. Not on the picture however is the properties view which shows all of the selected elements' properties.

2.4 UML Profiles

The concept UML Profiles is a profiling mechanism to create domain specific parts for a model saved within a profile. This way these profiles can be reused for multiple models to extend the metamodel in a lightweight fashion without changing it. A profile package may then consist of multiple stereotypes and package imports. These stereotypes then include various attributes that can be used within the model. As already stated in the introduction EMF Profiles uses the UML Profiles concept for its project. It is not entirely the same concept as a few adaptation had to be made in order to work with EMF. The difference between the two concepts will be detailed in the next section.

As thoroughly described by Langer et al. [LWWC12] the profile mechanism is an integral part of UML. It is located on the meta-metalevel M3 in the UML architecture model, as shown in Figure 2.5 as package Profiles. Using this setup it is easy to create a profile application as concrete instance in level M1. On the metalevel M2, just as the UML metamodel itself, resides the metamodel aProfile. This would be the metamodel of a concrete domain-specific profile. From this package application, instances can be created in M1. The profile mechanism in UML is therefore built just as UML itself, as it can be seen in the architecture.

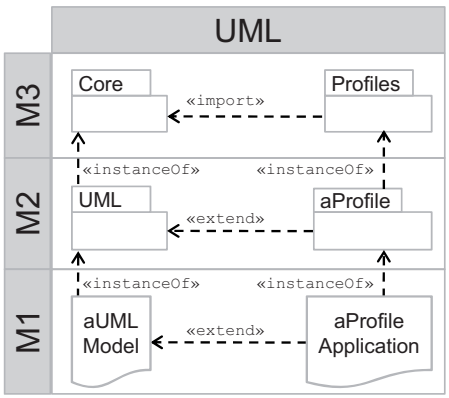

Figure 2.5: UML Architecture (Taken from [LWWC12]).

This concept has been a huge boost for the spread and usage of UML. Overall development time can be reduced through these reusable profiles. Also these lightweight extensions greatly increase the flexibility throughout the development. To summarize, UML may very well be the main reason why UML is currently - and already for some time - the most used and widespread modeling language.

2.5 EMF Profiles

EMF Profiles now uses all these concepts and frameworks above to also provide a profile mechanism for EMF. EMF Profiles was originally hosted on Google Code [EMFb]. After the service closed in January 2016 the project was transferred to GitHub. It is now available under the project lead Philip Langers' account [EMFa]. The branch chosen for this thesis to develop the framework on, was the master branch. There is a second one called develop, but it includes some untested and/or not finished work and should therefore not be used as base to work on.

EMF Profiles can be installed into Eclipse through the installation manager under Help/Install New Software. The update-site is located at modelversioning.org[2].

Comparison with UML Profiles While UML has the profile feature as integral component in its metamodel, EMF does not. This shortcoming of EMF Langer et al. [LWWC11] tried to overcome as recapped in the following paragraphs. A main problem was that in EMF it is not possible, or at least not very viable, to implement a profile package at the meta-metalevel. Hence it is also not possible to instantiate a profile in M1. This is however necessary to even use the profile mechanism. They eventually proposed two approaches as depicted in Figure 2.6, where they decided to go with approach (b) Metalevel Lifting by Inheritance.

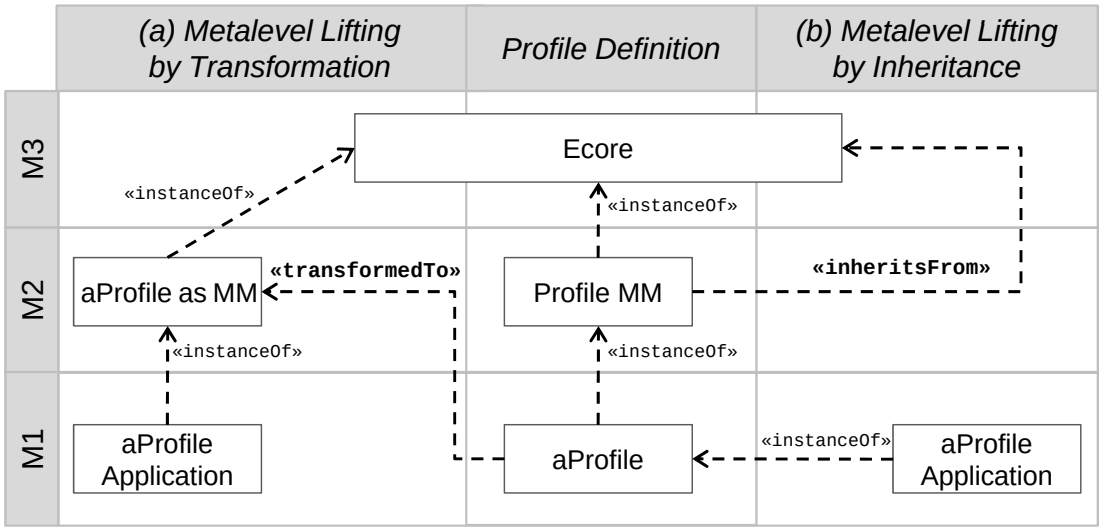

Figure 2.6: EMF Profiles Metamodel Lifting Approaches (Taken from [LWWC12]).

In this approach they created a new package on level M2 named Profile MM, the metamodel for the profile class. A profile can then be directly instantiated, because

[2]EMF Profiles update-site:
`http://www.modelversioning.org/emf-profiles-updatesite/`

of the inheritance from EClass. In EMF to be able to create a concrete instance of a metamodel in M1, it must be an instance of the meta-metaclass EClass in M3. So for the stereotype metaclass, which is defined in the metamodel Profile MM, the definition of EClass as a super type was also necessary as shown in Figure 2.7. This way also a stereotype application of a specific stereotype can be created.

The main problem of a "missing" metalevel to instantiate every entity of the profile mechanism was overcome by creating a multi-functional profile entity, whose functionality spans over two metalevels, while only residing on one. A concrete profile is as an instance of its metamodel, Profile MM on level M2, and located on level M1. But as it also includes stereotypes that can be instantiated, because of their instance-of relationship to EClass, it also serves again as metamodel. This opens up the possibility to create the horizontal instance-of relationship shown in the figure between the concrete profile (aProfile) and its profile application (aProfile Application). This relationship actually represents the additional vertical relationship that was needed.

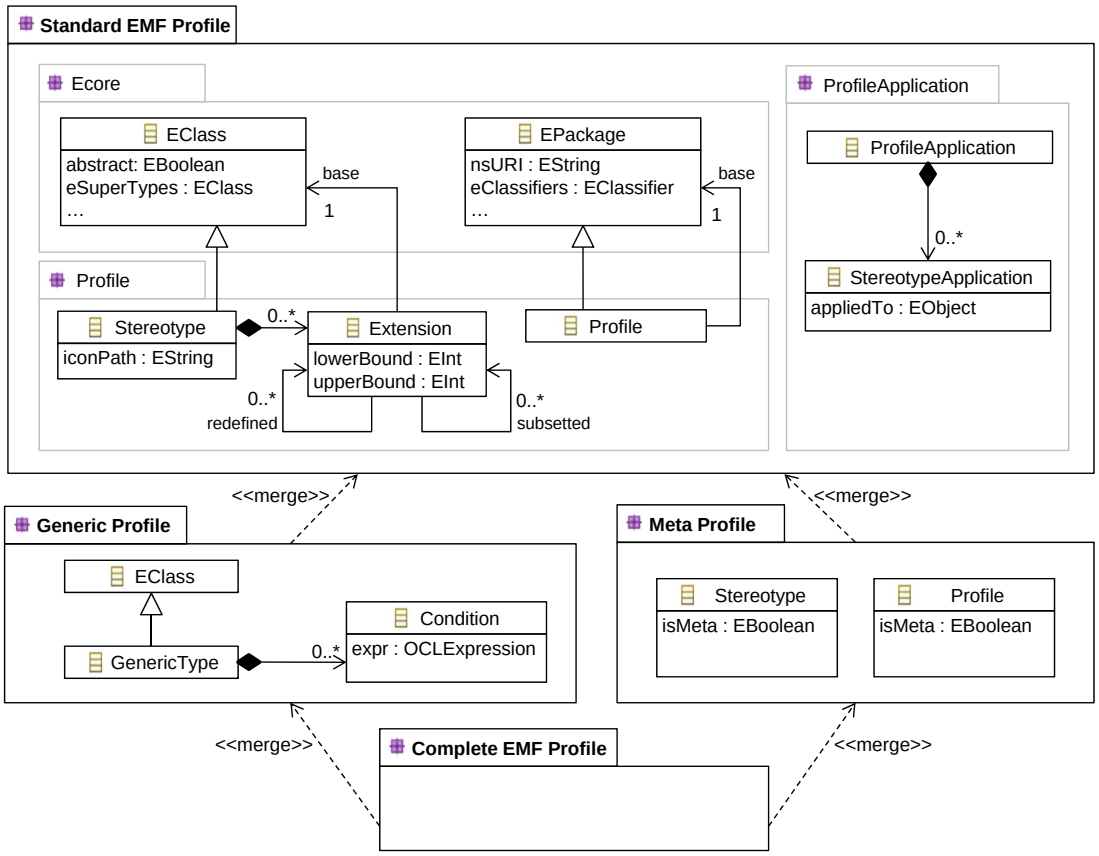

Figure 2.7: EMF Profile Metamodel (Taken from [LWWC12]).

Metamodel The main parts of the EMF Profiles project are modeled within two ecore diagrams. These two files serve as metamodels, where most of the code was generated from, using EMF. As already depicted above in Figure 2.7, the EMF Profiles project was developed according to this combined metamodel. This resulted in the already mentioned two ecore metamodels, one for the profile in Figure 2.8 and the other one for the profile application in Figure 2.9.

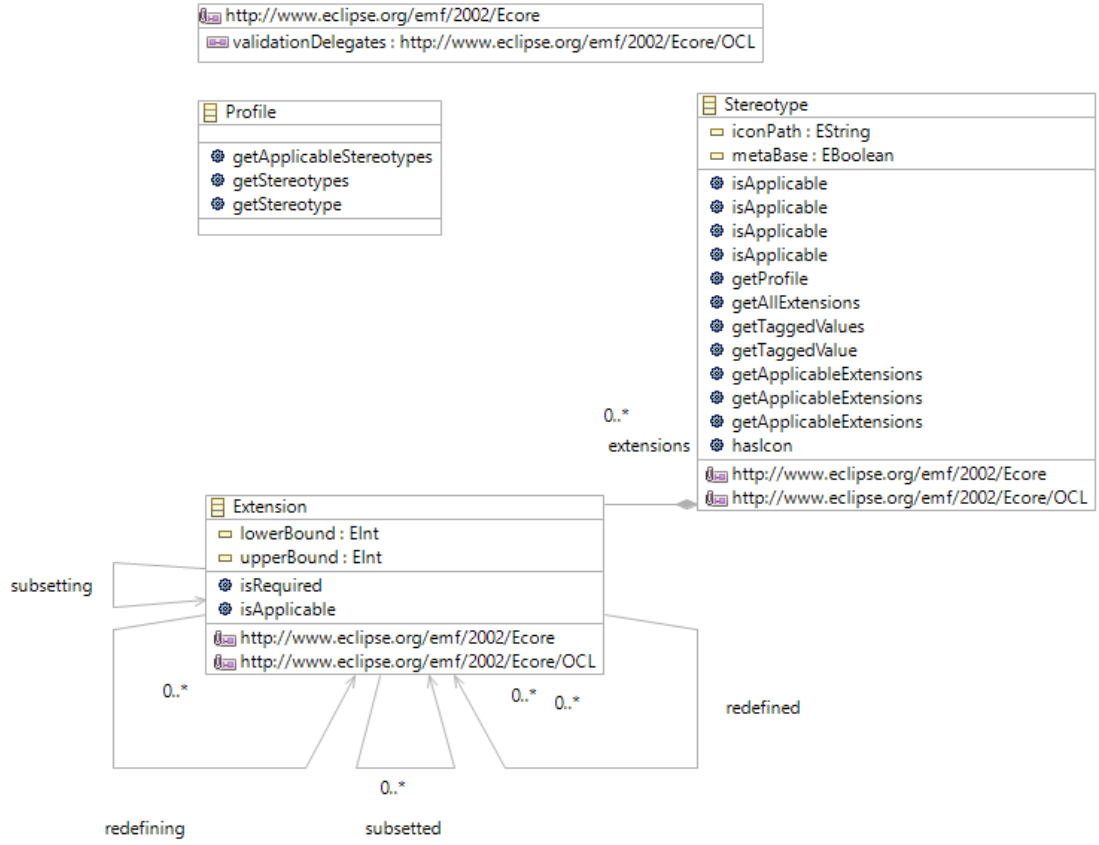

Figure 2.8: EMF Profiles Metamodel in Ecore

The profile metamodel consists of three classes: Profile, Extension and Stereotype. This metamodel is saved in the file emfprofile.ecore, in the main EMF Profiles project under the model folder. The Profile class inherits from EPackage and serves as container for stereotypes. This can be seen by the operations to retrieve stereotypes defined in the ecore metamodel. Therefore the profile manages a collection of stereotypes that are part of it. Next, the class Stereotype includes all references to extensions that are connected to it. It also includes the tagged values saved as EAttributes and provides methods to retrieve them. Other important operations are the isApplicable ones. These a boolean whether or not the stereotype can be applied onto a given eObject. Finally in the class

Extension the source and target of the extension arrow are referred. Also its restrictions on the amount of times it can be applied on the same model object is saved by the upper and lower bounds attributes.

The second metamodel covers all aspects regarding the profile and stereotype application. It is saved in the same folder as the profile metamodel and is named emfprofileapplication.ecore. It consists of four classes: ProfileApplication, ProfileImport, StereotypeApplication and StereotypeApplicability. The first one is part of the registry and keeps track of the imported profiles and applied stereotypes. The imported profiles are references to the class ProfileImport, which again has a reference to the actual Profile, whose import it represents. The class StereotypeApplication represents an actual connection between an object and an applied stereotype. Therefore it has references to the model object, the extension of the stereotype it is targeted by and the profile application as opposite of the containment reference of ProfileApplication. It also has an operation to return the base stereotype this application is part of.

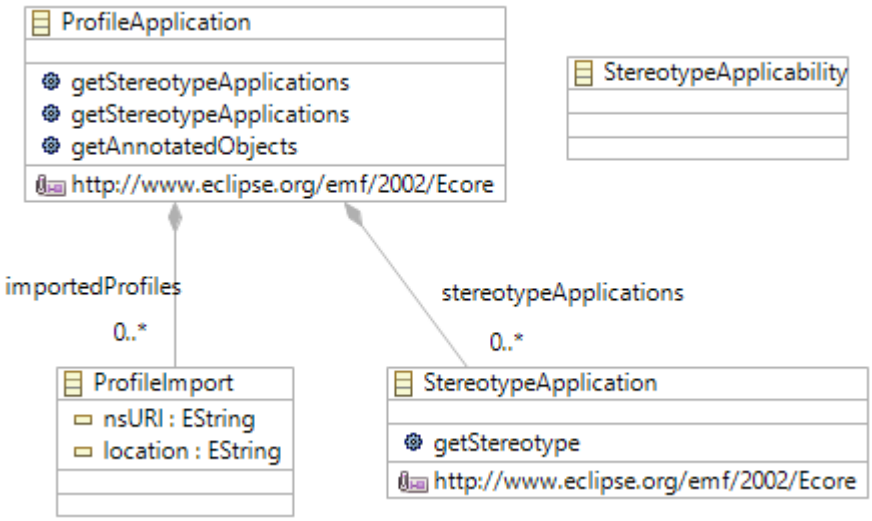

Figure 2.9: EMF Profiles Application Metamodel in Ecore

Noticeable in both metamodels are the multiple Ecore and OCL annotations. There purpose is to declare validation delegates for their parent class. These delegates are then executed in the editor to validate the state of the created model. Therefore OCL annotations have to be created within a class, providing an identifier and the actual expression. Secondly an Ecore annotation has to be created to declare all listed OCL identifiers as constraints. In the generated editor the model elements can then be validated by using the corresponding command in the right-click menu. This functionality is specifically provided for ecore models by the OCLinEcore sub-project.

Components The EMF Profiles project is, as already mentioned, based on a DSML generated with EMF. It consists of fifteen projects in the Eclipse workspace, whereof four of those projects were generated through the basic EMF generator. These include the main project, which also contains the Ecore models and genmodels, as well as the edit editor and test projects. Another one was also generated, but by GMF not EMF. The other projects are different registries and deployment projects. A detailed information on each project will be given later in this section. Two projects are purely for deployment. The other thirteen non-deployment projects are also plug-in projects. After deployment they are available as jar-archives at the update-site and can be installed into an Eclipse IDE. The technology used for these plug-ins is the Equinox/OSGi framework as described above, on which the Eclipse platform is built on.

- **org.modelversioning.emfprofile**

  This is the main project where the model code resides. It includes the two ecore metamodels. The first, emfprofile, is used for the core model structure of profiles, stereotypes and Extensions. The second one, emfprofileapplication, is the metamodel representing everything related to the application of profiles and stereotypes as well as to the registries. In this project are also all GMF related metamodels and generator files.

- **org.modelversioning.emfprofile.application.decorator.gmf**

  This project includes the support for GMF based editors. Its main purpose is to check whether or not the current editor is GMF based and decorate it accordingly, so it can be used for EMF Profiles' profile and application registries.

- **org.modelversioning.emfprofile.application.decorator.reflective**

  Again this project was created to add support for certain editors, in this case reflective based editors.

- **org.modelversioning.emfprofile.application.registry**

  This bundle is responsible for the core logic of the application registry. This is the registry where all stereotype applications are saved to. The main use of it is for the EMF Profile Applications view, where all applied profiles and stereotypes are shown. This view is not to be confused with the second registry view Registered EMF Profiles, where all available profiles are shown.

- **org.modelversioning.emfprofile.application.registry.ui**

  As addition to the last project this one provides all user interface related classes. This includes the main application view, wizards and the likes. The EMF Profile Applications view shows all applied stereotypes for a selected model object in the currently opened editor. The view can also be switched to show every stereotype application within the model in the opened editor.

- **org.modelversioning.emfprofile.diagram**

  This project is mostly generated by GMF. It includes the necessary parts for the editor in which the profiles are created. The generation is done through the gmfgen file in the main org.modelversioning.emfprofile package.

- **org.modelversioning.emfprofile.edit**

  The edit code in this project was auto-generated through the genmodel. It provides a wizard to create new instances of the model.

- **org.modelversioning.emfprofile.editor**

  In this second auto-generated project are the required classes for the editor to show and modify the created model.

- **org.modelversioning.emfprofile.feature**

  This project is for deployment. It provides a so called feature which includes a category. Within this category there are all the thirteen plugins of the EMF Profiles project. Eclipse offers a mechanism for applications to be updated or installed in to the Eclipse instance. This is done by providing update-sites for these applications, where Eclipse can download the application from. On these update-sites the offered applications consist of one or more features which can be installed separately. These features serve as logical folders to organize the containing plugins.

- **org.modelversioning.emfprofile.project**

  The purpose of this project is to provide the EMF Profiles nature and utility methods to it. A nature in Eclipse is a notation to a project on its structure. Another example of a nature would be the Java nature, where the required plugin dependencies, binary and source folders are created and registered. In this case the required - and therefore created if missing - structures are a diagram file, an icons folder and a plugin XML-file.

- **org.modelversioning.emfprofile.project.ui**

  Based on the framework of the last project this one provides the wizards and their background-logic to create an EMF Profiles Eclipse project.

- **org.modelversioning.emfprofile.registry**

  This project creates and administers the EMF Profiles registry. All EMF Profiles projects within the workspace and all such plugins are registered. Only herein registered profiles are available to apply to a model later.

- **org.modelversioning.emfprofile.registry.ui**

  This bundle provides the necessary user interface to the profiles registry. Its main purpose is the Registered EMF Profiles view, where all profiles that were found in the registry above are shown.

- **org.modelversioning.emfprofile.tests**

  The fourth and last of the auto-generated projects is tests. For each of the model classes some jUnit test cases were generated. Additionally to those, there are also some manually created test cases for newly developed features.

- **org.modelversioning.emfprofile.updatesite**

  In Eclipse an update-site is an address for a project, where the most recent version, but also older ones, can be downloaded from. In this case the updatesite project is linked to the feature project, which again provides the projects to deploy. Through this projects site.xml a build process can be initiated. All linked plugins will be compiled into jar files and provided under a new version number.

Profiles & Stereotypes The profile feature is similar to that of UML Profiles. The starting package is a profile. Within this profile one or more stereotypes can be defined. These stereotypes can later be applied to model objects, where first the profile itself has to be applied to the model once. All these profile and stereotype definitions are made in a graphical GMF-based editor provided by EMF Profiles. The corresponding diagram file (*.emfprofile_diagram) is automatically opened upon creation of an EMF Profiles project.

In the definition of stereotypes certain restrictions are also determined. The most important one is the definition on what class the stereotype can be applied to. This is done by pointing an extension arrow to an imported metamodel element or a self-created class. Another restriction is the amount of times the stereotype can be applied to the same object.

The stereotypes serve as container of the actual features that store additional information. In EMF Profiles these are called Tagged Values. They are entities, set to a certain type, and can then hold a value of that type. Valid types are all basic ones such as String, Integer or Enum, but also a few more sophisticated ones, for example references or collections. Once these stereotypes are applied to model objects, they extend them with the data provided by the tagged values without recreating their metamodel.

For the profile registry to be able to assess all provided profiles within the Eclipse instance, it uses the extension point mechanism. Each EMF Profile workspace project is already a plugin project, so within its structure there is also a plugin.xml file created. This XML-file declares to offer an extension to the extension point using the id org.modelversioning.emfprofile.profile. The EMF Profiles registry plugin, which has an extension point defined for this id, then collects all those extensions into its registry. Once a profile is in the registry it can be used to be applied to a model.

An example profile including two stereotypes is depicted in Figure 2.10. It defines a class Book which can be decorated by up to two different stereotypes. The first one named EBook offers a tagged value named format. This stereotype can only be applied once. Tags, the second stereotype, provides a tagged value called name. The type of name is

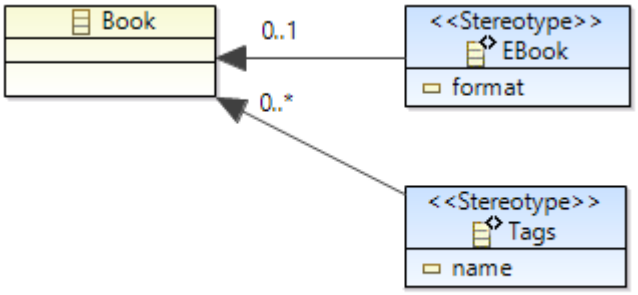

Figure 2.10: Example EMF Profile with two stereotypes.

String, which can not be seen in this figure, but in the properties view of the editor. Tags can be applied unlimited to decorate a Book object with String tags to categorize it.

Summary To summarize this section, the environment for the prototype in this thesis is mainly dependent on the EMF Profiles project. The used environment consists of the Eclipse Modeling Framework and the EMF Profiles project. These environments furthermore take use of the Java object-oriented programming language, Equinox/OSGi framework, Domain Specific Modeling concept, Ecore language, XMI persistency, the Graphical Modeling Framework and UML Profiles.

Now that the environment and basis of this thesis are defined, solving the two research problems within this context is explained in the following. Therefore in the next section possible approaches towards a solution will be analyzed.

CHAPTER 3

Methodology

So far the used frameworks and technologies were explained. This chapter now covers the selection process of frameworks, technologies, components, concepts and course of actions as well as considerations along the way, towards finding approaches as solutions to the stated problems. In the first section a small overview of the methods used in this chapter to select these frameworks is presented. The next section includes decisions and selections towards finding an approach solving the constraint issue. The third section also covers an approach definition, this time however in regards to the problem of missing runtime behavior for stereotypes.

All following approaches are chosen to work with or be implemented with Java. This is mainly due to the fact that according to the GitHub statistics, the EMF Profiles project is written entirely in Java. So it was reasonable to use it for this extension too, and not to add another language and dependencies into the project. Thus keeping the project lean and easier to maintain and extend.

3.1 Overview

In this thesis a prototype in the context described in the prerequisites Chapter 2 “Prerequisites” is provided to resolve the two research problems, mentioned in Chapter 1 “Introduction”. To be able to do so, an approach on how to create the prototype had to be designed first. This design approach had to include technologies or methodologies as solution for each of the two problems. An analytical approach was used to evaluate these technologies regarding their fit for the design. This design evaluation is documented in the following two sections.

In these sections the actual problems are reiterated, by defining the concrete shortcomings or differences to the desired behavior in the current EMF Profiles project. Then options on what could be done to achieve the desired behavior are present, including necessary

criteria for a valid solution. Next, the design search process, as explained above, is document. It was conducted by first searching through methods and technologies that could satisfy the behavior defined in the options, and secondly actually evaluating them regarding their fit for the design.

By using this prototype, developers of software projects gain a powerful tool to reduce time and effort in the development process. Through the additional constraint feature the error rate of profiles created with EMF Profiles is reduced. Furthermore EMF Profiles' area of application is expanded through the possibility to also stereotype and reuse runtime behavior. Developers can therefore use EMF Profiles more often to solve problems or assist their work. They then again profit from its benefits, such as the lightweight extension mechanism and code reusability, and also from its underlying technologies such as the EMF framework and its code generation ability. To summarize, all these benefits aid the software development and maintenance process to be easier, faster and produce less errors, which also saves costs.

In addition to the prototype as resulting artifact of this thesis, it also produced the acquired design itself. This design can be used by other researchers as a reference solution for similar problem statements.

3.2 Constraints

In this sectiona solution to the first research question "How can EMF Profiles be expanded to support the creation of modeling languages, which by definition have strong modeling constraints?" is proposed. The implementation of this proposed solution is documented in the next chapter, Chapter 4 "Realization", under section two. Thereafter the implementation is evaluated in Chapter 5 "Evaluation".

Problem Definition Profiles are created to decorate a model object with additional data. The main intentions to do so are on the one hand the clean and separated adaptation of models, on the other hand the re-usability of those features. In both these cases the profiles provide an approach with less probability to make errors. This, however, only applies to the features of the stereotypes themselves, not on errors regarding its application. Potential restrictions on the application of stereotypes can only be checked manually, since there is currently no way to extensively restrict the application. The only restrictions that are possible are the quantity, such as lower and upper bound on how often a stereotype can be applied to the same object, and the class type a stereotype can be applied on. Advanced restrictions based on properties of the object are not possible.

Such tedious manual checks should not be necessary. Not only are they still erroneous, but also the restrictions may not even be available to the users. Profiles are intended to be used by multiple users for multiple uses. Hence the people who created the profile are not necessarily the ones who use them afterwards. Therefore every profile should be self-explanatory. Every restriction should be within the profile, so misuse can be eliminated. This thinking follows the Japanese principle of Poka Yoke [Shi86], which

originated in the production industry. According to this principle it should not even be possible for users to make mistakes. So the goal must be to reduce or prohibit errors by design. The tools we use should not even allow to make errors. In a production environment this could be parts that are shaped to only fit into designated places and nowhere else. In our case it would be stereotypes that can only be applied to objects which entirely fulfill the stereotypes constraints. An important software engineering rule also dictates to avoid or resolve errors in an early stage of development, since it is much cheaper than in later stages. This rule would also be adhered that way.

Requirements A valid approach to solve this issue should therefore have some kind of annotation mechanism for stereotypes, to store the constraints in any fashion. The constraint definition should be available at the time of profile creation and it should not be possible to change it after the profile was deployed. Furthermore at the point of stereotype application next to the checks on bounds and class type, there should be an additional check if the annotated constraints would be true in this context. The stereotype application should then only continue if the constraints are true and otherwise show an error message.

While it may seem reasonable to include the constraint annotation in the stereotype, after a closer look it seemed the wiser choice would be to put it in the extension class. A stereotype can have more than one extension, but also to different classes. Constraints may factor in some class attribute, which are of course class dependent. Some other classes may not even have the same attribute or may be allowed only within a different range. Even if the constraints are the same for different classes, there should at least be the option to write separate constraints on other extensions.

The constraint language itself should be:

- String-based,

  The easiest way to store the constraints would be as a String property. It could be modified and stored without extra editors and with common, already pre-built mechanisms. Also retrieving the String for evaluation is simple.

- commonly known and used and

  Developers should not have to learn a new language just for this task. A widespread constraint language that is also not too hard to learn would be ideal. Thereby also leaving the possibility open to use the constraints interchangeable for other purposes. Chances are also higher that the constraints are already present in this language. They then don't have to be transformed, which would cut down a step and speed up the development even further while also eliminating an error prone task.

- able to process object attributes.

The purpose of these constraints is to restrict the application of stereotypes based on values of the object or other objects within the model. Therefore the constraint language has to be able to read out such attributes and use them for comparisons.

Selecting OCL Based on these criteria an unstructured literature and web research were done, which lead to the decision to select the Object Constraint Language (OCL) as standardized by the OMG [stad]. The reasons to choose OCL are manifold, but in short: OCL is quite perfect regarding the requirements. It is String-based and can easily be edited and saved as a property of Extension. It is well-known and already used in plenty applications including UML. It can also handle objects and their attributes and can even use typing with the command oclAsType. Beside the fulfillment of the above determined criteria, there are also additional pro-points to chose OCL for this sections approach. The language is already used as part of MOF and therefore also in EMF. Ecore models can be annotated with OCL constraints using EAnnotations, to allow a validation of the model in the editor. So a modeler who uses EMF can presumably also already use OCL.

OCL is used to formulate restrictions and queries for certain aspects of a model. It allows to add rules to a system, where the standard expressiveness is not sufficient enough or the text-based OCL string is just easier to handle, manipulate and persist. Based on the standard specifications, OCL expressions do not have side-effects. Thus any executed expression will only return values, such as a boolean value for constraints, but will not change the state of the model. A state can, however, be changed by a program based on the outcome of these expressions.

OCL Usage OCL is widely used, for example in UML, where it is also part of the official UML specification [stag] by OMG. The next largest area of application for OCL is in model transformation languages. The purpose of model transformation is to automatically process a model for different reasons, such as creating another model from it or generating source code. Major languages in this area that use OCL are the Atlas Transformation Language (ATL)[mis] and Query/View/Transformation (QVT) as specified by the OMG[stab]. In these languages OCL is mainly used in the query phase to gather the required model elements. In the actual transformation part it is also used to calculate certain properties based on the queried elements or read out their attributes. OCL can furthermore be used for any MOF-based metamodel, to further restrict or query model elements.

OCL Features According to its specification OCL is a formal language to describe expressions on models. These expressions can be of the following different kind:

- **Constraints:** Restrictions on model elements that need to be upheld in order to return true and therefore be valid.
- **Queries:** Expressions used to filter a set of objects or properties of those objects, based on the rules defined in the query.

While constraints keep the model or system in a valid state, queries are used to collect, calculate or derive values and elements, which will usually then be further processed by the model code. In this thesis the OCL constraint feature is only used for the design approach to restrict the application of stereotypes.

OCL Invariants OCL constraints can further be broken down into six types of constraints, as shown in the following listing.

- **Invariants:** Are restrictions on objects that need to be assured permanently. The only exception is during the time when an object is executing a method.
- **Pre- and Postconditions:** Define conditions that must assert to true before (precondition) or after (postcondition) a certain methods is executed.
- **Initial & Derivation Rules:** Define rules for the initial or derived value of an objects' attribute.
- **Guards:** Defines a guard for state transitions that needs to be true in order for the transition to fire.

From these constraints only invariants are relevant for this thesis. They offer restrictions on objects and can thereby be used to evaluate these restrictions on a model element, that should receive a stereotype application.

OCL Structure The syntax of OCL expressions is designed to be intuitive and not too mathematical, so average modelers without a vast mathematical background can easily use it. Each OCL statement starts by first defining its context. This expresses for which type of model element the statement was intended. An example code can be seen in Listing 3.1, where the statement starts with the *context* keyword, followed by the actual name of the elements' type. Based on this keyword OCL also sets the available environment including reachable paths as well as the attributes on the element.

Every OCL statement is an indirect instance of the class OCLExpression, which is again a subtype of TypedElement. Hence all statements are typed elements and therefore have a return value. A constraint statement for example always has a Boolean return value. Regarding types, OCL supports the use of:

- basic types, such as Integer, Boolean, String etc.
- set-valued types, such as the collections Set(T), Bag(T) etc.
- user-defined types, such as classes, interfaces or enumerations.

For these types there are also some predefined operations available to use. Basic types have quite the same operations available as they would have in a programming language. For integer these are for example summation, subtraction, multiplication or the function *abs()*, which returns the absolute value of an integer. For Strings, operations such as substring or size are available. Collections can use the *exists* operation, *forAll* or *sum*. Objects of user-defined classes can also use their class-specific methods such as *getName*. With all these types and operations OCL offers plenty of possibilities to restrict the stereotype application based on states or property values of the model.

OCL Example As stated before, the most important OCL constraints in the field of modeling are invariants, pre- and post-conditions. The first one is for objects, the other two are for methods. So for this case invariants are needed. An example for the structure of such an invariant would be the following term:

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

Listing 3.1: Example OCL invariant.

The first part, *context Person*, declares the context of the statement. This is the class type of the object on which the constraint is later evaluated. The declaration is important to set the variables and methods available to the object. The keyword *inv* defines the constraint to be an invariant. The last part is the actual invariant statement, a boolean expression which should assert to true. In this statement the before mentioned operations can be used. Additional assertions can be added by using the operators *and*, *or* or *xor*.

OCL Console The Eclipse sub-project OCLinEcore not only provides the functionality to process OCL expressions, but also provides an additional console to test these expressions in the context of a model as shown in Figure 3.1. The console can be selected by opening the standard Console view and choosing *Interactive OCL* in the *Open Console* dialog on the upper right side of the view. By selecting a model element the context of the expression can be set. Then any OCL expression can be evaluated on the object by typing it into the lower half of the window. The result of the expression as well as the expression will be displayed in the container above.

Approach Definition The approach to implement can now be composed using OCL. For the additional String property it is necessary to alter the emfprofile Ecore metamodel. In the class Extension there has to be one EAttribute of Type EString added, the Ecore equivalent of a Java String. This additional property will hold the String for later evaluation. The EAnnotation should be editable and the bounds should go from zero to one. The code will automatically be generated upon reloading the genmodel file and then selecting generate all. The constraints can then be saved to each Extension individually.

Now the only thing that is left, is the evaluation of the constraint at the time of application. This should be implemented in the method isApplicable in the class Stereotype. No metamodel changes are necessary for that, only an adaptation of the already existing

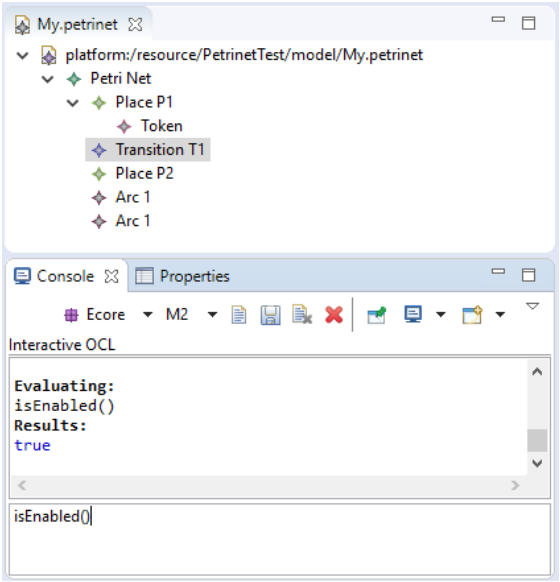

Figure 3.1: OCL Console in Eclipse.

methods. There are four methods with the same name in the class Stereotype, each with different parameters. The method to check the OCL constraint should be called within those methods.

3.3 Runtime Behavior

In this section an approach to resolve the issue of missing runtime behavior for stereotypes in EMF Profiles is proposed. This in turn should lead to an answer on the research question "How can EMF Profiles be expanded to support the adaptation of runtime behavior?". Therefore possible technologies were researched and examined. The findings of this task and the examination are documented in detail in the following subsection called "Analysis of Approaches". Based on the advantages and disadvantages of each way over the others, one was selected and used to develop an approach definition as documented in the second subsection.

As stated in the title this work pursues supporting the Models@Runtime concept in EMF Profiles. As accurately stated by Fouquet et al. [FNM+12] "Models@Runtime aims at taming the complexity of software dynamic adaptation by pushing further the idea of reflection and considering the reflection layer as a first-class modeling space". So the focus of this field is on self adaptive systems, which can reflect on them-self, and react dependent upon it. This field of study became popular not only in recent years and since 2006 has its own workshop, as part of the annual MoDELS conference. As one of the main platforms for model driven engineering, or even the de-facto standard as stated by many papers [FNM+12] [BGS+14], EMF is also an important tool for Models@Runtime

environments. By extending EMF Profiles in a way to support the dynamic runtime functionality mentioned in the introduction, it can contribute to this field within EMF.

To achieve Models@Runtime behavior using EMF Profiles the following additions are necessary. Once a stereotype, which includes changed runtime behavior, is applied to an object, the object should behave different according to the behavior defined in the stereotype. These behaviors or additional operations that execute at runtime should be overwritten. The crucial part is however that only those objects of a certain class should behave different, which have a corresponding stereotype applied to them. Hence the change will most likely interact at an objects' runtime level not at class definition. Furthermore the approach has to be lightweight as EMF Profiles is now, so no changes to the models metamodel are to be made.

3.3.1 Analysis of Approaches

In this subsection each of the considered approaches are detailed. They were analyzed whether or not they are suitable to be used for the runtime behavior. The analyzing included gathering strengths and weaknesses or conditions, that proved the approach to be unfit, if some arose. Then the ones, where breaking conditions were found, were discarded. For the rest a trade off was performed to get a final selection. Four major concepts were analyzed as documented below: inheritance, a few patterns and combination of patterns, reflection and aspects. Finally there is also a short summary of the concepts, important results and the selected technology, to start with defining an approach in the following subsection.

Requirements A valid approach has to fulfill some criteria to be used as solution in this matter. The frameworks and technologies in the next subsections were evaluated according to whether or not the following requirements are fulfilled:

(1) The change of behavior has to be possible at runtime not just at design time.

(2) The change of behavior has to be possible on the object level not on class level. So only an object with the corresponding stereotype should execute the new behavior, but not others of the same class.

(3) An object can have multiple and eventually different stereotypes applied to it. The approach therefore has to support multiple behavior changes on the same object.

(4) The approach also has to maintain the EMF Profiles property of being a lightweight extension. Hence no modifications to the model, its class files or to code that utilizes the model should be necessary.

Inheritance

The first way that came to my mind on how to solve the issue was the use of inheritance [RH04]. The desired behavior is exactly what inheritance enables. Methods of the base

class will be overwritten by the inheriting class. This way the same method can be called, but on another object which executes different behavior.

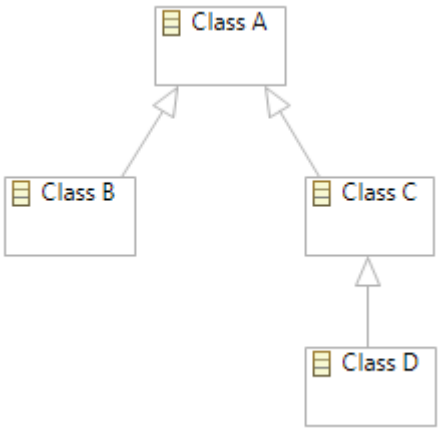

Figure 3.2: Inheritance

Inheritance is one of the main features in object-oriented programming. Figure 3.2 depicts the basic scheme of inheritance. Class A is the main class and one or more classes can inherit methods and variables from it. In this case those are the classes B and C. It then is again possible for another class to inherit from those classes B and C. In the figure, this is class D. Such a behavior is called multi-level inheritance and is also allowed in Java. The only thing that is not allowed in Java regarding inheritance is to inherit from more than one class at once.

Although inheritance delivers the required functionality, in this special case it can not be used, because the changing behavior is needed at runtime and not at design time. The lightweight nature of stereotypes allows to add and remove them during runtime. With inheritance however, the subtypes and therefore applicable classes must be defined beforehand and cannot be changed afterwards. Also inheritance would apply for a whole class and not, as required here, individually for each object. Because of this it is not possible to rely, at least not purely, on inheritance as a solution. So to summarize, inheritance does not comply to the requirements (1) and (2).

Patterns

Since plain inheritance was not useful, software design-patterns were tried next, to overcome its deficiencies. The goal was to use or combine patterns, which can switch methods dynamically at runtime and also do this on the object-level. To achieve this, different patterns as documented in the next few paragraphs were analyzed. The patterns that are analyzed here were all proposed by the so-called Gang of Four [JGVH95] and hence are called GoF-patterns.

Composite Pattern The composite pattern defines a part-whole relationship between classes. For example a file system where there are files and folders. Each folder can then again have files or folders in it, and is a composition of those. The actual files on the other hand represent leafs. The pattern definition is depicted in Figure 3.3. Composite patterns are used when the difference between compositions of objects and actual objects does not matter.

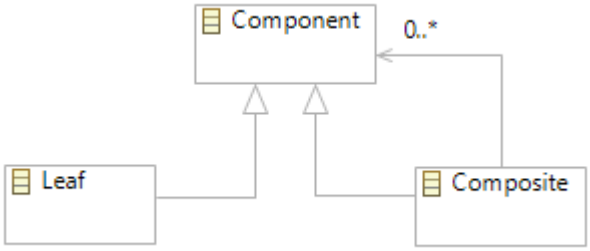

Figure 3.3: Composite Pattern.

This is a valid feature for the approach, since it should not matter if the original method is called or - in case a stereotype is applied - another method is called. The stereotypes code must therefore be a composition of the original type to provide its method. The call to a certain method would go to the abstract class Component. The actual object, where the method is executed, would either be the original object, e. g. the leaf, or a Composite object representing the stereotypes. In this Composite object multiple stereotypes could be represented as children. Using this pattern, the differences between actual implementation and its composition can be hidden. A call to a method would be conducted identically, regardless if the original behavior or a stereotype behavior will be executed.

While this pattern satisfies requirements (1), (2) and probably also (3), it would not satisfy requirement (4). To be able to select the proper object, whose method should be called, it is necessary to intercept the call or change the calling object. For the latter option it would also be necessary to know all possible callers beforehand. Hence both options are not viable and requirement (4) is not fulfilled.

Strategy Pattern The initial purpose of the strategy pattern is to switch underlying algorithms, while always calling the same method. These algorithms can be any method and do not have to be algorithms per se. The structure of the strategy pattern is given in Figure 3.4. The interface IStrategy provides a method for a specific context. All actual implementations of the method are designed towards the common interface, which makes them completely interchangeable.

In this case the strategy pattern could be used to provide an interface for a behavior, regardless if it is the original method or a method supplied by a stereotype. It would

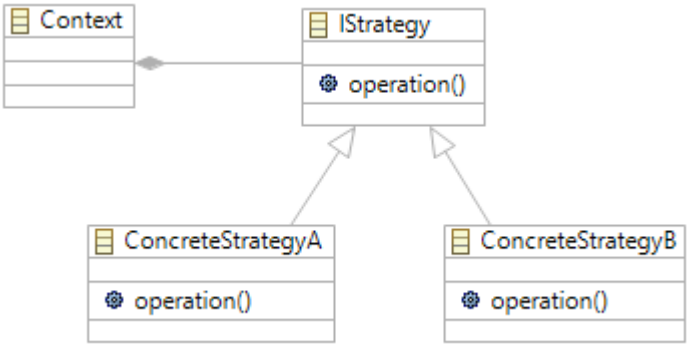

Figure 3.4: Strategy Pattern.

however also be necessary to intercept the call to the original method and redirect it accordingly, which is not possible.

Decorator Pattern This next pattern decorates an existing class with additional variables or methods. According to the Gang of Four it is a flexible way of using the inheritance concept, and ideally what was the goal in this section to achieve with patterns. The schematic concept of decorator patterns is depicted in Figure 3.5. It basically is a combination of the last two examined patterns, the composite and strategy patterns. Therefore it also combines their benefits, namely being able to use inheritance at runtime and also, through the additional layer under the Decorator, switch method and hence the actual code to be run.

Using these individual extensions, objects of a class can be decorated with additional methods of stereotypes. The decorator object includes a pointer to the current component reference. All underlying concrete methods are called through this pointer. To include this decorator pattern and run these methods, however, again calls to the original method have to be redirected.

Observer Pattern The observer pattern, as shown in Figure 3.6, offers a way to react upon certain events. A notification method of the observer object will be called at specific points within the code. The actual notification code that runs in these situations is dependent on the concrete implementation, i.e. ConcreteObserverX. This way arbitrary notification handlers can be invoked at points in the code, which would usually not be accessible.

All the previous patterns revealed that a mechanism to intercept method calls to a model object is necessary in order to achieve the desired behavior. This pattern may be a partial solution to that problem. An event system using this pattern is already part of the EMF framework. Each change event of a model object in EMF triggers a notification to inform observers and listeners. Through this notification system it would be possible to react

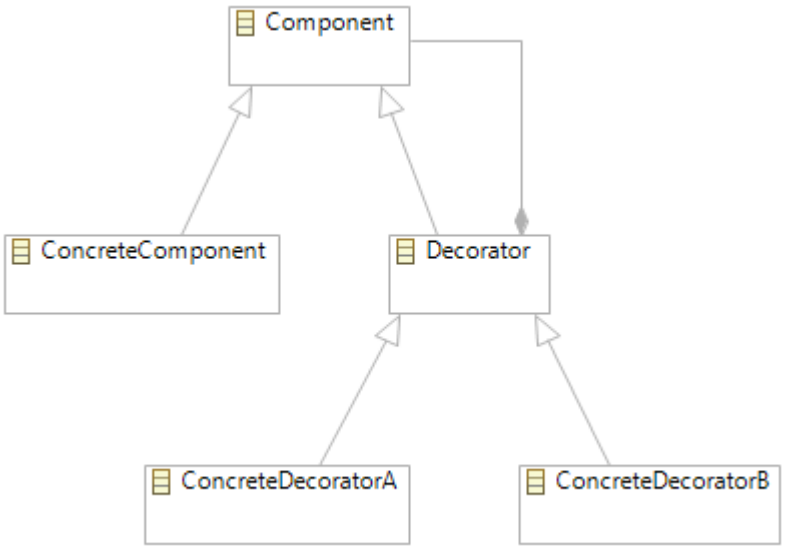

Figure 3.5: Decorator Pattern.

upon or maybe even intercept calls to a method. In these situations it would be possible to run the new stereotype behavior, but most likely only in addition to the original code. Hence this would mean that it is not possible to modify the original behavior.

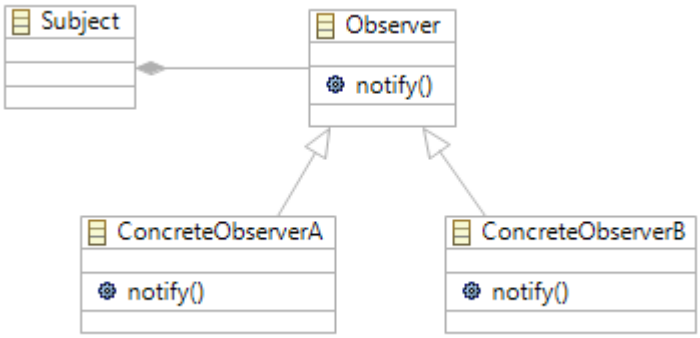

Figure 3.6: Observer Pattern.

Further Patterns & Combinations After evaluating these design patterns it became clear, that the issue of intercepting method calls cannot be resolved using patterns. So the last two patterns that were selected through the search process, the visitor and proxy pattern, were only analyzed rudimentary. Patterns are mostly designed to solve one specific problem, but none of them was designed to solve the problem in this section. Also combinations of the mentioned patterns were tried in order to receive a valid approach.

The combination of patterns is a widely used way to solve design problems, as certain problems often appear hand in hand.

Even combining patterns did not achieve the desired result. The focus in this task was to use the in EMF already existing observer pattern in combination with other patterns, since the observer pattern is an already existing feature, able to intercept a method call and to serve as an entry point for additional behavior. But as explained before, using patterns it is not possible to dynamically change behavior at runtime. In most cases it would be necessary to alter the class or object, which executes the method call on the model object. Therefore it would also be necessary to know all these calling objects beforehand, which is impossible and also not feasible, as it would also mean the plugin code has to be adapted each time other code in the IDE has changed. This setup is immensely contradictory to the desired lightweight, dynamic and reusable approach. Since none of these approaches seemed to overcome this main issue, different, more sophisticated technologies were analyzed.

Reflection

The next analyzed approach was Java reflection [FF04], since patterns could not, even in combination, overcome the issue of the unknown caller object. To solve this, an approach that could intercept a method call and react upon it was needed. Using Java reflection, methods can be called at runtime through variables. A call would therefore not be defined at design time, but dynamically at runtime. A sample generic method call is listed in Listing 3.2. It shows the retrieval of a method as an object, from another object, by providing the methods' name as String. With this method object it is then possible to start various actions, such as invoking the method, as in line two of the sample.

```
1 Method method = eObject.getClass().getMethod("methodName", null);
2 method.invoke(eObject, null);
```

Listing 3.2: Reflection example in Java.

Java reflection offers a way for the program code to inspect and modify itself. More precisely it can inspect interfaces, classes, methods and fields, and is also able to instantiate, invoke or alter them. Java reflection can therefore change the behavior of a class dynamically at runtime. Classes themselves can be loaded at runtime rather than design time, and - using a custom classloader - also be reloaded. Due to these features and possibilities it is therefore also a main framework in the field of Models@Runtime. Java reflection is already included in the Eclipse Java framework, so no additional features have to be installed.

Reflection: Drawbacks Next to these features there are also some drawbacks. The official documentation[1] at Oracle's website lists the following drawbacks:

[1] Java Reflection documentation: `https://docs.oracle.com/javase/tutorial/reflect/`

- Performance Overhead: Reflection tasks are quite resource intensive and will slow down performance of a code section.
- Security Restrictions: Required runtime permissions may not be available to reflection, depending on the security restrictions.
- Exposure of Internals: Reflection bypasses the standard Java accessing rules, defined through the keywords *private* or *protected*, and may therefore result in unexpected behavior.

For the EMF Profiles project, probably only the performance issue is relevant, while the other two are a bit insignificant. Additionally to these technical deficiencies, there is also another issue mentioned in [AGJ+11]. According to the statement, using Java reflection is tricky for programmers and may even lead to a rise in unexpected software errors, since the API does not provide a preview of the results of adaptations.

Reflection: Suitability In addition to these official drawbacks, there are also concrete problems regarding the qualification as valid design approach in this section. While Java reflection allows to react upon certain aspects of classes and objects through introspection, it still cannot intercept a method call without changing the target class. There may be a possibility to generically and automatically change these classes and add a check routine to them, whether or not a certain stereotype is applied. This way the changes would be object-specific, while no manual and permanent class changes are necessary. Hence this would still be a proper design approach for this thesis.

Proxy Next to plain Java reflection, another component of the framework was analyzed regarding this matter. The so-called Proxy class (*java.lang.reflect.Proxy*), based on design principles of the Proxy pattern, can be used to create instances at runtime that are implementing an interface. These instances can also implement multiple interfaces. This would enable the use of proxies to implement the behavior of multiple stereotypes.

Three parts are necessary to create a new proxy instance [2]:

- A classloader designed to load the specified proxy class.
- One or more interfaces, which will be implemented by the dynamic class.
- An invocation handler, which receives all method calls to the class and decides upon its internal logic what to do with them, e.g. invoke another method.

All dynamic proxy classes and their instances implement the Proxy class. Through the invocation handler all incoming method calls can be intercepted and a defined action be

[2] Oracle documentation on the Proxy class:
`https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html`

taken. This would be ideal for the EMF Profiles extension, but would also mean that the class of an object, decorated with a stereotype application, has to implement the proxy interface.

Spring & CGLIB The final approach which was evaluated in the area of reflection, was using the Spring framework together with the CGLIB library[3]. The former is a Java-based framework - and therefore usable in Eclipse - to aid software development. The latter is a software library for Java byte code generation and manipulation included in the Spring framework. The combination of these two frameworks also allows method interception in Java.

Through the Enhancer[4] class dynamic subclasses can be created. This feature is more powerful than the Proxy mechanism described before, since it also allows subclasses to extend other classes and also provides hooks for individual implementations to intercept methods. The interface MethodInterceptor[5] is one of these implementations, already provided by the framework.

With these technologies it would be possible to intercept method calls to a class and run a check regarding the stereotype applications. Dependent on the applications other methods could be invoked, such as the ones provided by those stereotypes. This approach would therefore achieve all four requirements in this section.

Aspects

The final approach considered in this section, was to use aspects in the sense of aspect-oriented programming (AOP). This subsection describes the fourth and only successful approach to dynamically alter the behavior of models based on the applied stereotypes. The first part contains an overview of and introduction into aspect-oriented programming. Therein the basic concept of aspects is presented and how they collaborate with common object-oriented programming. After that the concrete Java implementation called AspectJ as well as a few examples are shown, to provide insight of the possibilities that aspects provide.

Aspect Oriented Programming In contrary to the first impression, due to the name similarity to object oriented programming, it is not an independent new programming concept rather an addition to it. Aspect-oriented programming was designed as solution to so-called cross cutting concerns. The paper Aspect-oriented programming by Kizcales et al. [KLM+97] thoroughly describes the problem of cross-cutting concerns. According to the paper there are often implementations that cannot efficiently be captured with object-oriented methods and end up scattered throughout the code. This leads to code

[3] CGLIB project website: `https://github.com/cglib/cglib`

[4] API documentation for the Enhancer class:
`http://cglib.sourceforge.net/apidocs/net/sf/cglib/Enhancer.html`

[5] API documentation for the MethodInterceptor interface:
`http://cglib.sourceforge.net/apidocs/net/sf/cglib/MethodInterceptor.html`

tangling, which renders the code less readable and intuitive and is therefore difficult for development and maintenance of a software. Also the design principle separation of concerns [HL95] is then not adhered any more.

All these problems arise when a concern has to interact with multiple classes. Probably the best example to show such behavior is logging. The call to a logger has to be made in multiple different classes. One way is to log every exception, so in every catch block there has to be a method call to the logger. These method calls however do not have anything to do with the rest of the class in terms of semantics, and should therefore be in a separate class. Changing or removing those lines is a tedious work. Also, if the logging is only for testing purposes and has to be removed in the production code, the severity of this problem becomes even more apparent.

Aspects can separate these concerns from each other by handling cross-cutting concerns secondary. The aspects content could be to call the logging method each time a method throws an exception, except within the logging class. The actual aspect-oriented changes happen at build time. First the object-oriented classes will be built, just as usual. Then the aspects contents are weaved into the class files as defined. The resulting files are again class files, the aspect-weaver only changes the original class files according to its advices and modifications specified in the aspects.

Using this concept can greatly increase clarity over a software project. It eases maintenance since concerns are separated to modular points in the code. This produces clean code and follows the separation of concerns principle [SSR+05], thus increasing the codes' value.

Using aspects is also a way to overcome an issue that can arise when using subtype polymorphism, namely the circle-ellipse problem or also called square-rectangle problem[6]. They are a textbook example of where object-oriented programming reaches its limitations. The core of these problems is that it is not always possible to define one class as complete subtype of another class. For example a circle is a special kind of ellipse, which has the same values for its x- and y-axes. It therefore should be a subtype of ellipse, but for that it has to inherit all methods of ellipse. This also includes changing the length of the axes independently from each other, which should not be possible for a circle as it would not be a circle anymore. The change of this method to for example also change the other axis to the same length would, however, not be the expected behavior of an ellipse anymore. In Java only this approach to inherit classes is possible. Using aspects with inter-type declarations this problem can be solved.

Due to the flexibility of aspects and their capabilities to alter already existing class files and change their runtime behavior, AOP is often used for Models@Runtime related development. This becomes apparent when looking through the published papers of a Models@Runtime workshop.

[6] Circle-ellipse problem on Wikipedia:
`https://en.wikipedia.org/wiki/Circle-ellipse_problem`

How does it work? Throughout the runtime of a program, there are occurring many so-called join points. These can be the call of a method or its execution, but also an exception that has been thrown and many more. Using aspect-oriented programming these join points can be selected, and reacted upon. When the program reaches this point a piece of code can then be executed, which is defined by the aspect. At the time execution every variable that is available at this point is also available to the piece of code provided by the aspect.

There are three important components to a working aspect:

- **Aspect:** An aspect is comparable to a class in Java. It serves as a container within a package for advices and pointcuts.
- **Pointcut:** A pointcut is a selection of one or multiple join point events within the runtime. Join points can be for example method calls or their actual execution.
- **Advice:** Advice is the actual code to run, it is comparable with a method in Java. By connecting it to a pointcut it can execute at those, by the pointcut specified, points in the runtime.

To define a pointcut also a position is necessary. This further specifies the time of executing the advice in the context of the selected join point. Three positions for that are possible to choose from. Their different behavior is explained below:

- **Before:** Using this position the advice will run before the selected pointcut, for example before a method is executed.
- **Around:** This position lets the advices run instead of the join point. Using it is the only way for an advice to specify a return value. It therefore needs a return type notation prior to the position in the pointcut.
- **After:** Similar, but opposite, to the before operator, this position indicates that the advice will run after the pointcut.

AspectJ So far aspect-oriented programming was explained in general, this was the theoretical basic concept. Now the concrete Java implementation called AspectJ[7] is further described. The complete AspectJ developers guide is available within the documentation section of the project site[8], it thoroughly explains every part of the AspectJ usage and architecture. The basic parts of the language are recapped as follows.

To be able to provide all these aspect-oriented features AspectJ has to reach deep into how the classes are compiled. This is not possible without having full control over the

[7] AspectJ project website: `http://www.eclipse.org/aspectj/`

[8] AspectJ programming-guide:
`https://eclipse.org/aspectj/doc/released/progguide/index.html`

compilation process. Therefore the original Java compiler has to be replaced by the AspectJ compiler, also called ajc in short. It is completely based on the Java compiler, but has the necessary additions to handle aspects as well. So a Java file will be compiled entirely equal by both compilers, provided no aspects are configured to weave it.

AspectJ supports the following times, where the weaving-process of aspects into the class files can happen:

- Compile-time weaving: This is the most common approach to weave classes. For it to work the source code has to be available. Right at compiling, these files will be processed by the AspectJ-compiler, just as they would be by the Java compiler but with the additional inputs by the woven aspects. For certain features of aspects only this weaving-variant can be chosen. For example if features are added to a class that are also used by another class.
- Post-compile weaving: If the classes are already compiled into class files or jar files post-compile weaving is used.
- Load-time weaving: This is the same approach as post-compile weaving. The only difference is that the weaving process of a class is deferred until before it is defined to the Java virtual machine (JVM). It therefore needs to intercept the class loading process.

Another form of weaving is runtime weaving. According to the documentation it is not supported by AspectJ, but there are certain coding patterns to achieve this behavior. In runtime weaving classes are woven dynamically at runtime, while already being defined to the JVM. This means a class may behave as it was defined in its source code, but then the advice may be woven into it, changing its behavior. This behavior may very well be exactly what is needed for the problem stated in this section. As already mentioned it is not supported by AspectJ, but there are patterns to achieve it anyway.

```
1 @Before("call(* org.modelversioning.emfprofile..*(..)) && this(stt)")
2 public void applyStereotype(Stereotype stt) {
3     System.out.println("Applied Stereotype: " + stt);
4 }
```

Listing 3.3: Example AspectJ Annotation.

AspectJ Annotations To be able to define these functionalities, AspectJ introduces new syntax to Java. The pointcuts and advices explained above, have to be declared somewhere and connected to the methods or classes they relate to. For that AspectJ started using additional Java annotations to declare pointcuts and the likes within Java classes. Using the @-operator annotations of classes, variables and methods can be defined, such as @Aspect to define a class as aspect or @After to define the position of an advice. An example code of this usage is displayed in Listing 3.3, which shows a method that is advised by a call-pointcut with the position before. Whenever a method

within the class Stereotype is called, a message is written to the console before the call is executed.

AspectJ Files This approach is still possible, but outdated. The newer approach is to use separate aspect files, which include the necessary information. This way the original classes are not polluted even more with additional code, hence cleaning out the code and honoring the separation of concerns principle. AspectJ files end with the file extension .aj. They are structured just like Java classes with a few modifications. The imports of other packages and classes at the top are the same. Then the actual aspect definition follows, which is similar to the class definiton of a Java class except for the keyword *aspect* rather than *class*.

Within the content of these aspects pointcuts, advices and inter-type declarations can be defined. Pointcuts and advices can either be defined as separate or combined statements, the former is used to allow the reuse of the statements for other combinations. These two are the dynamical parts of an aspect, as they react upon the runtime for their activation. Inter-type declarations on the other hand, are static. They modify the structure of a class by adding or altering variables, creating an inheritance relationship to another class or let the class implement an interface.

Perhaps the most important and most used join points are those related to methods. Especially the call and execution pointcuts are very useful. There are a few differences to consider:

- The pointcut call intercepts a method call to run an advice and offers all included parameters.
- Quite similar the execution pointcut is triggered at the actual execution of the called method.
- Since they are active at different times in the runtime, they also have different access to variables.
- Furthermore additional pointcuts, such as within, behave differently when they are combined in the same pointcut statement with call or execution.
- When using call join points, only direct calls to the method on the object are registered. If however the method would be called through a superclass, the join point would not act upon it.

The examples in Listing 3.4 show code snippets for the two pointcuts. First a call pointcut named notify is declared. It has catches every call to the isApplicable method, which returns a boolean value and is part of Stereotype. The stereotype object it is called upon is also retrieved as parameter. Defining it this way using an identifier, it can later be used and combined with others, thus serving as a reference.

Next, the execution pointcut named addCheck is created. It is activated each time a method named isApplicable, which has a boolean return type and Object parameter, is executed. This time it is not fixed on objects of the class Stereotype. Also through using execution instead of call, each time this method executes is captured. Even if the call was made to a supertype of the object. Finally a piece of advice is combined using the *before* position operator and the beforehand declared notify pointcut. So before the, by notify, captured call to the method is made, a console message will be created using the stereotype object retrieved through the parameters. Altering the stereotypes' or objects' attributes at this time would be able to change the outcome of the subsequent applicability check, as this happens before the method starts to execute.

```
1     // Call
2     pointcut notify(Stereotype s): target(s) && call(boolean isApplicable(Object));
3
4     // Execution
5     pointcut addCheck(): execution(boolean isApplicable(Object));
6
7     // Advice
8     before(Stereotype s) notify(s) {
9         System.out.println("Check␣on␣applicability:␣" + s.getName());
10    }
```

Listing 3.4: Example AspectJ Pointcut.

There are also plenty of other pointcuts, for example handler(SomeException) to catch the execution of an exception or cflow(call(..)) to restrict the valid pointcuts to those which are in the control flow of the inner pointcut. None of these however seem to be relevant to a solution of the problem at hand.

Feasibility towards Solution Using aspects for the approach seems very promising. By using these code fragments within the plugins, they could bring their own runtime functionality with them, altering the runtime behavior dynamically. A new technology would be brought into the project, which is not desirable because of the goal to keep the project lean. But the benefits seem promising and aspects are already widely used within the Models@Runtime development.

To use AspectJ within Eclipse it is necessary to install the AspectJ Development Tools (AJDT) sub-project of Eclipse. It provides the AspectJ functionality for the Eclipse platform. This includes conducting setup tasks to effectively use aspects, such as setting the AspectJ compiler instead of the Java compiler [CC05].

Summary

This concludes the analysis part of this section, which will now be quickly summarized. The main difficulties these analyzed approaches had to overcome, were as follows:

- The behavior has to be modified for an object and not for the whole class.

- This behavior should also only run instead of the original in case a certain stereotype is applied to it. Thus the original behavior also has to be preserved.
- The class itself cannot be altered at design time.
- Classes which call the method are not known, nor can they be modified.

The first approach that was looked into, was inheritance. This approach however can not be used to solve the problem at hand, because of the static definitions at design time, which cannot be altered at runtime. To work around this problem the use of patterns alone and in combination was tried. Again none of the analyzed patterns lead to a solution. They solved the mentioned problem, but since the calling object is not known beforehand and the stereotyped object can also not be altered, these approaches did not provide the required functionality.

The only approaches able to intercept a method call have to have some kind of reflective capabilities. So next, Java reflection was tested to solve the issue. This powerful technology can inspect parts of the program at runtime and also modify it to a certain degree. Classes can be modified so method calls to its objects can be intercepted. This is, however, not trivial. In this area also dynamic proxies and the CGLIB library were evaluated. Both these concepts allow to intercept method calls and run custom code instead. With dynamic proxies, the class has to implement an interface before methods to it can be intercepted. This is not necessary using the CGLIB.

Finally aspect-oriented programming was analyzed, specifically AspectJ. It is able to intercept method calls and also insert code fragments into existing classes. The code to intercept and alter methods and classes can be completely separated from the remaining source code, thus offering a clean and compact solution. In the end AspectJ was used for the approach design, which is documented in the next subsection. This way less code generation and no class reloading is necessary. Furthermore the runtime feature is cleaner separated from other concerns. Lastly AspectJ is not solely available for Eclipse and can therefore be independently used in other frameworks. This is beneficial to the resulting approach defined in this thesis, which should serve as generic solution to similar problems, regardless of their domains.

3.3.2 Approach Definition

This last part contains the actual approach definition for the implementation process into the EMF Profiles project. There are a few separate parts to this approach which are denoted by their own paragraph.

There are three possible options on how to implement the aspects.

- There could be another core EMF Profiles plugin which manages all aspects. This way no projects have to be reconfigured to support the AspectJ nature and dependencies. The downside of this approach, is that the plugin then holds all

aspects of profiles that are developed on that platform. So without manually changing the plugin they can only be distributed together. This however is in contrast to the intended clean, modular and lightweight decoration of existing models.

- The next option is to create a separate aspect project for each profile. This new project will be created with Java and AspectJ natures and the corresponding dependencies and configurations. Thereby each profile has its own aspect project, which would correspond to the modularisation and eliminate the deficit of the core plugin option. Also the original profile plugin project does not have to be modified. This approach would however produce two plugins for just one profile, which is also not feasible as it is good practice to have strong encapsulation.
- The third and last approach, which was chosen to be used out of the three, creates the aspect within the profile project. The project itself has to be adapted to support Java and AspectJ, For that it needs to have a binary and source folder, dependencies to the Java and AspectJ runtime as well as a META-INF configuration file. In this option the aspect is encapsulated with its project. It is also possible to have multiple profiles with aspects within the same project, since this is on the development agenda of the EMF Profiles project.

When a developer creates a profile, there has to be a possibility to attach runtime code to a stereotype as well as a method which should be overwritten. Each profile plugin should then also contain an aspect file to be able to change the behavior of a class. In this aspect file the beforehand attached code should be included and run instead of the selected method. Finally there is also a check needed within the aspect, so the new code only runs if the corresponding stereotype is applied to the object. For this to work, there also has to be some kind of identification to a stereotype. The following main parts that are needed to this end were identified:

Plugin The current EMF Profiles project does not use aspects, also EMF itself is not shipped with any aspect capabilities. So to process aspect features an additional plugin is needed. AspectJ Development Tools (AJDT) provide the Eclipse platform specific tool support for AspectJ. AJDT is an Eclipse bundle and has its own Eclipse project page [Asp]. Since it is specifically designed for Eclipse, it already provides commonly used features in an easy manner. For example there is a button in the context menu to automatically convert the project to an AspectJ project. This method is also available programmatically through an API. After installation and configuration the plugin automatically handles the aspect weaving and also provides context highlighting for the aj source files.

Operation Tool For the graphical editor, where profiles can be created, an additional Operation element needs to be created in the GMF tool section. This includes the GMF changes, graphics and underlying semantics, as well as a new class in the metamodel.

The class Operation should handle the needed properties that are the new Java code, a position (f.e. around) and a selection of methods to choose from. All this should be a subcomponent of stereotype in the editor, so the operation class has to be referenced in the stereotype class.

Aspect Generator After the definition of the profile is finished, a new aspect has to be generated. The generation process for the aspect should run each time the profile is edited and saved. The first generation also has to set up the profile for the use of aspects. Since previous profiles do not have the proper natures, dependencies and configurations, this is a necessary step. Two additional project natures have to be added, the Java nature and the AspectJ nature. Both these natures also require some dependencies and configuration. Current profile projects solely rely on the plugin.xml for configurations and do not have a META-INF. Without this configuration file, however, aspects cannot work. Furthermore the newly created files and folders all have to be included into the build file, so a standard export procedure will create a working plugin with aspect features.

The AspectJ plugin has to have a notation in its META-INF configuration file, into which other bundles it should be weaved. If the operator *call* would be used in the generated aspects, every caller class would have to be known. This is not possible. Therefore only the operator *execution* is a viable choice. Then only the class itself has to be known, as every call to it will eventually end up in an execution of the method within the class. The class that is to be weaved is the extended class in the profile editor. Its method is selected by the developer as joinpoint of the operation in the stereotypes properties view. The rest of the aspects content are the body of the advice and the position for the pointcut, both are again properties of the operation and easy to retrieve.

Callback Routine The content of the generated aspect files needs to include a callback mechanism to check whether or not a specific stereotype is applied to the current model object. Since each profile plugin is a part of it own, this task is not so straight forward. The plugin cannot interfere with method calls to the model class defined in its aspect, if for example EMF Profiles is not even used. Yet it still has to check every method-execution if a stereotype is applied. Stereotypes alone do not have a unique identifier. Also after compilation there is no connection between the aspect code and the profile itself, let alone the stereotype the code came from. Thus an *UID*, consisting of the unique namespace identifier plus the stereotypes name, was proposed to be used. The stereotype name is unique within the profile and the profiles namespace is per definition also unique. With this unique stereotype identifier it is possible to compare stereotypes from a plugin with those in the registry. A simple check routine as method of the profile application registry singleton is sufficient.

All put together In the end the profile developer saves the source code for the new behavior into the stereotype definition. After saving it the project is automatically reconfigured and generates the aspect. The developer can then, as usual, export the project as plugin and use it in any Eclipse instance with AspectJ enabled. In a model

with an object of the class that is extended through the stereotype, the stereotype can be applied, provided all other restrictions are met. Once applied, the new behavior will execute if the joinpoint-method is called on the object. Each other object of the same class in the model will still execute the original method once called.

CHAPTER 4

Realization

The following sections document the implementation of the approaches for both the constraint problem and the runtime problem. Therefore the concepts, frameworks, languages and the likes defined in the last chapter were used to extend the EMF Profiles project. The end-result of this section should be a working prototype of EMF Profiles, including the newly developed extensions to allow runtime behavior and additional constraints. The evaluation of this implementation is conducted in chapter five - Evaluation, where the prototype was used in a case study to acquire metrics for comparability.

The First section covers preconditions that are necessary for the next parts to properly work, such as required plugins. In the second section documents the process of implementing the constraint support. After that, in the third section, the implementation of the runtime behavior for stereotypes is documented. The hereby documented implementation process should allow skilled developers to recreate the prototype and achieve the same result. Therefore important code snippets are described in detail with all their connections and purpose. Furthermore each of the two implemented features are detailed within their own section, to also allow the independent implementation of just one of them.

4.1 Preconditions

This section explains all necessary preconditions to run EMF Profiles in the same state, that was used for this thesis to start. All used versions and requirements are listed as well. This again should allow to recreate the extensions on the same base as used here.

The first and main precondition is the Eclipse Framework. It requires a recent Java SDK to be installed on the system. Eclipse has multiple different packages to download from its website. Each of these packages is just the core Eclipse framework with a bundle of plugins pre-installed to fit a certain area of application. EMF Profiles requires the

Eclipse Modeling Tools package. For this thesis the version Neon.1 was used. This package already includes the most important plugins for Java, Ecore, code generators and transformation. Required plugins which are not in the package, but probably existent on an instance that is already used for modeling, are Xtext (including Xtend), GMF Tooling and OCL in Ecore.

After that all essential frameworks and plugins are installed. So the next part is to import the actual EMF Profiles project from its GitHub project page [EMFa]. There are a few options on how to get the project. To checkout the project with Git or SVN use the link provided on the page. It is also possible to just download the whole package as ZIP file and import it into Eclipse. The important thing to remember is to use the master branch and not develop. The latter is for development purposes and includes not yet finished or untested code. At this point EMF Profiles should work and it is best to create a profile and try it out first to ensure everything is working correct until here.

The best way to this is to use another EMF instance. EMF Profiles is already shipped with the necessary feature and plugin declarations for a working update-site. In the first instance open the project updatesite and select the site.xml. Then select build all to create the contents for the update-site. Once the build process is finished you can use this update-site in the second EMF instance to install EMF Profiles. After a restart of the second instance it should be possible to create a new EMF Profiles project in the workspace. In this development workbench a profile and the model code have to be created. The latter provides the model from which later a concrete model will be created in the runtime workbench. Within this concrete model, the model object can then be extended by EMF Profiles. The model itself can either be installed into the development workbench as plugin or reside as project in the workspace. When the definition of a profile is finished and the model is available the runtime workbench can be started. This is done by right-clicking a project and select Run As / Eclipse Application.

Within the runtime workbench, check whether the two property views work correctly. Those are the Registered EMF Profiles and EMF Profile Applications views. If none of them throw an error the Registered EMF Profiles view should show the beforehand created profile. After this the concrete model can be created and opened in a reflective or GMF-based editor, such as the Sample Reflective Ecore Model Editor. In the EMF Profile Applications view using the Apply Profile button, a new profile application can be created. From there on the stereotypes can be applied to model objects they were designed for. If the application works and is also shown in the view afterwards, the test was successful.

4.2 Constraints

In this section the EMF Profiles framework will be extended by an additional evaluation system for OCL constraints. This prototype will be implemented according to the approach definition and consists of two subsections. Each documenting one of the two tasks planned in the conception. The first will describe the changes to the metamodel,

while the second one will describe the integration of OCL and the implementation of the checking-method. The main feature of this section is to provide a possibility to declare OCL constraints for each extension of a stereotype. The application of a stereotype to an object is then only possible if the declared constraint is valid. Therefore enabling an additional option to restrict the stereotype application and consequently reducing the amount of faulty models.

4.2.1 Metamodel Changes

As a first step the metamodel has to be adapted, because the class Extension needs another String property to store the constraints. These changes have to be made in the core emf-profiles ecore metamodel, not the second metamodel which is called emfprofileapplication. In the Extension class a new child EAnnotation was created. The name for this EAnnotation is constraints and can be edited in the properties view. Its EType property has to be changed to EString with the URI http://www.eclipse.org/emf/2002/Ecore#//EString, not the local referenced one starting with the /resource/ identifier. Other important properties are the lower and upper bound of the attrbite which have to be set to 0 and 1 respectively. To modify the attribute later in the editor, the property changeable has to be set to true. From here on all crucial properties are set and the ecore file can be saved.

Since the ecore file has been changed the genmodel file needs to be reloaded. This can be done by right-clicking the file in the workspace and selecting *Reload*. In the genmodel file the option *Property Multi-line* was also changed to true, to enable a bigger text field for entering the constraints. These OCL strings can get long, and within a single line property this would lead to confusion. After this change the projects can be regenerated by right-clicking the *EMFProfile* package and selecting *Generate All*. This concludes the metamodel changes, the only thing that is left is the evaluation of those Strings at the time of application.

4.2.2 Constraint Evaluation

To be able to include an additional check into the application procedure, where and how the current evaluation takes place had to be searched first. The initial entry point to apply a new Stereotype is the Apply Stereotype menu item in the right-click menu of a model object. The purpose of this menu item is to open a dialog window, showing all applicable stereotypes based on found profiles in the profile registry. A developer can then select one of the stereotypes and apply it to the object. The application registry will then be updated. It triggers a command which is specified in the plugin.xml of the two editor-specific projects for GMF-based and reflective-based editors. Each of them is only responsible for its kind of editor, hence there are two commands. The commands delegate the action to a command handler class named ApplyStereotypeHandler. These registered handlers differ regarding to the specifics of the editor, but within their logic they both call the same method to further process the command.

For that the handlers call the method applyStereotype of the singleton instance of ActiveEditorObserver. The function of this class is, according to the Javadoc written by its author, to manage the mapping of opened editors to the generated id for an opened model in the editor. The method searches through the registry to retrieve all applicable stereotypes for the model object and starts the dialog window with these results, using the method ApplyStereotypeOnEObjectDialog.openApplyStereotypeDialog.

Once an applicable stereotype has been selected out of the list and confirmed using the OK button, the method applyStereotype of class ProfileApplicationDecorator will be executed. This methods only purpose is to call the appropriate method within the ProfileFacade, a class created according to the facade design pattern [GHJV95]. Thus the ProfileFacade serves as limited interface for profile management, which also includes access on each stereotype within a loaded profile. Through three convenience methods named apply and isApplicable, finally the actual isApplicable method on the stereotype is called. If this boolean method returns true the facade will apply the stereotype by creating a new stereotypeApplication and setting its extension and appliedTo attributes.

The content of this method is as follows:

```
1 return isApplicable(eObject)
2     && getApplicableExtensions(eObject, appliedExtensions)
3         .contains(extension);
```

The first line consists of checks, whether or not the stereotype is a metabase or abstract and if there are any extensions on the stereotype, that fit the class of the object. The rest of the statement retrieves a list of all applicable extensions, which still have remaining applications left based on their upper bound and the already existing applications, and checks if it contains the supplied extension. Here the constraint check is just an additional method that needs to return true, so the following line had to be added:

```
1     && checkOCLConstraint(eObject, extension);
```

Within the new checking method the OCL in Ecore environment needs to be initiated and the String constraint converted to an OCL invariant. Since the constraint is only a part of an OCL invariant, the other part, namely the context, has to be set. The extension itself is just a feature of EMF Profiles and most likely will not be the target of the constraint. So it would make the most sense to take the applicant object as default context. This way each constraint does not have to navigate to the object first in order to use the constraint, that may even already be created in this context beforehand for other purposes. The resulting source code can be seen here:

```
1 private boolean checkOCLConstraint(EObject eObject, Extension extension) {
2     if(extension.getConstraints().isEmpty()) {
3         return true;
4     }
5     Constraint invariant = null;
6     OCL ocl = OCL.newInstance();
7     Helper helper = ocl.createOCLHelper();
8     helper.setInstanceContext(eObject);
9     try {
```

```
10          invariant = helper.createInvariant(extension.getConstraints());
11      } catch (ParserException e) {
12          // No valid Constraint
13          return false;
14      }
15      if(invariant == null){
16          // No valid Constraint
17          return false;
18      }
19      Query invariantQuery = ocl.createQuery(invariant);
20      return invariantQuery.check(eObject);
21  }
```

Listing 4.1: Method to evaluate the OCL expression.

The used classes for the evaluation, Constraint, OCL, Helper and Query, are all part of the package *org.eclipse.ocl.ecore*. It is crucial to use those classes and not their correspondent counterparts within the *org.eclipse.ocl* parent package, since these do not provide the necessary features to handle Ecore models. OCLHelper provides convenience methods for building queries. With its method *setInstanceContext* the context of the expression will be set, by using the class type of the supplied object. Next, the method *createInvariant* will create the whole invariant statement for the beforehand set context, adding the *inv* operator and the actual constraint using the supplied String parameter. This invariant can finally be queried and evaluated.

To quickly test the implementation, a small library example was used. A stereotype EBook can be applied to objects of class Book, but only for a specific title. Therefore the simple constraint *title = 'something'* is enough. The stereotype could not be applied unless the title was 'something'.

4.3 Runtime Behavior

In this section the implementation of the second approach regarding the runtime behavior for stereotypes will be documented. The implementation follows the formal approach defined in chapter 3 section 3 and additionally documents the more specific decisions taken in the process as well as important code fragments. The section is furthermore divided into four subsections, each regarding a separate part of the implementation process. The first subsection covers the necessary metamodel changes. The second one covers the changes that were made to the graphical editor in which the profiles are created. The third subsection includes the main component towards runtime behavior, the aspect file generator. Finally a conclusion of the implementation completes the section.

4.3.1 Metamodel Changes

To store additional information about the aspects and the method body, a few changes are to be made in the metamodels. These changes have to be done in the emfprofile.ecore file mainly, but also in emfprofileapplication.ecore. Both of them are within the org.modelversioning.emfprofile package, just as for the constraint implementation. First a

new class called Operation has to be created to hold the method body String, the selected joinpoint and the position of the advice. The position itself is an enumerator holding one of three possible values, thus the EEnum Position has to be created beforehand. Then the operations have to be referenced within the assigned Stereotype class. Finally an additional operation has to be created within ProfileApplication to retrieve stereotype applications. These changes assure all required data can be saved within a profile. The use of this data to change the behavior of models will be described in later subsections.

Position Referencing a position for the advice needs an enumerator class. Within ecore models this class type is called EEnum. The count of literals it contains is fixed to only three options: Before, Around and After. Each of those EEnum Literal also contains a numeric id to identify them. I ordered them by the time they come into effect, regarding the execution of a method as joinpoint: Starting with *Before* as 0, then *Around* with 1 and *After* with 2.

Operation The main part of the necessary changes is the new class Operation. It holds all the relevant information for one method, that should be executed. This includes the new method body that should be executed, the original method it should be executed instead and the aspect position. Since all these data relate to each other, it makes sense to encapsulate them within their own class rather than including them in the Stereotype class. Starting with the modifications I first created a new EClass named Operation, which inherits from EOperation, the Ecore type representing a method. This way Operation inherits already implemented features to represent a method, like parameter or exception handling. Not within those properties is however a way to specifiy the methods body. Therefore an EAttribute child has to be created. I named it body and selected EString as its type. This lets the Operation keep the source code that should later be executed serialized as a String.

The next attribute to create is the position as to where the advice should run relative to the pointcut. For this I choose an EAttribute named position and used my beforehand created enumerator Postition as type, so the developer can then select one of the three predetermined values. EAttributes let you choose a default value, this is the value that is used if no other value is entered in the properties view. For position I chose the Around literal as default, because, compared to After and Before, it seems to be the most desirable as it simply replaces the original method. Furthermore it is the only option to specify a return value. The final addition to the class Operation is the EReference named joinpoint. Its purpose is to save a reference to the original method, which should be the joinpoint serving as pointcut. The references type is set to EOperation, thus allowing the selection of any method that is defined within the metamodel of another class. The lower and upper bounds of each of these three child elements are 0 and 1 respectively. According to common naming conventions the class names were chosen to start with a capital letter, their child elements names start with a lower case letter.

Stereotype The last changes to the metamodel are within the existing Stereotype class. To retrieve the operations and reference them to the stereotypes, two EOperations have to be created. The first one is to retrieve all specified Operations of the stereotype, hence I named it getOperations and set its bounds from 0 to -1. No parameters are necessary for this operation as it does not further filter the returned operations. The second EOperation is used to return a specific Operation by its name. I called it getOperation and set its bound from 0 to 1. Furthermore I created a parameter of type EString as child, which will represent the name of the desired Operation. Both of these EOperations of course have Operation set as their return type.

Profile Application The only changes to the second ecore file, the emfprofileapplication.ecore, are within the class ProfileApplication. Here an additional operation is needed to retrieve stereotype applications for the aspect code as described later. The EOperation is named getStereotypeApplication and has two parameters. One of type EObject named eObject, and one of type EString named stereotypeId. The first one represents the object that the stereotype has been applied to, and the second represents the applications stereotype. As return type I selected StereotypeApplication and set the upper bound to 1.

Generation & Adaptation After all these changes were made to the ecore metamodels, the model files can be regenerated. This is done by reloading the ecore file into the genmodel file and generating all source code, through the right-click menu on the package (e.g. EMFProfile). Prior to the generation a small configuration to the emfprofile generator model has to be made. The body attribute in Operation should have its multi-line property set to true. This enables a separate editor window with multiple lines to edit the attributes String-value later in the profile editors property view. After this change the code can be generated. The two new methods within the Stereotype class were generated only as husks, since no information on the content was provided to the generator. These two have to be implemented manually, so the first action to take in the implementing class StereotypeImpl within the package org.modelversioning.emfprofile.impl, is to add a NOT after the @generated annotations to save them from being overwritten. Their purpose is, as stated above, to retrieve all Operations or respectively one specific Operation from a stereotype. The source code to achieve this is quite simple, as depicted in the following code fragment in Listing 4.2. The second method just uses this method and filters the names according to the supplied String parameter.

```
1 public EList<Operation> getOperations() {
2     EList<Operation> operations = new BasicEList<Operation>();
3     for(EOperation eoperation : getEOperations()){
4         if(eoperation instanceof Operation){
5             operations.add((Operation)eoperation);
6         }
7     }
8     return operations;
9 }
```

Listing 4.2: Manually created method getOperations within StereotypeImpl class.

4.3.2 GMF Editor

Profiles and stereotypes in EMF Profiles are defined through a graphical editor, more specifically a GMF-generated editor. There are two main features to the GMF editor, the design area and the tool palette, which provides the elements to be placed on the design area. The palette of the editor currently includes Stereotypes, Tagged Values and Extensions in the EMF Profiles group, and additionally the full Ecore features in a separate group, such as EClass or Attributes. For the definition of operations a new element has to be created in the EMF Profiles group. As defined in the metamodel, the name in the palette will also be operation.

The GMF-based editor is auto-generated through a gmfgen file, which will then process the input data into the editors Java source files, similar to the genmodel file in EMF. The way to a fully functional editor is through a series of configuration and combination steps of model files which in turn derive into the final gmfgen file. For the additional feature these generation input files have to be adapted. First a tool has to be created in the palette box. Then the graphics, determining how the component will look like in the editor area and where it can be applied to, have to be defined. After that the connection between the tool and its logic, based on the generated model, has to be established. Finally, in the beforehand mentioned gmfgen file, some configurations to fine-tune the java code generator can be done.

The overall picture of the combination and generation process as well as the progress of the current GMF project is depicted in its own eclipse view, the so called GMF-Dashboard. As it can be seen in Figure 4.1, the generation process is aided tremendously by this view. It offers a clear overview on how the process works and furthermore provides links to quickly configure, combine or generate files. The starting point of the generation process is the emfprofile.ecore domain model of the EMF. It just serves as input and does not have to be adapted for GMF. The metamodel was of course adapted by adding additional functionality, but this was done within the EMF domain to obtain the model code and not for GMF per se. After this, the next models that have to be created and or edited are the Tooling Definition Model, Graphical Definition Model, Mapping Model and the Diagram Editor Generation Model in this order. All these files and their changes are described in detail in the following paragraphs.

Tooling Definition Model The modifications in the tooling definition model are only small ones. This model keeps the logical representation of the tool palette. No graphical representation of these entities are adjusted here, nor is the linking to the domain models functionality. To this end the models main node is the *Tool Registry*, followed by the only palette the EMF Profiles project has, since it only uses one editor, the *Palette emfprofilePalette*. This palette includes all used tool groups. For the operation tool the *Tool Group EMF Profiles* has to be extended. Then a new child of type Creation Tool has to be created. There are only two properties to set for it: the title, which I set to *Operation*, and the description, which I set to *Add a new Operation to a stereotype*. The last task to do in the tooling definition is to order the created tool within its group. I

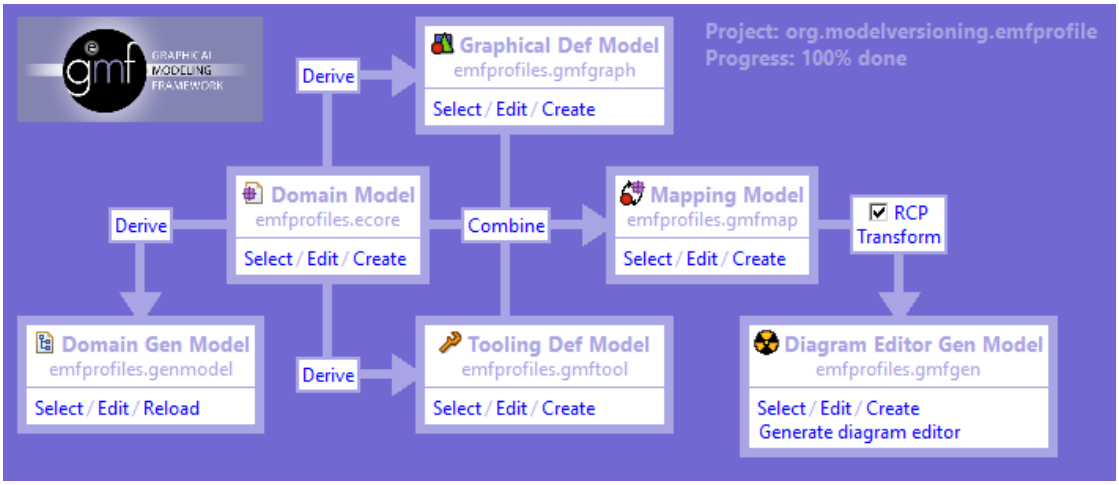

Figure 4.1: GMF-Dashboard view

chose to order it as third item, between the Tagged Value and Extension tools. This position should make it more obvious that the operation is a subcomponent of Stereotype, just as Tagged Value.

Graphical Definition Model In the Graphical Definition Model the graphical representation for the tools has to be created, which will later be combined with the Tooling Definition Model into the Mapping Model. In the editor the new Operation tool should be displayed just like Tagged Values, but in their own compartment. So a second sub-compartment for Stereotype is needed as displayed in Figure 4.2. The lower part containing the two operations is the goal to achieve.

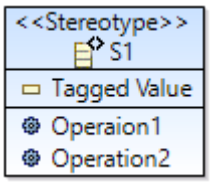

Figure 4.2: Stereotype in EMF Profiles editor.

The main node in the gmfgraph file is the Canvas called emfprofile. Entering it shows one Figure Gallery where the main design parts to be displayed are located. One figure is for example the Stereotype figure, without any tagged values or labels on it, consisting of a rectangle, background color and labels. In this node I created a Figure Descriptor named OperationDisc for the additional label that is used in the editor to change the operations name. Next I created a Label child element named OperationLabel, to assign the text label to the descriptor. This is the only graphical figure necessary and can now be used by other components outside of the gallery. So again in the main node I

created a Compartment named OperationComp and a Labels Diagram Label named Operations. The first one defines the logical compartment area, which will later be placed on the bottom of the stereotype figure, the second one defines the Label area. In the compartments properties I set the figure to the already existing Figure Descriptor StereotypeFigure, so it has the same design as the stereotypes have. For the diagram label I set the figure-property to the beforehand created Figure Descriptor OperationDisc. These are all the adaptation in the graphical model. In the next steps the graphics have to be mapped to the tools and logic.

Mapping Model The mapping model combines the information of the graphical and tooling models. It also maps these model informations to the actual model logic, as to which classes or entities these tools actually represent. Within the main node Mapping of the gmfmap file is an element called Canvas Mapping. It can only be created once and includes the linking to the visual representations and tooling as stated above. This element is of course already created in the EMF Profiles project and does not have to be updated. The only changes to this file are within the *Top Node Reference <eClassifiers:Stereotype/Stereotype>*, which represents the Stereotype class.

In its child element Node Mapping, a new Child Reference and Compartment Mapping have to be created. In the Compartment Mapping the Compartment OperationComp (StereotypeFigure) has to be selected as compartment property. The Child Reference declares the affiliation between the operation and the stereotype. Two Properties have to be set: Under Compartment the just created Compartment Mapping has to be selected and as Containment Feature the type EClass.eOperations:EOperation has to be set. This concludes the mapping as a child to the stereotype, now the actual operation has to be mapped. Therefore the child reference needs to have a child element of type Node Mapping and within that element a Feature Label Mapping. The label mapping references the label to the property of the Operation that should be editable by this label, in this case the name. Hence the Feature to display has to be set to ENamedElement.name:EString. Furthermore the Diagram Label property has to be set to Diagram Label Operations. For Node Mapping three properties have to be set. The first one being the Element property which has to be set to the type Operation, then for Diagram Node again the label Diagram Label Operations has to be selected. Finally the editor tool Creation Tool Operation has to be linked in the Tool property.

Diagram Editor Generation Model With all the mapping between the models completed, the only task left is to adjust the generator details in the generation model. By right-clicking the gmfmap file in the workspace, the command Create generator model... can be used to create or update a gmfgen generator model. After selecting the gmfmap and genmodel files in the wizard, the gmfgen file will be created. No further modifications to the model are required. The actual diagram source code can then be generated by using the command Generate diagram code in the gmfgen files right-click menu. From this point on the editor should show the additional operation tool and its properties. To quickly check everything is working as intended, the runtime instance of

Eclipse can be started which compiles the projects into active plugins. After creating a new EMF Profiles project or opening a diagram file of an existing project, the new editor is shown.

By default a star icon is chosen to decorate the operation element in the tool palette. This icon is rather meaningless and should be changed to a more meaningful one, so developers can quickly recognize it by the icon and associate it with methods. Since it is already known to represent operations, I decided to take a gear icon used in Ecore for the EOperations class. The icon can be found in the org.eclipse.emf.ecore.edit package. After extracting the GIF file out of the bundle, I renamed it to Operation.gif and overwrote the existing default icon file in the folder-path icons/full/obj16 of the package org.modelversioning.emfprofile.edit.

Another adaptation which is a bit more important, is the filter method to retrieve a selection for the joinpoint property in the properties view of the editor. Currently by default all EOperations that are defined within the involved ecore files are shown. So if for example a stereotype EBook has an extension to the imported class Book, then the property would show each operation that is defined within the emfprofile metamodel and the metamodel of the book, such as getTitle. So this would by default show all EMF Profiles operations such as getContainerClass, getEOperation and so on. This behavior will not be intended, but also be confusing because of the sheer mass of results.

To tackle that problem I decided to filter the results using a property descriptor in the operations item provider. Each property entry has its own descriptor method, which is automatically generated by EMF based on its type and the configurations made in the generator model, such as multi-line, editable, sorting etc. The joinpoints property descriptor is located in the class OperationItemProvider, which is part of the project org.modelversioning.emfprofile.edit, within the package provider. Therein the method addJoinpointPropertyDescriptor creates a new ItemPropertyDescriptor object. At this point I altered the code to override the method getChoiceOfValues to filter the results and only return EOperations of classes that are connected to the stereotype by an extension. This was achieved by adding the piece of code in Listing 4.3.

```
1 {
2     @Override
3     public Collection<?> getChoiceOfValues(Object object) {
4         Operation operation = (Operation) object;
5         Stereotype stereotype = (Stereotype) operation.eContainer();
6         List<EOperation> joinpoints = new ArrayList<EOperation>();
7         for(Extension extension : stereotype.getAllExtensions()){
8             joinpoints.addAll(extension.getTarget().getEOperations());
9         }
10        return joinpoints;
11    }
12 }
```

Listing 4.3: Filter method for joinpoint property.

The operation tool is now properly represented in the editor and can be used to extend stereotypes with it. The item as part of a stereotype and also its properties are able to

be saved and loaded through the XMI serialization mechanism of the editor, as the rest of the profile. Until here the creation part of the new runtime feature is finished. The next part is the actual functionality to alter the runtime behavior. This will be done using an aspect generator, as described in the following subsections.

4.3.3 Aspect Generator

Each profile that uses at least one operation in a stereotype has to have an aspect file for it. Therefore the file should be generated when a profile is saved in the editor. This functionality requires a generator mechanism which will be the main part to achieve altered runtime behavior through stereotypes. Necessary tasks are to make the project ready to handle aspects, to create an aspect file with basic functionality, to create pointcuts and advices for operations within this file based on the position type (eg. around or before), to include it in the build process and also register it properly for later use in the plugin.

The whole generator mechanism is built in a way so profiles can still be used without the need for AspectJ, if they don't use operations. As soon as an operation is placed into a stereotype, upon the next save the project will be converted and the aspect will be created. In the following subsections the implementation process of this generator mechanism will be explained.

AspectJ Preconditions

To be able to use AspectJ, a few prerequisites are required. The default Eclipse or the Eclipse Modeling Tools package for that matter, do not support AspectJ out of the box. The AspectJ Development Tools (AJDT) bundle [Asp] has to be installed first, which includes the AspectJ libraries and provides additional features to integrate it into the Eclipse IDE. It also takes care of choosing the AspectJ compiler instead of the default Java compiler. Currently there is only a development version of AJDT available for Eclipse Neon. In fact the last release version was provided for Eclipse 3.7 (Indigo), up until current editions only development version are provided. These versions nevertheless work stable and as intended. For Eclipse Neon I installed the development build for Eclipse 4.6.

After its installation there is also a bit of configuration necessary for the aspects to be woven. Aspect get woven into compiled class files. To do so the AspectJ runtime plugin has to be started before the desired class file is compiled. The runtime should therefore be started as soon as possible. This also shows some limitations of AspectJ as it cannot weave into the earliest core files that are loaded and are essential to even start the compilation and Eclipse. To achieve this the run configurations for an Eclipse Application have to be modified. This can be done in the tab Plug-ins, where for org.eclipse.equinox.weaving.aspectj the parameter Auto-Start has to be set to true and Start Level set to 2. The start level is the indicator in which order the plugins will be loaded. A lower start value means earlier loading. Most plugins are set to default, which

is a variable that can be changed but is most commonly and by default set to four. If more than one plugin have the same start value then their loading is done alphabetically. According to forum posts there is a workaround, which can weave into those plugins anyway, but that is neither trivial nor needed for this thesis.

Class Structure

Now that all preconditions are fulfilled, the actual implementation of the generator can begin. As stated above the generation process should start when a profile is saved. The check whether or not a file will actually be generated - this is the case, if the profile includes an operation element - is also placed inside the generator. Saving a profile in the diagram editor calls the method doSave of the class EMFProfileDiagramEditor, which is in the project org.modelversioning.emfprofile.diagram under sub-package part. This method is called regardless if either the save button in the menu is pressed or the command CTRL + S is used. The class is mainly generated by GMF, so is the doSave method. The generated method was extended with the beforeSave hooking method. In this method I placed the call to the generator and used the profile and the diagrams URI as parameters. If no object of the generator exists, one will be created, otherwise the existing object will be used, hence for one editor there is one aspect generator.

The generator class resides in the main project org.modelversioning.emfprofile within the sub-package org.modelversioning.emfprofileapplication.util and is named ProfileApplicationAspectGenerator. Due to the call and the generator being in different projects, a new dependency to the generator classes project has to be created in the diagram project. This concludes the generators basic class structure. The following section will deal with its content.

Project Setup

The original profile projects in the workspace, were designed to provide only the basic features. Those are the diagram and optionally some icons. However no Java files or anything that has to be compiled were necessary. That is why there is also no support for Java compilation, such as the needed source and binary folders. AspectJ requires the project to have the aspect nature to work, but to include the aspect nature the project already has to be a Java project. The only natures EMF Profiles projects have are the equinox plugin nature and the EMF Profiles natures. These setup tasks have to be performed on the first save if the project does not already have those natures. But also other tasks to update the project have to be performed, like creating a manifest file or updating the build file.

Natures The first thing to do when the generators method to create an aspect is called, is to check the projects natures. If it has no Java nature or no AspectJ nature, they have to be added. With the Java nature also the following required files, folders and configurations are created:

- Java Nature: To identify the project as a Java capable one, the project description has to be updated. This is achieved by adding the Java nature id *JavaCore.NATURE_ID* to the description file.
- Source and Binary Folders: A Java projects needs registered source and binary folders. They are created by the method createSrcAndBin. The bin folder is registered as output location for the project. Also a package for the aspect file is created in the source folder for later use.
- Build Properties: The build configuration file has to be adapted to include the newly created files and folders. These are the input and output locations as well as the manifest file. The method in which these updates are done is called createBuildProperties.
- Classpath: The main part for a Java project to function is the Java runtime environment. Therefore the JRE has to be registered within the classpath as developed in the setClasspath method. Furthermore the source folder and plugin dependencies have to be registered.

Projects can be converted to an AspectJ project by using the command in the right-click menu. This command executes a utility method within the AJDT plugin that is public and can therefore also be called from within source code. In my generator this part is handled by the configureAspectJNature method, with the important part being the call to AJDTUtils.addAspectJNature.

Manifest EMF Profiles projects solely rely on the plugin.xml configuration file, for dependencies and plugin declarations. With the current OSGi environment, this approach is deprecated, as such informations should only be in the manifest.mf file. The purpose of the plugin.xml remains only for the definition of extensions and extension points. Other meta data that it currently includes, such as id, name and version, should all be written to the manifest. This data and the file itself are also required for the aspect feature. The aspect compiler for example uses informations out of the manifest file, such as where the aspect files in the project are located or which bundles they supplement. The method in the generator to create this file is called createManifest. The most important, aspect-related keys within the manifest are:

- Export-Package: Exports the aspect, so it can be considered by the compiler.
- Eclipse-SupplementBundle: Lists the bundles that include classes in which the aspects should be weaved into.
- Require-Bundle: This item includes the main runtime libraries for AspectJ through the bundle org.aspectj.runtime, as well as again all bundles that should be weaved into, so their classes can be used in the body of an aspects advice.

Aspect Content

The only part that is left now, is the generation of the aspect file with its contents. This is entirely done by the method saveAspect, which builds a String that will at the end be written as file into the beforehand created package. The whole method is built as a template where variable parts are inserted at specific points. The first parts needed to be created are the package definition, various import declarations and the default public aspect construct. For each operation an advice will then be generated. Here it is important to differentiate between the position types. While around advices have a type and can return values, before and after do not. An example for the pointcut part of an advice can be seen in Listing 4.4.

```
1     boolean around(petrinet.Transition eObject):
2         target(eObject) && execution(boolean isEnabled()) {
```

Listing 4.4: Generated example pointcut.

I referred the pointcut to the method that was selected as joinpoint in the stereotype operation. Separately I also used its class type with the full package name as context to further restrict the application. By declaring the target object and generically naming it eObject, I also provided the target variable to be used in the body property of the operation. As stated in the approach definition I used the operator execution instead of call, since I do not have control over the calling object. This way only the declared method is altered by the advice and every call to it, regardless which bundle made the call, triggers its execution and therefore the advice code.

Callback Routine Up until now the advices in the generated aspects are invoked each time a pointcut is found, for instance when a method is executed. For the full runtime the new source code from within the body property will run instead of the original one, for every object of the targeted class and in every editor or other code that calls the method. This behaviour is not intended, rather the advice should only be executed in an editor where the profile is loaded and only if the corresponding object has a stereotype application of the respective type. To achieve this another query has to be generated in the advice before the body execution. In this query it should be checked in current editors registry, whether or not the object has a stereotype application of the stereotype that provided the operation.

For this to work it is also necessary to have a mechanism to uniquely identify a stereotype. As a solution to that, the universally unique namespace identifier, which is entered by the profile creator in the property view under the property *Ns URI*, and the stereotype name were combined. The Ns URI is per definition globally unique and the stereotype name is unique within a profile, so this combined String then represents a unique identifier for each stereotype even in different EMF instances. The String is then written into the aspect advice to be available for later comparison.

The EMF Profiles stereotype application registry keeps a separated registry for each supported editor. Therefore the advice has to retain the model ID for the editor in

which the target object resides. The only way this is possible is to retrieve it through a method in a singleton class as entry point, because there is no reference to any EMF Profiles objects in the weaved code at runtime. To achieve this the ActiveEditorObserver singleton class was used, which provides utility methods regarding the currently active editor. It was extended by an additional method, which returns the model ID of the last active editor.

```
1 String stereotypeID = "http://test/petrinet/priority/Priority";
2 String modelId =
3     ActiveEditorObserver.INSTANCE.getModelIdForLastActiveWorkbenchPart();
4 StereotypeApplication stereotypeApplication = ProfileApplicationRegistry.INSTANCE.
      getStereotypeApplication(modelId, eObject, stereotypeID);
5 if(stereotypeApplication != null) {
6     [...body...]
7 } else {
8     return proceed(eObject);
9 }
```

Listing 4.5: Generated example callback routine.

The third line in the advice is the actual check in the registry. With the gathered information another new method in the singleton class ProfileApplicationRegistry can be called. In this method the registry looks for a ProfileApplicationManager object for the provided model ID, and if found searches its profile applications the provided object. If it is found the applied stereotype is compared to the provided id. This comparison method getStereotypeApplication was added earlier in the metamodel to the class ProfileApplication.

Now that it is evident which objects do have a stereotype application, the only thing that is left is to react upn the result. Therefore a simple if query is sufficient to differentiate if the provided body should run or not. While for the positions before and after nothing more has to be done after the check, with around as position this is not the case. The operator around specifies that the advice should run instead of the original method, as it now always runs and checks the applications the original method has to be invoked if no stereotype application has been found. An example of this case can be seen in Listing 4.5, where *return proceed(eObject)* is called.

4.4 Summary

In this chapter the implementation process that followed the approach defined in Chapter 3 was presented. Important source code fragments were included and the structure of the newly developed features was illustrated. The full source code of the prototype can be inspected and downloaded at the GitHub repository [Pro].

- Using OCL statements the application of a stereotype can now be further restricted, through an additional property in the extension. No further requirements are necessary for this restriction mechanism.

- The context of the OCL invariant is already inferred as the target object and the invariant operator is also already hard-coded, so the only part of the statement that has to be provided is the constraint expression.
- The implemented runtime feature lets stereotypes carry an operation including source code. Model objects can be decorated with such a stereotype and have their runtime behavior changed, without changing any source code or metamodel.
- The plugin is encapsulated and can therefore be deployed to any other EMF instance, provided it has the AJDT installed. It also works without AJDT being configured, if no operations are used within profiles.
- The extended EMF Profiles project does use a new AJDT bundle dependency for the generator class. Furthermore the AspectJ runtime library is necessary in the actual profile projects, respectively profile plugins if they are deployed.
- In the editor each save of the profile generates or regenerates an aspect. The first save also sets up the project to support aspects, if it does not already.
- For every method that is used as joinpoint by any stereotype in any profile plugin within the EMF instance, an additional check code is inserted. This code does not alter any runtime behavior unless the object has a stereotype applied to it which uses the runtime feature.
- To check if the weaving process is working correctly, the debug mode can be turned on, as described in the AspectJ problem diagnosis guide. The compiler will then print every weaving action to the console.

CHAPTER 5

Evaluation

This chapter evaluates the beforehand implemented framework as solution to the research questions. The evaluation is divided into five separate sections. It starts with the overall setup for the evaluation process in section one. The case study is based on a petri net metamodel from which a model was generated and extended by a simulator. The purpose of the simulator is to have a runnable place/transition net, according to the specifications by the ISO/IEC-15909 series. Then the second section documents the actual evaluation of the case study. Three petri net extensions are implemented as profiles using the new prototype, and their feasibility regarding the specifications are analyzed.

The third section presents the results of the evaluation and compares them to the research questions to achieve appropriate answers to them. Afterwards in the section critical reflection, benefits and shortcomings of the prototype and the approach as a whole are discussed. Next, open issues that exist in the evaluated approach are listed and discussed. Finally threats to validity are documented, which could distort the results of the evaluation.

5.1 Test Framework Setup

This section covers the necessary pre-work towards the evaluation of the prototype. In order to use the prototype later, a model to work with had to be created first. This model is a basic Petri net with default features and restrictions as documented in the next sub-section. After that a simulator was created on top of the model, as documented in the second sub-section. Then the actual extensions were created as stereotypes and evaluated, but these tasks are documented in later sections.

5.1.1 Petri Net

The evaluation was concluded through the use of Petri nets [Pet77]. There are plenty of other possible modeling languages that could have been taken. The following reasons were decisive for the use of Petri nets: the language is well known, has a simple structure with only a few components in its metamodel and the language, and its models are easy to understand. Furthermore the additional standardized extensions that are available for Petri nets are ideal to be used as stereotypes.

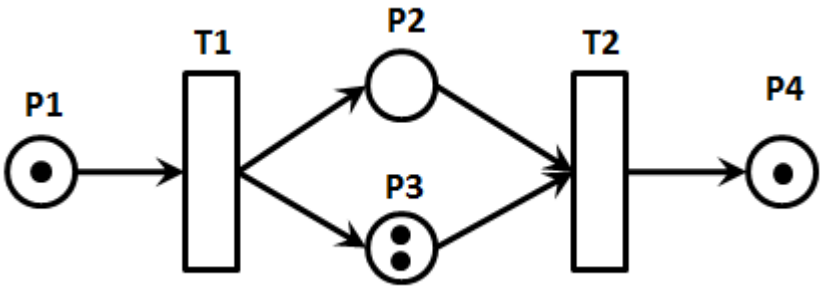

Figure 5.1: Petri net example.

An example graphical representation of a Petri net can be seen in Figure 5.1. The main features of a Petri net are places (P1 - P4), which can have tokens, and transitions (T1 - T2), which serve as decision maker in between places. These elements are connected to each other by arcs, where an arc can only connect a place to a transition or vice versa. It cannot however connect two transitions or places with each other. The purpose of a Petri net is for a token to travel through the net along arcs from one place to another, based on the outcome of transitions. Each of the incoming arcs of a transition has their enabling method. This method returns true if the amount of tokens on the connected place is equal or higher than its weight attribute. This attribute is an integer value, which is by default set to one. If all those incoming methods return true, the transition will fire. Petri nets enable a way to simulate decision system.

Petri Net Metamodel To be able to use Petri nets for the evaluation I built the language as DSML using EMF. The metamodel is depicted in Figure 5.2. After completing the metamodel the full source code was generated. Using the generated editor I created a sample Petri net model to evaluate.

The petri net metamodel consists of six classes within the petrinet EPackage. The first class is **PetriNet**, the top-level class, which represents the whole petri net model itself. Its child elements are two EReferences and one EAttribute. The first one is a reference to nodes with multiplicity from zero to infinity. This collection of nodes includes all places and transitions as explained later. The next reference collection is called arcs and

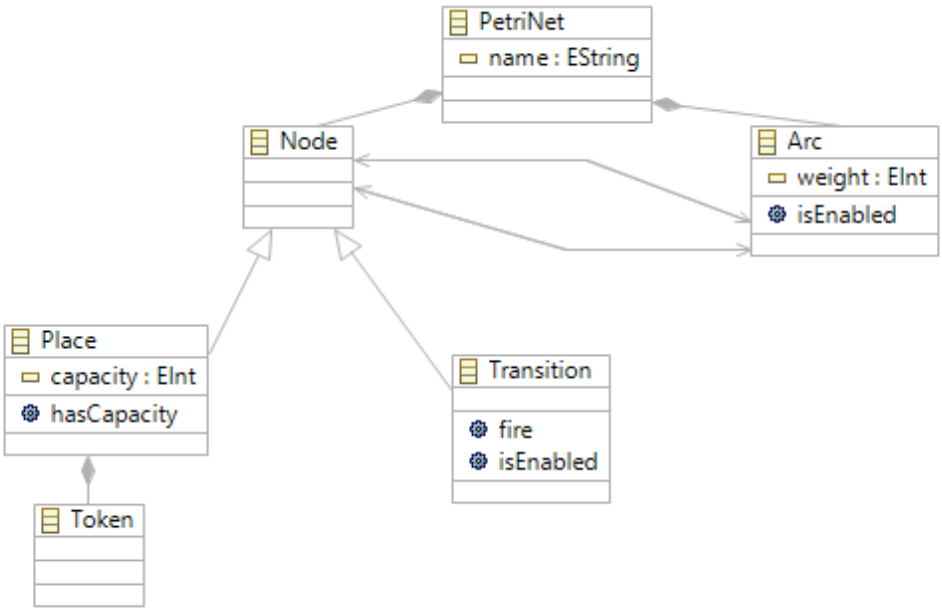

Figure 5.2: Petri net metamodel in ECore

includes all arcs in the petri net. Lastly there is an EAttribute called name with the type EString. This is a single String to keep a name for the petri net for identification.

The class **Node** is an abstract class and can therefore not be instantiated. Its purpose is to serve as an abstraction of Place and Transition classes, which both have connected arcs to it, either as source or target or both. Therefore the class Node has two EReference, one for each the outgoing and the incoming arcs, as children. Their multiplicity has a lower bound of zero and an upper bound of infinite.

The first of the two implementations of Node is the **Transition** class. Additionally to the inherited references of Node it has two EOperations defined. The first operation is fire. It has no return type or parameters and is responsible to fire the transition – that is to collect the designated amount of tokens on each source place and add again the designated amount of tokens to the target places –, but only if the operation isEnabled returns true. IsEnabled checks whether or not the transition itself is enabled. In the basic petri net logic, this is the case if all the places connected to the transition on an incoming arc have at least the amount of tokens, that is defined as weight attribute in the connecting arc.

The second implementation of Node is the class **Place**. Place serves as a storage for tokens and is the only element in the petri net to hold tokens. The maximum amount of tokens it can simultaneously store is defined by the EAttribute capacity of type Integer. Furthermore there is the EReference collection named tokens, which can be empty and has

no upper bound. The third and last child element of Node is the operation hasCapacity. Its purpose is to check whether or not a specified amount, given as integer-parameter, in addition to the count of tokens already on the place, would be lower than the defined capacity and to return a Boolean value accordingly.

For the connections between a place and a transition, the **Arc** class was defined. It includes two EReferences source and target. Both have to reference exactly one object of type Node. An EAttribute weight saves an Integer, which defines the amount of tokens to be transported upon firing the connected Transition. In case the Arc connects a Place to a Transition (incoming Arc), weight is the amount of Tokens required in the Place so the Arc is enabled and the Transition can fire. It is also the amount of Tokens that will be removed from the Place upon firing. In the reverse case, when the Arc connects a Transition to a Place (outgoing Arc), the weight defines the amount of Tokens that will be created in the target Place. These requirement calculations are provided by the operation isEnabled as specified in the metamodel. In the context of a Transition, the sum of incoming Tokens – the combined weights of incoming arcs – does not have to be the same amount as the sum of outgoing Tokens – the combined weights of outgoing arcs. The weights are not dependent on any other weights. In the standard case however, a Transition is a mere deflector where incoming Tokens pass the Transition and flow on to a targeted Place. The amount of incoming and outgoing Tokens is - in this case - exactly the same.

At last there is the **Token** class, which represents a Token that can travel from one place to another place. The places must be connected through Arcs and a Transition. The decisions when and where the Tokens will travel is up to the Transitions. A Token has no further properties in the metamodel. It also has no inherited or other properties in the model editor later on, since it does not inherit any class.

5.1.2 Simulator

Petri net models do have runtime semantics and behavior, therefore a simulator is needed to provide this runtime behavior for the actual Petri net models. For this purpose I created the class SimulatorCommandHandler in the package petrinet.handlers along with FireTransitionCommand in the same package.

FireTransitionCommand This command extends the EMF class CompoundCommand. The purpose of this approach is to have a command combined with multiple sub-commands. The only functionality this command has, is to remove tokens from places that are connected by incoming arcs with the transition. The amount of tokens to remove is defined in the weight property of each of those arcs. After the removal new tokens are created for each place connected by an outgoing arc. Again the amount of tokens to create is defined in the weight property of the arcs. Each of those remove and create actions is a separate sub-command. This way these actions can easily be undone and redone.

SimulatorCommandHandler The simulation handler class provides two commands for the right-click menu, to start the simulation. One action is to execute the fire method of a transition when right-clicking it. The second command is provided on the right-click menu of a PetriNet element. It will not only fire one transition but all transitions in the Petri net for as long as there are enabled transitions. Both these commands also log their actions in the Java console. The core action of these commands is to call the fire method on the transition object. This method then calls its isEnabled method to check whether or not it can fire and does so if true.

5.2 Evaluation

Additionally to the core place/transition Petri net three extensions were chosen, namely priority net, timed Petri net and inhibitor arcs. These extensions were implemented as profiles of EMF Profiles. The evaluation documents to what extent the standardized definitions of each of the extensions could be implemented through the use of stereotypes. The finished Petri net extensions should include every restriction and behaviour, specified in the definitions by Hillah et al. [HKLP12].

These three extensions were chosen because they require constraints to be upheld in order to use them for an object, and also they offer runtime behavior that relies on changes in already existing methods. Furthermore it also still makes sense to use all three extensions together within a model, so this scenario, whether or not they play well together, can also be tested. After their design in the EMF Profiles editor, the profiles can be exported as plugin for the use in a runtime instance to extend models.

5.2.1 Extension: Inhibitor Arc

The first extension to build using the new EMF Profiles prototype is that of inhibitor arcs. Inhibitor arcs are a special kind of arc. In contrary to default arcs which are enabled if a specific amount of tokens are on the connected place, inhibitor arcs are enabled if no tokens are on the place. Hence only incoming arcs can be inhibitor arcs in the context of transitions. According to the specifications the only difference is the logic of the enabling rule.

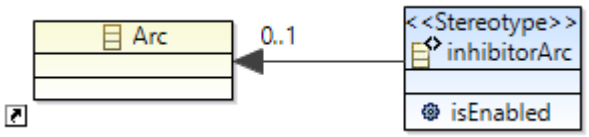

Figure 5.3: Profile - Inhibitor Arc

The inhibitor arc functionality is created as a stereotype as shown in Figure 5.3, according to definition five in Hillah et al. [HKLP12]. The stereotype has to extend the class Arc. As it can be seen there is only one additional operation necessary and no tagged

values. The operations position is set to Around and the selected joinpoint was the method isEnabled of Arc. The bodies content can be seen in Listing 5.1. It overwrites the enabling rule to return true only if no tokens are present in the source Place and otherwise return false. The functionality was fully implemented as stereotype.

```
1 Node node = eObject.getSource();
2 if (node instanceof Place) {
3     if(((Place) node).getTokens().size() == 0)
4     return true;
5 }
6 return false;
```

Listing 5.1: Inhibitor Arc profile: content of body property.

Finally I also created an OCL expression for the property in extension, displayed in Listing 5.4. It restricts the application of a inhibitor arc stereotype if the weight property is not set to zero. This would otherwise be illogic as there are no tokens on the place when it fires, but with a positive weight number there would be some tokens deleted from it. The second and third part of the expression check the source and target to be of type Place and Transition respectively, so the arc is definitely an incoming arc.

```
weight = 0 and source.oclIsTypeOf(Place) and target.oclIsTypeOf(Transition)
```

Listing 5.2: OCL invariant in extension of Inhibitor Arc profile.

5.2.2 Extension: Priority

The Petri net extension Priority dictates an additional - for example integer - value as property of transitions which serves as a priority order. The relevant definitions are number six to eight [HKLP12]. Transitions can then only fire if no other transitions in the net with a higher priority value are enabled. The ordering policy of the value is not fixed to ascending or descending by its specifications, but of course fixed to just one of them within the same Petri net. The value can either be statically as in my example or dynamically returned by a method, which relies on factors within the net. As for the previous extension the only affected method is the enabling rule isEnabled. This time however not within the Arc class but within the Transition itself.

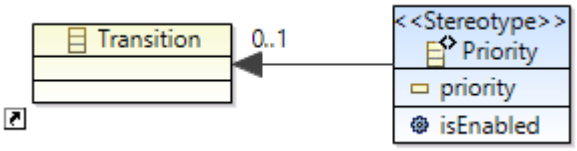

Figure 5.4: Profile - Priority

To achieve this behavior I created a tagged value within the stereotype. I named it priority and set its type to EInt with a default value of 0. Secondly I created a new operation named isEnabled, that uses the transitions isEnabled method as joinpoint. It

overwrites its behavior by selecting Around as position and entering the body shown in Listing 5.3. Within this aspect code I run through all transitions in the Petri net and check whether or not they have a stereotype application of the same type and a tagged value named priority. If so I further check if it is higher than the own priority and if the transition is enabled. So for this code I used the ascending policy where higher value means a higher priority.

```
1  if (eObject.eContainer() instanceof PetriNet) {
2      int priorityA = (int) stereotypeApplication.eGet(
3          stereotypeApplication.getStereotype().getTaggedValue("priority"));
4      for (Node node : ((PetriNet) eObject.eContainer()).getNodes()) {
5          if(node instanceof Transition) {
6              StereotypeApplication sACompare = ProfileApplicationRegistry.INSTANCE.
                   getStereotypeApplication(modelId, node, stereotypeID);
7              int priorityB = sACompare == null ? 0 : (int) sACompare.eGet(sACompare.
                   getStereotype().getTaggedValue("priority"));
8              if(priorityB > priorityA && ((Transition)node).isEnabled()){
9                  return false;
10             }
11         }
12     }
13 }
14 return proceed(eObject);
```

Listing 5.3: Priority profile: content of body property.

For the extension I entered the OCL code in Listing 5.4, to ensure all places connected to an outgoing arc have more or equal capacity than the weight of the arc specifies.

```
out->forAll(
    target.oclAsType(Place).capacity >= weight or target.oclAsType(Place).capacity < 0)
```

Listing 5.4: OCL invariant in extension of Priority profile.

5.2.3 Extension: Time

The third and last extension I implement as stereotype is the time extension. As per definition nine to twelve [HKLP12], transitions in a timed Petri net can only fire after certain time intervals passed since the last firing, simulating a cool down phase. For these intervals there are one or more clocks which continuously count forward. Again there are different policies on how the intervals and clocks relate to the enabling rule of the transitions.

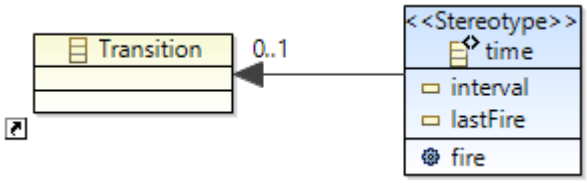

Figure 5.5: Profile - Time

I realized these specifications within the stereotype extending the class Transition and using two tagged values and an operation. The first tagged value is named interval and of type EInt. Its default value is set to zero and it serves as numerical interval given in seconds. The second tagged value is named lastFire and of type EDate. Its default value is a date in the past and it serves as timestamp of the last firing. Finally the operation fire uses the Around position and the fire method of Transition as joinpoint. The body content can be seen in Listing 5.5. With this source code I implemented a local clock for each of the transitions. The interval starts from the last time the transition fired. As long as the upper bound of the interval is not reached it cannot fire again. Finally I update the lastFire value to the new timestamp.

```
1 int interval = (int) stereotypeApplication.eGet(stereotypeApplication.getStereotype().
      getTaggedValue("interval"));
2 Date lastFire = (Date) stereotypeApplication.eGet(stereotypeApplication.getStereotype()
      .getTaggedValue("lastFire"));
3 long ms = interval * 1000 + lastFire.getTime();
4 if(ms > System.currentTimeMillis())
5     return;
6 if(eObject.isEnabled())
7     new FireTransitionCommand(eObject);
8 stereotypeApplication.eSet(stereotypeApplication.getStereotype().getTaggedValue("
      lastFire"), new Date(System.currentTimeMillis()));
```

Listing 5.5: Time profile: content of body property.

In this profile I also created the same OCL expression as for the priority extension.

5.3 Results

Now that all components are implemented they can be evaluated and the results assessed. For this I divided the section into four sub-parts. First I present a quick demonstration of the prototype in action. The next part deals with the constraint support evaluation, how many and how well constraints could be implemented. The third part evaluates the same for the runtime behavior. Lastly I also assessed general metrics of the prototype to make it entirely comparable to other approaches. These finding are documented in the last subsection.

5.3.1 Demo

How the finished prototype looks like in EMF is shown in Figure 5.6. In this example a model element got decorated with a priority profile. The model *My.petrinet* can be seen on the upper left side of the figure. It is a basic Petri net model consisting of three places, which are connected serial by a combination of arcs and transitions. The first transition *T1*, connecting the places *P1* and *P2*, has a priority stereotype applied to it. This can be seen in the lower window in the *EMF Profile Applications* view, provided by EMF Profiles. Based on the setting, this view shows all stereotype applications for the selected element or the whole model.

After selecting this stereotype application, its values can be seen and modified in the default Eclipse *Properties* view. In this example that includes only the *Priority* integer value. The other properties are all read-only properties, which are created for every profile. The operation that is also part of the priority profile is not shown in this view, as it is also read-only and represents the runtime logic that is behind this application.

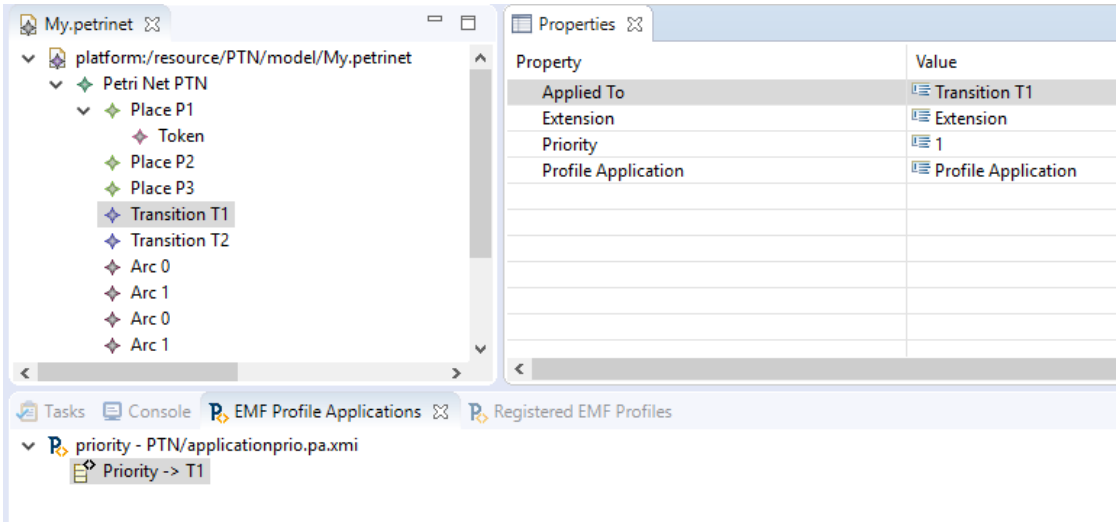

Figure 5.6: Example profile application, shown in EMF.

A graphical representation of a Petri net, which is extended by the beforehand created profiles, is depicted in Figure 5.7. For this figure I used the starting example of a graphical Petri net from Figure 5.1 and added the various stereotype applications as annotations.

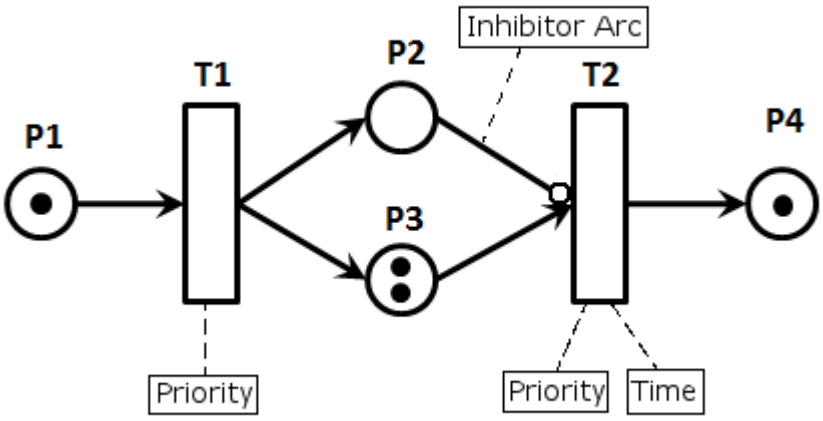

Figure 5.7: Petri net example, including annotated extensions.

While the first transition *T1* only has a priority application, the second one additionally also has a time component applied to it. Hence the second transition is only enabled, if its priority is higher than the first transitions' or if the first transition is not enabled. Furthermore through the time stereotype, the transition is only enabled within certain time-frames. After the transition fired, it requires a cool-down phase in which it cannot be fired again. The third profile type is the inhibitor arc, which is applied to the incoming arc of the transition *T2*. It therefore also affects the transitions' enabled state. Only if there are no tokens on the inhibitor arcs source place it is enabled. The following transition is again dependent on its incoming arcs to be enabled. These stereotypes can now freely be applied onto model elements and directly alter the Petri nets' runtime behavior.

5.3.2 Constraints

The research question for this part was "How can EMF Profiles be expanded to support the creation of modeling languages, which by definition have strong modeling constraints?". These modeling languages with strong constraints are the extensions defined in the previous section according to the specifications in [HKLP12].

- For inhibitor arcs a constraint was necessary to restrict the stereotype from being applied, if the arcs weight is not set to zero.
- Furthermore the inhibitor arc stereotype can also only be applied to incoming arc.
- The third implemented constraint for priority and time stereotypes, was to not allow the application if places on outgoing arcs have a lower capacity than the weight of the connecting arc.

All these required restrictions were fully realized. So to answer the research question: EMF Profiles can be expanded by a constraint within the extension of a stereotype, to restrict the application of stereotypes for each different applicable class. The constraint itself may be an OCL invariant expression.

5.3.3 Runtime Behavior

Regarding the runtime behavior issue the research question was "How can EMF Profiles be expanded to support the adaptation of runtime behavior?". Each of the three picked extensions has their own runtime functionality replacing or extending the original Petri net behavior.

- The inhibitor arc stereotype alters the enabling rule of arcs to invert its behavior, returning true only if there are no tokens on the source place.

- For the priority extension the stereotype alters the enabling rule of a transition object, to check all other transitions in the Petri net for a higher priority. If this is the case and that transition is also enabled, the transition object is not enabled.
- The time extension stereotype alters the fire behavior of a transition. Using the stereotype it is also dependent on a time interval, starting with the last firing of the transition, that has to be surpassed before the next firing.

Again all required functionality was fully implemented. The answer to the research question two is as follows: EMF Profiles can be expanded by using the AspectJ technology. Through the generation of aspects including stereotype-specific advices, it is possible to dynamically add, modify or remove the behavior of single model objects.

5.3.4 General Metrics

Through the implementation of the extensions as stereotypes and using the prototype, I detected the following further possibilities and limitations of the approach:

+ No metamodel changes are necessary to the model.
+ No classes have to be changed in the model.
+ No methods have to be manually changed in the model. The aspects however automatically alter the advised joinpoint methods.
+ Concurrent use of multiple stereotypes on the same model object is possible.
− Orthogonality problems can arise, if multiple stereotypes alter the same method and even more if they also use the same pointcut-position.
− Constraints regarding the application of other stereotypes are not possible. For the inhibitor arc stereotype it would for example make sense to restrict its application only to arcs, where the target transition also has non inhibitor arcs. Otherwise the only incoming arc is constantly true and the simulator would end up in an endless loop. This is however not possible in the prototype.
− One additional plugin - AJDT - plus its configuration is needed for the approach to work.
− An additional technology has to be learned to use the approach. It is not necessary to completely understand AspectJ, but some basics are required, for example what positions are and what they do or how to use the proceed operator. The aspect code is generated entirely. Only imports may have to be manually added, depending on the body properties code, but these are not AspectJ related.

5.4 Critical Reflection

The evaluated prototype has proven to be a solution to both the proposed problems. It has drawbacks and added new issues, but it is still a prototype and most of these issues seem to be solvable by further work. A minor issue for example is the way stereotypes are uniquely identified through the aspect. The technique to add the profiles URI and the stereotypes name to form a unique id is depending on the developers to uphold best practices and not use the same combination in different profiles. One constraint issue remains and an issue regarding the runtime behavior emerged throughout the case study, as discussed in the next section.

Overall however the approach was able to entirely implement the specifications for the selected extensions in the case study. Where are now profiles for each of the extensions, that can be used and removed as desired. The concurrent use of these profiles is also possible in this case, however, as stated above, can be problematic in some cases.

The prototype now offers a powerful profiling mechanism to add information and modify behavior. The behavioral parts effect is, according to the conceptual reference model of aspect oriented programming by Wimmer et al. [WSK+11], an enhancement, but can also be used for replacement and deletion.

5.5 Discussion of Open Issues

There are two quite important open issues and a few lesser important ones. The first is the missing constraining ability for other stereotypes. Also with the new prototype there is no way to restrict a stereotype from being applied based on another stereotype application. The second one is about orthogonality. Multiple stereotypes altering the same method can lead to undesired behavior. There should be a way to avoid such collisions or at least have a documentation what concepts and combinations to avoid.

The remaining issues are just minor ones and mostly about usability. Firstly, the OCL constraint String property is bad to use for more than one constraint, as they all have to be added by an and operator which renders the String unreadable after a few decent expressions. The body property window for an operation is a plain String-window and does not have any syntax marking. There should be a widget that supports Java syntax to ease the development. Finally within the generated aspect there can be unresolved classes that are used in the body property. These can be quickly added in the aspect file using the IDE, but should be generated automatically or at least half-automatically through a window that lets the user choose the appropriate package.

5.6 Threats to Validity

This was my personal critical reflection of the evaluation, there are however to a certain degree still threats to its validity, which need to be addressed. According to Wohlin et al.

[WRH+12], when conducting a case study, there are four aspects of validity to consider, which could bias the outcome of the evaluation. A reflection of these threats to validity in the context of this study is presented in the following paragraphs.

Construct Validity Construct validity examines to what extent the chosen evaluation set-up really measures what was intended by the research questions. Both my research questions are about mechanisms to achieve a desired functionality. For the evaluation set-up I chose to build a Petri net and develop a few of its extensions as EMF Profiles. As a result I analyzed how much of these extensions' specifications could be implemented as a profile. Either a functionality could be implemented or not. These metrics are therefore mostly binary and have a low threat to construct validity. There are no statistical metrics and no metrics that were delivered or evaluated by other people, hence no dependencies or misinterpretations by others are possible.

Internal Validity Internal validity is concerned with the threat that investigated factors are affected by unknown additional factors or to a different extent than it is known. For the actual evaluation of the extensions no relations had to be examined. The general metrics, however, include the point of orthogonality problems. I did not further analyze this problem, but there could exist some unknown factors which also affect the ability to combine two stereotypes on the same element. Other than that no threats to internal validity should exist.

External Validity When conducting case studies, there is no statistical data about the evaluated system available, just one case. External validity deals with the threat that results of the case study evaluation are not relevant to other cases or not generalizable. In this thesis I evaluated the prototype regarding its suitability for Petri net extensions. There are, however, plenty of other and different fields of application for the prototype. So there is a threat that the results may not be as meaningful for other cases. The threat is, however, also not too high, as the evaluated metrics cover characteristics for constraints and runtime behavior. If any other cases would be evaluated with these functionalities, the results would probably be quite similar.

Reliability The reliability threat to this thesis is concerned with the chance that the evaluation results are dependent on the specific researcher, and another researcher could not receive the same results when conducting the evaluation. This is not an issue in this case study. The prototypes' functionality is given as input to this evaluation, and it is evaluated against specifications. As long as I did not misinterpret the specifications another researcher would most likely come to the same conclusions, since there are no estimated values, subjective criteria or other metrics that could be interpreted differently. Only if there is bad or too few documentation on a part of this case study, the result could differ.

CHAPTER 6

Related Work

This chapter presents papers, frameworks and mechanisms that are related to this work. These can either be related by dealing with similar problem definitions or offering similar concepts. To receive these related papers and concepts, a systematic literature review as well as unstructured literature researches were conducted. So, the first section starts by documenting this information gathering process. In the second section the findings are presented and explained as well as compared to the design approach and resulting prototype of this thesis.

6.1 Literature Study

This section covers the information retrieval process for this master thesis. The goal of this exercise was to get an overview of the DSML area, the Models@Runtime concept and the EMF Profiles project. Furthermore implementation approaches or solutions to similar problems as the ones faced in this thesis were searched. Lastly also common metrics to be used as indicators of software development quality were researched. These software metrics were taken into account in the evaluation phase in chapter five.

The chosen approach to obtain these informations was to do a systematic literature review (SLR) as proposed by Biolchini et al. [dABMN+07] and in addition unstructured literature researches. The benefit of the SLR approach is the structured and well defined process used to search through databases and obtain relevant papers, articles and books. This way the search process becomes traceable and repeatable.

An SLR starts by defining research questions for which relevant literature should be found. For my search this were the two problem statements defined in Chapter 1 “Introduction”. Based on these research questions a list of keywords was defined, plus some others based on the greater context of this thesis, such as Models@Runtime. Using these keywords, different search strings were built and used to get results from the following selected

databases: ACM Digital Library[1], DBLP[2], Springer Link[3] and IEEExplore[4]. Next, an analysis matrix was created to collect all publications, sort out duplicates and perform a quick analysis based on the abstract of the papers. Some resulting publications are referenced within this thesis, and a few are covered in detail in the next section.

6.2 Comparison with existing Approaches

This section serves as overview on related work that offers similar approaches or deals with similar problems. They are listed as follows describing their purpose and also comparing them against the two approaches defined and developed in this thesis.

6.2.1 Tool Support

This thesis worked with the EMF Profiles project, which again is based on the DSML tool EMF. EMF is the main tool used for MDE [FNM+12], but as stated in the introduction chapter, there are also other important DSML tools available. According to my research none of them offer any similar profiles concept, let alone my newly developed runtime mechanism. There is one small exception though, the so-called AToMPM [SVM+13] web-based modeling tool, which offers a basic stereotyping concept. This tool is however, based on its usage and its capabilities, not comparable to EMF, MetaEdit+ and the likes. From this point of view EMF Profiles is quite unique, at least in the DSML domain. As already stated in the beginning there are UML profiles too.

6.2.2 UML Profiles

Regarding these UML profiles the first difference to mention, is that they are not available for DSML development. Hence the EMF Profiles project was founded. They do however offer the same profile and stereotype mechanism to add semantics of a specific domain using tagged values. The important part is that they can only add semantics. With the hereby developed prototype it is also possible to alter existing semantics or even remove them, by overwriting them. Furthermore the concept of defining operations in stereotype, to dynamically alter the runtime behavior of the model, is a novelty. Such a mechanism is not available for UML profiles.

Constraints UML profiles also includes a constraint feature for stereotypes. It is however different from the feature implemented in this thesis. While here a constraint mechanism was developed to restrict the application of stereotypes onto model elements, in UML profiles OCL constraints are defined within stereotypes to be, just as tagged values, passed on to the model element. Their purpose is to restrict its attributes based on these expressions.

[1] ACM Digital Library: `http://dl.acm.org/`
[2] DBLP Computer Science Bibliography: `http://dblp.uni-trier.de/`
[3] Springer Link: `https://link.springer.com/`
[4] IEEExplore: `http://ieeexplore.ieee.org/`

Runtime Behavior These were the available concepts for the default UML profiles mechanism. A paper by Tatibouet et al. [TCGT14] proposes the addition of a mechanism to formalize the execution semantics of UML profiles using fUML. This was done as a step to structure semantics of UML profiles, as they are usually not present in a structured, automated processable manner. The approach further allows to directly execute those models using UML profiles. They conducted this by introducing new classes, which override the runtime behavior of their parents with default object-oriented concepts. This approach is however limited to the modification of the standard UML execution behavior. So not as in my approach, which enables overriding the behavior of every class, except those compiled before the AspectJ framework is started.

Orthogonality Another paper by Noyrit et al. [NGTS10] proposes an approach to use multiple DSMLs, implemented as UML profiles, in combination. These are typically not designed to be combined with each other. Therefore, just as it was experienced in this thesis, orthogonality problems can arise when multiple such UML profiles are combined. So, the paper deals with the same problem as this thesis did, but in the context of UML profiles rather than EMF Profiles. Nevertheless these findings can just as much be used for the EMF Profiles domain.

Their proposed approach was to create composite profiles that combine the concepts out of multiple component profiles. They found out that there is always some human intervention necessary in order to create these compositions. Noyrit et al. also stated in their paper that much more research on the matter is necessary, mainly to further increase the degree of automation required to create composites.

6.2.3 Java Annotations, C# Attributes

Regarding the dynamic changing of runtime behavior there are similar constructs in the programming area. Most notably Java annotations and its counterpart C# attributes are relevant. They too alter the behavior of methods at runtime based on the notations in the source code, written above a method. The similarity is even more apparent when considering that AspectJ at first also used Java annotations to express its commands. They are however written directly into a class and therefore not as modular and lightweight as my approach. Profile plugins could, using this approach, not be passed on to another instance and still work.

Java Annotations Typical Java annotations are for example *@Override*, to mark that a method overrides an equally named method of a superclass, and *@Deprecated*, to tag methods that they will not be supported anymore in future releases and should therefore no longer be used. These annotations serve developers as information in the JavaDoc descriptions and also for the compiler to issue warnings. In addition to those default annotations there are also meta-annotations, which annotate other annotations.

One of these is *@Retention*[5]. This meta-annotation allows to set a policy when the subsequent annotation will be discarded. By default this would be after compilation at runtime. However, using the value *RetentionPolicy.RUNTIME* the subsequent annotation is available throughout the runtime. This way enables accessing Java annotations at runtime, using reflection.

The application of Java annotations and the creation of a custom annotation are shown in the following example code in Listing 6.1. The fourth code line shows the application of the custom *Entity* annotation and the provided key-value pair. From line fourteen onwards, the creation of the *Entity* annotation is shown. The *@Target* annotation restricts the application of the following custom annotation to the *TYPE* class. Through *@interface*, a new annotation can be defined. In this case the annotation is named *Entity* and has the key-value pair name as String. After this declaration the custom annotation can be used for classes, interfaces and enumerations, as specified by *ElementType.TYPE*.

```
1 package [...];
2 import javax.persistence.Entity;
3
4 @Entity (name = "Order")
5 public class Order {
6   [...]
7 }
8
9 [...]
10
11 package javax.persistence;
12 import java.lang.annotation.*;
13
14 @Target (ElementType.TYPE)
15 public @interface Entity {
16   String name() default "";
17 }
```

Listing 6.1: Java annotation example. Taken from [BGWK16].

New Features Java annotations first started to annotate methods and classes with meta information. This concept quickly evolved to the comprehensive annotation system it is now. With the current Java 8, Oracle introduced additional new features to make the annotation mechanism even more powerful[6]. Through the new type annotations it is possible to not only change the behavior of methods, but also influence further aspects of the program flow. These new features were achieved by including the checker framework[7] into Java.

The main goal of the checker framework is, according to its project website, to help developers to detect and prevent errors in Java programs. It does so by making as much

[5] Annotation Type Retention API:
https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Retention.html

[6] Oracle blog entry on Java 8 features:
https://blogs.oracle.com/java-platform-group/entry/java_8_s_new_type

[7] Checker Framework project page: https://checkerframework.org/

source code information as possible able to be annotated and therefore machine readable. These informations are currently mostly written into documentations or are only in the heads of developers, who design the program accordingly. But the program is then only designed to avoid these invalid values and states as much as possible, since there are always unknown code paths or other sources of errors. Therefore the program itself can never fully guarantee to uphold these restrictions or behave as intended. By making them machine readable, they can be validated automatically and assured by the compiler, thus producing better code while reducing the need for manual checks as well as the error rate.

Type Annotations Through the checker framework a new kind of annotations was introduced in Java 8, the so-called type annotations. According to the checker framework project page, for each of these new annotations a plugin is added to the compiler, which is responsible to check the constraints affiliated with these annotations. The goal of all these mechanism within the checker framework is to minimize runtime errors. To this end the framework also allows developers to write custom checker plugins. The actual new type annotations are listed below:

- **@NonNull:** This type annotation assures a variable cannot be null. The compile-time checker would otherwise issue a warning, after analyzing all possible paths, whether or not the variable could receive a null value.

- **@ReadOnly:** The read-only annotation prohibits any modification of an object. Again, the compiler will verify this property at compile-time and issue warnings accordingly.

- **@Regex:** Verifies whether or not the content of String variables are valid regular expressions. These verifications only check the correct formatting, not, however, any application of these expressions onto Strings.

- **@Tainted & @Untainted:** Marks variables or objects as incompatible to each other. A tainted variable can for instance not be used as parameter for a method, that is marked as untainted. The variable must first be somehow validated and then marked as untainted, to be used in such a method. This mechanism may be, for example, used to validate untrustworthy user input before processing it. This increases the stability and security of a program, by avoiding attack scenarios like SQL injections. Beside such security concerns, it can of course also be used to just validate input regarding a valid syntax or forbidden symbols.

- **@m:** This last annotations deals with correctly annotating and using measurement units. It ensures that multiple numbers are only used in the proper unit and have been correctly converted.

These pre-defined type annotations basically cover some of the most common causes of defects in software projects. Additionally, through the use of custom annotations and checkers, even more specific, project-related checks can be implemented.

Constraints for Java Annotations An article by Cordoba-Sanchez et al. [CdL16] describes a DSML language named "Ann", to further constrain Java annotations. "Ann" was, just as my prototype in this thesis, implemented as Eclipse plugin for EMF. It was developed because of some shortages in the current framework. For instance the application of the custom annotation created in Listing 6.1 can only be limited to the preset enumeration values, given by the framework. In this example it is restricted to classes, interfaces and enumerations. It is, however, not possible to further restrict these types; For example, to restrict a class based on its internal structure or attributes.

As a solution to this problem, Cordoba-Sanchez et al. proposed the use of the language "Ann". This mechanism is able to validate Java annotations, and can therefore prohibit certain applications that are not intended. A developer can design a new annotation using "Ann" by defining the syntax and static semantics. The necessary Java code is then generated from these designs. The resulting code consists of two files. The first one declares the annotations' syntax definition, while the second one offers an annotation processor, which checks constraints regarding the static semantics. Constraints can either be requirements or prohibitions that need to be fulfilled in order to apply an annotation.

Differences Java annotations and C# attributes can both be used to alter the behavior of programs at runtime based on the notations. This ability is equal to my prototypes'. They are, however, fully integrated into the source code definitions and hence not as lightweight and reusable as my approach. Also, it is possible to directly write complete Java code into EMF Profiles, whereas hereby additional classes have to be written to interpret the annotations. A benefit over my approach is that Java annotations and C# attributes are part of a default installation. My prototype, however, has to be installed additionally and is also dependent on further non-default plugins and frameworks, such as AJDT.

Furthermore, the ability to restrict their application is not as manifold as through the new constraint feature of my prototype. The DSML "Ann" improves this shortcoming for the default Java annotations, but is still limited. Whereas in my prototype OCL invariants can be defined, which can even include the result of a method or a variables value in its expression.

6.2.4 Petri Net Generator

An article by Hillah et al. [HKLP12] proposes an extension framework for Petri nets. Just as my prototype, this extension framework was implemented in EMF. Using this framework, Petri net types can be created through constraints and enabling rules. The

resulting types are implemented in PNML (Petri Net Markup Language)[8], an XML-like markup language designed for Petri nets. The framework can, however, only generate the syntax of these new Petri net types, but no semantics. The resulting extensions are in contrast to profiles by my prototype completely orthogonal.

[8]Petri Net Markup Language website: `http://www.pnml.org`

CHAPTER 7

Conclusion and Future Work

This final chapter reiterates what this thesis' goal was to begin-with, what has been achieved by the end and which future work unfolded. This chapter concludes the thesis.

7.1 Conclusion

The purpose of this thesis was to receive answers to the research questions "How can EMF Profiles be expanded to support the creation of modeling languages, which by definition have strong modeling constraints?" and "How can EMF Profiles be expanded to support the adaptation of runtime behavior?". Answers to these questions are important, because of the high vulnerability to error, due to the missing constraint feature at stereotype application. And secondly, because not all aspects of a xDSML can be stereotyped using the original EMF Profiles. It currently offers no way to stereotype runtime behavior. Most languages or features that should be stereotyped, however, have their own behavior and therefore need to be able to alter existing or add new runtime behavior. Adding these features will ultimately lead to increased re-usability as well as reduced development time and amount of defects. It also assures that developers can use EMF Profiles more often for their projects, and thus helps to further spread the use of DSMLs and code generation.

The research methods used in this thesis followed the general approach in design science. After designing an approach, an artifact - the prototype - was created accordingly. This artifact was then evaluated using a case study. So to answer these research questions, requirements for each of the two problems were worked out. Based on these requirements possible mechanisms and technologies that could be used for a working approach were searched and evaluated. For that purpose the following were analyzed: OCL, inheritance, various software patterns, reflection and AspectJ. The final approach consisted of OCL invariants to offer restrictions for the application of stereotypes, and AspectJ to invoke the stereotypes' runtime behavior.

After implementing a prototype according to this approach it was evaluated, as described in Chapter 5 "Evaluation". A case study was used to assess its capability to resolve the issues. For this case study Petri nets and their extensions were chosen. A default Petri net was implemented as DSML in EMF. Furthermore three Petri net extensions were created as EMF profiles. The case study should show how many of the extensions' specifications could be implemented as EMF profile and how many could not. The result was that every specification of the three selected extensions - inhibitor arcs, priority and timed Petri net - could be implemented. Furthermore additional metrics were collected about the approach, so these strengths and weaknesses can be compared to other approaches as answer to the research questions. The approach proved to be lightweight, as no classes or metamodels had to be modified. Beside the additional AspectJ dependencies that are needed, the approach has one additional noteworthy shortcoming. If multiple stereotypes are applied onto one object and both modify the same method, orthogonality problems can arise. The executed behavior may not be entirely as intended.

The results of this thesis are on the one hand a working prototype of the EMF Profiles project, including means to restrict the application of stereotypes and to enable modifying the runtime behavior of model objects by applying a stereotype. Furthermore the thesis extended the knowledge on how runtime behavior can be influenced in a lightweight fashion, by providing an evaluated design approach for it. This design for a dynamic runtime mechanism may for example be used within the research field of Models@Runtime. All source code created in this thesis is available at GitHub[1].

7.2 Future Work

With the development of this prototype and its evaluation other problems and interesting research topics emerged. Three points unfolded, which are either too far from this thesis' scope or simply too much work to be solved in here, but nevertheless are useful or interesting.

Inter-Stereotype Constraints Regarding the constraint issue that was dealt with in this thesis, there is still a matter open. These OCL constraints do not support the entire spectrum of restrictions that could be needed. While they are able now to cope with the issue of missing constraints regarding the model object and every further object that is accessible from it, they can not be used to restrict the application of a stereotype based on whether or not another stereotype is already applied or has a certain value. While this does not prohibit to build a model element according to its own specifications and restrict it that way, it could however not prevent illicit or undesired interdependencies with other stereotypes. In this case a model would be developed which, in reality, is not possible. Such a constraint mechanism is an interesting feature to look into. A research question for this could be "How can EMF Profiles be expanded to support inter-stereotype

[1] GitHub repository for the prototype: `https://github.com/cmodw/emf-profiles/`

constraints?". To achieve this with EMF Profiles the constraint mechanism would have to be able to retrieve informations from the EMF Profiles application registry.

Orthogonality Another shortcoming of the implemented prototype that came up in the evaluation, was missing orthogonality. If multiple stereotype are applied onto the same model object and alter the same method using operations and aspects, it can lead to undesired and also undefined behavior. For example when one stereotype alters a method using the *before* position and changes a variable, a second stereotype using *around* can overwrite the method, its variables and even its execution entirely. Furthermore when multiple around aspects advice the same method, AspectJ has its own routine on the order in which to apply the advices. For this issue a research question could be "How to avoid collisions of stereotypes, when using operations in EMF Profiles". For that matter it would be interesting to analyze which combinations of certain factors, such as position, type of advice and the overall structure of the method, are problematic.

Regarding this topic, the already mentioned paper by Noyrit et al. [NGTS10] should also be considered. It defines an approach to overcome orthogonality issues using multiple UML profiles. While this approach currently almost always needs human intervention, future work could further automate it. These results are interesting as they probably also serve as a good base for the orthogonality issue of my prototype or may even be entirely adopted.

Reflection Approach As part of the approach definition also the area of Java reflection was analyzed as solution to the runtime problem. As stated in Chapter 3, the decision was made to use AspectJ instead, since it seemed to be the more promising choice. The approaches including the Java reflection Proxy class or the CGLIB library, presented within Chapter 3, were not completely dismissed as it seemed they would basically work. But, because of AspectJ's advantages, it was decided not to take them. So another possible future work would be to simply choose these two approaches instead of AspectJ to develop and evaluate a prototype. The research question would then again be one of my original questions: "How can EMF Profiles be expanded to support the adaptation of runtime behavior?".

Bibliography

[AGJ+11] Uwe Aßmann, Sebastian Götz, Jean-Marc Jézéquel, Brice Morin, and Mario Trapp. A Reference Architecture and Roadmap for Models@run.time Systems. In *Models@run.time - Foundations, Applications, and Roadmaps*, pages 1–18, 2011.

[Asp] AspectJ Development Tools. http://www.eclipse.org/ajdt/. Accessed: 2017-01-02.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[BGS+14] Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. Neo4EMF, A Scalable Persistence Layer for EMF Models. In *Proceedings of the 10th European Conference on Modelling Foundations and Applications, ECMFA 2014, Held as Part of STAF 2014*, pages 230–241. Springer, 2014.

[BGWK16] Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer, and Gerti Kappel. Leveraging annotation-based modeling with Jump. *Software & Systems Modeling*, pages 1–25, 2016.

[CC05] Adrian M. Colyer and Andy Clement. Aspect-oriented programming with AspectJ. *IBM Systems Journal*, 44(2):301–308, 2005.

[CdL16] Irene Córdoba-Sánchez and Juan de Lara. Ann: A domain-specific language for the effective design and validation of Java annotations. *Computer Languages, Systems & Structures*, 45:164–190, 2016.

[dABMN+07] Jorge Calmon de Almeida Biolchini, Paula Gomes Mian, Ana Candida Cruz Natali, Tayana Uchôa Conte, and Guilherme Horta Travassos. Scientific research ontology to support systematic review in software engineering. *Advanced Engineering Informatics*, 21(2):133–151, 2007.

[Ecl] Eclipse Modeling Framework. https://www.eclipse.org/modeling/emf/. Accessed: 2017-01-02.

[EMFa] EMF Profiles Repository on GitHub. https://github.com/planger/emf-profiles. Accessed: 2017-01-02.

[EMFb] EMF Profiles Repository on Google Code. https://code.google.com/archive/a/eclipselabs.org/p/emf-profiles. Accessed: 2017-01-02.

[FF04] Ira R. Forman and Nate Forman. *Java Reflection in Action*. In Action Series. Manning, 2004.

[FNM+12] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems, MODELS 2012*, pages 87–101. Springer, 2012.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.

[GNT+07] Jeff Gray, Sandeep Neema, Juha-Pekka Tolvanen, Aniruddha S. Gokhale, Steven Kelly, and Jonathan Sprinkle. Domain-Specific Modeling. In *Handbook of Dynamic System Modeling*. 2007.

[HKLP12] Lom-Messan Hillah, Fabrice Kordon, Charles Lakos, and Laure Petrucci. Extending PNML Scope: A Framework to Combine Petri Nets Types. *Transactions on Petri Nets and Other Models of Concurrency*, 6:46–70, 2012.

[HL95] Walter L. Hürsch and Cristina Videira Lopes. Separation of Concerns. Technical report, 1995.

[JGVH95] Ralph Johnson, Erich Gamma, John Vlissides, and Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming, ECOOP'97*, pages 220–242. Springer, 1997.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008.

[LWWC11] Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. From UML Profiles to EMF Profiles and Beyond. In *Proceedings of the 49th International Conference on Objects, Models, Components, Patterns, TOOLS 2011*, pages 52–67, 2011.

[LWWC12] Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. EMF Profiles: A Lightweight Extension Approach for EMF Models. *Journal of Object Technology*, 11(1):1–29, 2012.

[mis] ATL Transformation Language, https://eclipse.org/atl/. Accessed: 2017-01-02.

[MVA10] Jeff McAffer, Paul VanderLei, and Simon Archer. *OSGi and Equinox: Creating Highly Modular Java Systems*. Addison-Wesley Professional, 1st edition, 2010.

[NGTS10] Florian Noyrit, Sébastien Gérard, François Terrier, and Bran Selic. Consistent Modeling Using Multiple UML Profiles. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems, MODELS 2010, Part I*, pages 392–406, 2010.

[Pet77] James L. Peterson. Petri Nets. *ACM Computing Surveys*, 9(3):223–252, 1977.

[Pro] Prototype Repository on GitHub. https://github.com/cmodw/emf-profiles/. Accessed: 2017-01-02.

[RH04] Peter Van Roy and Seif Haridi. *Object-Oriented Programming*, pages 489–568. Concepts, Techniques, and Models of Computer Programming. MIT Press, 2004.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[Shi86] Shigeo Shingo. *Zero Quality Control: Source Inspection and the Poka-Yoke System*. Taylor & Francis, 1986.

[SSR+05] Arnor Solberg, Devon M. Simmonds, Raghu Reddy, Sudipto Ghosh, and Robert B. France. Using Aspect Oriented Techniques to Support Separation of Concerns in Model Driven Development. In *29th Annual International Computer Software and Applications Conference, COMPSAC 2005, Volume 1*, pages 121–126, 2005.

[staa] ISO/IEC 19508:2014 - Information technology – Object Management Group Meta Object Facility (MOF) Core.

[stab] OMG Meta Object Facility (MOF) 2.0 - Query/View/Transformation Specification, Version 1.3, 2016, http://www.omg.org/spec/QVT/1.3/.

[stac] OMG Meta Object Facility (MOF) Core Specification, Version 2.5.1, 2016, http://www.omg.org/spec/MOF/2.5.1/.

[stad] OMG Object Constraint Language, Version 2.4, 2014, http://www.omg.org/spec/OCL/2.4/.

[stae] OMG Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.2.1, 2016, http://www.omg.org/spec/FUML/1.2.1/.

[staf] OMG Unified Modeling Language ™(OMG UML), Version 2.4.1, 2011, http://www.omg.org/spec/UML/2.4.1/.

[stag] OMG Unified Modeling Language ™(OMG UML), Version 2.5, 2015, http://www.omg.org/spec/UML/2.5/.

[stah] OMG XML Metadata Interchange (XMI) Specification, Version 2.5.1, 2015, http://www.omg.org/spec/XMI/2.5.1/.

[SVM+13] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. AToMPM: A Web-based Modeling Environment. In *Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition*, pages 21–25, 2013.

[Tan09] Wei Tang. *Meta Object Facility*, pages 1722–1723. Encyclopedia of Database Systems. Springer, 2009.

[TCGT14] Jérémie Tatibouet, Arnaud Cuccuru, Sébastien Gérard, and François Terrier. Formalizing Execution Semantics of UML Profiles with fUML Models. In *Proceedings of the 17th International Conference on Model-Driven Engineering Languages and Systems, MODELS 2014*, pages 133–148, 2014.

[TK05] Juha-Pekka Tolvanen and Steven Kelly. Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences. In *Proceedings of the 9th International Conference on Software Product Lines, SPLC 2005*, pages 198–209, 2005.

[Vog15] Lars Vogel. *Eclipse Rich Client Platform.* vogella series. Lars Vogel, 2015. http://www.vogella.com/tutorials/eclipse.html.

[WRH+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. *Experimentation in Software Engineering.* Springer, 2012.

[WSK+11] Manuel Wimmer, Andrea Schauerhuber, Gerti Kappel, Werner Retschitzegger, Wieland Schwinger, and Elisabeth Kapsammer. A survey on UML-based aspect-oriented design modeling. *ACM Computing Surveys*, 43(4):28:1–28:33, 2011.