Karlsruhe Institute of Technology

An approach to maintainable model transformations with an internal DSL

Master thesis of

Georg Hinkel

At the Department of Informatics
Institute for Program Structures
and Data Organization (IPD)

Reviewer:	Prof. Dr. Ralf Reussner
Second reviewer:	Prof. Dr. Walther Tichy
Advisor:	Dr. Lucia Happe
Second advisor:	Dr. Thomas Goldschmidt

Duration:: 1st May 2013 – 31th October 2013

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

PLACE, DATE

.....

(**Georg Hinkel**)

Abstract

In recent years, model-driven software development (MDSD) has gained popularity among both industry and academia. MDSD aims to generate traditional software artifacts from models. This generation process is realized in multiple steps. Thus, before being transformed to software artifacts, models are transformed into models of other metamodels. Such model transformation is supported by dedicated model transformation languages. In many cases, these are entirely new languages (external domain-specific languages, DSLs) for a more clear and concise representation of abstractions. On the other hand, the tool support is rather poor and the transformation developers hardly know the transformation language.

A possible solution for this problem is to extend the programming language typically used by developers (mostly Java or C#) with the required abstractions. This can be achieved with an internal DSL. Thus, concepts of the host language can easily be reused while still creating the necessary abstractions to ease development of model transformations. Furthermore, the tool support for the host language can be reused for the DSL.

In this master thesis, NMF Transformations is presented, a framework and internal DSL for C#. It equips developers with the ability to specify model transformations in languages like C# without having to give up abstractions known from model transformation standards. Transformation developers get the full tool support provided for C#.

The applicability of NMF Transformations as well as the impact of NMF Transformations to quality attributes of model transformations is evaluated in three case studies. Two of them come from the *Transformation Tool Contests 2013* (TTC). With these case studies, NMF Transformations is compared with other approaches to model transformation. A further case study comes from ABB Corporate Research to demonstrate the advantages of NMF Transformations in an industrial scenario where aspects like testability gain special importance.

Zusammenfassung

In den letzten Jahren hat sich das Konzept der Modellgetriebenen Softwareentwicklung (MDSD) zunehmend verbreitet. MDSD zielt darauf ab, Traditionelle Softwareartefakte aus Modellen zu generieren. Diese Generierung geschieht über mehrere Stufen, sodass Modelle erst mit Hilfe von Modelltransformationen in andere Modelle transformiert werden. Solche Modelltransformationen werden von dedizierten Modelltransformationssprachen unterstützt, die in vielen Fällen komplett eigene Sprachen (externe domänenspezifische Sprachen, DSLs) sind. Dies trägt dazu bei, dass Abstraktionen kompakter und klarer repräsentiert werden können. Auf der anderen Seite ist die Werkzeugunterstützung meist unzureichend und die Transformationssprache kennen die Sprache nicht.
Eine mögliche Lösung für dieses Problem ist, mit Hilfe einer internen DSL die für Modelltransformationen nötigen Abstraktionen direkt in die Sprache einzubinden, mit der Entwickler typischerweise arbeiten (in den meisten Fällen Java oder C#). Dadurch können bestehende Konzepte aus der Hostsprache wiederverwendet werden und gleichzeitig Abstraktionen geschaffen werden, die die Entwicklung von Modelltransformationen erleichtern. Auf der anderen Seite kann die Werkzeugunterstützung der Hostsprache teilweise übernommen werden.
In dieser Masterarbeit wird NMF Transformations vorgestellt, das eine interne DSL für C#, zur Verfügung stellt. Transformationsentwicklern wird damit die Möglichkeit gegeben, Modelltransformationen in C# zu spezifizieren, ohne auf die aus Modelltransformationsstandards bekannten Abstraktionen verzichten zu müssen. Transformationsentwickler werden dabei wie aus C# gewohnt von Visual Studio unterstützt.
Die Anwendbarkeit und die Auswirkungen der Sprachgestaltung von NMF Transformations auf Qualitätsattribute für Modelltransformationen wird evaluiert durch drei Fallstudien. Zwei Fallstudien wurden bei dem *Transformation Tool Contests 2013* (TTC) eingereicht, um NMF Transformations mit anderen Ansätzen vergleichen zu können. Eine weitere Fallstudie von ABB Corporate Research wurde bearbeitet, um den Nutzen von NMF Transformations in einem industriellen Umfeld zu demonstrieren, wo Kriterien wie Testbarkeit besondere Bedeutung haben.

Contents

1. Introduction

1.1. Motivation

In the recent decades, software systems always became more and more complex. The common way to handle this complexity is to use a higher level of abstraction and thereby reducing the complexity. Model-driven software development (MDSD) [SV05] offers ways to write software in a higher level of abstraction. Domain-specific models that abstract from implementation details are used as primary software artifacts and transformed into other artifacts like code or more specific models. To allow an automated transformation, these models have to conform to formally defined structures. This formal definition of models is again a domain and thus, also the structure of models is modeled in a metamodel. The metamodel again has its metamodel, called meta-metamodel. The most common meta-metamodels, MOF and Ecore [MEG+03], are self-descriptive, i.e. it is possible to describe the Ecore metamodel in Ecore. In most cases it is enough to consider three meta-levels. However, in theory there could be arbitrary many. In the Model-Driven-Architecture (MDA, [S+00]) the originals, i.e. the modeled part of the reality, are referred to as the level $M0$. The models are at the level $M1$, the metamodel on $M2$ and the meta-metamodels $M3$.

To get traditional software artifacts, model transformations are used to transform the models into software artifacts. Writing these transformations is usually supported by Model Transformation Languages (MTLs), dedicated domain specific languages for model transformation. However, these transformations can be very complex and are thus usually divided into multiple steps. Instead of using a single transformation to generate the software artifacts out of the models, the input models are transformed into one or many intermediate models. These intermediate models also have to conform to a formally defined metamodel and are then used as inputs to other transformations that eventually create the desired software artifacts.

Most of these software artifacts are not models but have a text format. To create artifacts that conform to such text formats, model-to-text-transformations (M2T-transformations, see [CH03]) are used. In many cases, they are text templates that require the input model to be semantically similar to the artifact that have to be generated. On the other hand, transformations used to transform models into models of a different metamodel are referred to as model-to-model-transformations (M2M-transformations). The target models can be used as inputs to M2T-transformations or further M2M-transformations. While M2T-transformations mostly just print the model into text, M2M-transformations can contain

complex transformation logic. It is also possible that the input and the target domain are the same and a M2M-transformation. In this case, the transformation only changes the input model. These transformations are referred to as in-place transformations.

However, requirements for software always change and so do the software artifacts [Leh74]. If these artifacts happen to be generated, either the generating mechanism or its input has to be changed. Changing the generated software artifacts is considered to be not maintainable [SV05]. Instead, MDSD aims to implement the very most of these changes by altering the models. However, in the long run eventually a changed requirement also forces model transformations to be changed.

MDSD is an approach to limit the impact on these changes as hopefully most of the changes can be implemented by simply changing the models and rerun the transformations. However, there might be changed requirements that cannot be handled by just changing the models. This is especially the case when the understanding of the domain has changed so that either the source or target metamodel changes. Whereas the other model-driven artifacts like metamodels usually change rather seldom (e.g. when the understanding of the domain changes), model transformations as the generating mechanism for most of the traditional software artifacts like code are likely to be subject to change quite often. Thus, the maintenance of the model transformations is crucial for the success of a model-driven software development project. As they more often transform semantics, the maintainability especially of M2M transformations is of great importance to model-driven projects.

On the other hand, M2M-transformations are still quite unpopular. Thus, dedicated transformation languages like QVT (see [Obj11] or section 3.2) still suffer from insufficient tool support. Tool support has been shown to be a major factor for the decision whether a company would adopt MDE [Sta06]. As recent research shows, the tool support still has not matured [MGSF13]. But as the tool support for general purpose solutions is continuously getting better, developers start to expect good tool support that boosts their productivity. This tool support is also rapidly improved by the tool vendors. As far more developers work on the improvements to general purpose language tools, it can be expected that the gap between the tool support for general purpose solutions and MTLs will continue to increase.

In addition, most external DSLs have a limited expressiveness. As model transformation tasks have an intrinsic complexity ([SK03a] assume that MTLs must be Turing-complete), this limited expressiveness often leads to the fact that MTLs copy concepts from general purpose languages.

A possible way to overcome these problems is to use transformation languages that are implemented as internal DSLs [Fow10], i.e. they utilize the language features of an existing language (host language) and are thus able to reuse these concepts. The main benefit from this approach is that the transformation language is supported by tools in the same way as the host language. Developers that are familiar with the host language are likely to familiarize with the transformation language.

The host language for these transformation languages are mostly ones only used by relatively few developers. These languages include for example Ruby utilized by RubyTL [CMT06] or Scala utilized by ScalaMTL [GWS12].

However, an important aspect of maintenance is also the background of the users. In the case of model transformation languages, the users are the developers of model transformations that are mostly unfamiliar with functional or dynamic programming paradigms. Thus, this approach targets at providing the necessary abstractions for model transformations on a mainstream language like C#. Using a mainstream language as their host languages, internal DSLs can also inherit the tool support of the host language. In case of

C#, this tool support includes a fully featured debugger, various profilers and refactoring support.

Such internal model transformation languages already exist. An example that uses a mainstream language as its host language is NMF Transformations which is part of the NMF[1] framework. The purpose of this thesis is to evaluate the effect of this particular internal DSL for writing maintainable model transformations. Despite this, this thesis is also the first document to describe NMF Transformations. As a consequence, various refactorings of NMF Transformations in design and implementation are also a contribution of this master thesis. The main improvement in the design of NMF Transformations is the clear separation of the model transformation framework and the internal DSL built on top of this framework. In this way, NMF Transformations follows the design guidelines for DSLs specified in [Fow10]. Furthermore, the NMF Transformations Core framework satisfies the framework design guidelines from [CA08] and has a high test code coverage. Thus, it is more appropriate for a use in a production environment.

1.2. Structure of this Master Thesis

At first, chapter 2 shows related work. It is an overview given about existing transformation engines with external (dedicated) DSLs (section 2.1). Section 2.2 gives an overview on model transformation engines that originally come from the field of graph transformation. Next, section 2.3 gives an overview of MTLs implemented as internal DSLs much like NMF Transformations before section 2.4 compares the existing approaches with internal and external DSLs. Section 2.5 introduces some existing work about maintainability of model transformations.

Next, chapter 3 introduces some foundations this thesis is built upon. Section 3.1 gives an overview about the approach of model-driven software development. Section 3.2 introduces the transformation standard QVT by the Object Management Group (OMG) as standardized examples of transformation languages. Section 3.3 introduces some C# language features that are referred to in later chapters as the internal DSL of NMF Transformations makes use of them. Section 3.4 introduces the quality attributes of model transformations which are referred to later in the master thesis to evaluate NMF Transformations.

Chapter 4 introduces two example model transformation problems that will be referred to throughout the thesis for various purposes. However, these problems are only toy examples. The first of these examples is about transforming finite state charts to Petri Nets, see section 4.1. The second scenario is about transforming a simple person metamodel into a more sophisticated FamilyRelations metamodel, as in section 4.2.

Chapter 5 discusses how to support the task of model transformation with dedicated languages, so called Model Transformation Languages. This chapter is divided in the first section that discusses the special properties of the model transformation domain, whereas sections 5.2 and 5.3 discuss supporting model transformations with internal or external DSLs before section 5.4 concludes this chapter.

Chapter 6 analyzes the need of a framework or dedicated language for writing model transformations and shows the problems that usually occur when writing model transformations with general purpose C#-code. These problems arise from the cyclic nature of many models (section 6.1) and the need of correspondence links (see section 6.2), from inheritance relations of some model elements (see section 6.3), from rather complex patterns (see section 6.4) or from optimizations over various non-deterministic results (see section

[1] `http://nmf.codeplex.com`

6.5). Furthermore, some problems known from existing model transformation approaches are listed like the need for Higher-Order Transformations (see section 6.6), Transformation composition (section 6.7) and Testing of transformations, see section 6.8. Finally, section 6.9 concludes this chapter.

Chapter 7 introduces NMF Transformations and discusses how it solves the problems from chapter 6 in a way that is natural to C# developers. Section 7.1 first introduces the semantic model behind NMF Transformations. Next, section 7.2 introduces the architecture of NMF Transformations, followed by the stage model in section 7.3. Section 7.4 introduces the NMF Transformations Language (NTL), the internal DSL built on top of NMF Transformations. However, NMF Transformations/NTL is not a solution for each and every problem and thus, section 7.5 explains which problems cannot or were not tackled so far. Section 7.6 concludes the efforts on NMF Transformations.

Chapter 8 discusses how features of NMF Transformations, specifically the language features of NTL impact on the maintainability of the resulting model transformations. To accomplish this, the maintainability of model transformations is broken down into the quality attributes of model transformations from section 3.4 that are reviewed with a focus on maintainability in mind. The sections of this chapter go through these quality attributes and discuss the impacts of the language features and discuss possible needs of further evaluation.

The next three chapters introduce the case studies to address the further evaluation mentioned in chapter 8. The first two of them come from the Transformation Tool Contest 2013 (TTC[2]). The TTC 2013 consisted of three case studies but the third case is more closely related to NMF Optimizations that is described in a separate technical report [Hin13]. The last case study is a case study from the industry, from ABB Corporate Research.

Chapter 9 describes the first TTC case study which has been the Flowgraphs case. The goal of this case was to generate a control flow graph and data flow graph out of Java code model. The case is explained in more detail in section 9.1. Section 9.2 explains the planned evaluation for this case study. Next, section 9.3 explains the NMF solution to this case, before section 9.4 also introduces the other solutions, but in a very brief manner. Section 9.5 gives an idea of how a solution of this case could look like in general purpose code. Section 9.6 reports the results of this case on the TTC workshop. Section 9.7 performs the evaluation of this case study with respect to the evaluation criteria from section 9.2.1 and finally section 9.8 concludes this case study.

Chapter 10 describes the Petri Nets to State Charts case study. The structure of this chapter is the same as the previous chapter except that there is no section on a solution of this case study in general purpose code this time. The rest of the chapter is exactly structured in the same way.

Chapter 11 describes the case study from ABB Corporate Research. It consists of writing an extensible code generator for OPC UA which is a communication standard from the Open Platforms Communications Group[3] (OPC) that is largely used in automation. At first, section 11.1 briefly introduces the Address Space Model of this standard. Next, section 11.2 describes the transformation that is to be created. Same as in the other case studies, section 11.3 explains the planned evaluation for this case study, before section 11.4 describes the solution with NMF Transformations. Section 11.6 performs the validation in regard to the previously defined validation criteria and finally section 11.7 draws conclusions from this case study.

[2] `http://planet-research20.org/TTC/`
[3] `https://www.opcfoundation.org/`

The validation results of the three case studies are summarized in chapter 12. At first, the three case studies for the validation are compared to each other to get insights how to classify their results. This is done in section 12.1. The folowing section each deal with one of the quality attributes defined by van Amstel that needed further validation (as discussed in chapter 8). The understandability is discussed in section 12.2, the modifiability is the subject of section 12.3 before sections 12.4 and 12.5 deal with the consistency and conciseness.

Finally, chapter 13 concludes this thesis and summarizes the results in section 13.1. Section 13.2 shows up assumptions and limitations of this work. Finally, section 13.3 looks forward to future work.

The master thesis also contains some appendix chapters that take a further look at other aspects but have not included in the main part as they cover topics not as closely related to the main contribution.

Chapter A introduces the open-source project NMF and briefly introduces the parts it consists of. Chapter B gives some more implementation details on NMF Transformations. Thus, it can be used as a reference to see the full diagrams when only parts will be presented in chapter 7. Chapter C shows the implementation of the example transformation cases from section 4 with NTL. These implementations are referred to in chapter 7 and thus, one may use this chapter as reference to see the context in which these parts are used originally. Chapter D collects the data from the TTC. This includes the results from the open peer reviews as well as comments and remarks on NMF Transformations that were made on the TTC conference. This chapter is also for reference. Chapter E contains the evaluation sheet for the ABB case study.

A possible way to measure the maintainability of code is the use of metrics as discussed in chapter F. As NMF Transformations is an internal DSL, section F.1 discusses how the metrics that are measured by Visual Studio apply to model transformations created by NMF Transformations. Section F.2 tries to adapt existing metrics for model transformation languages to NMF Transformations. However, these metrics somehow fall out of the range of this master thesis and are thus put into the appendix.

1.3. Contribution

The main contributions of this master thesis are as follows:

- **Refactoring/restructuring of NMF Transformations:** As NMF Transformations is an open-source framework that existed before this master thesis started, the contribution of this master thesis is explained here in a bit more detail. NMF Transformations is a part of NMFwhich is a set of projects aiming to support model-driven software development at the .NET platform (see appendix, chapter A for details). The whole project was initiated in July 2012 by myself. I am also the only one who contributed to this project so far. By the end of 2012, the Transformations project was already able to run the toy example transformations that are also shown in this thesis (see section C). However, this thesis is the first document to describe NMF Transformations. As a consequence, many design flaws were detected during the writing process to describe NMF Transformations. As an example, the clear separation of framework and internal DSL on top of it is an outcome of the master thesis.
- **Validation:** The biggest contribution of this master thesis is the validation of NMF Transformations in the three case studies presented in chapters 9, 10 and 11.

With these case studies, NMF Transformations is validated against other state-of-the-art model transformation languages and tools. The discussion on metrics as in the appendix can also be seen as a minor contribution of this master thesis towards the NMF Transformations approach to maintainable model transformations.

- **Optimization:** Furthermore, to satisfy the optimization problems that are presented in section 6.5, NMF Optimizations was created. Initially, the intention was to implement the optimization framework as a part of NMF Transformations. It turned out that it was easier to create a separate project and integrate both projects with each other, allowing an easier separate usage. However, NMF Optimizations also introduced a lot research questions that would have blown the space limitations of this thesis. Thus, NMF Optimizations is only described in a separate technical report [Hin13].

Furthermore, a contribution is to review the metrics for object-oriented design implemented in Visual Studio for their usage with NMF Transformations. Adaptions from existing metrics for model transformations are discussed. However, this topic is not as closely related to the maintainability evaluation and thus, this contribution can be found in the appendix (chapter F).

1.4. How to read this thesis

Dependent on personal interest and background, only parts of the master thesis might be important to readers of this master thesis.

- **C# developers that need to specify model transformations:** As model-driven software development is quite unpopular in the .NET community, most C# developers may find the foundations on model-driven software development and model transformation with QVT useful (section 3.1 and 3.2). To understand the code examples, reading the chapter 4 might be helpful. These transformations are used in chapter 6 where the problems of model transformations are described and how one would solve these problems with C# code. In the following chapter 7, NMF Transformations is introduced that tries to overcome the problems from chapter 6. Chapter 11 shows how NMF Transformations is applied to create a code generator in an industrial context to replace general purpose solutions. If Visual Studio is used as the IDE, the chapter F might be helpful as it instructs how to read the metrics computed by Visual Studio in scenarios incorporating NMF Transformations.
- **Transformation developers of other model transformation languages:** For transformation developers that want to get an overview on NMF Transformations, the advise is to read chapter 5 to understand why to create a transformation language as an internal DSL. Chapter 6 summarizes the problems in model transformation that NMF Transformations tries to solve. Chapter 7 introduces the approach of NMF Transformations. To understand this chapter, it might be helpful to read about the used language features from C# in section 3.3 first. Next, chapter 8 discusses how these concepts influence the quality attributes for model transformations by van Amstel. With this background, the case studies from the TTC in chapters 9 and 10 might be interesting to see how NMF Transformations compares to other model transformation languages. Further, the ABB case study from chapter 11 may give pointers that allow to extend the scope of model transformations if the model transformation language is not tight to a specific metamodelling foundation. Finally, chapter 12 summarizes the validation results and shows the advantages and disadvantages of NMF Transformations.

- **Developers of transformation languages:** For developers of other transformation languages, the master thesis might be helpful to determine the concepts of NMF Transformationsand how they impact on the maintainability of the resulting model transformations. The most important chapters are here the case studies on the TTC where the influence on the language features on the model transformation quality attributes by van Amstel is discussed. Also the chapters 5 and 6 might be interesting as they describe the problems that may occur in model transformations that any model transformation language must cope with. Next, chapter 7 describes the language features of NMF Transformations. This description might improve the understanding of the TTC case studies. As an internal DSL, NTL uses the language features of C#. The more sophisticated language features are described in section 3.3.

2. Related Work

Model transformations are a very important artifact in model-driven software engineering and as MDE gains acceptance in both research and industry [MFM+09], it is crucial to look at the maintenance. Sendall and Kozaczynski even state model transformations as the "heart and soul of model-driven software development" [SK03a]. Thus, there are a plethora of MTLs available [CH06, SS09].

For some of these model transformation languages, implementations exist for the cases of the TTC and thus, their advantages and disadvantages compared with NTL are analyzed in chapters 9 and 10.

2.1. MTLs with external DSL

External DSLs are entirely new languages. This yields the benefit that developers of the transformation language do not have to respect any constraint for the language. They can design their MTL in a way such that writing a model transformation feels as natural as possible for a transformation developer. However, all the tools that support development in this language also have to be created, many of them from scratch. This includes a parser, a compiler or interpreter, a debugger, a profiler and several features that model editors use to have such as syntax highlighting or code completion. Some of these features like a parser and editors with syntax highlighting can be generated by tools like xText [EV06]. However, others like a debugger usually cannot be generated. Furthermore, building a debugger for declarative transformation languages is often complicated not only because of a complex implementation but mainly because the execution semantics of the transformation language makes it hard or even impossible.[1] Given that none of the MTLs has achieved a great acceptance, this effort is put rather seldom. As a consequence, the debugger support while writing a model transformation is rather poor if there is a debugger present at all. This gets even worse when it comes to more sophisticated tool support like a profiler. NTL as an internal DSL inherits the tool support for C# including rich debugging, refactoring and profiling support.

Examples of external MTLs are the languages of QVT that are presented in the sections 3.2.1 and 3.2.2 in more detail. These languages are standardized by the OMG. However, they lack of good tool support. Furthermore, QVT-O copies many concepts from general purpose languages and QVT-R has the problem that there are hardly implementations for an engine available.

[1] A further discussion of external DSLs for model transformation can be found in section 5.2

2.2. Model Transformation as Graph Transformation

Models can also be seen as object graphs. Thus, graph transformation languages also attracted quite a lot of interest transforming models in the context of model-driven engineering. A prominent example is GrGen.NET [GK08] that has won multiple awards at the Transformation Tool Contest 2011[2]. GrGen.NET originally came from compiler construction but the nature of a generic graph rewriting tool also allows its usage as a model transformation engine [GDG08]. Graph transformation languages often share the disadvantages of external DSLs as they can be seen as graphical languages. However, some concepts like simple (e.g. string) attributes and object inheritance do not transfer well to graph transformation.

An important flavor of model transformation by graph transformation is Triple Graph Grammars (TGGs). Originally proposed in the nineties [Sch95], TGGs serve to transform a source graph into a target graph using graph transformation rules that consist of a graph triple. The left hand side (LHS) graph specifies patterns that the transformation engine looks for during a transformation processing and replaces elements matching to the graph elements on the right hand side (RHS) where elements of the LHS and the RHS have a certain correspondence conforming to the correspondence graph. These graphs can be specified in a single graph, e.g. by using stereotypes to mark elements of the LHS, RHS and correspondence graph. The advantage of this procedure is that the roles of the LHS graph and the RHS graph can easily be interchanged and thus, the transformation is bidirectional [GGL05]. Furthermore, the specification of a TGG can also be used for a transformation that supports check operations. The applicability of this approach for model transformations has been shown in [GGL05, Kön05]. A further major advantage is that model transformations with TGGs can be specified visually by creating graphs. The tool support for model transformations using TGGs has been surveyed in [KS06].

An example of a TGG implementation that tries to overcome the limitations of model transformation through graph transformation is eMoflon for which there is a proof-of-concept solution for the Flowgraphs case. This solution (and thus also eMoflon) is discussed in chapter 9.

2.3. MTLs with internal DSL

Internal DSLs use another language, usually a general purpose language, as host language. They are usually implemented as libraries for that host language with an API that can be used similar to an entirely new language.[3] Because they allow a very flexible syntax, dynamic or scripting languages like Python or Ruby are often used as host languages [CMT06]. However, the maintainability of a software artifacts also depends on the background of their users and dynamic languages are not as popular as mainstream languages like Java or C#. Furthermore, the tool support for such languages is often not as good.

Another language that is often used as a host language for internal MTLs is Scala [Pic08, Slo08]. Besides the flexible syntax, Scala is a strongly typed language and internal MTLs based on it like ScalaMTL can automatically inherit this type safety [GWS12]. ScalaMTL uses implicit conversions for tracing. However, the concept of implicit conversions for tracing yields some problems, e.g. if multiple passes of a model transformation are required.

[2] `http://planet-research20.org/ttc2011/index.php?option=com_content&view=article&id=148&catid=17&Itemid=203`

[3] A further discussion of internal DSLs for model transformation can be found in section 5.3

There is a continuous pass between shallow internal DSLs and ordinary libraries of the host languages. An example of the latter is Paisley [yWL12] which demonstrates a query-command syntax [Fow10] as pure library for Java. However, this transformation language misses important features like the avoidance of cycles.

One of the most important properties of an internal DSL is its ability to provide a concise syntax for the model transformation tasks. As internal DSLs have limits in their syntax, this may get difficult.

2.4. Comparison of MTLs

Table 2.1 tries to compare a number of model transformation languages (external and internal DSLs) for their support of the model transformation problems in chapter 6. This comparison is very rough as it only looks whether there exist concepts at all to solve these problems. To review these languages how well the offered concepts solve these problems is a research question on its own. Some further observations and thus comparisons to other model transformation languages are made in the TTC case studies (see chapters 9 and 10).

Based on this comparison, it may seem that external MTLs are generally suitable for model transformation tasks. As a reason, these languages have been designed specifically for these tasks but do not have any restrictions for their syntax. However, this comes at the price of poor tool support as all tool support has to be created almost entirely new. In contrast, internal MTLs can inherit the tool support provided for their host language. The more popular the host language, the better the tool support usually is. However, this discussion is intensified in the TTC case study chapters 9 and 10.

Furthermore, table 2.1 reads that NMF Transformations is the only solution that has built-in support for optimization tasks. This support is provided by NMF Optimizations that is not described in this master thesis but in a separate technical report [Hin13]. The reason that other MTLs do not have support for optimization tasks may be that domain-specific optimizations have rarely been a subject of research. To the best of my knowledge, NMF Optimizations is the only framework providing support for domain-specific optimization so far and it has not been evaluated in practise. Thus, the fact that NMF Transformations is the only solution with optimization support should be dealt with care. However, it is a clear advantage of NMF Transformations that extensions like NMF Optimizations are possible at all. For external MTLs, this is much more complicated and not possible without an access to the solutions source.

The table also ignores the modeling framework the MTLs are based on, as most MTL are indeed based on certain modeling frameworks. In contrast, NMF Transformations can operate on plain objects either as input or output models. This has several consequences. On the one hand, NMF Transformations cannot rely on the structure that is provided by a modeling framework. On the other hand, the scope of NMF Transformations is widened.

2.5. Maintainability of model transformations

Maintainability is an important aspect when developing a MTL if not the most important one. Van Amstel et al. have conducted a complex analysis [vAVDB11a, vA11] for ATL, QVT-O, XTend and ASF+SDF. The analysis in this thesis applies similar techniques for NMF Transformations. As the master thesis is the first document to describe NMF Transformations, it has not been evaluated before.

Name	Host language	Correspondence & Tracing	Cyclic object models	Inheritance	Patterns	Optimization tasks	Higher-Order Transformations	Transformation Composition	Testing [a]
QVT-O	N|A	✓	–	✓[b]	–	–	–	✓	✓
QVT-R	N|A	✓	✓	–[c]	✓	–	?	–	–
ATL	N|A	✓	✓	✓	✓	–	✓	✓	–
ETL	N|A	✓	✓	✓	✓	–	–	✓	–
RubyTL	Ruby	✓	?	?	–	–	–	?	?
FunnyQT	Clojure	✓	✓	✓[d]	✓[e]	–	–	?	?
ScalaMTL	Scala	✓	–	?	✓	–	–	?	?
Paisley	Java	?	?	?	✓	–	–	?	?
NMF	C#	✓	✓	✓	✓[f]	✓[g]	–	✓	✓

✓ = Dedicated support
– = No dedicated support
? = Not described in the literature, but possible extension

[a] As black-box testing is always possible, this asks for a unit testing support
[b] Disjunct mappings yield a maintenance problem (see section 6.3).
[c] QVT-R solves this problem through pattern matching (see section 6.3). However, there is no inheritance concept between relations.
[d] With the same restrictions as QVT-O
[e] Unlike ATL and ETL, FunnyQT supports a single input element type, only. However, the Petri Nets to State Charts case showed that FunnyQT is capable of more complex patterns through embeddings in general purpose code.
[f] With the Relational extensions, see section 7.4.6
[g] With NMF Optimizations, see [Hin13]

Table 2.1.: Comparison of MTLs

One important approach to analyze and therefore eventually improve the maintainability of model transformation is the definition of metrics. Thus, some approaches [vAvdB10, KGBH10, vAvdB11b] have defined maintenance metrics for transformations written in several MTLs. A discussion on these metrics can be found in the appendix of this thesis (see section F).

There are other approaches to analyze transformation languages for more specific attributes like modularity. For model transformations, this is achieved through composition of model transformations. The techniques for composition are analyzed in [CM08, Wag08, CM09, WVDSD10]. [WKK+12] provides a comparison framework for these approaches.

3. Foundations

In this chapter, some foundations are introduced that this thesis relies on. At first, the approach of model-driven software development is briefly explained in section 3.1. Second, in section 3.2 the two transformation languages with the Query View Transformation (QVT) standard by the OMG are briefly introduced. This is made because these languages are used as a reference. Last, in section 3.3 some more sophisticated language features of C# are introduced that are massively used in the API of NMF Transformations. Section 3.4 introduces the quality attributes of model transformations that will be used to guide through the validation sections of this master thesis.

3.1. Model-driven software development

Model-driven software development (MDSD) or Model-driven engineering (MDE) is an approach to handle the problem of ever-increasing complexity in the software development. Instead of code, domain specific models are the central software artifacts. All other software artifacts like code, documentation or test cases are then generated from the models using transformations. To make transformations possible, the models have to conform to a formal definition. As this formal definition is once again a model in the domain of metamodeling, it is called a metamodel. It describes the structure of the models that conform to this metamodel. As metamodels in turn are models in the domain of metamodeling, they also have their own metamodel, referred to as the meta-metamodel. Most available meta-metamodels are self-descriptive. This prevents endless conformance sequences. The OMG standardized the meta-metamodel MOF (Meta Object Facility). However, the most common meta-metamodel in practice is Ecore, an implementation of the EMOF (Essential MOF) standard. Ecore is implemented as part of the Eclipse Modeling framework (EMF) [MEG+03]. ling, it is called a metamodel. It describes the structure of the models that conform to this metamodel. As metamodels in turn are models in the domain of metamodeling, they also have their own metamodel, referred to as the meta-metamodel. Most available meta-metamodels are self-descriptive. This prevents endless conformance sequences. The OMG standardized the meta-metamodel MOF (Meta Object Facility). However, the most common meta-metamodel in practice is Ecore, an implementation of the EMOF (Essential MOF) standard. Ecore is implemented as part of the Eclipse Modeling framework (EMF) [MEG+03].

Sometimes this structural description encoded in a metamodel still allows to create models that are not valid as they do not represent the concepts in the reality properly. To restrict

the validity of models, static semantics is used to ideally only allow models that have a correspondence in reality. The static semantic is usually expressed through invariants that have to hold for specific objects. These invariants are typically expressed using the Object Constraint Language (OCL). Static semantics restrict models, so they are conformant to a metamodel. In contrast, dynamic semantics describes the correspondence to objects or concepts in the reality. Dynamic semantics are usually expressed in natural language.

Models that represent systems are usually hand-crafted in editors. There are two fundamentally different types of editors, graphical and textual editors. Graphical editors provide a graphical user interface and show the models as diagrams of some sort. The model-driven developer may then edit the models by editing the diagram or creating new diagrams. Textual editors rely on a grammar that describes how models can be described in text. The syntax of these editors, either as textual syntax or as description of how the graphs should look like, is referred to as concrete syntax as they describe what is visible to the model developer. In contrast, the metamodel and the static semantics are referred to as abstract syntax, as they describe the abstract concepts that are expressed through the concrete syntaxes. An abstract syntax can have multiple concrete syntaxes as there might be multiple editors to edit the same models. Most often, there are multiple editors that each edit different parts of the model. Abstract and concrete syntax together form domain specific languages (DSLs).

The models obtained from the various editors are then transformed to either other models or traditional software artifacts. This process is usually supported by Model Transformation Languages (MTLs). In many cases, the overall abstraction level of the model representing the whole system is relatively high and thus model transformations transforming these models directly to the desired software artifacts as code are very complex. Moreover, it is not only the semantics that has to be transformed, it also is the syntax. Therefore, it is a widely adopted approach to split the transformation of the semantics from the transformation of the syntax. Thus, a metamodel is created that describes the structure of the target software artifact. A model transformation can then transform the input system level model to a model of that target semantic whereas a separate transformation takes the model with the semantics already fit to the target software artifact and only transforms the syntax, i.e. prints the model in the format according to the type of software artifact that is to created. These two types of model transformations fundamentally differ. The first one takes models as inputs and creates models. It is referred to as Model-to-Model-transformation (M2M-transformation) whereas the latter one is referred to as Model-to-Text-transformation (M2T-transformation).

As the goal of M2T-transformations is to fit a model in a given structure, these transformations contain a lot of static information like keywords or the structure of the targeted format and are thus mostly formulated as text templates. However, as they aim to transform syntax rather than semantics, it is often difficult to include complex transformation logic into these transformations.

In contrast to M2T, M2M-transformations transform models conforming to one or multiple source metamodels into a model conforming to a target metamodel. Usually, the metamodeling foundation is used to load and save models to files. But the transformations usually do not care how the models are serialized. The focus is rather set to the transformation of semantics. There are different paradigms of model transformations present and a whole row of different transformation engines. They can roughly be split up transformation approaches using dedicated transformation languages as external DSLs and approaches using any language as host language for an internal DSL that is used to describe the transformations. Some examples of each are presented in chapter 2. However, a prominent example of external DSLs for model transformation is QVT that actually consists of two external

DSLs. The whole concept of QVT is explained in more depth in the section 3.2.

3.2. Query-View-Transformation (QVT)

QVT [Obj11] is a standardized MTL designed by the OMG. Model transformations in QVT are usually written in one of the languages QVT Operational (QVT-O) or QVT Relational (QVT-R). QVT Operational is described in more detail in section 3.2.1. QVT Relational is described in more detail in section 3.2.2. Both of these languages are meant to be compiled to QVT Core, a language not intended to be written by humans directly. An implementation of a transformation engine would then be able to execute the QVT Core code after QVT-R or QVT-O have been compiled to QVT Core. However, existing implementations only execute either of them directly. Instead, approaches exist that can translate QVT-R into QVT-O [RRMB08].

The concepts of the QVT languages are explained using an example transformation from UML models to a relational database scheme. The transformation code is taken from the QVT specification by the OMG [Obj11]. However, the description here is only an overview. For more details see the specification [Obj11].

3.2.1. QVT Operational

QVT-O is an imperative MTL based on mapping rules. Much like imperative languages like Java, Mappings in QVT-O specify how a model element is transformed into another in an imperatively manner. For this purpose, the side-effect free query language OCL (Object Constraint Language) has been extended with some imperative constructs like statements, assignments and loops. In this way, all the OCL language constructs can still be used when writing model transformations.

Transformations

Transformations in QVT-O need to specify the domains used as input or targeted domain. There can be multiple input or target domains specified. Furthermore, much like usual programs in imperative languages, a transformation has to specify a main method.

```
1 modeltype UML uses SimpleUml ("http://SimpleUml");
2 modeltype RDBMS "strict" uses SimpleRdbms;
3
4 transformation Uml2Rdbms(in uml:UML, out rdbms:RDBMS);
5 // the entry point for the execution of the transformation
6 main() {
7  uml.objectsOfType(Package)->map packageToSchema();
8 }
```

Listing 3.1: A transformation in QVT-O

Listing 3.1 shows an example of a transformation in QVT-O (taken from [Obj11]). It transforms UML models into relational database schemes. The statement in line 7 is used to call Mappings. These concepts are explained in the following paragraph.

Mappings

Mappings are the heart of a transformation in QVT-O. They define how an element (or multiple) should be mapped to another element. Although some dependencies and guards can be set declaratively, the main part of the Mapping is written in an imperative manner. Listing 3.2 shows an example of how such a Mapping looks like.

```
1 mapping Package::packageToSchema() : Schema
2  when{ self.name.startingWith() <> "_"}
3 {
4  name := self.name;
5  table := self.ownedElement -> map class2table();
6 }
```

Listing 3.2: A example Mapping in QVT-O

Mappings are the language features that differ most from pure object-oriented programming. They consist of a context, optional multiple parameters and a return type. Unlike methods in object-oriented programming, their execution is traced. QVT-O has a trace model where it saves which elements have been mapped together with the result of that Mapping. Transformation developers can access these trace links by the language feature `resolve`. However, resolve only looks at trace links already created when the execution is at this point. If the trace link is intended to see all trace links, it is possible to use another language feature namely `late resolve` that resolves the trace link after all Mappings have been executed. The result that is written into the trace by default, is a newly created model element conforming to the return type of the Mapping. However, it is possible to overwrite this behavior. For this purpose, a Mapping can consist of three section. The first section, the init section, is executed before the local implicit `result` variable is written to the trace and therefore can be overwritten. The populate section of a Mapping is to contain code that populates the properties of the resulting elements. Finally a dedicated end section is executed before the execution flow leaves the Mapping. The structure of a Mapping is also shown in listing 3.3. There, X denotes the type that is decorated by the Mapping. Furthermore, the listing shows the syntax for multiple target elements (whereas the syntax for a single result is shown in listing 3.2).

```
1 mapping<dirkind0> X::mappingname
2 (<dirkind1> p1:P1, <dirkind2> p2:P2) : r1:R1, r2:R2
3  when{...}
4  where{...}
5 {
6  init{...}
7  population{...}
8  end{...}
9 }
```

Listing 3.3: The structure of a QVT-O mapping

Here, the `when`-clause denotes a precondition whereas the `where`-clause may contain a post-condition.

Queries, Constructors and Helpers

Queries, Constructors and Helpers are constructs to help structure transformations and reuse some parts in multiple transformations. Whereas Mappings have to reside in the same file as the transformation, Queries, Helpers and Constructors can be swapped out to other files, called libraries. Helpers and Queries are methods that can be attached to a type. They can have multiple input parameters and a return type but do not have to. They can actually be imagined as being much like methods in object-oriented programming besides the fact that they can be defined outside the class they use as context object. The difference between Helpers and Queries is that Helpers may have side effects on

their parameters whereas Queries must not. Constructors are very similar, much like constructors in object-oriented programming they build a model element using several parameters.

Intermediate data

QVT-O has the handy ability to extend metaclasses by properties only used during a transformation. Alternatively, new intermediate metaclasses can be created that are also only valid during transformation.

```
1 intermediate class LeafAttribute
2 {
3  name:String;
4  kind:String;
5  attr:Attribute;
6 };
7 intermediate property Class::leafAttributes
8  : Sequence(LeafAttribute);
```

Listing 3.4: Definition of Intermediate Properties and Classes in QVT-O

Listing 3.4 shows an example where Intermediate Classes and Properties are used to flatten the class hierarchies.

Disjunct Mappings

To e.g. support inheritance, QVT-O includes a mechanism to define a mapping as disjunction of an ordered list of other mappings. Instead of the disjunct mapping, the first mapping whose guard (`when`-clause) returns true.

```
1  mapping UML::Feature::convertFeature () : JAVA::Element
2   disjuncts convertAttribute, convertOperation, convertConstructor
       () {}
3
4  mapping UML::Attribute::convertAttribute : JAVA::Field {
5   name := self.name;
6  }
7  mapping UML::Operation::convertConstructor : JAVA::Constructor
8   when{self.name = self.namespace.name;} {
9   name := self.name;
10 }
11 mapping UML::Operation::convertOperation : JAVA::Method
12  when{self.name <> self.namespace.name;} {
13  name := self.name;
14 }
```

Listing 3.5: A disjunct mapping in QVT-O

In the example of listing 3.5, the disjunct mapping feature is used to transform a UML feature into a Java feature. The mapping disjunction makes it possible to avoid type casts, which are considered as being hard to maintain. Instead, it is the task of the transformation engine to check the types and the preconditions of the disjuncted mappings.

Transformation composition

QVT-O supports the composition of transformations. In general, there are two popular ways to compose a transformation of multiple others:

1. Transformation chaining: This means that transformations are executed one after another.
2. Transformation composition: A transformation extends another transformation by adding or removing elements.

Transformation composition is discussed in more detail in section 6.7.

In QVT-O, each of these composition techniques is represented by a dedicated mechanism. To use transformation chaining, a transformation that uses other transformations has to declare this usage at its head using the *access* keyword. In the main method, the other transformation can then be instantiated and called using the *transform* method. This method returns a status with information whether the transformation was successful. To extend a transformation, the extending transformation has also to declare that in the header using the *extends* keyword. An extension can define further transformation elements like mappings and replace the main method. Listing 3.6 shows an example.

```
1  transformation CompleteUml2Rdbms(in uml:UML, out rdbms:RDBMS)
2   access transformation UmlCleaning(inout UML),
3   extends transformation Uml2Rdbms(in UML, out RDBMS);
4
5  main() {
6   var tmp: UML = uml.copy();
7   // performs the "cleaning"
8   var retcode := (new UmlCleaning(tmp))->transform();
9   if (notretcode.failed())
10   uml.objectsOfType(Package)->map packageToSchema()
11  else raise "UmlModelTransformationFailed";
12 }
```

Listing 3.6: Transformation composition in QVT-O

In the example, the above `Uml2Rdms` from listing 3.1 has been extended by an in-place cleaning transformation. The only extension that the `CompleteUml2Rdbms` does is to replace the main method and chain the cleaning transformation first.

3.2.2. QVT Relational

QVT-R is a declarative MTL. Rather than specifying how elements of the source model are to be transformed into elements of the target model, the transformation rules in QVT-R form relations between them. These relations are then to be enforced by the transformation engine implementation. In this way, a QVT-R transformation rather synchronizes two models with several synchronization rules rather than explicitly transforming the input model. In this way, QVT-R is an example of a declarative transformation language that even provides support for bidirectionality.

Relations in QVT-R

The relations defined by QVT-R act as constraints. This procedure can be seen as similar to math relations as a (binary) relation from elements of two sets M_1 and M_2 is defined as $R \subset M_1 \times M_2$. In a similar way, the relations in QVT-R limit the input and target domain

elements. However, in some cases it makes sense to limit the scope of these constraints and allow exceptions, e.g. when there is another constraint that has to hold for a certain element. Furthermore, to divide implementations of such constraints a bit, relations can further define other relations that need to hold, also. This yields the basic structure of a QVT-R relation. One or many domains can be specified that are equivalent to the sets M_1 and M_2 in the above mini-example. A relation can have a `when`-clause specifying when the constraint is enforced and a `where`-clause specifying other constraints that need to be enforced. Listing 3.7 shows an example relation (taken from [Obj11]).

```
1  relation ClassToTable /* map each persistent class to a table */
2  {
3   domain uml c:Class {
4    namespace = p:Package {},
5    kind='Persistent',
6    name=cn
7   }
8   domain rdbms t:Table {
9    schema = s:Schema {},
10   name=cn,
11   column = cl:Column {
12    name=cn+'_tid',
13    type='NUMBER'},
14   primaryKey = k:PrimaryKey {
15    name=cn+'_pk',
16    column=cl}
17  }
18  when {
19   PackageToSchema(p, s);
20  }
21  where {
22   AttributeToColumn(c, t);
23  }
24 }
```

Listing 3.7: A QVT-R relation with two domains

The example shows the relation that maps persistent classes to tables in a transformation that transforms UML models to relational database schemes. The domain sections specify patterns that form the equivalent to the sets M_1 and M_2, the set of domain elements to which the relation should be applied. The relation further specifies that for each `Class` in the *uml* model. There should be a corresponding `Table` element in the *rdbms* model that has the structure specified in the according domain section, provided that the package of that class corresponds (stand in a relation to) the schema of the table. Furthermore, the attributes of the class and the columns of the table have to conform to the relation *AttributeToColumn*.

Furthermore, there are different types of domains, domains where transformation engine should only look for patterns (*checkonly domains*) and domains where the transformation engine may create model elements to enforce the patterns (*enforce domains*). Some implementations even do not allow to not specify the type of the domain.

Top relations

Some relations within a transformation may be specified as *top relations*. These top relations are the relations that need to hold for every matched elements. Thus, they may

not specify a `when`-clause. Relations that are not marked as top relations are only enforced if they are called explicitly in a where section of another relation. Listing 3.8 [Obj11] again shows the above transformation from UML models to database schemes. The relations *PackageToSchema* and *ClassToTable* are enforced for every package or class that match the relations pattern, respectively. The third relation *AttributeToColumn* is only enforced when called explicitly from another relation.

```
1 transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) {
2  top relation PackageToSchema {...}
3  top relation ClassToTable {...}
4  relation AttributeToColumn {...}
5 }
```

Listing 3.8: A QVT-R transformation with top relations and non-top relations

3.3. C# language features

An internal DSL has to make use of the features of the host language it is embedded in. Being implemented as library conforming to the Common Language Specification (CLS), NMF Transformations can have various host (.NET-)languages. However, in this thesis we concentrate on C# as host language. Therefore, this section gives an overview about some of the rather special language features of C#, especially in comparison to Java.

Generally recognized as an imperative language, C# has changed in the last years and continuously got elements known from functional programming. Most of the features that are presented in the subsequent subsections were introduced with C# 3.0 which has been a part of the .NET framework 3.5 released in November 2007. Both the implementation but especially the public API of NMF Transformations use these features massively and so they are described here in a bit more detail. The entire documentation can be found in the MSDN[1].

3.3.1. Local type inference

C# allows local type inference for local variables. This means that instead of declaring a variable with its type, the compiler can automatically interfere the type of the local variable by its initialization. Thus, for example the following two statement blocks in listing 3.9 are equivalent.

```
1 string a = "";
2 int b = 1;
3 StringBuilder sb = new StringBuilder();
4 Guid uid = Guid.NewGuid();
5
6 var a2 = "";
7 var b2 = 1;
8 var sb = new StringBuilder();
9 var uid = Guid.NewGuid();
```

Listing 3.9: Local type inference examples

Being purely syntactical sugar, the `var` keyword is mostly used to minimize the typing effort. However, the coding guidelines of some companies (including Microsoft) avoid the

[1] `http://msdn.microsoft.com/en-us/library/vstudio/618ayhy6.aspx`

usage of this keyword as there is a threat that the developer is unaware of the type of a variable.

The purpose of the `var` keyword is to support anonymous types as presented in the next section.

3.3.2. Initialization lists and anonymous types

Initialization lists are used to simply initialize an object instance by setting its properties without implementing a dedicated constructor. This makes the two statements in listing 3.10 equivalent.

```
1 var person = new Person();
2 person.Name = "John";
3
4 var person2 = new Person() { Name = "John" };
```

Listing 3.10: Object initializers in C#

The main purpose of this procedure is that the above statement in listing 3.10 can now be written in a single line which is important for other language features like Language Integrated Query (LINQ, see section 3.3.5).

These initialization lists further make it possible to use anonymous types by simply omitting the type name. This can be used to avoid implementing a dedicated class that simply consists of several properties. Anonymous types require to use the local type inference with the `var` keyword as the name of these types is generated by the compiler and thus unknown at development time.

```
1 Person john = ...;
2 Person marry = ...;
3
4 var couple = new { Husband = john, Wife = marry };
```

Listing 3.11: Using anonymous types in C#

3.3.3. Extension methods

Extension methods are another language feature of C# since version 3.0. It is syntactic sugar to add methods to sealed types like interfaces. These extension methods are then mapped to static methods that take the context object instance as first parameter. A method is marked as extension method by decorating the first parameter with the `this` keyword. The `var` keyword is just an instruction to the compiler to infer the type of a local variable from the context.

Extension methods may only be defined as static methods inside of static classes (i.e. classes that must not be instantiated). In order to use an extension method, the namespace of the declaring class must be imported with a using statement.

3.3.4. Lambda-expressions (Closures)

One of the key differences of functional programming to imperative programming is the treatment of functions as objects. Therefore, it is no surprise that these languages have concepts to define functions very easily. Initially introduced in the lambda calculus [Chu36, Chu40], lambda expressions are nowadays used in many functional programming languages and since version 3.0 released in 2007 in C#. However, in the literature, they

are often called Closures. Lambda expressions allow to define method within the context of a method. In this way, a method implemented as a lambda expression can make use not only of its own parameters, but can also refer to the parameters of the method it is defined in. As the compiler (at least in C#) has to fill the gap between the definition of that lambda expression involving other variables than just its own parameters, this may be a reason that Lambda expressions are often called Closures. The transformation in an object-oriented design is hidden to the developer by the compiler. This concept makes it much easier to use higher order functions. As an example, listing 3.12 shows how the higher order function Iterate that iterates a given function until it returns false could be easily implemented.

```
1 public static Action<T> Iterate<T>(Func<T, bool> iterateAndTest)
2 {
3  return item => while (iterateAndTest(item)) {};
4 }
```

Listing 3.12: Usage of Lambda-expressions for higher-order functions

3.3.5. Monads

Monads are a concept initially taken from category theory, but have been applied to functional programming in 1991. They consist of a type constructor, a unit function and a bind function. The type constructor takes a type and puts it into a monadic container. This container may decorate the type with some additional information. The unit function is used to create an instance of the monadic container out of an instance of the decorated type and the bind function composes two monadic containers to a new monadic container.

An example in C# is the `IEnumerable<T>` type that can be seen as a monadic container of the type T. It describes a collection of items of type T that can be enumerated using an enumerator obtainable from that interface. Here, the unit function is a function that wraps an item of type T into a collection with exactly one item. The bind function is a method that takes a collection and a function to obtain child collections and returns a collection of child items. An example of how the bind function implemented as an extension method could look like in presented in listing 3.13

```
1 public static IEnumerable<U> Bind<T,U>(this IEnumerable<T> items,
2  Func<T, IEnumerable<U>> func) {
3  foreach (var item in items) {
4   foreach (var subitem in func(item)) {
5    yield return subitem;
6   }
7  }
8 }
```

Listing 3.13: An implementation for the bind function of the IEnumerable<T> monad

In the above code example, the `yield return` statements makes it possible to specify an iterable collection (an *IEnumerable<T>* in .NET) without specifying an iterator. The iterator is constructed by the compiler based on the method implementation. As this language feature is not used in the public API of NMF Transformations, it is not introduced here. Instead, a documentation can be found in the MSDN[2].

However, monads in C# are a bit different. For performance reasons, the bind function has been extended with an additional selector method and a third selection type. Furthermore,

[2] `http://msdn.microsoft.com/en-us/library/vstudio/9k7k7cf0.aspx`

it is named `SelectMany` rather than bind. A possible implementation of this method for the `IEnumerable<T>` monad is presented in listing 3.14.

```
1 public static IEnumerable<V> SelectMany<T,U,V>(
2  this IEnumerable<T> items, Func<T, IEnumerable<U>> func,
3  Func<T,U,V> selector) {
4  foreach (var item in items) {
5   foreach (var subitem in func(item)) {
6    yield return selector(item,subitem);
7   }
8  }
9 }
```

Listing 3.14: An implementation for the bind operation of the IEnumerable<T> monad

If we consider the simple scenario where we have two lists of strings and want to have a collection with all possible concatenations of these strings, we can simply write use our extension method from listing 3.14, see listing 3.15.

```
1 IEnumerable<string> items1;
2 IEnumerable<string> items2;
3
4 foreach(var s in items1
5  .SelectMany(s1 => items2, (s1,s2) => s1 + s2)) {
6         Console.WriteLine(s);
7 }
```

Listing 3.15: Using the IEnumerable<T> monad with the LINQ query syntax

With this implementation of `SelectMany`, it is possible to use the query syntax of LINQ. To make this possible, the bind operator `SelectMany` must be implemented as an extension method and the namespace of the implementation class must be imported with a using statement. The above example using the query syntax is demonstrated in listing 3.16. The second `from` statement and the `select` statement are translated into a call of the `SelectMany` extension method shown in listing 3.14.

```
1  IEnumerable<string> items1;
2  IEnumerable<string> items2;
3
4  var itemsCombined = from s1 in items1
5                      from s2 in items2
6                      select s1 + s2;
7
8  foreach(var s in itemsCombined) {
9         Console.WriteLine(s);
10 }
```

Listing 3.16: Using the IEnumerable<T> monad with the LINQ query syntax

However, the query syntax allows more flexibility. To use an enhanced syntax, other extension methods have to be implemented. Here, we introduce the further extension methods that are used in the implementation of NMF Transformations. This includes only the `where` operator. The goal is to be possible to enhance the API such that a syntax like in listing 3.17 is possible.

```
1  IEnumerable<string> items1;
2  IEnumerable<string> items2;
3
4  var itemsCombined = from s1 in items1
5                      from s2 in tems2
6                      where s1 != s2
7                      select s1 + s2;
8
9  foreach(var s in itemsCombined) {
10         Console.WriteLine(s);
11 }
```

Listing 3.17: Using the IEnumerable<T> monad with the LINQ query syntax, enhanced

The first and quite obvious extension method that is needed for this syntax is `Where`. Listing 3.18 shows its signature.

```
1  public static IEnumerable<T> Where<T>(this IEnumerable<T> items,
       Func<T,bool> filter);
```

Listing 3.18: The signature of the Where extension method

The semantics of the `Where`-method is that it should filter the monadic instance with the given filter method. If there was only one `from` statement in listing 3.17, we would be done here. However, there are two of them and thus, there is a further method needed for the above syntax to not cause a compilation error. In listing 3.16, the second `from`-statement calls `SelectMany` where the type placeholders T, U, V are set to *string* and the selection method returns the concatenation. However, in listing 3.17 there is a filter in between based on both string values. Thus, `SelectMany` is called with a compiler generated tuple of strings as type parameter V. This tuple is then filtered by a `Where` implementation as in listing 3.18. Thus, we need a further extension method to perform the final selection that returns the concatenation. The signature is presented in listing 3.19.

```
1  public static IEnumerable<U> Select<T,U>(this IEnumerable<T>
       items, Func<T,U> selector);
```

Listing 3.19: The signature of the Select extension method

3.4. Quality Attributes of Model Transformations

In his PhD-thesis, Marc van Amstel describes a row of quality attributes for model transformations [vA11] to reflect the internal quality of model transformations. These quality attributes have been derived with respect of development and maintenance and are thus suitable to make statements on the maintainability. The following sections will introduce these quality attributes. For more detail, especially for the discussion why these quality attributes have an impact on the maintainability, the reader is referred to [vA11].

3.4.1. Understandability

Understandability is a general code attribute and refers to the effort to understand what the code actually does. Understandability itself has a high complexity as it depends on a row of subsequent attributes like the background of the developers that are trying to understand a given piece of code.

For traditional software artifacts, understandability has been shown to make up the biggest part of the effort involved in most maintenance tasks [CMT96, SWM97].

Tool support for the understandability of a model transformation includes visualizations of its internal structure. An example how such tool support could look like for a model transformation language is presented in [RNHR13] where a tool support for QVT is presented.

Understandability is a subjective criterion that can only be evaluated by using questionnaires.

3.4.2. Modifiability

"Modifiability refers to the amount of effort for adapting a model transformation to provide different or additional functionality." [vA11]. For traditional software artifacts, perfective changes make up the biggest proportion of such adaption scenarios [LS81]. Of course, modifiability is closely related to understandability as developers have to understand a model transformation in order to change it.

In traditional software artifacts, a large proportion of modifiability consists of discoverability. That is, the developer can discover the features of a framework without referring to the documentation and be able to use a framework guided only by the tool support. This tool support consist of auto-completion and suggestions.

3.4.3. Reusability

Reusability in the context of model transformations is the extent in which parts of the model transformation can be reused for other (related) model transformation. That is, in order to be reusable, a model transformation language must provide means to compose a model transformation of parts of other model transformations written in the same language.

Reusability is an important attribute for the maintainability of a model transformation as it helps to avoid duplication of code which yields maintenance problems as this code eventually has to be changed.

3.4.4. Modularity

"Modularity is the extend in which a model transformation is systematically separated and structured." [vA11]. Modularity is particularly useful to quickly identify the change impact of any changes occurring when evolving the model transformation. Furthermore, a structure of well separated parts of a model transformation also means that these parts can be developed separately.

3.4.5. Completeness

The completeness is the extend in which a model transformation fulfills its requirements. Unlike the other quality attributes, completeness is also an external quality attribute as it measures the conformance of the model transformation to the requirements. As such, the completeness depends on these requirements.

3.4.6. Consistency

"Consistency is the extend in which a model transformation is implemented in a uniform manner. " [vA11]. That is, a uniform programming style is used throughout the whole model transformation.

Possible reasons for inconsistent model transformations include that multiple developers may be working on a model transformation, but also that the transformation language in use does not allow to fulfill all requirements in a convenient way. In the latter cases, transformation developers may decide to change the programming style or even paradigm to fulfill these more sophisticated requirements.

3.4.7. Conciseness

"Conciseness refers to the extend in which a model transformation is free of superfluous elements." [vA11]. Although Van Amstel mainly refers to unused parts of a model transformation, conciseness also refers to the density in which the model transformation is specified. For model transformation languages, this includes the avoidance of syntactic noise, i.e. code that is only required for the parser but does not have a semantic.

4. Example Transformations

In this section, some examples for typical model transformations are introduced. These sample transformations are used later to explain problems and approaches to solve them. They are rather toy examples than model transformations from practice and the solutions thus do not offer much insight to model transformation.

4.1. Finite State Machines to Petri Nets

Finite state machines have gained a high popularity in computer science. They can be used to describe processes in an intuitive way. Finite state machines consist of multiple (finitely many) states that can have transitions to each other that are triggered on a certain input. There is exactly one start state and arbitrary many final states. Transitions have exactly one origin state and one target state. If one considers a online purchasing process, it starts when the customer navigates to the shopping portal. The customer then puts items into the shopping cart and eventually navigates to the checkout. Thus, he enters his personal details, the shipping address and the payment method and finally confirms the purchase. This process can be described in a finite state machine where the steps to complete the purchasing process are represented by the states of that finite state machine. The transitions are triggered by the customer entering a URL to his browser. The states of the finite state machine may form cycles like after inserting an item to the shopping cart, the customer might insert another item. Also more complex cycle are possible. A customer might proceed to the checkout, fill in the personal details, add items to the shopping cart and cancels the purchase process afterward. This causes the finite state machine go back to the state where the customer is inserting items to the shopping cart.

Figure 4.1 shows a metamodel of finite state machines.

Petri Nets are a different way to describe processes. Instead of states, Petri Nets consist of places. Whereas finite state machines have the semantics that they are in one state at a time, the semantic of Petri Nets is that tokens flow through the Petri Net. Furthermore, the transitions in a Petri Net may have multiple origin and target places. They are triggered if there is a token on each origin place. The then destroy these tokens and create new tokens at each of the target places. In the example above, the requests in the shopping system can be seen as tokens and the states of a request can be seen as places. Of course, many customers might insert items to shopping carts in parallel.

Figure 4.2 shows a metamodel of Petri Nets.

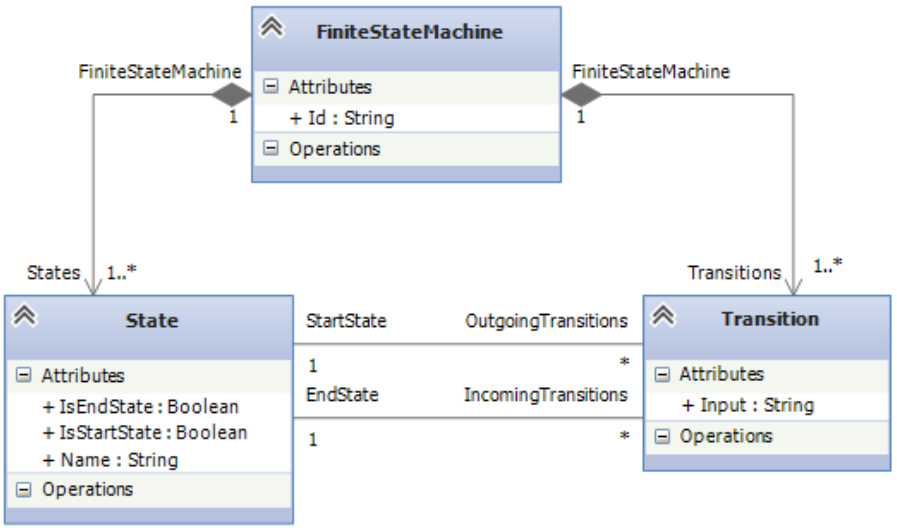

Figure 4.1.: A metamodel for finite state machines

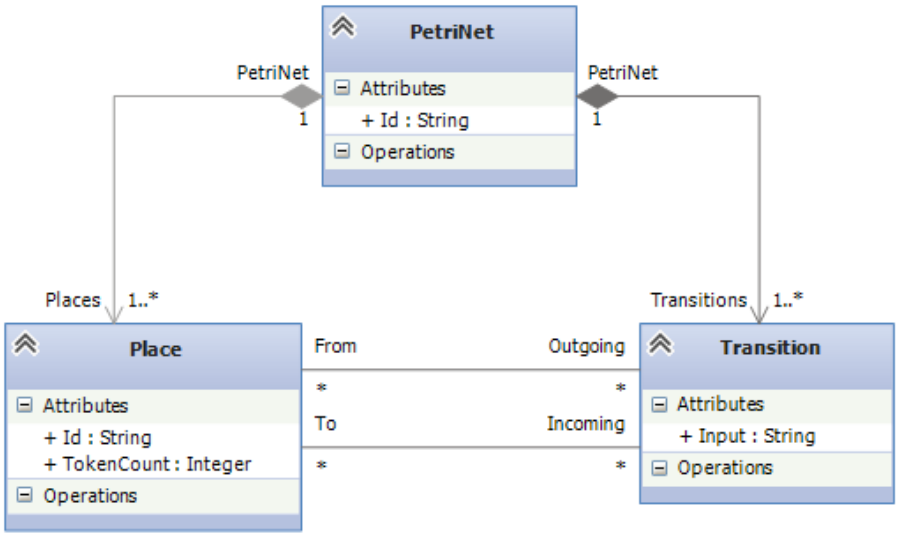

Figure 4.2.: A Metamodel for Petri Nets

In the above example, a Petri Net allows several kinds of analysis on the data. Given the response times of the software components represented by the places and a usage scenario of customers, it is possible to identify the bottleneck component by extending the Petri Net to a Queuing Petri Net [KB03]. Thus, possible performance problems can be unveiled in early development stages.

The model transformation from models of finite states to models of Petri Nets can be specified through the following requirements:

- For each state in the finite state machine, there should be a corresponding place in the Petri Net.
- For each transition in the finite state machine, there should be a corresponding transition in the Petri Net.
- For each start state in the finite state machine, there should be a transition without an origin and and one target place set to the corresponding place of the start state.

- For each end state in the finite state machine, there should be a transition without a target place and one origin place, which is the corresponding place of the end state.

In other words, the resulting Petri Net should have the same behavior as the finite state machine with the exception that it is possible to describe how multiple requests go through the system at the same time. Requests drop in from the transition corresponding to the start state and flow through the Petri Net. This transformation is even valid if an invalid finite state machine model with multiple start states is transformed.

This model transformation is a rather simple one. The main challenge is to establish a correspondence between states and places.

4.2. People to Family Relations

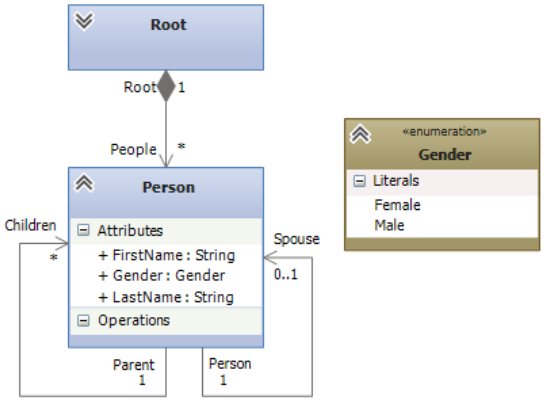

Figure 4.3.: The *People* metamodel

Another example of a model transformation is a transformation between two different metamodels expressing the same domain. In this example, there are two metamodels depicted in the figures 4.3 and 4.4 that actually both express the same domain. This domain describes several people and the relations between them. However, whereas the *People* metamodel in figure 4.3 is rather simple, the *FamilyRelations* metamodel in figure 4.4 is more sophisticated.

Similar to the transformation in section 4.1, this transformation includes the task to map person objects from the source model to corresponding person objects in the target model. However, this transformation includes an inheritance hierarchy in the target metamodel, although in the source metamodel, the equivalent concept is encoded in an enumeration. Thus, males and females must be treated differently by the transformation. Furthermore, the transformation includes some domain specific transformation knowledge, as the transformation must know that for instance the sisters of a woman are the daughters of her parents excluding herself.

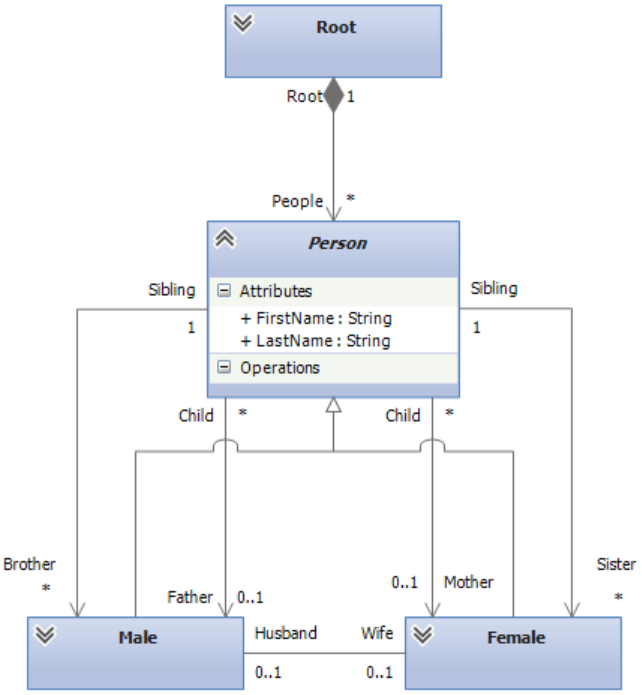

Figure 4.4.: The *FamilyRelations* metamodel

5. Supporting model transformation with dedicated languages

Debating about model transformation languages, it is important to discuss how a model transformation language should be implemented. Section 5.1 first discusses the specifics of the domain of model transformation. Afterward, the sections 5.2 and 5.3 discuss to use an external language or an internal language for model transformations. Finally, section 5.4 draws conclusions how to implement a maintainable model transformation language.

5.1. The domain of model transformation

Transforming models can be seen as a dedicated domain in the sense of model driven engineering. As a result, some transformation languages like ATL encourage this way of thinking, expressing a transformation as a model in this domain [BBG+06]. This way of consideration of a model transformation has huge advantages when it comes to higher-order transformations. Being models themselves, transformations can be analyzed or generated using normal model transformations.

However, the domain of model transformation has some characteristics that makes it different to other domains. Sendall and Kozaczynski have stated several requirements in [SK03a]. According to this paper, it is a central requirement for model transformation languages that

> "A transformation language must provide for complete automation and must be expressive, unambiguous, and Turing complete".

Complete automation means that a transformation language must provide a tool that enables users to run the model transformation automatically. It is a desirable tool support to integrate model transformations in the development workflow so that any transformations run automatically whenever required (e.g. when the source model changes or when the target artifact is needed). Expressiveness means that a transformation developer ought to be equipped with language features to express the essence of the model transformation without too much syntactic noise. Unambiguity means that a model transformation must have a deterministic result, i.e. given the same input models, it should result in equivalent target models. The Turing complexity relates to the complexity of the language. A Turing complete language can describe an arbitrary Turing machine and thus all algorithms that can be implemented by a Turing machine.

Whereas complete automation and unambiguity are common requirements for a domain specific language, the Turing completeness is not. In fact, Kelly and Pohjonen listed it as worst practice [KP09] for the underlying abstract syntax represented by the metamodel. Fowler even goes one step further and generally calls Turing complete languages general purpose languages, even though they might be dedicated for a specific domain like for example R is dedicated for statistics [Fow10]. As reason that Fowler puts such a strong emphasis on the limited expressiveness, it changes the characteristics of a DSL in comparison with general purpose languages in the way how these languages are implemented and used. While a general purpose language is used for many purposes, a DSL has its tiny little purpose that is entirely solved by that DSL.

This requirement is also debatable. It is also a question of what exactly a model transformation is, especially how far a model transformation is able to transform the semantics of a model. Indeed, it is easy to find a model transformation problem that only requires a regular language to be solved but it is hard to find a good example to demonstrate that MTLs should be Turing complete. When debating that model transformations indeed should be Turing complete, the best argument is that especially in-place transformation sometimes include model optimization tasks of arbitrary complexity. Furthermore, the lambda calculus is shown to be Turing complete. However, in thesis, we tend to follow the argumentation of Sendall and Kozaczynski that model transformation languages should be Turing complete.

The problems arising when DSL are getting more and more expressive, they start copying duplicating concepts. Since they have proven their usefulness, domain dedicated general purpose languages often duplicate concepts that are already included in main stream general purpose languages (which in turn Kelly and Pohjonen list as separate worst practice). Assuming Turing complexity further yields some other problems like the undecidability whether higher-order transformations terminate (an immediate consequence from the Halting problem). This leads to the question whether model transformation should be seen as general purpose task, to be supported by general purpose languages.

Another factor that advocates calling model transformation languages general purpose languages is the scope of these languages. Where DSLs are usually created to support a very specific task, model transformation languages are used throughout the community of that MTL instead of just inside a single or a few projects.

However, the work done so far in domain-specific languages yields a useful categorization when talking about model transformation languages. Although they have different scope and an increased complexity, the way of how to implement these languages either as entirely new language like an external DSL or as framework providing a language-like syntax remains essentially the same. In fact, there are examples of both kinds of general purpose languages. Examples of external general purpose languages include the main stream general purpose languages like Java or C# and many others like Ruby. However, many consider Ruby on Rails a different language, though still general purpose, but clearly Rails is the equivalent to an internal DSL in general purpose languages.

Furthermore, in contrast to most general purpose languages, MTLs have a focus on a certain task, although this task cannot clearly be bounded. The task of a model transformation language is like to transform anything into anything. However, there are quite some assumptions that can be made for the environments where model transformation languages are going to be executed which is not true for languages like Java or C#. We can assume that a model transformation task is part of a model-driven software development process and mostly runs without interactions (although interactive model transformations exist). There are also some common high-level abstractions for model transformation implemented in most model transformation languages. These concepts include mapping

concepts and tracing concepts. That is, most model transformation languages save the information that particular model elements correspond to each other. The lack of such abstractions in mainstream languages like C# or Java makes these languages infeasible for model transformation tasks [SK03a].

Thus, model transformation languages have a position somewhere in between domain-specific languages and usual general purpose languages, much like R is considered a general purpose language by Martin Fowler but indeed has a though washy domain (to support any statistics work). Where Fowler states there is a fuzzy borderline between domain-specific languages and general purpose languages, languages like R or model transformation languages are pretty clearly on that borderline.

This position being somewhere in between general purpose and domain-specific languages yields an again different way of how such languages are used or implemented. Where general purpose languages are implemented by a very small group of developers in comparison to their users and on the other side domain-specific languages have a tiny amount of users, model transformation languages are also in between. The proportion of users in relation to the language developers is again somewhere between general purpose languages and DSLs.

However, as the borderline between DSLs and general purpose languages is fuzzy enough, the introduction of another intermediate kind of languages must cause an amount of confusion that would likely overwhelm the benefit of that categorization. Thus, in this thesis we continue to treat model transformation as a domain, although possibly one with an intrinsic Turing complete complexity. As this complexity is intrinsic in the domain, we will continue to call the languages dedicated to this domain domain specific languages, as their Turing complexity is caused by their complex domain. However, it is still useful to keep in mind that this categorization is not that clear and model transformation languages can also easily be referred to as general purpose languages.

5.2. MTLs as External DSLs

External DSLs are domain specific languages that use an entirely new language. Thus, language engineers that create an external DSL also have to create all the tool support. This includes a parser and either an interpreter or a compiler at minimum. An interpreter is a component that executes a model at runtime whereas a compiler uses some sort of intermediary representation. In the context of model-driven engineering, the compiler is usually referred to as code generator.

In general, external domain specific languages have the great advantage that there are no restrictions on the language. Thus, in contrast to internal DSLs that are embedded in a host language, they can provide a clear syntax avoiding unnecessary syntactical noise like unnecessary punctuation. In practice, the only restriction usually is that the parser generator has to be able to create a parser for the language but as most such systems work with BNF or EBNF[1] syntax, this is not a hard restriction.

Where general purpose languages lack of suitable high-level abstractions, external DSLs can provide a simple syntax for these high-level abstractions. A transformation engine can then execute the execution semantics of these high-level abstractions. The actual abstraction level of these concepts may vary. As an example, QVT-O is still an imperative language and thus the abstraction from the actual computational model is quite thin. In contrast, the abstractions included in QVT-R are much higher hiding the computational

[1] (Extended) Backus-Naur-Form, a special syntax for grammars

model entirely from the transformation developer, which is why it is called a declarative language.

However, as we are dealing with a complex domain, these external DSLs somehow have to find ways to represent this complexity. The threat here is that these transformation languages may duplicate the concepts known from general purpose languages which is considered a worst practice [KP09]. As an example, QVT-O extends the OCL query language with some imperative constructs like assignments and loops. The threat arising from this practice is that the language becomes less useful as the language anneals to main stream general purpose languages which usually have much better tool support. In turn, the DSL gets less useful.

Even though such an external transformation language may be using the general purpose language constructs and thereby reinventing the wheel, there is still another threat that the transformation language is inventing it slightly different. As an example, an assignment in QVT-O is written as := instead of = like in most C-based languages like Java or C#. As a reason, QVT-O is extending the syntax of OCL which already uses the = operator for equality (where C-based languages use the == operator). This difference in the syntax might seem unimportant as the learning effort is the same for both syntax variants. In fact, it is not because the C-like version is the version most developers are used to. As an immediate consequence, developers developing in both QVT-O and a C-based language like Java or C# can get confused about when to use which version of an assignment. Especially developers developing mostly in that C-based language are likely to end up trying to specify an assignment statement in QVT-O using the =-operator. As this is a valid statement in QVT-O, the compiler does not throw an error and the developer starts wondering why the transformation is not producing the expected output. The root of the problem is hard to find, as there is only a colon missing. Such effects can be annoying and thus both increase the maintenance costs and lower the acceptance of the model transformation language among developers.

5.3. MTLs as Internal DSLs

Internal DSLs are domain specific languages that use the syntax of another language, commonly referred to as host language. Most internal DSLs use dynamic or scripting languages as their host language. The obvious benefit from this procedure and most often the reason to use an internal DSL is that the tool support of the host language can be used. Thus, neither a parser nor interpreter or compiler have to be built. All this can be reused from the host language which makes an internal DSL much easier to implement. An example is the trace analysis languages from Barringer and Havelund. Whereas the language `Ruler` is implemented as an external language, the language `TraceContract` is implemented as an internal DSL. Although `TraceContract` offers greater functionality and easier adaptability, its implementation is close to a magnitude smaller in size [BH11].

In the example of Barringer and Havelund, the authors referred to their language as a shallow internal DSL. This means that the DSL is actually exposing normal language features. In the case of `TraceContract`, the pattern matching from Scala is used for the trace analysis, also.

The approach of a shallow internal DSL exposing features of the host language can also solve the dilemma mentioned at the beginning of this chapter. By including the features of the host language, an internal DSL might make use of these features to express the parts of the abstract syntax that are responsible for the Turing completeness. In most of the cases where a language is Turing complete, there is only a part of the language that makes the Turing completeness. Consider for example C# without any class members,

statements or expressions. What remains is the namespace and class structure, even including inheritance and generic classes. However, this remainder is not Turing complete. The Turing completeness is introduced foremost by the expression syntax.

A DSL, whether internal or external, always needs a semantic model behind it [Fow10] which is also referred to as the abstract syntax. Especially internal DSLs easily fall into the trap of not creating a semantic model behind the scenes but perform operations directly. In his book, Fowler calls this a command-query syntax which does not fulfill the definition of a DSL, since a DSL always needs such a semantic model. Having said that, it is not specified how this model could look like and what the metamodel is.

In case of an internal DSL for an object-oriented language, the metamodel consists of classes and the model is mostly created by method calls and afterward represented by instances of these classes. In C#, the references of these classes are represented by properties. However, unlike typical metamodels conforming to meta-metamodels such as Ecore, the type of these properties is not restricted by either other metaclasses or primitive objects. They can be typed with arbitrary types. This includes types that wrap Turing complete functionality, such as interfaces or functions[2]. As C# has the language feature of delegates, properties can easily be typed with functions that may contain the Turing complexity.

If, however, the host language has no language features like function types and lambda expressions, this procedure can still be done by specifying the Turing complexity through method polymorphism which is also available in languages like Java that do not support functions as objects. In this way, the functions are specified through inheritance of certain classes and overriding these methods. Both attribute specification through inheritance and by simply setting attributes in the form of lambda expressions can be combined.

The traditional arguments against shallow internal DSLs are that such languages are the lack of analyzability (due to the used Turing completeness of the general purpose host language) and the lack of conciseness (which is an issue of all internal DSLs). Assuming a the Turing complexity requirement, the analyzability argument can be invalidated as the domain itself is Turing complete and thus lacks analyzability. However, as the Turing complexity is hidden in some attributes, the semantic model indeed can be analyzed ignoring just these attributes. The latter lack of conciseness also has the consequence that domain experts must understand the host language and are thus unable to read the DSL in many cases. Hence, the DSL is losing one of its greatest benefits, at least according to [Fow10]. However, also external DSLs do not necessarily need to be understandable by domain experts, just because they are dedicated languages, especially as the domain is, in contrast to usual domains in MDE, Turing complete.

However, internal also have drawbacks when compared with external DSLs. The requirement that the language must be embedded in the host language yields a worse conciseness. How badly the conciseness is affected by the host language depends on both the host language and the nature of the internal DSL. Usually, dynamic or functional programming languages are better suited as host languages than for example Java or C# as they offer a flexible syntax and thus allow for more concise embedded DSLs.

The lack of conciseness yields a threat of losing one of the most important properties that a DSL can have: Its easy understandability by domain experts. Where users of an internal DSL have to be familiar with the (mostly complex) syntax and abstractions from the host language, users of an external DSL only have to know the syntax of that DSL. However, model transformations describe an execution semantics that has an intrinsic complexity and thus, it is questionable how external languages can manage to specify

[2]However, Ecore offers an extension point by assigning properties the type *EJavaObject*

model transformations in a concise manner. This is an open research question and will be discussed later in this thesis.

But it is not only the conciseness. Internal DSLs cannot control the parsing process and thus, developer may create statements that are invalid in terms of the intended language. As for example, if the internal DSL is using method calls to build up its semantic model, these method calls might be encapsulated in a conditional statement where the condition e.g. relies on some configuration parameters. As it is in general impossible to evaluate these conditions at compile time, such cases make static code analysis of the internal DSL hard. As a consequence, maintainability measures such as metrics are hard to compute, as it is difficult to derive the semantic model at compile time. At least, it can be detected that static code analysis can parse the semantic model by means of static code analysis. For example, if the method calls to make up the semantic model are not encapsulated in conditional statements or loops, static code analysis can conclude that these method calls are made exactly once. As we can assume that the static code analysis knows the effects of such method calls to the semantic model, the semantic model can be parsed statically.

5.4. Conclusions

Model transformation can be seen as a domain in the sense of model-driven engineering. However, model transformations need to be Turing complete, thus making the domain Turing complete. Thus, also the language to support model transformations must be Turing complete and hence, the definition for domain-specific languages from Fowler [Fow10] must be extended for this purpose to allow Turing complete languages. Although external DSLs can offer a syntax easier to read and thus understandable by domain experts, they are likely to reinvent mechanisms already known from general purpose languages in a slightly different way. This yields maintenance problems as developers face several different versions of the same concept.

Internal DSLs can utilize their general purpose host language and use it to express the Turing complete parts of a semantic model. In this way, the language features of the host language can be reused to specify the Turing complexity involved in the model transformations. With its domain focus, the internal DSL remains to specify the structure of the semantic model. This can be easily accomplished as the remainder is not Turing complete.

6. Model Transformation Problems

Besides other aspects like its purpose, the usability of a transformation language depends on the background and the preferences of its users [SK03a]. However, there are few dedicated transformation developers as most developers still develop in general purpose code. Thus, in order to provide better support for model transformation developers, it is crucial to understand where the problems of model transformations are when dealing with them using these general purpose languages. These problems then lead to functionality that is to be supported by MTLs.

The following sections introduce some issues that arise when performing typical transformation tasks. This list does not claim completeness. Instead, it is based on observations on a row of small and middle-sized model transformation projects. It is also discussed how the standardized languages QVT-O and QVT-R use language features to overcome these issues. In these sections, the general assumption is that code is generated for all input and output metamodels of a model transformation and thus model elements can easily be represented by objects of classes representing the metaclasses of the metamodel. We further assume that it is possible to modify the code generation process to allow improvements for developing model transformations e.g. by extending these classes with some code required by any pattern.

6.1. Correspondence & tracing

6.1.1. Problem description

Elements in the target model of a model transformation usually do not depend on all of the input model elements. In this way, the target model elements depend on the model elements of the input models that influence them. In some cases, this dependence may form a more or less obvious correspondence. The easiest example is a copy transformation where each of the target model elements corresponds to its original model element. The original model element and the corresponding element form a relation which is why some MTLs including QVT Relations name their top level language constructs relation.

Transformation developers usually need to query these relations that are somehow defined in the transformation. The support for this type of queries is generally recognized as trace. In many MTLs including QVT-O this trace is explicitly used to get a corresponding element for a given input model element.

6.1.2. Solutions in general purpose code

To save trace links, usually hashtables are used. The usage of the hashtable also yields the advantage that the hashtable can also serve as tracking mechanism to see which elements have been transformed. However, hashtables usually do not allow more complex queries. Furthermore, this procedure requires some kind of bookkeeping from the developers, which eventually becomes confusing.

As these correspondence links have to be set up manually, there are only set up where they are required. However, as requirements change, setting this correspondence might be necessary for other objects as well. This yields a drawback of creating the correspondence link for just some of the elements as the change impact of extending the transformation can be large.

6.1.3. Solutions in QVT-O and QVT-R

Being an entirely declarative language, QVT-R does not have an explicit support for tracing as in this section. Instead, these problems are hidden by the transformation language. The tracing functionality is hidden in the complex pattern matching in QVT-R where QVT-R can declare a constraint that two objects have a correspondence according to another relation.

On the other side, QVT-O has a powerful and explicit trace support with the `resolve` keyword. This trace support is even bidirectional with `invresolve`. Several modifications like `resolveIn` or `resolveOne` exist to support different type of tracing queries.

6.2. Cyclic object models

6.2.1. Problem description

Considered as an object graph, many models contain cycles. However, where trees and acyclic graphs have gained a good popularity in computer science, cyclic graphs have not. This might be because as soon as a graph is cyclic, one cannot traverse it without tracking which parts of the graph already have been traversed in order to prevent an endless loop. In general purpose code, this is typically done using lists of the items already visited.

As the metamodels of input models can be very complex, it might contain a lot of cycles. Each of these cycles might cause an endless loop, unless handled with a tracking mechanism. Thus, the transformation developer ends up with an amount of these tracking mechanisms.

Furthermore, inheritance involved in a complex metamodel can cause that developers are hardly aware of these cycles. This yields the risk that a cycle is forgotten and causes an endless loop. It also causes the risk that a newly introduced tracking mechanism remains undocumented and another developer that is asked to maintain the code does not see its necessity.

6.2.2. Solutions in general purpose code

To prevent cycles, transformation developers using general purpose languages usually end up using lists of elements already visited. However, if also a tracing mechanism is used, this also can be utilized to prevent endless loops.

6.2.3. Solutions in QVT-O and QVT-R

Much like the tracing support from the last section, the transformation engine is responsible to call each QVT-R transformation relation once at most.

However, QVT-O does not have a dedicated support to call each mapping only once.

6.3. Inheritance

6.3.1. Problem description

Most metamodeling frameworks like for example EMF [MEG+03] allow inheritance. Inheritance relations are used to established an "is-a" relation between elements, in MDE between metaclasses. As a result, instances of the super class are substitutable by instances of the subclass. Often, the subclasses form different ways to fulfill the tasks assigned to the superclass. Thus, to transform the elements of the superclass, it is necessary to know the true subclass of this element.

The most obvious solution to check the elements for their type is considered to violate good object oriented design as it involves explicit type checks that yield a maintenance problem. The proposed way to solve this problem with a good OO design is to utilize method polymorphism, i.e. the derived classes of some base class override certain methods. However, this usually requires the availability of methods to be overridden which means that the model explicitly supports a certain model transformation task. In many scenarios, this is not even possible as the code for the models is not accessible.

However, even if the code is accessible, a transformation only describes a single aspect and thus, the code for the classes in a metamodel should not contain code for a specific transformation. As we assumed the code representing the metamodel to be generated, we cannot assume this code to contain virtual methods and implementations to support a certain transformation task.

6.3.2. Solutions in general purpose code

In object oriented design, the visitor pattern [ERRJ95] has been introduced to overcome this issue. Metamodels rather seldom change and thus, it is a promising approach to build a visitor pattern for each of the inheritance hierarchies appearing in the metamodel. With these visitor patterns, a subtask of a model transformation can be implemented as a visitor, thus avoiding type checks. To be used for model transformation, the *visit* method only has to be changed to return an instance of a generic type T.

In order to use a visitor pattern, the model code has to support it. As we assumed that the code generation process for the metamodels can be modified, it is possible to generate code to implement a visitor pattern on the models. Even in case that the model code is not accessible, Palsberg and Jay already already showed that the *accept*-method of a visitor pattern can be omitted when reflection is available [PJ98].

However, in contrast to many general purpose languages like Java or C#, most metamodeling frameworks even allow multiple inheritance. As a consequence, a metaclass may not be part of a single inheritance hierarchy, only. Instead, there might be many of these hierarchies. Furthermore, it is unclear which inheritance levels are important for a specific transformation task. The approach to generate the visitor pattern for all metaclasses that are leafs in the inheritance hierarchy might lead to a manageable amount of different visitor patterns, but the implementation of the visitors might suffer as many of the *visit* methods point to the same code. Thus, generating visitor patterns for each and every occurrence of inheritance hierarchies is not feasible.

The approach suggested by Palsberg and Jay is more flexible as no *accept* methods of visitor patterns have to be implemented in the model code. Instead, the metaclasses involved to fulfill a certain transformation task can be selected per task. However, the measurements in the article show that it is also very slow, mainly of the poor performance of invoking a method through reflection. For model transformations executed at development time, as most of model transformations are, maintenance usually is more important than

performance. This is because model transformations can be executed as preprocessing in a nightly build.

6.3.3. Solutions in QVT-O and QVT-R

Again, QVT-R solves this problem with a complex pattern matching by specifying which elements should be transformed. By describing a procedure how to create copy transformations using QVT-R, [GW08] shows how to use the pattern matching to support inheritance problems. However, solving this issue with patterns requires a very special paradigm that many developers do not familiarize with.

QVT-O has a dedicated support for such inheritance related issues. These can be handled by disjunct mappings. With these disjunct mappings, it is further possible to use guards which makes the approach more flexible than just supporting inheritance issues. However, a disjunct mapping must not contain any statements. Thus, to avoid duplicate code, statements common to a super class must be created separately in an abstract mapping. Thus, the inheritance must be denoted at two different places: In the disjunction of the disjunct mapping and in the mapping itself extending the abstract mapping.

6.4. Patterns

6.4.1. Problem description

Model transformations often face problems that require to transform an element out of tuples of other elements. An example can be where a model element can represent a context of another element. As the context may serve for multiple other elements and the original element is not aware of any context, the transformation has to match these model elements, for example using a more complex pattern matching.

6.4.2. Solutions in general purpose code

In general purpose code, complex pattern matching based on tuples of elements is not directly supported. Instead, the developer must create these tuples and filter them on his own.

6.4.3. Solutions in QVT-O and QVT-R

QVT-R has a great intrinsic support for specifying complex patterns as transformations are specified only through patterns.

In QVT-O, a mapping can have multiple input elements. However, beginning from the second argument, the latter arguments serve as parameters. Implementations of QVT-O do not create tuples to match these parameters automatically.

6.5. Optimization Tasks

6.5.1. Problem description

Some transformations, especially often in-place transformations, require to optimize the model in terms of domain specific optimization criteria such as cost heuristics. The transformation has to output not just any valid result but the result that best suits these optimization criteria, e.g. has the lowest cost.

Such optimization tasks can include some sort of backtracking mechanism but do not have to. Sometimes, it is possible to solve the optimization task entirely by means of

heuristics. However, retrieving an applicable heuristics and method to be sure that certain requirements are met is often very time-consuming. Depending on the size of the models and how often the resulting transformation is executed (and thus how valuable development effort is in comparison to the execution times), it can be much cheaper to simply run a brute-force algorithm instead of providing special algorithms to solve the transformation task in a deterministic manner.

However, optimizations are not limited to model transformations. Rather, optimizations can be a part of multiple processes, including business logic processes. As optimization problems usually are NP-hard, most optimizations tasks are solved by general purpose code. As a reason, general purpose languages provide the flexibility to use case-specific polynomial time approximations.

The idea of optimization tasks within model transformations can be extended to the idea to split the properties of a model element. Some properties are specified using a DSL and others are automatically inferred by an optimization process in order to minimize some cost function like the response times of the resulting software system.

In contrast to optimizations that are part of a daily business, optimizations running on models to set certain parameters run relatively seldom, operating on rather small problem sizes. However, such model optimizations may be changed rather often as new models may require entirely new optimization tasks. As a consequence, the performance of how these optimizations tasks are executed gets less important meanwhile the effort of specifying and maintaining such a model optimization becomes critical.

6.5.2. Solutions in general purpose code

Developers in general purpose code have to implement optimization tasks by manually implementing a suitable algorithm. Usually, developers attribute to some common optimization problem like bin-packing or the traveling salesman problem and use the known approximation algorithms to solve their optimization in a reasonable execution time. However, the process of finding an appropriate well known optimization problem and fitting it to the actual optimization task is both time-consuming and complicated.

If no suitable known optimization problem can be found, developers have to implement an own algorithm. The easiest algorithm to implement is usually a brute force algorithm. The downside of this approach is that the execution time of such brute force algorithm is deterministic exponential whereas for many known optimization problems, approximations exist that achieve a low polynomial time complexity.

For such situations, frameworks like NMF Optimizations [Hin13] exist that allow a concise specification of optimizations that can then be solved by brute force algorithms.

6.5.3. Solutions in QVT-O and QVT-R

Neither QVT-O nor QVT-R include support for non-deterministic optimization. However, QVT-O offers functionality to clone objects, which might be helpful when performing such optimization tasks. Furthermore, if there was a framework that supported these tasks, transformation developers would have difficulties to integrate this framework into the transformation. In QVT-R, this is impossible. Some QVT-O implementations offer ways to access black box functionality that can be implemented in a general purpose language like Java.

More general, if a model transformation task includes an optimization task but the (external) model transformation language does not support optimization tasks, specifying such a transformation gets very complicated. As external MTLs like QVT-O and QVT-R have

a limited set of features that are dedicated for their domain, applying them in scenarios that are not foreseen by the language developers yields a huge maintenance problem. Unlike general purpose language features, language features of an external domain-specific language ought to have a domain-specific semantic. Assuming that MTLs are Turing complete (see chapter 5), this argument is attenuated. For example, QVT-O also reintroduces language features of normal general purpose languages. The threat for the transformation developer is to try expressing the optimization problem with the abstractions for model transformations like mappings. This creates a set of workarounds that are harmful for the maintenance of the transformation as it is unclear which mappings belong to the transformation and which of them are just used as workarounds.

The usual solution is to use extension points of the language to e.g. implement rules in a mainstream general purpose language like Java or C#. An external MTL can reference this general purpose code, while an internal DSL can provide means to inline this code into the model transformation. However, as the transformation is then implemented in multiple programming paradigms, this hampers the solution consistency.

6.6. Higher-Order Transformations

6.6.1. Problem description

Higher-order transformations are transformations that either accept other model transformations as input for the transformation or produce a model transformation as output, much like higher-order functions in functional programming. Higher-order transformations proofed their usefulness in many cases [TJF+09].

6.6.2. Solutions in general purpose code

Source code of a general purpose language can also be seen as textual representation of a model. Approaches that represent this are available for several popular general purpose languages. For Java, there is JaMoPP [HJSW09] whereas the metamodel of such a code representation in .NET is part of the Base Class Library (with the class structure of .NET used as the meta-metamodel) which is in turn part of the .NET framework. These classes for this representation are in the namespace `System.CodeDOM` and provide a language-independent code model. However, Microsoft is currently working on Roslyn[1], which includes a more fine grained source code model.

Most of these models can also be used to generate code, thus making a higher-order-transformation that outputs a model transformation similar to an ordinary M2M-transformation task. However, as usual general purpose code lack of suitable high-level abstractions, such model transformations tend to be very complicated and verbose. As a reason, they do not only have to transform the semantics of the input model, but also have to unroll the transformation abstractions to general purpose code.

Analyzing a transformation written in general purpose code is very hard as the structure of the transformation is dependent of the transformations architecture. This could be anything depending on the personal flavor of the developer that created the transformation. Analyzing the code to derive the structure also suffers from the halting problem, i.e. the question whether such an analysis does even terminate for each and every model transformation is undecidable.

[1] `http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx`

6.6.3. Solutions in QVT-O and QVT-R

The OMG has only standardized *how* transformations work and the features of the languages QVT-O and QVT-R. It is not specified *what* transformations in that languages are [BBG+06]. Other MTLs that are not standardized but support higher-order transformations include ATL. In ATL, a model transformation itself is a model and thus can be generated as an output of another transformation or analyzed. In theory, this procedure is also possible for QVT-O and QVT-R but there is no such implementation yet, besides generating the model transformation code with e.g. a M2T-transformation.

In theory, the argument with the analyzability holds for the analysis of every model transformation language. However, model transformation languages ought to have an abstract syntax (same as every domain-specific language [Fow10]). A higher-order transformation that reads another model transformation can restrict on a non Turing complete part of this abstract syntax and thus is analyzable.

6.7. Transformation Composition

6.7.1. Problem description

Modularity is a key factor for reusability and adaptability in software development and thus, the need for composition mechanisms has already been denoted in [SK03a]. The composition of model transformations is usually divided in internal and external transformation composition [Kle06]. Internal transformation composition means that a model transformation is composed of multiple transformations of the same language whereas external composition allows to compose multiple model transformations of different languages. Thus, where internal transformation composition must be supported by the transformation language, external transformation composition is about interoperability between multiple transformations possibly of different paradigms. Hence, despite its importance, external transformation composition is out of the scope of this chapter.

Internal model transformation composition can further be divided to transformation chaining and transformation composition. Transformation chaining means that multiple transformations that may be written in entirely different languages are executed one after another. Thus, model transformations must be able to call other model transformations as part of them. Transformation composition means that a model transformation is created from parts of another transformation or many of them. This is usually achieved by reusing parts of the transformation such as transformation rules.

6.7.2. Solutions in general purpose code

Most general purpose languages evolved over far more time and with much more intensive research among them than any domain-specific languages will ever be, also including model transformation languages. As modularity and thus reusability have always been important in software development, the usual mainstream general purpose languages including Java and C# provide great support for modularity. Thus, transformation chaining is easy to accomplish as the transformation is mostly represented by methods that can easily be chained.

The extension of an existing transformation is more difficult. The usual way to extend an existing implementation in object-oriented design is to use polymorphism or program against interfaces instead of concrete classes (or a combination of both). This yields the trap that only code can be replaced or extended that is separated in virtual methods or hidden behind an interface. Whereas methods in Java are virtual by default, in other languages like C# they are not. Furthermore, it is not possible to change the behavior of parts of a method. As a consequence, a model transformation in general purpose code can only be extended where the developer set dedicated extension points.

6.7.3. Solutions in QVT-O and QVT-R

The support of QVT-O for transformation composition has been presented in section 3.2.1. The specification of QVT-R does not include methods for transformation composition.

6.8. Testing

6.8.1. Problem description

Humans and thus transformation developers make mistakes. In general, the earlier such a mistake is detected, the better and cheaper it is to repair this unintended behavior. For this purpose, whenever developers can make mistakes, testing is required to hopefully reduce the amount of mistakes that make their way all through the development process. Using automated testing, tests are written that mostly test small portions of a software development artifact and verify that it is producing the expected result. These tests can automatically be executed causing an alarm bell to ring as soon as a code change makes these tests fail. The ability of good testing to early detect bugs makes testability a strong part of the overall maintainability of a software artifact.

As a consequence, software-development processes like test-driven development (TDD) have gained popularity in recent years. In TDD, the tests are written before the implementation to ensure that the code is properly tested and the tests do not depend on the implementation.

6.8.2. Solutions in general purpose code

For most modern main stream general purpose languages, there are testing frameworks available, often more than just one. The most prominent testing frameworks are JUnit[2] for Java and NUnit[3] or MSTest[4] for .NET. These test frameworks usually allow to specify unit tests that can set up a test environment, perform some operations under test and check the results, usually by using assertions on the result to compare the actual result with the expected result.

Setting up the testing environment in this task involves creating stubs or mocks for all components that the component under test needs in order to separate the component under test as much as possible from the rest of the software system to allow for isolated testing conditions.

In industry, the development teams in large companies like for example Microsoft consist of as many dedicated testers as developers. This may show the importance of testing in software development projects.

These automated tests are often included in the build process, such that a build automatically triggers the execution of tests, thus automatically causing those alarm bells to ring in case that tests have failed.

Furthermore, some tools like Pex[5] [TDH08] allow automated white-box testing and thus automatically create test cases for a given code.

[2] http://junit.org/
[3] http://www.nunit.org/
[4] http://msdn.microsoft.com/en-us/library/ms182469.aspx
[5] http://research.microsoft.com/en-us/projects/pex/

6.8.3. Solutions in QVT-O and QVT-R

Both QVT-R and QVT-O do not have dedicated support for testing model transformations included. Thus, it is only possible to test the complete model transformation as a black box. However, conducting tests like this also needs a proper foundation for model comparison. As a reason, the specification of a model transformation may not define the order in which certain model elements need to be generated. As a consequence, the expected model and the resulting model cannot just be compared for whether they match exactly (which in turn is also difficult as equal models do not necessarily have the same serialization).

However, recent publications proposed a testing framework for QVT-O that is entirely implemented as QVT-O transformation where tests for QVT-O transformations are also specified as QVT-O transformation [CFM10]. This has the advantage that transformation developers do not need extra tool support as the IDE supporting the development of the transformation automatically also supports testing these transformations.

Other approaches like [FSB04, WKC08, SBM08, HLG13, WS13] aim to automatically generate test models to support testing the transformations. However, as soon as a transformation language is Turing complete, this cannot be achieved in general as an immediate consequence of the Halting problem. Furthermore, most testing of model transformations is still done by using black-box-testing, i.e. the whole transformation is tested as it instead of testing parts of the transformation separately.

6.9. Conclusions

Writing model transformations in a mainstream general purpose language like Java or C# is not a maintainable option as such languages lack suitable high-level abstractions to support developers specifying their transformation. The lack of these abstractions yields several problems that need to be dealt with. All of them can be handled by general purpose languages. However, these solutions are not always maintainable solutions.

These problems that are hard to solve with general purpose code include the following:

- Model transformations establish a correspondence between elements of the target model and the source elements that they are representing, which often needs to be queried.
- Seen as a graph, models may contain cycles and thus developers must developers must prevent endless loops in the model transformation.
- Models often contain inheritance hierarchies so that developers need ways to transform polymorph instances. Dynamic programming can solve this problem but only at the cost of losing the type safety and thus the IDE support.
- Model transformations may contain complex pattern matching.
- Especially in-place transformations may need optimizations as part of the transformation.
- Higher-order transformations need to read or write such model transformation languages.

However, model transformation languages must preserve some quality critical properties of general purpose code. First of all, model transformation languages must provide means for testing parts of a transformation rule separately. Furthermore, transformation languages must provide means to compose a model transformation of other model transformations. Solutions for QVT-O are available for these issues. For QVT-R, however, such support

does not exist yet. This may be caused by the fact that the computational model of QVT-O is very similar to the one used in Java and thus for example testing can be supported in quite a similar way.

Furthermore, specifying optimizations with current model transformation languages is supported by neither QVT-O nor QVT-R directly. Instead, transformation developers have to embed such optimizations in the transformation language which is hard to accomplish and yields inconsistent solutions.

7. NMF Transformations

This chapter presents NMF Transformations, an open-source framework to support model transformations to address the problems featured in sections 6.2 and 6.3. NMF Transformations consists of the framework NMF Transformations Core and the internal DSL NMF Transformations Language (NTL) built on top of it. NMF Transformations Core provides a metamodel for model transformations while NTL provides an internal DSL designed to use NMF Transformations within C# more easily. NTL further provides extensions to NMF Transformations to solve the problem of complex patterns, see section 6.4. Neither NMF Transformations nor NTL provide support for optimization tasks as in discussed in section 6.5. For this purpose, NMF offers NMF Optimizations which is described in [Hin13]. The problem of higher-order transformations is also not covered besides the general remarks on higher-order transformations made in section 6.6.

Although it is compliant to the Common Language Specification (CLS) and thus it is possible to use NTL in any .NET language, it is specifically designed for a usage in C#. Thus, the master thesis concentrates on the C# syntax. NTL allows to specify transformation models that conform to the abstract syntax of NMF Transformations Core. The metamodel for these models is Turing complete, although the Turing complexity is encapsulated in functions that are used as attributes of the transformation models. The structure of the metamodel itself is not Turing complete. However, the Turing complexity is intended as a consequence of the discussion from chapter 5. NTL tries to provide support that is as familiar as possible to C# developers. This is achieved by following the design guidelines of the .NET framework documented by Cwalina and Abrams [CA08] and letting transformation developers specify the population part of model elements in usual methods with access to a trace functionality.

Being part of NMF, NMF Transformations is a framework to create rule-based transformations using any .NET languages. NMF Transformations expects the input arguments as plain CLR objects. It cannot read models from files. However, to interoperate with e.g. EMF, NMF does contain a library and a code generator that is able to generate classes for the metaclasses contained in a Ecore metamodel (see section A.3). The code generator uses the `System.CodeDOM` namespace and thus, it is possible to generate this code in multiple languages. Further, the code generator uses attributes to allow other libraries to serialize and deserialize these models into an XMI format compatible with EMF (see section A.4).

First, section 7.1 introduces the abstract syntax of the transformation models. Next, section 7.2 explains the architecture of NMF Transformations and thus explains how the abstract concepts of NMF Transformations get an execution semantic. This is followed by section 7.3 that introduces the stage model before section 7.4 shows how to specify model transformations using NTL. In section 7.5, the drawbacks of NMF Transformations are explained to show what has not been achieved with this approach yet. Finally, section 7.6 concludes this chapter, summarizing NMF Transformations.

7.1. Abstract syntax

As discussed in [Fow10] and in section 5.3, an internal DSL like NMF Transformations needs a semantic model, an abstract syntax. However, the abstract syntax behind NMF Transformations is not always directly represented in code and especially we abstract from the decision whether a certain element is a class or an interface here for better clarity and will instead use simplifications. The changes from this simplified abstractions to the implementation model will be described later in this section.

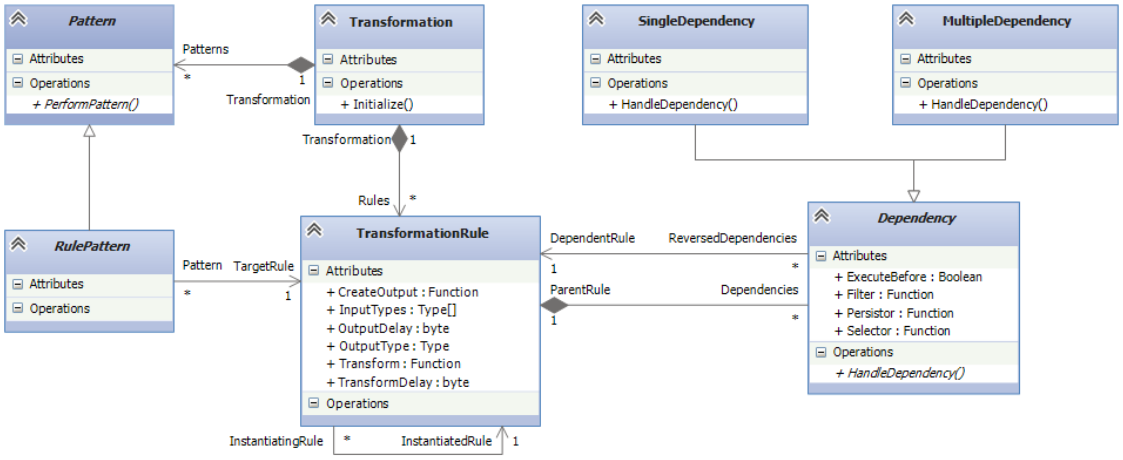

Figure 7.1.: The conceptual abstract syntax of NMF Transformations

The conceptual abstract syntax behind NMF Transformations is presented in a class diagram in figure 7.1. A transformation in NMF Transformations simply consists of transformation rules and patterns. Patterns can be tied to a transformation rule and thus have that transformation rule assigned to them.

Transformation rules may have dependencies and can instantiate other transformation rules. Transformation rule instantiation is a concept to support the transformation of inheritance hierarchies in either input or output models. Basically, a transformation rule can be marked instantiating for another transformation rule and thus create the output for this other transformation rule. To other transformation rules, this procedure is transparent, e.g. the correspondence created by calling the instantiated transformation rule is still available to the trace. These instantiation links can further be annotated with filter expressions to restrict the inputs where a transformation rule can instantiate this other transformation rule. By default, the computations are only filtered by the runtime type of the input arguments and simply use the first transformation rule that matches.

Transformation rules specify how their output is to be created, in case they are not instantiated, and how the inputs should be transformed to the output element. This is specified through the attributes `CreateOutput` and `Transform`, which are typed as functions. These functions make up the biggest part of the transformation where most the actual work can be done.

Transformation rules can further set a delay level, both for creating their output and for transformation. This output is used to inform the transformation engine that a transformation rule needs to generate its output in a second, third, or even later transformation pass, e.g. in order to have increased trace capabilities. More details to this technique will be presented in the section on the transformation stage model, section 7.3.

Dependencies are links between transformation rules. This enables the transformation engine to know which other transformation rules to call, together with the inputs for these transformation rules. They can either be single dependencies (causing to call another transformation rule once per handled input) or a multiple dependency (causing multiple other calls). Dependencies in general have a transformation rule that they depend on. They may further filter the computations that trigger them, may specify a selector that selects the input of the dependent transformation rule and a persistor that persists the output of the dependent transformation rules together with the output of their parent transformation rule. Once again, these attributes are typed as functions, following the discussion from chapter 5.

Furthermore, a transformation may specify patterns. These form an extension point where the metamodel can be extended. Such extensions are easy to implement, as the metaclasses of an internal DSL are usually mapped to classes. These patterns may rely on a certain transformation rule. Instead of referencing e.g. a C# class to specify the implementation of the extension, extensions are meant to be made by extending model by these pattern implementations.

As one can easily see, the model itself is not very complex, with only seven metaclasses in total. Instead, the complexity is hidden in the attributes which are typed as functions. The inner structure of these functions is not modeled in the abstract syntax of NMF Transformations. Instead, the execution semantic of .NET is reused for this, same as the syntax, because developers are used to it. Following Fowler 5, the transformation model can be considered an adaptive model that represents an alternative computational model and incorporates imperative C# code to specify how to for example select the inputs for a dependent transformation rule. However, the greatest incorporation is the `Transform` method that incorporates a whole method, thereby most often specifying the main parts of the transformation. By some sense, NMF Transformations can be considered a language to specify when to call which `Transform` method with which parameters. Thus, the incorporated imperative language is not only an extension point to express what could otherwise not be modeled with the abstract syntax, instead the functionality represented by incorporated imperative language represent core parts of the transformation model. All that NMF Transformations adds to the C# language that gives NMF Transformations a flavor of a language is how transformations are specified in code.

The diagram in figure 7.1 also lacks of type safety, which is important since many of the .NET languages and especially C# are type safe (although extensions for dynamic languages exist in C#). Furthermore, the type safety has the huge advantage that it enables rich tool support of the IDE. Thus, NMF Transformations tries to save as much as possible of the type safety of C#. This is the main reason that the metaclass `TransformationRule` is actually divided into a plethora of classes in the implementation. These derived classes provide type-safe support. However, in some cases it is not possible to catch all type mistakes during compiler time. As an example, the consistency of transformation rule instantiations regarding the types of the input and output types of these rules can only be checked at runtime rather than at compile time.

The implementation model of NMF Transformations further includes some other classes that were not represented yet. Many of them remain hidden to transformation developers and are for internal use only. They are made public entirely for extension

purposes of the NMF Transformations framework itself and are thus moved into a separate namespace, following the guideline from [CA08].

An important exception to that rule are the classes related to the transformation context. First of all, there is the transformation context. NMF Transformations is actually aware that model transformations do not need to run only sequentially. At least, multiple model transformations may run with the same model transformation, but on different inputs, achieving simple task parallelism. NMF Transformations supports this use case by introducing a transformation context. This transformation context contains all information that is used for a specific pass of the model transformation. The transformation itself is stateless (except for caching that is not yet implemented). Thus, a model transformation can run with different inputs in parallel, sharing the transformation initialization. This procedure has also advantages in a sequential world, as multiple transformations can run one after another without exposing the transformation initialization (and only dispose the transformation context).

Representing the state, the transformation context is responsible for the most important functionality in the transformation, such as calling a transformation rule with a specific input and the trace functionality. To enhance reusability, the trace functionality is moved into a separate component and hidden behind a separate interface, same as the transformation context.

Another important concept of the implementation model that has not been presented yet is the computation. Computations represent that a transformation rule has been called within a transformation context for a given input. As transformation rules may only be called once per context and input parameters, they are unique for these parameters. But the computation object also holds the state of such a call, most importantly, the output. However, as creating the output may be delayed, the output is not always accessible. Computations also provide events to inform clients that either the output is created or the main computation, i.e. the `Transform` method, has been called.

Furthermore, the metaclass `TransformationRule` is renamed to `GeneralTransformationRule` in the implementation model. This practice is following the naming guidelines of [CA08] to reserve the best names for the classes that will be used most frequently.

As a consequence, the same diagram with the concepts represented by the abstract syntax of NMF Transformations in the implementation looks a bit different. As patterns are marked abstract, they are turned into interfaces for the implementation to avoid any influence to inheritance hierarchies. Furthermore, there is an additional extension point introduced as an interface for the dependencies. This allows developers to extend the transformation rules with own dependencies which actually serve as hooks to own code. The functions `CreateOutput` and `Transform` have been moved to the computation class to allow developers extend the computation class of their needs with additional properties shared by these functions and let these methods depend on them. If this is not required, as in most cases, these methods can be specified together with the transformation rule in a `TransformationRule`. However, the class `TransformationRule` is not shown in figure 7.2 for better clarity.

Figure 7.2 shows a class diagram containing a fragment of this implementation model with the dependencies corresponding to the diagram in figure 7.1, except for the patterns. The classes *Action*, *Predicate* and *Func* are delegate types and represent functions and can contain arbitrary C# code (as well as any code in other .NET language).

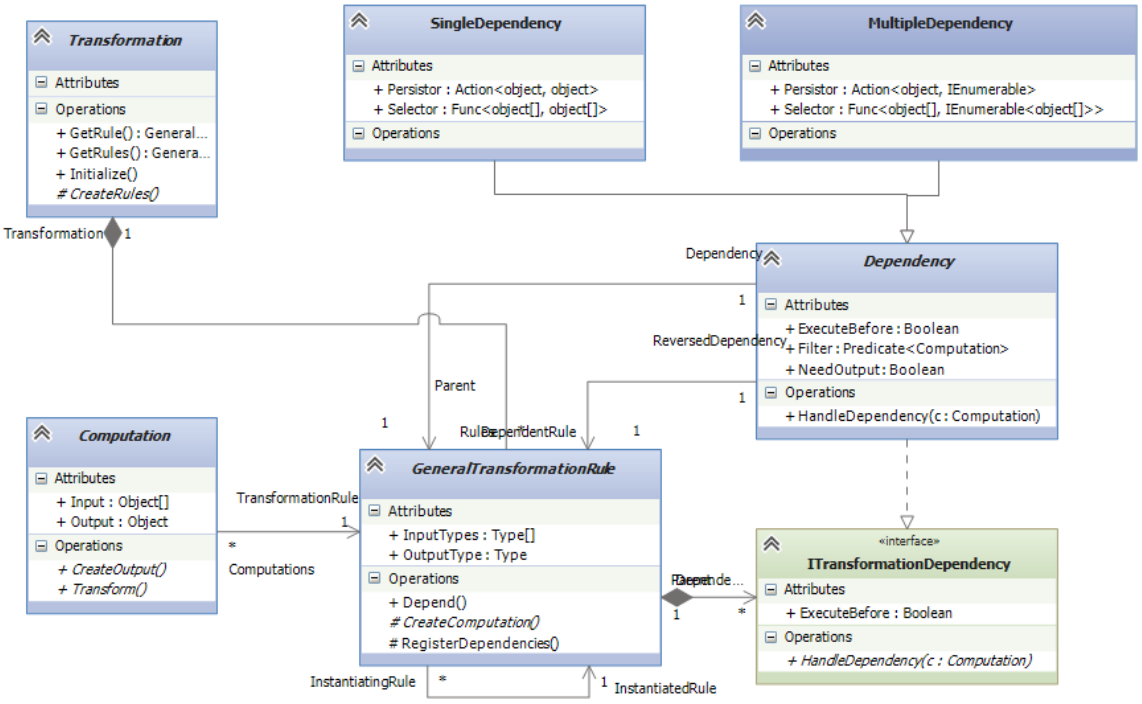

Figure 7.2.: The abstract syntax of NMF Transformations (fragment)

7.2. Architecture of NMF Transformations Core

From the large, NMF Transformations is divided into the framework NMF Transformations Core and the internal DSL NTL built on top of the framework. The framework and the internal DSL are well separated into two different namespaces. While the classes that make up NMF Transformations Core reside in the namespace *NMF.Transformations.Core*, the classes of NTL reside in the namespace *NMF.Transformations*. Furthermore, the implementations of NMF Transformations Core and NTL reside in different assemblies. The hierarchy of the namespaces already offers a more general concept. Although NMF Transformations Core has no reference to NTL, the best names are reserved for the classes that developers are most likely to work with. As the most prominent example, the abstract concept of a *transformation rule* from NMF Transformations Core is implemented in the class `GeneralTransformationRule`. This is done to reserve the name *TransformationRule* to a class that developers are more likely to work with, namely the type-safe simple transformation rule from NTL that is much more specific than `GeneralTransformationRule`.

This general architecture is illustrated in figure 7.3. This class diagram also shows the relational extensions that are presented in section 7.4.6. However, NTL actually supports the relational extensions with own convenience methods that depend on the relational extensions as they represent a facade of them. Thus, NTL and the relational extensions are shipped in the same assembly. However, NMF Transformations Core is shipped in a separate assembly. This general structure of having a framework and an internal DSL where the framework does not depend on the internal DSL makes it possible to reuse NMF Transformations Core independent from NTL. This extensibility of NMF Transformations is discussed in more detail in section 7.4.9.

The architecture of NMF Transformations Core is divided into a static and a context-aware part. The static part consists of transformations and the transformation rules and patterns they consist of. Static means that a transformation and its rules are created and initialized independently from any models that need to be transformed. In figure 7.4,

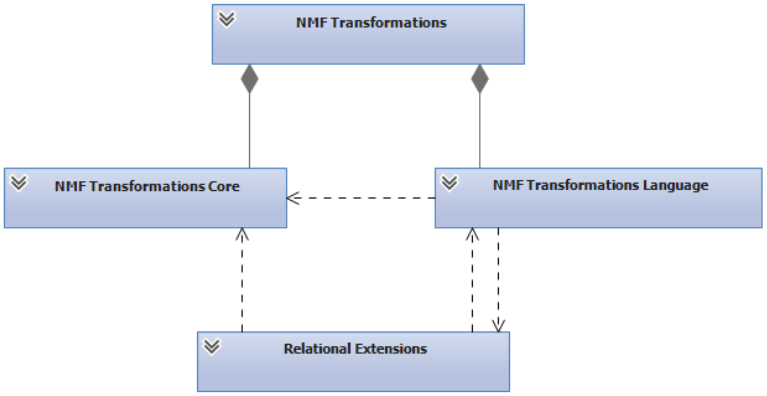

Figure 7.3.: The architecture of NMF Transformations from the large

the static classes of NMF Transformations are displayed on the left side while the context-aware classes are on the right hand of the diagram (patterns and pattern contexts are omitted in the diagram). As one can see in the diagram, the classes representing the transformation and its rules do not have any references to classes that depend on a context.

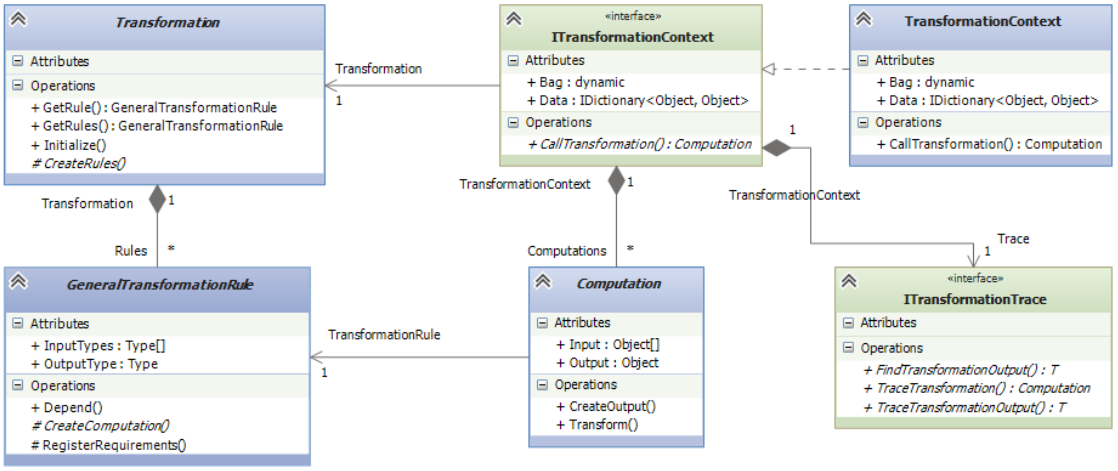

Figure 7.4.: Core concepts of NMF Transformations

The classes of the static part describe the transformation structure. This includes the dependencies between the transformation rules not visible in the diagram in figure 7.4. With these dependencies, the transformation engine is able to decide which transformation rule to apply in a specific situation. The transformation, its transformation rules and the dependencies form a semantic model for the transformation as described in [Fow10]. However, instead of having plain attributes of limited expressiveness (as for example strings), many references of this semantic model express an execution semantic that is represented in code as they represent functions. As in [SK03a], this code has to allow the semantic model for a Turing complete execution model. As Turing complete semantics are best expressed in general purpose languages, developers are used to, these references are set by implementing certain methods like the `Transform` method of a transformation rule or providing these Turing complete parts by specifying lambda expressions.

In contrast to the static part, the dynamic part is responsible to provide support when a transformation passes with a certain input. Thus, the transformation context represents the transformation pass whereas the computations represent that a transformation has been called within the context. Similar as a transformation consists of multiple transforma-

tion rules, the transformation context consists of multiple computations. However, usually there are a lot more computations within a transformation context than transformation rules within a transformation as most of the transformation rules apply to multiple input elements. To put an example, a simple copy transformation of the Palladio Component Model [BKR09] metamodel contains about 7000 computations but only 17 transformation rules (one for each metaclass within the Ecore metamodel except for *EGenericType* and *ETypeParameter* that are not transformed at all).

As the dynamic part is responsible for the pass of a transformation, it is also responsible for the computational model of the transformation. The transformation context implementation is hidden behind an interface `ITransformationContext`. The implementation of this class is responsible to decide how the transformation is executed. This interface has a default implementation `TransformationContext`. Many of the explanations given in this chapter regarding the execution of a transformation are dedicated to this particular implementation. One of these implementation details is the fact that transformation rules may only be executed once per context and input. In truth, this is specified in the transformation context implementation only. As this implementation is hidden behind an interface, it is easily possible to change this behavior, whenever e.g. it is not necessary to check whether there already is a computation with the given specifications.

Fowler calls such a model an Adaptive Model representing an alternative computational model [Fow10]. Indeed, transformation developers cannot see how the transformation is executed just by looking at the code for their transformation, which is basically a list of transformation rules. However, by hiding the implementation of how the semantic is executed behind an interface, the transformation developer can override the default behavior by putting a custom transformation context into the transformation engine. This e.g. allows easy separate testing of transformation rules as discussed in section 7.4.8.

The trace functionality is moved to a dedicated component in order to allow some applications to have access to the trace but not to the transformation context in order to provide a read-only access to the transformation. The trace functionality with its loads of convenience methods is also hidden behind an interface. This interface only consists of a few orthogonal trace methods. NTL as the internal DSL extends this trace by using extensions methods to provide a whole lot of convenience methods, given that NTL knows how the concepts of NMF Transformations Core are used to perform the transformation, e.g. that input type can be derived statically by the type parameters of the `GeneralTransformationRule`. Thus, NTL can provide the transformation developer a type-safe access to the trace functionality without knowing the underlying implementation.

7.3. Stages of the transformation

A model transformation in NMF Transformations runs in several different stages. First, the transformation is initialized, also initializing the rules and resolving dependencies between rules. This happens once and independent of the input of the transformation. The further steps only apply when the transformation is asked to transform a certain object. In this case, a context object is created for the transformation to ensure the transformation can be run threadsafe. Second, the transformation rules applied to each possible match. In this stage, also output for each computation is created, which in turn is recognized in the trace. However, some rules can delay the creation of their outputs. When all transformation rule dependencies are executed, these delayed outputs are created. Next, the main `Transform` methods of each computation are invoked, doing the main transformation work. The execution of the `Transform` methods can also be delayed. Finally, the transformation context is closed and the transformation is finished. These stages will be further explained in the subsequent paragraphs.

The stage model implementation of NMF Transformationsis specific to the `TransformationContext`, which can be replaced by another transformation context implementation. Another transformation context implementation might thus use a different stage model. However, the abstract syntax of transformation rules only allows the two methods `CreateOutput` and `Transform` that specify how the transformation is to be executed. Unlike the `TransformationContext` implementation that operates on plain transformation rules as `GeneralTransformationRule`, another implementation may further rely on additional interfaces or subclasses.

7.3.1. Initialization

Of all stages within a transformation, the initialization is the one that can be done without the transformation context. As an initialized transformation is reusable, the initialization stage may execute rather complex code as it is only executed once and thus not critical to the overall performance.

In the initialization stage, the transformation rules within a transformation are created and initialized, hereby resolving the dependencies to other rules.

7.3.2. Create Patterns

In this stage, the context object for a transformation is created and initialized. Together with the transformation context, also the pattern objects registered at the transformation are executed and pattern appliances are created. These are pattern objects aware of their transformation context.

7.3.3. Execute dependencies

This one of the main stages in the transformation. The transformation engine invokes the start transformation rule to transform the requested input to the requested output. This rule is by default inferred by the types involved in the `Transform` method of the transformation engine (see listing 7.2 for an example). By invoking this transformation rule, the dependencies of that transformation rule are being executed. These dependencies in turn invoke other transformation rules and following this procedure, (almost) all Computations are created that are required to fulfill the transformation request. However, some dependencies might rely on outputs of other computations that have been delayed. These dependencies are executed as soon as the required output is computed, which is in the next stage. The execution of these dependencies might, of course, trigger other dependencies.

If a transformation rule is invoked with a certain input, first the transformation engine checks whether there already is a computation with this input and the requested transformation rule. If so, this computation is returned. If not, the transformation performs the required steps to invoke the transformation rule. This roughly involves registering the computation for tracing, executing the dependencies of the transformation rule that apply before the computation, handling the computation and executing the dependencies that apply after the computation. Handling a computations involves steps like creating its output and inserting into the transformation order. However, transformation rule instantiation makes it a bit more complicated than that.

7.3.4. Create delayed outputs

Computations can delay their output in 255 levels. The advantage of this technique is that computations of all earlier delay levels (e.g. level 0 with non-delayed computations) are available for tracing purposes. This is possible, because Computation objects are always aware of their transformation context.

7.3.5. Transform

In this stage, the `Transform` method is executed. Similar to the creation of outputs, this can be delayed up to 255 levels. The `Transform` method is similar to the *populate* section of QVT-O with the main difference that in QVT-O, the populate section of a mapping is executed straight after the trace entry has been created. In NMF Transformations, the `Transform` method is executed after *all* outputs of any computations have been written to trace entries. However, the purpose is the same: In this method, the properties of the output objects ought to be initialized ("'populated"').

This stage only executes the `Transform` method of the computations in the order that has been derived in the previous stage when the dependencies where executed.

7.3.6. Finish Patterns

In this stage, the pattern appliances have a chance to do whatever they need to do in order to perform their task.

7.4. NMF Transformations Language (NTL)

This section introduces NTL, the internal DSL used to specify model transformations in e.g. C#. As the definition of NTL complies to the Common Language Specification, it is also possible to use NTL with other languages, both this may seem a bit odd for developers used to these programming languages.

The main purpose of NTL is to add a layer of type-safety to NMF Transformations Core. The classes in NMF Transformations Core always work with objects. NTL adds convenience methods utilizing the generics feature of C# to provide methods to specify dependencies in a type-safe manner. Furthermore, this section will also introduce the relational extensions. These help to specify complex patterns as introduced in section 6.4.

First of all, section 7.4.1 introduces how to specify model transformations at all in NTL. Afterward, section 7.4.2 shows the specification of transformation rules. Section 7.4.3 continues with the specification of dependencies between transformation rules. Section 7.4.4 gives an overview on the tracing functionality which is part of NTL. Section 7.4.5 introduces how to use transformation rule instantiation. Section 7.4.6 shows the relational extensions to NTLwhich provide support for more sophisticated dependencies. Next, section 7.4.7 introduces how to compose model transformations written in NTLbefore finally section 7.4.8 discusses how to test model transformations written with NMF Transformations/NTL.

7.4.1. Specifying Transformations

Model transformations in NMF Transformations are represented by the abstract class `Transformation`. The one and only abstract method within this class is `CreateRules` which returns a collection of transformation rules. However, as separating the definition of a transformation rule and the usage is potentially harmful to maintenance, transformation rules in NTL are inferred from the nested classes by using reflection techniques. Thus, transformations in NTL are represented by the class `ReflectiveTransformation` which inherits from `Transformation`. A new transformation rule is specified through inheritance from `ReflectiveTransformation`. The transformation rules of this class are then represented by the public nested classes of that transformation class that inherit from `GeneralTransformationRule`.

However, this procedure is breaking the framework design guidelines from [CA08], but it is actually done to avoid the maintenance problem if new rules are added to an existing transformation. Furthermore, it enhances the conciseness and readability of the resulting transformation. The reason that this guideline exists and why it also has a negative impact on the language is that it lacks of discoverability. When specifying a transformation, developers must come to the idea to create nested classes inheriting from some classes representing transformation rules. The IDE cannot guide him to do that. However, it has been an important design decision to put maintenance and conciseness over discoverability.

On the other hand, nested classes are compiled into metadata whereas usual code is compiled to Intermediate Language (IL) code. As a consequence, the structure of the model transformation is also preserved after the compilation as the structure is reflected in the assembly metadata. Internally, the `ReflectiveTransformation` class uses this metadata to infer the transformation rules within the transformation. In the same way, the transformation structure can be inferred by dedicated tool support.

Using this class as a base class, it is possible to write model transformations with the scheme listed in listing 7.1.

```
1 using NMF.Transformations;
2
3 public class FSM2PN : ReflectiveTransformation {
4         // Transformation rules as public nested classes
5 }
```

Listing 7.1: A model transformation using the ReflectiveTransformation class

The example creates a model transformation for the scenario in section 4.1. However, the transformation does not contain any transformation rule yet. Thus, when requesting to transform a finite state machine to a Petri Net, the transformation engine would throw an exception saying that no rule could be found to transform a finite state machine to a Petri Net.

```
1 FSM.FiniteStateMachine fsm = ...
2 var transformation = new FSM2PN();
3 var pn = TransformationEngine.Transform<FSM.FiniteStateMachine,
    PN.PetriNet>(fsm, transformation);
```

Listing 7.2: Invoking a model transformation

The transformation is invoked as displayed in listing 7.2. The `TransformationEngine` class also has some methods to transform a collection of objects or process one or many elements without creating an output. The latter versions can be used for in-place transformations. Furthermore, it is possible to transform an element with an existing transformation context. In this way, the transformation can make use of the trace functionality of another transformation.

In some cases, it is not feasible to specify all transformation rule of a model transformation in a single class, but refer to transformation rules defined somewhere else. To define model transformation rules across file boundaries is actually very easy as C# does allow to split the definition of a class among several files. However, if one wants to access transformation rules from other model transformation rules, or rules defined stand-alone e.g. in separate libraries, this is possible through the method `CreateCustomRules`. In this method, it is possible to load a collection of transformation rules. Developers can also reflect the transformation rules contained in other transformations, same as the `ReflectiveTransformation` does. This can be achieved by using the static class `Reflector`.

However, these transformation rules are then wrenched from their context, i.e. they have a different transformation assigned to them. This is fine as long they do not rely on this transformation.

7.4.2. Specifying Transformation Rules

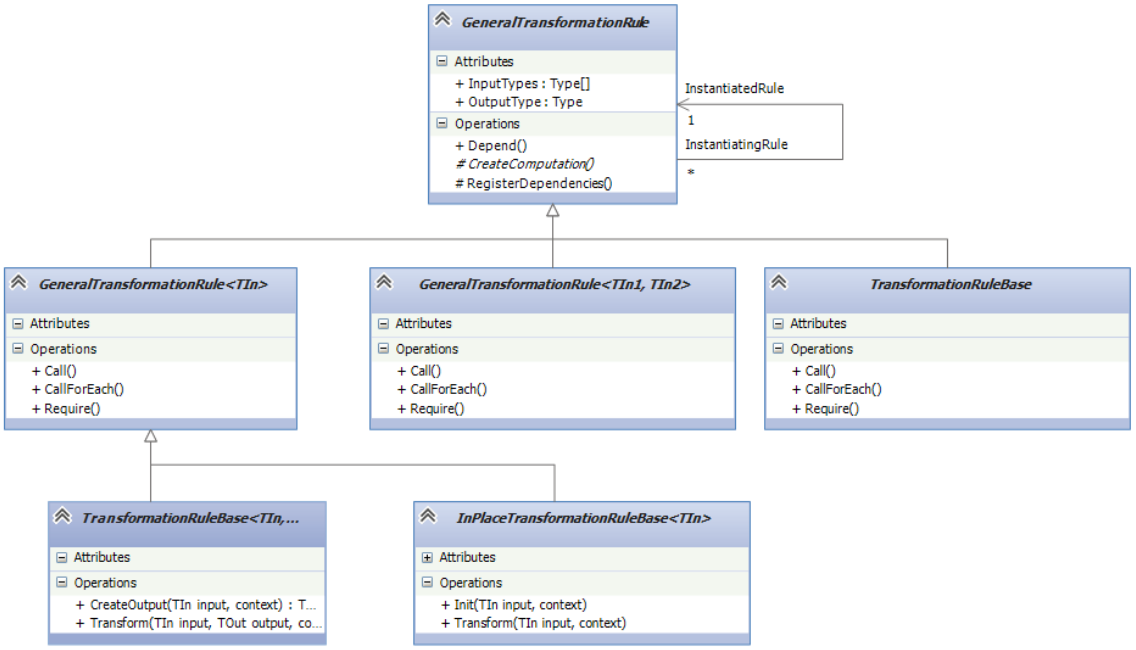

Figure 7.5.: The inheritance hierarchy of GeneralTransformationRule (simplified)

Transformation rules in NMF Transformations are represented by the class `GeneralTransformationRule`. However, this class is abstract and is not intended to be used directly. Instead, the classes `TransformationRule` with either two or three type parameters and the classes `InPlaceTransformationRule` with one or two type parameters are available to be used directly. The type parameters represent the signature of the transformation rules. For transformation rules deriving from `TransformationRule` for example, the first one or two type parameters represent the types of the input arguments whereas the last type parameter specifies the output type. These classes provide a more convenient way to specify the dependencies as they are aware of the types of input and output parameters. Figure 7.5 shows the inheritance hierarchy of the `GeneralTransformationRule` class. Several classes and the very most of the methods are omitted in respect of place limitations. If transformation rules with more than two input parameters are required, the transformation engine can be extended with further classes. Alternatively, the framework contains the classes `TransformationRule` and `InPlaceTransformationRule` with support for arbitrarily many input type arguments. However, the methods of these classes are not type safe.

Of multiple abstract methods in the `GeneralTransformationRule` class, only the method `CreateComputation` remains not implemented. In this method, a transformation rule must create a computation with the given transformation context and the given input. However, in many cases it suffices to use a general computation class without further attributes. This also improves the conciseness of the transformation. Thus, the framework further includes the classes `TransformationRule` and `InPlaceTransformationRule`, each in two versions for one or two input arguments. In these transformation rules, it is possible to specify the `Transform` method and the `CreateOutput` method directly instead creating a separate Computation subclass. However, if these simple transformation techniques do

not suffice as for example additional actions are required when dependencies are executed, transformation rules can easily define their own implementation of the `Computation` class in order to perform the necessary actions. However, note that the `Computation` class is not type safe.

```
1  using NMF.Transformations;
2  using NMF.Transformations.Core;
3
4  public class FSM2PN : ReflectiveTransformation {
5   public class Automata2Net : TransformationRule<FSM.
        FiniteStateMachine, PN.PetriNet>
6   {
7    public override void Transform(FSM.FiniteStateMachine input, PN
        .PetriNet output, ITransformationContext context)
8    {
9     output.ID = input.ID;
10   }
11  }
12 }
```

Listing 7.3: The transformation rule FiniteStateMachine2PetriNet

Listing 7.3 shows the definition of the transformation rule that is used to transform a finite state machine into a Petri Net. So far, it is only specified that the Petri Net should have the same name as the finite state machine. As the method to create the output for this transformation rule is not overridden, the `TransformationRule` will use the system activator to create an instance of the *PetriNet* class.

The code inside the `Transform` method can be arbitrary general purpose code, as this method is invoked (instead of being reflected).

7.4.3. Dependencies between transformation rules

Transformation rules can have dependencies. This means that a transformation rule may specify another transformation rule to be invoked with a certain input whenever the transformation rule is called itself. Such dependencies act as triggers. A transformation rule is only invoked either if it is the start rule with the transformation input, or it is triggered by one of these dependencies. Dependencies are specified in the `RegisterDependencies` method.

As NMF Transformations operates on plain CLR objects, it does not have an idea of their structure. It would be possible to read this structure using reflection techniques, but the performance of such reflection techniques is very poor, so that by a principle design decision, the structure of the objects that are to be transformed must be reflected in dependencies of the transformation rules.

Multiple types of dependencies exist. Dependencies of a transformation rule can either be executed before or after the computation. This has an influence on the order of when a computation is forced to generate its output and the order of when the Transform method is being called. Dependencies that apply before the computation are called *require*-dependencies, whereas dependencies applying after a computation are called *call*-dependencies. In general, the dependencies implemented in the framework invoke exactly one other transformation rule, possibly with a different input. However, in some cases it is required to invoke a transformation rule multiple times with a collection of inputs. Such multiple dependencies also exist. One can identify them by the suffix "Many".

As call dependencies are executed after the computation has set its output, it is also possible to let them rely on the output of the computation. These computations are referred to as output-sensitive. Beware that an output-sensitive dependency of a delayed transformation rule is only triggered as soon as the output is created. As a consequence, also dependencies of this computation are executed delayed unless the dependent computations have been created due to other dependencies.

Dependencies are set by calling the appropriate method in the `RegisterDependencies` method of a transformation rule. This method gets called when the transformation rule is initialized. A dependency object is created to represent this dependency and added to the dependencies of the transformation rule. The methods to register these dependencies are called after the according type of dependency. All of these methods have several overloads that allow to specify different aspects of the dependency in a type-safe manner. These aspects are represented by functions and the intended way to use the API is to call these methods with lambda expressions. To enhance the readability and understandability of the model transformation code, it is also a good practice to use named parameters. However, a transformation developer may also write separate methods. It is possible to specify selectors, persistors and filters. Selectors are methods that select the input for the dependent transformation rule, given the input (in case of output sensitive dependencies also the output) of the computation that executed the dependency. Filters filter the computations where the dependency is executed. Persistors are callbacks that can be used to add the resulting element of a model transformation to the target model.

Given the model transformation from listing 7.3, we can extend it by another transformation rule *State2Place* to transform states into places. In the *FiniteStateMachine2PetriNet* transformation rule, we now have to ensure that the states within a finite state machine are transformed using the *State2Place rule*. This can be accomplished in listing 7.4. Since the dependency is written as requirement, the states of a finite state machine will be processed prior to the finite state machine. Thus, when the `Transform` method of the finite state machine is called, the trace entries of the states are available.

```
1 public class FiniteStateMachine2PetriNet : TransformationRule<FSM
    .FiniteStateMachine, PN.PetriNet>
2 {
3  public override void RegisterDependencies()
4  {
5   RequireMany(Rule<State2Place>(),
6           selector: fsm => fsm.States);
7  }
8 }
```

Listing 7.4: Registering a dependency to State2Place

The `Rule` method in line 5 is used to get the instance of the specified transformation rule type. As the rule *State2Place* is a transformation rule transforming a state into a place, the C# compiler is able to infer the typecast of the lambda expression used as second parameter. As this lambda expression has one parameter and returns a collection of states, the compiler can infer that this is meant to be the selector method. Thus, it is unnecessary to write the name of this parameter. However, it increases the maintainability.

An important aspect of maintainability is that for a given requirement change, it is easy to find the code that needs to be changed to fulfill the updated requirements. As these updated requirements are likely to involve that only some certain target elements need to generated in another structure, it is desirable to have transformation code that is responsible for a certain target element as close together as possible. Thus, a maintainability

goal for a transformation language must be to provide means to achieve this as far as possible. NTL tries to achieve this goal by its basic structure of transformation rules and dependencies between them specified together with the transformation rules. However, it is an important aspect at which, the base or the dependent transformation rule, these dependencies are defined.

Consider again the last example. Does the specification that any state of the model root must be transformed via *State2Place* really to the transformation rule representing the model root? Another option (in the eyes of some people a more maintainable one) would be to keep the information on the concepts of states together and define this dependency straight at the *State2Place*-rule. As a result, the changed requirements that state machines no longer require states can be dealt with more effectively (which is infeasible for state machines, of course).

For this purpose, it is possible to reverse the dependencies. This is accomplished using the `CallFor` and `CallForEach` methods. The semantics is the current transformation rule is called whenever another transformation rule is called with one (CallFor) or many (CallForEach) sets of inputs.

Thus, we can also include the above dependency using `CallForEach` as done in listing 7.5.

```
1 public class State2Place : TransformationRule<FSM.State, PN.Place
    >
2 {
3  public override void RegisterDependencies()
4  {
5   CallForEach(Rule<AutomataToNet>(),
6     selector: fsm => fsm.States);
7  }
8 }
```

Listing 7.5: The rule State2Place with reversed dependency

Note that the code displayed in listings 7.4 and 7.5 have exactly the same effect. Once again, we could have omitted the parameter name.

However, in the example above, the resulting Petri Net and the places have nothing in common, as the places have not been added to the *Places* property of the Petri Net. It is possible to use the tracing functionality of NMF Transformations for this purpose. However, in many cases it is handy to register the outcome of such dependencies right at the same place where the dependency has been created. Thus, an overloaded version of these dependency setting methods also accepts the definition of a persistor. This persistor is called as soon as the dependent computations have their output set, especially before any `Transform` method has been called. Listing 7.6 shows how the above listing 7.5 would have to be extended to use persistors.

```
1  public class State2Place : TransformationRule<FSM.State, PN.Place
       >
2  {
3   public override void RegisterDependencies()
4   {
5    CallForEach(Rule<AutomataToNet>(),
6      selector: fsm => fsm.States,
7                  persistor: (net,places) => net.Places.AddRange(
                       places));
8   }
9  }
```

Listing 7.6: The rule State2Place with reversed dependency and persistor

There is no limit on the amount of dependencies, a transformation rule can have. A transformation rule can also have multiple dependencies to the same other transformation rule. Listing 7.7 shows the example of the transformation rule used to transform the transitions of a finite state machine. There are two dependencies to the *State2Place* rule, each with different selectors (the method that selects the input for the transformation rule defining the dependency) and persistors.

```
1  public class Transition2Transition : TransformationRule<FSM.
       Transition, PN.Transition>
2  {
3   public override void RegisterDependencies()
4   {
5    CallForEach(Rule<AutomataToNet>(),
6      selector: fsm => fsm.Transitions,
7      persistor: (net, transitions) => net.Transitions.AddRange(
           transitions));
8
9    Require(Rule<State2Place>(),
10           selector: t => t.StartState,
11                 persistor: (t, place) => t.From.Add(place));
12
13   Require(Rule<State2Place>(),
14           selector: t => t.EndState,
15                 persistor: (t, place) => t.To.Add(place));
16  }
17 }
```

Listing 7.7: The rule Transition2Transition with multiple dependencies

However, in some more sophisticated cases it might be necessary to create dependencies not for a single transformation rule but for all rules that match a certain type signature. Such dependencies are referred to as *wildcard*-dependencies. To specify them, the convenience methods to specify dependencies all have appropriate overloads that do not take a transformation rule. However, the compiler then can no longer infer the correct type arguments and thus, the developer has to specify these type parameters. The type parameters are also used to select the transformation rules that the dependency targets to.

7.4.4. Tracing

In some cases, the persistor methods for the dependencies do not suffice to fulfill the transformation requirements, e.g. because the correspondence represented by other transformation rules is necessary. In the example of finite state machines to Petri Nets, consider the transformation of end states. Section 4.1 specifies that for each end state in a finite state machine, there should be a corresponding transition created starting from the transformed place but with no target place.

As it is required that an end state is transformed to a transition, a transformation developer would certainly start with the transformation rule as displayed below in listing 7.8.

```
1  public class EndState2Transition : TransformationRule<FSM.State,
       PN.Transition>
2  {
3   public override void RegisterDependencies()
4   {
5    CallForEach(Rule<AutomataToNet>(),
6     selector: fsm => fsm.States.Where(s => s.IsEndState),
7     persistor: (pn, endTransitions) =>
8               pn.Transitions.AddRange(endTransitions));
9   }
10 }
```

Listing 7.8: An incomplete EndState2Transition rule

The above code creates an empty transition for the end state and must be extended that the *From* reference of the transition contains the end state that it belongs to. This of course can be achieved by using another dependency, either as call-dependency or as require-dependency, with a persistor that registers the corresponding place for the end state in the *From* reference of the transition. The transformation engine is responsible that the transformation rule to transform the state into a place is only executed once at most. The correspondence in the sense of 6.1 can be queried purely by using dependencies. However, although this procedure in this yet very simple case, it might be more complicated in other situations.

Thus, NMF Transformations also includes an explicit trace support that is further extended by NTL. This trace support further has the advantage to allow more sophisticated queries. However, in contrast to dependencies from section 7.4.3, it is a pure query functionality with no side effects. This means that if a corresponding place for a state is traced which is not or not yet transformed, a null reference is returned instead of triggering the transformation of that state.

Similar to QVT-O, the main trace functionality is represented by the `Resolve` method. Various variations of this methods exist to support a broad range of trace requests. The different variations of this method are listed together with a description in table 7.1. These trace methods are implemented as extension methods for convenience. Thus, they require to add a `using` statement to the `NMF.Transformations` namespace. The methods basically redirect the trace functions to the trace functionality shown in table 7.2.

In some occasions, it is necessary to trace not only the outputs but trace the computations instead. This can be accomplished by using the `Trace` method. Again, there are different variations that are explained in table 7.2. Unlike the `Resolve` method variants, these methods are defined by the trace interface of NMF Transformations Core. This interface is designed to be agnostic of the transformation context implementation (i.e. the transformation engine implementation). Thus, it is also agnostic of the assumption that

Trace method	Description
`ResolveIn`	Traces the output of the computation that transformed the given input element(s) with the given transformation rule.
`Resolve`	Traces the output of the computation that transformed the given input element(s) with the given signature.
`ResolveManyIn`	Traces the outputs of the computations that transformed the given collection of input elements with the given transformation rule.
`ResolveMany`	Traces the outputs of the computations that transformed the given collection of input elements with the given signature.
`ResolveInWhere`	Traces the outputs that were created by the given transformation rule and match the given criteria.
`ResolveWhere`	Traces the outputs that were created with a matching signature and further match the given criteria.
`FindAllIn`	Traces the outputs of all computations with the given transformation rule.
`FindAll`	Traces all outputs of computations with a given input signature.

Table 7.1.: Overview on the trace functionality to return the results

a transformation rule may only be called once per input. Thus, these methods always return collections. Furthermore, instead of computations, these methods return objects of a trace interface, `ITraceEntry`. This is done to support adding and removing entries to the trace which is presented later in this section. Thus, the trace entries have to be casted to computations.

`Trace` method	Description
`TraceIn`	Traces the computation that transformed the given input element(s) with the given transformation rule.
`Trace`	Traces the computation that transformed the given input element(s) with the given signature.
`TraceManyIn`	Traces the computations with the given transformation rule where the input is part of the given list. The order of the returned computations is not necessarily the same as the order of the corresponding input elements.
`TraceMany`	Same as `TraceManyIn` but instead based on the signature.
`TraceAllIn`	Traces all computations for the given transformation rule.
`TraceAll`	Traces all computations with inputs of the given signature.

Table 7.2.: Overview on the trace functionality to return the computations

With these trace methods, it is possible to finish the above transformation rule *EndState2Transition* with the `Transform` method as shown in listing 7.9.

```
1 public override void Transform(FSM.State input, PN.Transition
      output, ITransformationContext context)
2 {
3  var from = context.Trace.ResolveIn(Rule<StateToPlace>(), input);
4  output.From.Add(from);
5  from.Outgoing.Add(output);
6  output.Input = "";
7 }
```

Listing 7.9: The Transform method of the EndState2Transition rule

The trace is encapsulated by a dedicated trace component, hidden behind an interface. This interface only offers a small subset of the trace functionality, as most of the trace functionality is implemented as extension methods for this trace interface to allow a consistent functionality and to have the trace component be unaware of the existence of NTL. However, there is a default implementation to implement the trace functionality by linear scan on a collection of computations. This is done mainly to make it easier for developers who want extend NMF Transformations. Furthermore, this enables to extend the trace interface without breaking existing extensions. As the trace implementing methods are virtual, extensions should override the most important methods to accelerate the trace.

There is no equivalent concept like *late resolve* in QVT-O. However, it is possible to trace the computations instead of their outputs. Most trace operations only operate on trace entries, but these trace entries can be casted to computations. These computations have an event when their output is initialized. This event can be subscribed, making this solution more flexible (QVT-O only allows *late resolve*s assigned to attributes, not to local variables, whereas there is no restriction in NMF Transformations).

The trace interface also offers two further methods that allow transformation developers to extend the trace functionality by inserting trace entries. These trace entries have a lightweight interface that only represent the tuple of input and output arguments as well

as a transformation rule used as the trace key. Transformation developers can use this feature to freely remove and add trace entries to the trace component, independently of the computations that take part in the actual computation.

NTL uses this concept to provide another useful kind of trace functionality that encapsulates these methods that allow to change the contents of the trace. By default, a computation of a transformation rule is automatically recognized by the trace functionality. However, in some cases it is useful to create additional trace entries. The creation of such additional trace entries can be specified using some methods provided by the `TransformationRuleBase`-class. These methods and their meaning is listed in table 7.3.

Trace method	Description
TraceInput	Creates additional trace entries that make it possible to find an input of a computation based on some key. A method must be provided that selects this key, given the input of the computation. This method is available also in in-place transformation rules. Overloads exist that do not need a transformation rule as trace key, but create one.
TraceOutput	Creates additional trace entries that make it possible to find an output of a computation based on a key. This key may depend on both input and output of a computation. A method must be provided to select the key for a given computation. Overloads exist that do not need a transformation rule as a trace key, but create one.
TraceAs	Creates additional trace entries, where these trace entries may have completely different types. In this version, the transformation rules used as the trace key must be provided. Furthermore, the method takes two methods as optional parameters. These methods specify the input and the output of the new trace entries. If a null value is passed, NTL assumes to use the input or output of the computation. If this is not possible due to the transformation rule signature, an exception is thrown.

Table 7.3.: Special dependencies for further functionality related to trace support

Currently, C# does not allow to specify constraints on type parameters that the type parameters of a method must be base types of the surrounding class type parameters. As a consequence, the exception in the `TraceAs` function is only thrown at runtime, although this is a constraint that can be checked at compile time.

7.4.5. Transformation rule instantiation

The example of the transformation from the *People* metamodel into the *FamilyRelations* metamodel from section 4.2 demonstrates the need of means to support transforming inheritance hierarchies (also discussed in section 6.3) although the source metamodel of this transformation does not even include an inheritance hierarchy.

It is of course possible to solve this transformation task using only the dependencies from section 7.4.3. This would include separate transformation rules for transforming males and females. These transformation rules can then be triggered by dependencies for males

or females, accordingly. This solution does not even necessarily break the "Don't repeat yourself" principle as duplicate code can be avoided by a separate transformation rule without an output. This procedure is demonstrated in listing 7.10. In this excerpt of a transformation, only the transformation rules to transform males and the transformation rule common to both females and males is shown.

```
1  public class Person2Male : TransformationRule<Ps.Person, Fam.Male
       >
2  {
3   public override void RegisterDependencies()
4   {
5    CallForEach<Ps.Root, Fam.Root>
6     (root => root.People.Where(p => p.Gender == Ps.Gender.Male),
7     (root, people) => root.People.AddRange(people));
8
9    CallOutputSensitive(Rule<InitPerson>(), (input, output) =>
       input, (input, output) => output);
10  }
11 }
12
13 public class InitPerson : InPlaceTransformationRule<Ps.Person,
     Fam.Person>
14 {
15  public override void Transform(Ps.Person input, Fam.Person
      transformed, ITransformationContext context)
16  {
17   transformed.FirstName = input.FirstName;
18   transformed.LastName = input.Name;
19  }
20 }
```

Listing 7.10: An implementation sample for People to FamilyRelations without inheritance support

However, this solution yields a maintainability problem. As soon as there is a new subclass of Person in the source model that needs to be treated differently, the reversed dependency triggering the Person2Male transformation rule must be changed to transform only those Person instances that are not instances of this newly introduced subclass. If the dependency was not reversed, all dependencies distributed across the transformation would need to be updated. This badly contradicts the maintainability goal to minimize the change impact of introducing a new subclass.

This contradiction is also there in QVT-O although not as bad as in the above example. If a metaclass representing the input of a mapping gets another subclass that has to treated differently, this mapping has to be turned into a disjunct mapping and all references must be updated.

To escape this maintenance dilemma, NMF Transformations has a built-in support for inheritance. Transformation rules may declare that they are able to create the output of other transformation rules. This procedure is called instantiating. As a consequence, the computation of the instantiated transformation rule is not any more requested to create its output. Instead, the output is copied from the computation for the instantiating transformation rule. Furthermore, instantiation implicitly includes a require dependency. In contrast to e.g. disjunct mappings in QVT-O, instantiations can be nested, i.e. a trans-

formation rule instantiating another transformation rule can itself again be instantiated. Furthermore, instantiations can be attributed with a separate filter.

Marking a transformation rule as instantiating for another is done by calling the method `MarkInstantiatingFor`. Sadly, unlike Java, .NET does not allow constraints on type parameters in the direction that the type parameter is a super class of another type. Thus, the validity of the calls to `MarkInstantiatingFor` cannot be checked by the compiler. Instead, it is checked at runtime during initialization of the transformation.

Consequently, when there is some sort of inheritance, there is also some sort of abstract and sealed elements. Thus, the class `AbstractTransformationRule` represents a simple transformation rule which must be instantiated by another transformation rule. However, the whole magic behind this class is that an exception is thrown whenever the transformation rule is requested to create an output for a computation.

The above example rewritten using instantiation is shown in listing 7.11.

```
1  public class Person2Male : TransformationRule<Ps.Person, Fam.Male
       >
2  {
3   public override void RegisterDependencies()
4   {
5    MarkInstantiatingFor(Rule<Person2Person>(), p => p.Gender == Ps
         .Gender.Male);
6   }
7  }
8
9  public class Person2Person : AbstractTransformationRule<Ps.Person
       , Fam.Person>
10 {
11  public override void Transform(Ps.Person input, Fam.Person
       output, ITransformationContext context)
12  {
13   output.LastName = input.Name;
14   output.FirstName = input.FirstName;
15  }
16
17  public override void RegisterDependencies()
18  {
19   CallForEach<Ps.Root, Fam.Root>(
20    root => root.People,
21    (root, people) => root.People.AddRange(people));
22  }
23 }
```

Listing 7.11: An implementation sample for People to FamilyRelations with instantiation

In contrast to the implementation in listing 7.10, it is transparent for the *Person2Person* transformation rule that there is a special treatment for male people. Furthermore, this technique yields a good tracing ability, as the transformation rule *Person2Person* represents every transformation from a person in the People metamodel to a person in the FamilyRelations metamodel.

The instantiation mechanism seems quite similar to inheritance of classes. However, there is a huge difference. A transformation rule can also instantiate another transformation

rule with multiple input parameters. In this case, given a set of input elements for a transformation rule, there might be multiple transformation rules that can instantiate this rule. Instead of applying a sophisticated method of finding the best match, NMF Transformations just takes the first transformation rule that fits the criteria. The transformation rules are checked in the same order as in the Rules collection of the transformation which is the same order of how they are returned in the `CreateRules` method which in turn for reflective transformations is the order of how they appear in the code.

7.4.6. Relational Extensions

Dependencies as introduced in section 7.4.3 can only describe the circumstance that a computation directly triggers another. However, in some cases this is not sufficient. Some cases require more complex patterns including dependencies to more than one computation whose input arguments are not linked by any reference. If input elements are linked through a reference, it is usually possible (and more performant) to use ordinary dependencies instead. If they are not, then Relational Extensions provide means to concisely specify when a transformation is called.

The purpose of the relational extensions of NTL is to provide means to support complex patterns for transformation rules as discussed in section 6.4. The name Relational Extensions comes from the intention that these extensions make it possible to write transformation rules that come very close to the relations from QVT-R, except for their lack of bidirectional transformation support. Although the name suggests it, the Relational Extensions are not implemented as extensions, i.e. they are implemented within the same assembly. The reason are some convenience methods that simplify the syntax for the transformation developers. The design rationale behind this decision is that an easier syntax for transformation developers is yet more important than the clear code separation to separate assemblies.

The relational extensions use the pattern extension mechanism. Each relational expression registers a transformation rule pattern for the transformation rule in quest. The convenience methods mentioned above introduce a shortcut to provide these pattern objects using the query syntax of C#.

Thus, the *IEnumerable* interface from the Base Class Library of .NET has been extended to the interface `IGrowableSource` that represents a collection that notifies clients when an item is added to the collection. Thus, such growable sources can specify when a transformation rule is to be triggered and every new element of this pattern is then automatically forwarded to call the parented transformation rule.

As an example, consider once again the transformation from People to FamilyRelations introduced in section 4.2. Consider that this transformation is extended by the deduction of households. This might be required for marketing purposes as for example it usually suffices to send a brochure only once per household. To do this, the transformation can use the heuristic that spouses live together in one household and thus, either one or two adults live together in a household together with their children. We further assume that the adults are of different sex for the sake of simplicity. This is because to compare sets more easily, we need an order on them. This is easy when we have men and women as we can create ordered tuples by putting either of them in front. In this case, we decide to take the woman in front. The code to call the transformation rule to create a household model element is shown in listing 7.12.

```
1  public class Spouses2Household : TransformationRule<Female, Male,
       Household>
2  {
3   public override void RegisterDependencies()
4   {
5    WithPattern(from mom in Rule<Person2Female>().
         ToComputationSource()
6     from dad in Rule<Person2Male>().ToComputationSource()
7     where (mom.Input == null || mom.Input.Spouse == dad.Input)
8      && (dad.Input == null || dad.Input.Spouse == mom.Input)
9     select new Tuple<Female, Male>(mom.Input, dad.Input));
10  }
11 }
```

Listing 7.12: Applying pattern matching to create households

What exactly does the code above do? It specifies a patternto describe when the transformation should be called. If a transformation rule requires two input arguments, the public API of NMF Transformations requires a tuple of the involved input arguments. Thus, the pattern has to return tuples of the types according to the input type parameters of the transformation rule. These tuples are then used to call the transformation rule with the specified contexts.

The syntax in listing 7.12 should be familiar to C# developers, as this syntax is also applied in the Language Integrated Query (LINQ)[1] although the majority of the developers probably did not yet recognize that this syntax could be used outside of this framework. If we look at the type of the expected parameter of the `WithPattern` method, it expects a function that takes a transformation context as parameter and returns a relational source of tuples of the transformation rules input type parameters. However, this function type is used as a monad for the tuple type (see section 3.3.5) and since the appropriate methods are implemented, the C# query syntax can be used to specify this function.

However, instead of a unit function, the method `ToComputationSource` is implemented. Given a transformation rule, this method returns a function that returns for each transformation context an object that collects the computations of this transformation rule under the given context. Further overloads exist that can further filter the computations by a given filter method. The type of the object collecting the computations of this transformation rule is a relational source of computation wrappers. The reason to wrap these computations is that the computation class is an abstract non-generic base class that is not aware of the type of its input and output parameters. The wrappers, implemented as structures for performance reasons, wrap these computations and provide a typed access. Thus, the compiler knows that i.e. the expression `mom.Input` in line 7 must be of type *Person*, as this is the input type parameter of the according *Person2Female* transformation rule. This enables the IDE to support the developer with useful functionality like code completion. Furthermore, the filter specified in line 7 and 8 can completely be checked by the compiler during development.

By using a syntax that developers are familiar with, it is possible to let them specify patterns for transformation rules in a way that is familiar to them. Thus, the acceptance of the transformation language is improved and the maintenance effort is lowered.

The filter on the above pattern may get more sophisticated. If this is the case, the developer can move this code into a separate method that is called within the where statement. As

[1] `http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx`

the developer names this method accordingly, the query statement is still pretty much understandable in what it does. This method is (same as the lambda expression in the `where`-clause) then executed for every tuple of in this case females and males that passed previous filters.

```
1 WithPattern(from mom in Rule<Person2Female>().ToComputationSource
      ()
2  from dad in Rule<Person2Male>().ToComputationSource()
3  where IsAllowedPair(mom, dad)
4  select new Tuple<Female, Male>(mom.Input, dad.Input));
```

Listing 7.13: The above pattern using a separate filter method

A huge advantage of moving rather complex predicates into separate functions is that these predicates become testable. By setting the accessibility of these predicate functions to `internal` scope, test assemblies can be configured to access these internal methods while clients of the library cannot.

An example of this is shown in listing 7.13 where the complex predicate of a tuple of when to match a woman and a man is moved to a dedicated method accordingly named *IsAllowedPair*.

The order of the statements within the query expression is of great importance as it determines which extension methods are called in what sequence. Same like any query syntax, the query syntax within the Relational Extensions is translated into a sequence of calls to extension methods. As an example, listing 7.14 shows how the above pattern is translated into the sequence of calls to extension methods. Although this representation is more understandable in how the developer can get a glimpse of how the code actually works, it is also much less concise and thus less understandable in what the intention behind this code is.

```
1 WithPattern(Rule<Person2Female>().ToComputationSource()
2  .SelectMany(mom => Rule<Person2Male>().ToComputationSource(), (
       mom, dad) => new {mom=mom, dad=dad})
3  .Where(pair => IsAllowedPair(pair.mom, pair.dad))
4  .Select(pair => new Tuple<Female, Male>(pair.mom.Input, pair.dad
       .Input)));
```

Listing 7.14: The above pattern using method chaining

An important point of what this representation unveils, is that the unit function `ToComputationSource` is actually called for each female person that can act as a mom, thereby creating a new collection each time. Thus, it is better to move this call outside the pattern finally arriving at the code shown in listing 7.15.

```
1  public class Spouses2Household : TransformationRule<Female, Male,
       Household>
2  {
3   public override void RegisterDependencies()
4   {
5    var moms = Rule<Person2Female>().ToComputationSource();
6    var dads = Rule<Person2Male>().ToComputationSource();
7
8    WithPattern(from mom in moms
9     from dad in dads
10    where IsAllowedPair(mom, dad)
11    select new Tuple<Female, Male>(mom.Input, dad.Input));
12  }
13 }
```

Listing 7.15: Applying pattern matching to create households, updated

However, in this way the method responsible to filter the candidate tuples of moms and dads can hardly rely on the context of the underlying computations. This can only be achieved by extracting the transformation context from any of the computation wrappers by calling the reference `Computation.Context`. However, the computation source object may also specify to allow null values. In this case, the according computation wrapper wraps a null reference which obviously does not have a context set. A possible solution for this dilemma is to make the function that is hidden in listing 7.12 a little bit more explicit. Listing 7.16 shows the above pattern using a lambda expression instead of pure query syntax.

```
1 WithPattern(context =>
2   from mom in Rule<Person2Female>().ToComputationSource(context)
3   from dad in Rule<Person2Male>().ToComputationSource(context)
4   where IsAllowedPair(mom, dad)
5   select new Tuple<Female, Male>(mom.Input, dad.Input));
```

Listing 7.16: Using a lambda expression for more sophisticated filters

Using the version of listing 7.16, the transformation developer may access the transformation context within the where clause more directly, although this has not been done in the above example. The syntax, however, is very similar to the syntax in listing 7.12. However, this solution yields the problem that again, the method `ToComputationSource` is called for each mom candidate which yields a performance problem.

The keywords `from`, `where` and `select` are the only keywords within the query syntax of C# that have been implemented. Others, especially `group by` have not been implemented, as they do not make sense in this scenario.

The classes involved in the implementation of the Relational Extensions and thus representing the abstract syntax for the Relational extensions are shown in figure 7.6. The only purpose of the classes `CompositeSource`, `SelectSource` and `FilteredSource` is to implement the query syntax used in the Relational Extensions. As they implement monad methods, they have several type parameters which have been omitted in the diagram for better clarity.

7.4.7. Composing Transformations

Model transformations in NMF Transformations mainly consist of their transformation rules and patterns. As transformation rules are .NET classes, they can be easily moved to

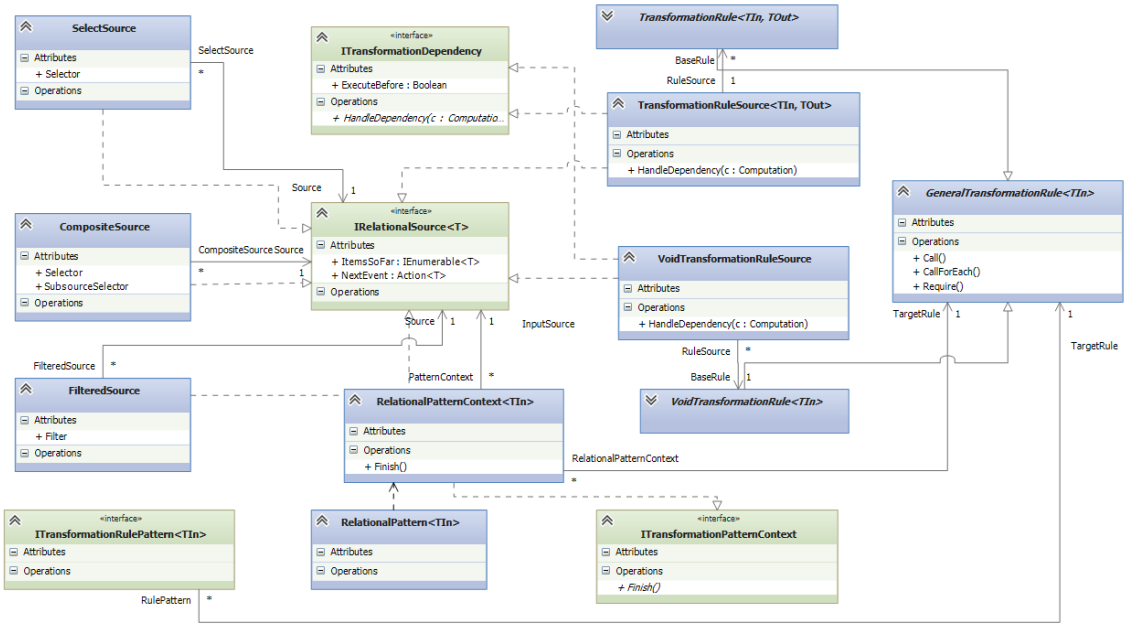

Figure 7.6.: The classes involved in the Relational Extensions

separate files and even separate assemblies. These assemblies can even be signed to be confident that they are not forged. When using a reflective transformation, these additional transformation rules can be specified by overriding the `CreateCustomTransformationRules` method. If the transformation directly inherits from the `Transformation` class, these rules must be specified in the same way as any other rules.

As running a model transformation in NMF Transformations is a special method call, same like in most other internal DSLs, model transformations can easily use other model transformations to call them. It is not necessary as in QVT-O to declare this in front. However, it may of cause be useful to import the namespace of this other transformation via a `using` statement. Thus, chaining is as easily accomplished as in general purpose code.

The composition of model transformations is more complex. Transformation rules in NMF Transformations are represented as instances of classes. These classes are to be instantiated within the transformation container class. By default, a reflective transformation instantiates all its public nested classes. However, this procedure can easily be overwritten thereby instantiating any classes representing any transformation rules. These transformation rules do not have any restrictions except that they must be accessible from the transformation that uses it. Especially, the classes representing transformation rules to be used in a composed transformation may also be embedded into another assembly. Thus, it is possible to build up a transformation library and put this into a separate assembly. As assemblies can be digitally signed in .NET, developers can be sure that these transformation rules are not being tampered with.

What makes transformation composition hard to predict is the configurability of transformation rules. Transformation rules always have a reference to the transformation that contains them. By casting this reference to the actual transformation type, transformation developers can make the transformation rule depend on some properties or, as transformation rules are nested classes, even private fields of the containing transformation. However, if such procedure is applied, the transformation rules cannot be used in a different transformation unless of course this other transformation is subclass of the transformation the transformation rule originally was designed for.

Model transformations that mainly use classes from other libraries should inherit directly from `Transformation` and override the `CreateRules` method directly as the reflection of transformation rules is meaningless. This method is to return a collection of transformation rules that afterward will be used as transformation rules of the Transformation. This collection is only retrieved once and copied into another collection. The intended way to specify the transformation rules is to use the co-method functionality of .NET. Listing 7.17 demonstrates this procedure.

```
1 public class FSM2PN : Transformation
2 {
3  protected override IEnumerable<GeneralTransformationRule>
       CreateRules()
4  {
5   yield return new Automata2Net();
6   yield return new State2Place();
7   ...
8  }
9 }
```

Listing 7.17: A transformation using transformation rules from other assemblies

In this way, a transformation can easily be composed of transformation rules either used by other transformations or contained in a transformation library. These transformation rules can be extended with reversed dependencies and instantiations thus calling transformation rules representing possible extensions. However, it is not possible to replace entire transformation rules.

However, in some cases it is necessary to include whole transformations instead of few transformation rules. In this case, any changes made to the included base transformation should be transparent to the extension transformation. Especially when the base transformation is itself extended by a further transformation rule, any transformation extending this base transformation should not be aware of this and should not need to reference this transformation rule.

There are two ways to achieve this. The first and simplest possibility would be inheriting from this transformation. If the base transformation is a `ReflectiveTransformation`, then the reflection step will reflect all transformation rules that are contained in either of the transformation classes. As the transformation class is a direct subclass of the transformation that is to be extended, all transformation rules will automatically work, even if they cast the transformation to the type of the extended transformation as such a cast operation is still valid. Thus, such an extension can be made almost without hesitation.

The second option is to use the `Reflector` class and load patterns and transformation rules from another transformation. However, this will only be possible if the transformation rules does not rely on the exact type of their containing transformation.

When using the first option, using the features of `ReflectiveTransformation`, it is also possible to override existing transformation rules. However, as transformation rules in NMF Transformations are represented by classes, overriding other transformation rules must respect the capabilities of the type system of .NET. As .NET only supports covariance and contravariance in interfaces, it is e.g. impossible to change the input or output types of a transformation rule. Thus, the overriding rule must inherit from the overridden rule. It can then override the main methods `CreateOutput` and `Transform`. To mark the transformation rule as overriding another transformation rule, it must be decorated with the `OverrideRuleAttribute`.

```
1  public class ExtendedFSM2PN : FSM2PN
2  {
3   [OverrideRule]
4   public class Automata2SophisticatedNet : Automata2Net
5   {
6    public override PN.PetriNet CreateOutput(...)
7    {
8     return new PN.SophisticatedPetriNet();
9    }
10  }
11 }
```

Listing 7.18: A transformation using transformation rules from other assemblies

For example, the code demonstrated in listing 7.18 shows how to extend the transformation from listing 7.3 in that way that instead of an ordinary *PetriNet*, an instance of the class *SophisticatedPetriNet* is created as root element for the resulting Petri Net. However, if other rules reference the *Automata2Net* rule, this reference will automatically be redirected to the new *Automata2SophisticatedNet* rule. In this way, all transformation steps can easily be extended by transformation composition. However, overriding a transformation rule must be made explicit by applying the `OverrideRuleAttribute`. If this attribute was omitted in listing 7.18, both transformation rules, *Automata2Net* and *Automata2SophisticatedNet* would coexist next to each other but transformation rules referencing the *Automata2Net* rule would continue to do so.

However, in some cases one might prevent that other transformation rules may override a behavior of a certain transformation rule. The simplest method to achieve that is to mark the class representing this transformation rule sealed. As overriding transformation rules must inherit and inheritance from this rule is made impossible, overriding this rule is impossible. If this was too strict, it would also be possible to prevent overriding certain parts of the transformation rules, e.g. the `RegisterDependencies` method, by marking that method as sealed.

The example of overriding the `RegisterDependencies` method also shows that this overriding mechanism - as it is based on inheritance of the transformation rule classes - is still a white-box technique. Thus, when overriding a transformation rule, the developer must know how this transformation rule works.

7.4.8. Testing

As mentioned in section 6.8, testing is an essential part of the software development process. Besides that an implementation of a model transformation language needs to be properly tested, it is essential for a model transformation language to provide utilities to test the resulting model transformations.

Same like any other software component, model transformations can of course be tested with a black box testing strategy. This means that the model transformation is treated as a black box and then fed with a model that is just large enough to demonstrate some problems that might occur. Afterward, the test compares the result that has been created by the transformation with the expected result that the transformation should have returned. Test cases for model transformation testing can be derived from the specification independent of the transformation language [Gue12]. ed with a model that is just large enough to demonstrate some problems that might occur. Afterward, the test compares the result that has been created by the transformation with the expected result that the

transformation should have returned. Test cases for model transformation testing can be derived from the specification independent of the transformation language [Gue12].

However, the specification of a model transformation is rarely as complete to only allow a single model as output. Instead, in most cases a variety of possible results is valid. Recent research tackled this problem by the introduction partial oracles [Gue12, FMSA13] that allow to check only those parts of the result model that are completely determined by the specification. However, such a partial oracle is not available for NMF Transformations.

Other approaches for model transformation testing generate test cases for a model transformation by analyzing the model transformation using white-box techniques [HLG13, WS13]. However, also these white-box techniques attempt to test the resulting model transformation as it. This yields the problem that a fault in a model transformation is hard to locate. Thus, it is desirable to test not only the whole transformation as it, but also tiny bit of the transformation.

Model transformations in NMF Transformations consist of transformation rules and transformation patterns. As the transformation rules can have complicated dependencies between them, the model transformation as a conglomerate of these transformation rules can be difficult to test. Thus, it is useful to be able to test the transformation rules and the transformation patterns separately.

NMF Transformations supports testing the transformation rules separately. There are two types of artifacts that need to be tested separately. The first type is methods that are part of the transformation rules, especially the `Transform` method and the `CreateOutput` method. The whole difference to any other methods of an arbitrary class really is that these methods are to be executed by the transformation context within NMF Transformations. However, unit tests can still call them directly as these methods are public.

However, these methods need an instance of the transformation context. For such testing purposes, NMF Transformations contains another implementation of the `ITransformationContext` interface, namely the `MockContext` that as the name implies mocks the transformation context. When trying to transform anything with a mock context, this results in a thrown exception. However, the mock context eases setting up the testing environment, especially including the trace support. As a reason, a mock context allows test engineers to add computations to the trace without actually calling any transformation rule and the dependencies. These computations are then represented by mocked computations that throw exceptions when asked to execute the `Transform` method. In this way, the mock context can provide a proper testing environment to test transformation rules separately using usual testing frameworks such as MSTest or NUnit.

Testing the `CreateOutput` method

In most cases, creating the output does not involve the tracing functionality and thus, the `CreateOutput` method can be tested straight away. In cases where the trace functionality is needed, the test environment can be set up same as for the `Transform` method which is described in the next paragraph.

Testing the `Transform` method

Setting up the testing environment for the `Transform` method usually involves setting up an environment for tracing. This can be done most conveniently by using the class `MockContext`. Instead of executing the dependencies of a transformation, looking for instantiations and performing every step that is necessary for the execution semantics, the `MockContext` simply creates a trace entry (if this does not exist yet) and returns the

resulting computation. Furthermore, unlike `TransformationContext`, the computation collection of the `MockContext` is not read-only. As there is also a mock class for computations, the developer can simply add such mocked computations that are then also recognized for the trace functionality.

```
1  [TestMethod]
2  public void EndState2Transition_Transform_Test() {
3   //set up testing environment
4   var fsm2pn = new FSM2PN();
5   var context = new MockContext(fsm2pn);
6   var endState = new FSM.State()
7     { Name = "Test", IsEndState = true };
8   var endPlace = new PN.Place() { Name = "Test" };
9   var endTransition = new PN.Transition();
10  var endState2Transition = fsm2pn
11      .Rule<FSM2PN.EndState2Transition>();
12  context.Computations.Add(fsm2pn.Rule<FSM2PN.State2Place>(),
        endState, endPlace);
13  //perform method under test
14  endState2Transition.Transform(endState, endTransition, context);
15  //assertions
16  ...
17 }
```

Listing 7.19: A test case to test the EndState2Transition rule

Listing 7.19 demonstrates this unit testing of single transformation rules by testing the *EndState2Transition* rule from listing 7.9. This rule uses the trace to find a corresponding place for the end state in order to register the resulting transition with the place. Adding a mock computation to the computations in the mock context as done in line 12 enables the `Transform` method of the *EndState2Transition* rule to find the `endPlace` when tracing the corresponding place for the `endState`.

However, the `Transform` method may also call another transformation rule directly. For this purpose, the `Computations` collection is an *ObservableCollection* that provides an event whenever the collection is changed.

Testing the `RegisterDependencies` method

Most method calls in the `RegisterDependencies` method are done to register dependencies. For this purpose, the `MockContext` also allows to call the dependencies of a transformation rule directly. Furthermore, in case of wildcard dependencies, it is especially useful to test how many dependencies were actually generated. This can be observed by checking the `Dependencies` collection of the transformation rule. Note that, of course, reversed dependencies are saved on their base transformation rules.

Testing the transformation patterns depends on the patterns that are being used. However, the `MockContext` mock of the transformation context may be useful for this purpose, too.

7.4.9. Extensibility

An important difference of internal DSLs compared to external DSLs is the extensibility. Internal DSLs are represented by frameworks on their own. By designing these frameworks in an extensible manner, the internal DSL can be extended not only by the developer of the framework, but also by the developer of the model transformations. As a consequence,

as soon as an internal DSL does not suffice for some sophisticated model transformation tasks, the whole language can be extended to offer these capabilities.

Thus, both the model transformation framework NMF Transformations Core and the internal DSL NTL must be designed extensible, so developers can easily extend these frameworks to extend the framework, if necessary.

For a framework like NMF Transformations Core, this extensibility is a core requirement. After all, it is what makes a framework different from a library [CA08]. Fortunately, there are a couple of guidelines for .NET frameworks (including .NET itself) that have been documented in [CA08] to make it possible that frameworks have the same look and feel as the .NET framework itself. Furthermore, these guidelines are enforced by static code analysis. This static code analysis is included in Visual Studio as Code Analysis. There are different rule sets available to automatically check solutions for design flaws. The framework NMF Transformations Core has been developed to satisfy all of the existing rules by Microsoft. There are only three points in the code of the framework that conflict with a code analysis rule. These points are subject to explicit design decisions against a particular and are documented in the code by an attribute suppressing the code analysis warning.

Most transformation developers will use NMF Transformations Core through NTL. Being an internal DSL, NTL can easily extended not only by framework developers but also by transformation developers. After all, Cuadrado et al. considered the extensibility important enough to design RubyTL, an internal transformation DSL with a stress on extensibility (based on Ruby, see [CMT06]). In fact, NTL is implemented as a separate framework that provides a language-like public API.

The scenario of extending an internal DSL is awkward. As a reason, internal DSLs usually rather concentrate on avoiding syntactic noise for the users of the resulting language rather than complying to guidelines for extensible frameworks. Thus, the design goals of extensibility and the avoidance of syntactic noise contradict. However, the implementation of NTL tries to stick to the guidelines regarding extensibility as far as possible, achieving a fair trade-off between extensibility and avoidance of syntactic noise. As a consequence, also the implementation of NTL is continuously checked by static code analysis although not all of the available rules are applied. Furthermore, NTL extends as less as possible concepts from NMF Transformations Corein order to raise the chance that it gracefully integrates with other extensions of NMF Transformations Core. In fact, the transformation rule types provided by NTL can not only be used with the default transformation type of NTL, they can used with any implementation instead. Furthermore, the `ReflectiveTransformation` also allows to load custom transformation rules. The main reflection-based functionality of `ReflectiveTransformation` has been moved to a separate component to allow other implementations of `Transformation` access the same functionality.

However, to have the public API of NTL behave like a language on its own, it intentionally breaks some of the design guidelines from [CA08] mentioned above. Such breaches to the design guidelines are explicitly documented in the source code by attributes suppressing the code analysis warnings. The implementation often breaks one of the following code analysis rules:

- **CA1011:Consider passing base types as parameters**: This guideline is broken for large parts of the public API as the API often uses more specific types for parameters specifying a rule parameter. This is used to have the C# compiler infer the generic type parameters of a method, as soon as the transformation developer specifies a transformation rule types with the runtime type. This is used to reduce the syntactic noise of the resulting C# code.

- **CA1004:Generic methods should provide type parameter**: As discussed in section 7.4.3, it is sometimes handy to be able to specify dependencies without knowing the exact type of the transformation rule. In this case, it is often necessary to specify the type parameters of a method explicitly although this is prohibited by CA1004. This does not have a direct impact on extensibility but instead has an impact on readability. However, this is an intended behavior, as the alternatives within the C# syntax seemed worse.

- **CA:1006:Do not nest generic types in member signatures**: Although the documentation encourages developers to not suppress this code analysis rule for the sake of easy-to-read public APIs, it has been turned off for the entire NTL project. As a reason, this rule occasionally turns out to be harmful to the maintainability and extensibility of the resulting framework. As an example, the rule prevents the usage of monad implementations.

- **Use the query syntax for anything other than collections**: This guideline is not enforced by means of static code analysis. However, it is broken intentionally by the relational extensions that implement the query pattern to reduce syntactic noise. As the query pattern originally has been developed for collections, its usage to specify the composition of functions may seem odd for some developers, but it helps to reduce the syntactic noise and thus to produce cleaner transformation code.

Except for the listed exceptions, the implementation of NTL satisfies all design guidelines for frameworks and thus extending it should feel as natural as possible to C# developers.

7.5. Drawbacks & Future Work

Although NMF Transformations provides support for a variety of model transformation tasks, it is not possible to solve every transformation problem with these tools. This section describes these scenarios where neither NMF Transformations nor NTL can help to solve them in a maintainable way. However, some scenarios may be supported in future versions. In these occasions, it is evaluated how these scenarios could be supported using NMF Transformations and NTL.

7.5.1. Trace serialization

The trace of transformations generated using NMF Transformations is not serialized, currently. Thus, it can only be queried while the transformation context is still in memory. As a reason, NMF Transformationsoperates on plain CLR objects and therefore does not know how to serialize these objects. Furthermore, it might be a good idea to only include a reference in the serialization of the trace instead of the full object. Serializing the full object would result in large trace files that also contain both input and output models.

However, the framework would allow a separate serialization component to serialize the trace. Furthermore, the trace component allows to add a computation to the trace only, such that NMF Transformations makes it possible to load the trace from a file and running a model transformation with that trace functionality as basis. However, this required to persist the references from the trace to the models. An implementation of a trace serialization would be specific for this referencing scheme. However, trace serialization has not been implemented for any such referencing scheme.

7.5.2. Change propagation

Change propagation means that a model transformation is aware of changes made to the output model after the transformation has been performed. In case of such a modification, the output model should change accordingly to reflect this change without having to run the model transformation once again. However, so far there is hardly any model transformation language available that supports this kind of change propagation and thus, there are a row of open research questions, including the question of the types of changes that are actually supported.

As NMF Transformations operates on in-memory representations of models, this could be possible. Many collection classes provide an event to notify clients when they are changed. The existence of this event can be queried e.g. by checking whether the collection class implements the interface *INotifyCollectionChanged*. In a similar way, a class can implement the interface *INotifyPropertyChanged* to inform clients that a property has been changed, again using an event. These events could be used to notify a framework and thus executing event handlers that previously had been set by the model transformation, executing a dependency on a saved transformation context and thus propagating this change to the target model by a previously defined binding.

Such a binding technology even exists in the .NET framework, more specific in the Windows Presentation Foundation (WPF)[2], and is used to bind elements of a user interface to properties of a view model. WPF essentially uses a different representation of properties, called dependency properties, that does not have the usual type restrictions. Instead, these properties allow to assign rather complex binding objects to them instead of values. These binding objects have the functionality that they provide a function to obtain the actual property value and also inform the client object whenever this value happens to change.

WPF actually even allows more complex expressions as paths. It tracks the path and is aware of any changes that may occur along this path. This means that a property of the target model not only can be bound to a property of the source model but also can be bound to a more complex expression. Consider for example a target model that has a person model with just a plain street. With the data binding technology this street property can be bound to the *Street* property of the first *Address* class instance of a collection named *Addresses*. Thus, that street is not only updated when the street property of the primary residence changes but also when the primary residence changes. As using the WPF technology is completely optional, the transformation developer can specify which correspondence is set once and which correspondences should be maintained.

Furthermore, the data binding technology can even be applied in both directions and a value converter can be placed in between. In this way, transformations with NMF Transformations and NTL can be used to establish a self-maintaining correspondence between multiple models.

The advantage of reusing WPF data binding for change propagation is clearly that this is a technology that is ready to be used. However, it has been designed for user interfaces rather than large models that need to be transformed. Furthermore, it requires a very representation for the target model that lacks of performance.

However, as establishing such a self-maintaining correspondence links between in-memory models to the best of my knowledge has not been done before, it would definitely be a subject of future work to demonstrate this capability. All that needs to done for this is creating new kinds of dependencies that are aware of changes and persist the elements in observable objects. However, extensions to the framework also yield extensions in the internal DSL to serve that framework and thus, also NTL would require some extensions.

[2] `http://msdn.microsoft.com/en-us//library/ms754130.aspx`

7.5.3. Bidirectionality

Bidirectionality in model transformations means that the same model transformation can be applied in both directions. Consider the example from section 4.1. A bidirectional model transformation servicing this transformation task would actually capable of transforming a Petri Net back to a finite state machine. This little toy example already shows that bidirectional transformations yield a row of problems as clearly the transformation from Petri Nets to finite state machines is not always possible. In such a scenario, a model transformation has then to recognize that the transformation is impossible to accomplish and thus has to cancel the operation with an error message.

This is a problem typical for bidirectional model transformations. It is hard to give proper conditions when a model transformation task is reversible and what the actual domain of this reversed model transformation is.

However, even if these problems would be solved someday, NMF Transformations still operates by executing methods of a Turing complete language. However, reversing an operation is hardly possible by reflecting how this operation affects a model. Thus, the reverse operation has to be specified separately same as the persistors of the dependencies. Hence, the efforts of specifying both directions is nearly the same as specifying both transformation directions separately.

On the other hand, some bidirectional model transformations are indeed asymmetrical in the way the one direction is more complex than the backward direction. An example of such a model transformation is the Petri Nets to State Charts case that will be introduced in more detail in chapter 10 (although bidirectionality was not a requirement in this case). To obtain a Petri Net from a State Chart, one simply has to look at the *Basic* and *HyperEdge* elements and ignore the compound states.

The way how dependencies are specified in NTL are not suitable for bidirectional model transformation and thus, it is infeasible to use NTL for bidirectional model transformations. However, NMF Transformations Core as the underlying framework can be reused for bidirectional model transformations when creating an additional internal DSL specifically for bidirectionality. However, this may be a subject of further research.

7.5.4. Model synchronization

Model synchronization means that two or more models are synchronized using certain correspondence rules. If one again considers the transformation problem from section 4.1, this meant that whenever there is a state in the finite state machine with the name ”s1”, there must be a corresponding place with the name ”s1” in the corresponding Petri Net. If there is one, the transformation may reuse this place, otherwise a state has to be created. Conversely, if there is a place named ”s1”, there must be a state in the corresponding finite state machine.

Much like the bidirectionality support, model synchronization cannot be accomplished with NTL. Furthermore, NMF Transformations Corecurrently suspects transformation rules to create their output dependent on their input and the transformation context only. For model synchronization, this is not feasible, as also the context of the transformation rule call is important. Thus, even NMF Transformations Core had to be modified to support model synchronization.

A very interesting version is the combination of change propagation and model synchronization. As NMF Transformations operates on in-memory representations, a model synchronization can be used to install a synchronization link between two models conforming to different metamodels and have them synchronized automatically. In this way,

the model transformation does not really transform models but rather installs a correspondence link between different models. An application scenario is Vitruvius [KBL13] where the idea is to use multiple models to specify different aspects of a system. A model transformation establishing a synchronization link between these models can then be used to keep these models synchronized.

7.5.5. Graphical syntax

In their paper on the requirements of model transformations [SK03a], Sendall and Kozaczynski have stated that a graphical syntax for the specification of model transformation would be a nice thing to have. Indeed, there are a couple of model transformation approaches, e.g. TGGs (see section 2.2) that do provide a graphical syntax to specify model transformations. However, although this paper has been referenced in chapter 5 quite a some times, NMF Transformations currently does not have a graphical syntax.

However, NMF Transformations provides a framework for model transformations that can be built on. As NTL further simplifies the syntax of how model transformations may be represented, it would be possible to have an additional graphical syntax that generates code that either directly interoperates with NMF Transformations or uses NTL for this purpose.

NMF Transformations Core is quite a low level framework. NTL raises the abstraction a bit but is still a low level language as it tries to provide the necessary flexibility. In case that NTL would turn out to be worse than other model transformation languages in terms of usability, especially because of the introduced verbosity due to choosing C# as host language, NTL may still serve as a target for code generation. The code generator is easier to write, as there is not such a big semantical gap that has to filled by the code generator. It is subject of further research to investigate how a code generator for languages like ATL, QVT-R or QVT-O could be built. The benefit of such a code generator would be that these languages get the possibility to operate on plain CLR-objects and thus make these transformation languages more platform independent. Furthermore, it would be subject of investigation if the execution speed of such transformations would improve as these languages are currently interpreted rather than compiled.

7.5.6. Test case generation

It has become a trend in computer science to generate test cases rather than specifying them separately. Such efforts have been made very recently for TGGs [HLG13, WS13]. However, automatic test case generation also exists for general purpose languages. Specifically for .NET, the Pex framework [TDH08] has been created to automatically generate test cases for C# code. It is an interesting question whether this test case generation can be applied to model transformations with NMF Transformations/NTL, also. However, this is an open research question and out of the scope of this thesis.

7.5.7. Parallelism

The sequential execution speed has stopped to rise for several years now. As a replacement, computers get the capability of computing an increasing number of tasks at the same time. This yields the problem that developers need to make the most of this capability and must distribute the computations to several subtasks, so that they can be computed in parallel. If these subtasks can work entirely independent from each other, everything is fine. However, often this is not possible. Instead, these subtasks have dependencies on each other in that way that one subtask needs another to be completed. The developer then must synchronize these subtask so that the first one in the example starts only when the latter is completed.

From a step backwards, what NMF Transformations actually does is dividing a complex model transformation task in multiple subtasks, namely computations. Furthermore, the transformation rules that define how to perform these subtasks have explicit dependencies. This is where the `MarkRequire` method of the `Computation` class comes into place. With this method, subclasses of the `Computation` class can override a behavior to be executed whenever a computation depends on another computation due to a dependency. This method can be used to synchronize the subtasks as the computation is perfectly aware that another computation needs to complete first.

Many parallelism techniques suffer in performance if the developer divides a task into too many subtasks. As a reason, these subtasks are often directly mapped to threads. These threads are more lightweight than processes, but having several thousands of them are usually still too many as they induce additional efforts in the operating systems scheduler. However, .NET includes an abstraction from operating system threads, namely *Task*s. Tasks are not represented by a thread but instead scheduled to a number of available threads by the framework, thus preventing the operating system from having a plethora of different threads. As a consequence, they are much more lightweight and creating thousands of them does not immediately yield a performance problem. Thus, they are also appropriate for smaller subtasks. Hence, the technology is a reasonable candidate to implement parallel execution of model transformations specified with NMF Transformations. Especially the transform stage can be run in parallel. In this stage, the trace is usually not modified any more. Furthermore, different computations usually run on different model elements, so they do not interfere.

However, there are also some problems (as usual when dealing with parallelism). First and foremost, accessing the model in a concurrent manner does require the model to accept that, i.e., if there is an access conflict (e.g. a data race) on a collection, the collection must support this. Thus, the model representation has to be modified for this. The second question is how to deal with computations that are not represented by tasks or how to let the developer specify which rules are to be run in parallel. To maintain the semantics, a parallel version would still have to respect the dependencies set by the transformation model. The third question is what speedups could be achieved with agnostic parallelism.

7.6. Conclusions

NMF Transformations provides means to write rule-based model transformations with an internal DSL utilizing .NET languages like C# as host language that transforms models represented as plain CLR-objects. These transformations have simple abstract syntax that hides its complexity in its attributes. The Turing complexity intrinsic to model transformation languages can be specified in normal general purpose code that developers are most familiar with. On the other hand, the internal DSL on top of NMF Transformations, NTL, provides means to use high-level abstractions for model transformations that are missing in plain general purpose languages.

NMF Transformations provides support specifically to transform complex inheritance hierarchies in a maintainable way. NMF Transformations includes a powerful trace functionality. However, tracing is only possible in the direction of the model transformation. More complex pattern matching is supported through the relational extensions of NTL.

The resulting model transformations can be tested with unit tests, where the transformation rules can be tested separately.

NTL also provides techniques for transformation developers to extend existing model transformations very easily and thus customize existing model transformations for changed situations.

8. Impact of NTL language features to maintainability

This chapter discusses how the language features of NTL can have an impact on the quality attributes of model transformations. For this purpose, the next sections evaluate the impacts of NTLto each of the quality attributes presented in section 3.4. It is also investigated whether it is necessary to further evaluate the impact on these quality attributes and how such an evaluation could be achieved.

8.1. Understandability

Understandability is a subjective criteria and it is a research field on its own to measure understandability by means of metrics. Thus, the impact of NTLs design on the understandability of the resulting model transformations cannot be determined for sure here as understandability also largely depends on the person that has to understand the model transformation, e.g. regarding his background.

Compared to creating model transformations with usual C# code, NTL probably raises the understandability of the resulting model transformations for most of the developers as it provides abstractions for model transformations. However, besides the actual model transformation, developers have to understand these abstractions. It is subject of further evaluation to investigate how this effort of learning these abstractions relate to ease the understandability of model transformations created using NTL. The understandability of NMF Transformations will be evaluated with prospective users of the code generator that is the outcome of chapter 11.

On the other side, if developers are familiar with the usual abstractions of other model transformation languages, it will be interesting how these transformation developers understand the way NTL exposes these abstractions. This is especially interesting as NTL is an internal language and thus includes syntax artifacts that may hamper the understandability of the model transformation code. This is a goal of the TTC case studies as during the TTC, multiple experienced transformation developers reviewed the NMF solution and were further asked to rate the understandability and conciseness of the solutions. Sadly, the developers were not asked for the understandability directly. But it may be possible to draw some conclusions for the transformation understandability from this combined evaluation, also.

8.2. Modifiability

Modifiability is closely related to understandability. However, there is a little difference. As soon as a developer understands the code of a model transformation, he still has to understand the concepts of the underlying model transformation language (provided there is one). Although code can be self-explanatory, it can still be hard to write such code.

8.2.1. Discoverability

This is where discoverability comes into place. Discoverability refers to the tool support offered to discover a framework or language during development without consulting a separate documentation. This means that tool support is telling the developers what features the framework or language can offer. It is closely related to the learnability and depends on the transformation language and tool. Thus, it is more related to the used transformation language and tool than to the model transformation itself. However, the effort that is required to change a model transformation clearly depends on the tool support and the used language in which the model transformation is specified. Thus, it is useful to consider the discoverability support of the used language and tools.

In the case of NTL, we use Visual Studio Ultimate as the tool to specify model transformations. The discoverability of an API has been shown to impact the learnability of a framework massively [RD11, SWM97]. However, we treat it as a part of the support for modifiability as discoverability support usually appears when a developer starts modifying the code, in Visual Studio in the form of IntelliSense support.

Being an internal DSL, NTL actually inherits the basic tool support for discoverability from Visual Studio for free. That is, developer can explore the NTL language by reviewing the public API that is presented to them via the IntelliSense feature of Visual Studio. As soon as a developer starts typing, Visual Studio pops up an IntelliSense window to show the keywords, functions or type names that are available in the context of the cursor position. The developer can then select the method or type he wishes to access in that IntelliSense list and Visual Studio automatically shows a tool tip showing the inline documentation for that method and its signature. When the developer starts to type in the parameters of a method, Visual Studio knows the parameter index and shows up the documentation for that parameter. In NTL, every publicly visible method (including `protected` methods) has this inline documentation set on the summary (which is the field that is actually displayed by IntelliSense) and all parameters. If the code is already written, Visual Studio also shows this inline documentation as soon as the developer moves the mouse over a member or type reference.

It has been a design goal of NTL to work with this IntelliSense feature as far as possible. However, the nature of an internal DSL make this infeasible at some points. The most prominent example of this is how transformations consist of transformation rules. The transformation developer has to know that he has to create a class that inherits from `ReflectiveTransformation` and specify the rules of the transformation as public nested classes. The IntelliSense feature of Visual Studio does not support the developer at this point. The supports starts where the developer begins to specify dependencies of a transformation rule. As these dependencies are specified through method calls in the `RegisterDependencies` method, the IntelliSense feature of Visual Studio can support the transformation developer by providing him with the information of what dependencies can be specified as it shows what methods are available to call.

However, that same IntelliSense feature does not prevent a transformation developer to specify a dependency inside the `Transform` or `CreateOutput` method that actually does not have an effect. In this way, NTL is discoverable to some extend, but further tool

support would be required to forbid unwanted calls like creating a dependency inside methods that are not meant for this.

8.2.2. Change impacts

Another important aspect of modifiability is the change impact of various kinds of changes. As they occur most often, we concentrate on perfective changes. The easiest form of perfective changes to a model transformation is when the source metamodel is extended thus allowing new sorts of input models. How these extensions to the source metamodel could look like depends on the used meta-metamodel but in many cases, such extensions consist of newly introduced metaclasses that are linked through inheritance or new references.

In NTL, features such as transformation rule instantiation have been designed to enable transformation designs that minimize the change impacts of e.g. perfective changes (see section 7.4). However, it is subject to further validation to validate how these features compare to similar features of other model transformation languages.

8.2.3. Debugging

Developers often make mistakes and so do transformation developers. Debugging is a valuable technique to locate these bugs more precisely as the developer can step through the execution of the model transformation.

As an internal DSL for C#, NTL inherits the debugging support from Visual Studio for arbitrary C# code. Thus, transformation developers can set breakpoints in the transformation code. These breakpoints may also rely on custom conditions or may only break on a certain hit count. Thus, the transformation developer is able to understand the execution of the model transformation very precisely. As NMF Transformations is open-source, he can also download the source code and see how the control flow is executed internally (although he then has to master the complex code of NMF Transformations Core). As soon as the normal execution stops in a breakpoint, the developer can step through the code and see what exactly happens. The state of the model transformation and the transformed models can be reviewed. As Visual Studios watch window allows to specify arbitrary watches, every detail of the transformation state can be watched during a breakpoint. The only exception is that the watch window does not allow to use lambda expressions in watches.

Furthermore, it is possible to change the code while the execution is on a breakpoint, and continue the execution with the changed code (in Visual Studio 2012, this is only possible when the transformation is run in 32bit-mode but this restriction will be withdrawn in Visual Studio 2013).

8.2.4. Testing

For avoid bugs in software artifacts, it is beneficial to use automated tests for any kind of software artifact. Thus, it is important to test also model transformations. Thus, the model transformation language must support the creation of unit tests to test the model transformation.

NTL has a dedicated support for tests as explained in section 7.4.8. However, it is subject of evaluation how these features can really be used to test the transformation rules of a model transformation separately and independent of the other transformation rules.

8.2.5. Refactorings

"A system used in a real-world environment must change to avoid becoming progressively less useful" and "As a system evolves, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying this structure" [Leh74]. What became famous as Lehman's laws has a direct impact to the maintenance of model transformations in the direct need of refactoring operations to preserve the quality of model transformation in terms of its structure. This includes simple operations such as renaming operations of transformation rules but also more complex refactoring operations that really change the structure.

Such refactoring operations are typical maintenance tasks. Thus, the tool support of a transformation language for refactoring operations is important for the maintainability of model transformations created with that language and tool.

As an internal DSL, NTLinherits the refactoring support for arbitrary C# code from Visual Studio. This e.g. includes renaming operations. As a transformation rule in NTL is a public nested class and Visual Studio supports the renaming of classes, transformation rules can be renamed fully automatically. Further refactoring support is offered by additional tools such as ReSharper[1] that offers a range of further refactoring operations such as "extract superclass" [FB99]. Many of these refactoring operations also apply on model transformations written with NTL. For example, the renaming operation can be applied to rename transformation rules. Other built-in refactorings such as extracting a method can be used to increase the modularity of the transformation rules.

However, it is also a subject of further validation how the refactoring support compares to the refactoring support of other model transformation languages.

8.3. Reusability

It has been discussed in section 7.4.7 how the language features of NTL make it possible for transformation developers to reuse transformation rules and whole model transformations across multiple model transformation rules. In this way, NTL indeed allows model transformations and parts of it to be reused. Furthermore, all actions that are done inside the `Transform` and `CreateOutput` methods can further be moved to dedicated components that further enhance reusability.

It is not further discussed how the language features of NTL for reusability compare to related language features of other model transformation languages, at least within this master thesis, due to limitations in space and time.

8.4. Modularity

Being an internal DSL with C# as its host language, code for a model transformation can be split across arbitrary many files and assemblies. However, splitting up an assembly in multiple different assemblies requires to use transformation composition techniques as a class in C# cannot be split across multiple assemblies. However, C# indeed allows to split a class definition across file boundaries by using `partial` classes. With this feature the C# compiler combines the fragments of a class declaration to a single type that gets compiled in an assembly. With this technique, it is possible to modularize the specification of a model transformation and have the different parts locked in the version control system by different developers. This enables developers to work on a single model transformation in parallel.

[1] `http://www.jetbrains.com/resharper`

However, the fact that NTL makes it possible to modularize the specification of a model transformation, this does not mean that the way the model transformation is split into parts would always make sense. A meaningful and clear separation of a model transformation into parts that can be developed independently is a non-trivial task. In many cases, it would make much more sense to create separate model transformations instead and combine these model transformations using the features from section 7.4.7. Thus, separating a model transformation in meaningful parts is more like a general open research question than a concern dedicated to NTL. Thus, the modularity of model transformations is not evaluated across multiple model transformation technologies.

8.5. Completeness

The completeness of a model transformation is the extend in which a model transformation conforms to its requirements and therefore first and foremost depends on these requirements. The impact of the language design of NTL only consists in the way whether NTL makes it possible to fulfill these requirements. However, NTL even allows to put arbitrary general purpose code inside the `Transform` and `CreateOutput` methods. Thus, it is possible to compute everything that is computable with general purpose code. Having said that, not all requirements can be gracefully supported by NTL. Thus, it is a subject of further evaluation to find out which types of model transformation can be supported by NTL.

Note that this statement does not hold for external DSLs so easily. E.g., it is unclear whether TGGs can fulfill every possible requirements for a model transformation. The worst thing that could happen to a model transformation developer is that some requirements must be fulfilled entirely in general purpose code. Other transformation languages may require to use general purpose pre- or postprocessing.

8.6. Consistency

The consistency is a quality attribute that the used model transformation language has the biggest impact on. After all, a model transformation can only be inconsistent when the model transformation language provides multiple programming styles. On the other hand, a model transformation can only be consistent if the requirements can be fulfilled with a relatively small number of concepts that can be used throughout a model transformation. A variety of different concepts to achieve the same results in different ways may lead to developers that mix these concepts within their model transformation, making the model transformation less consistent. Although this variety can be suppressed by code guidelines, transformation languages that offer very limited but powerful concepts may better prevent inconsistent model transformations, as they just not offer the choice for developers.

NTL itself takes advantage of this possibility of few but powerful concepts in that way that its entire functionality is built on the NMF Transformations Core concept of dependencies, the trace component and patterns. Leaving aside the relational extensions, all that NTL does is to expose these few concepts in a type-safe way thereby using loads of convenience methods. Furthermore, NTL also adds some tracing functionality that can be seen as newly introduced concepts. However, some effects that could be achieved by using this tracing functionality can also be achieved using the `Call` and `Require` methods. Furthermore, a transformation developer may choose whether to persist the outcome of a dependency in the persistor of the dependency or to use the trace functionality in the `Transform` method. Both programming styles achieve the same results (although executing persistence logic in the persistor is slightly more performant). Thus, NTL offers a possible way for inconsistency in the model transformations.

This problem gets worse as soon as relational extensions come into place. As they are to concisely specify when to call a certain transformation rule, they offer an alternative to the specification of dependencies. This existence of an alternative solution eventually moves developers to use this alternative and thus, the model transformation becomes inconsistent. Even if a single model transformation was consistent, many model-driven software development projects contain more than just one model transformation and inconsistencies among these model transformations also would yield similar problems as inconsistencies within a single model transformation.

However, NTL is designed to allow transformation developers to put arbitrary code in the `CreateOutput` and `Transform` methods. Thus, the consistency can easily be hampered when the model transformation uses general purpose code when there had been an appropriate abstraction provided by NTL. NTL cannot take care of this as this is a responsibility of the transformation developers. This only thing that can be done to prevent such situations is to clearly describe, which type of model transformation tasks can be gracefully supported by NTL. To do this, further evaluation is required to get insights to the type of model transformations that can be supported with NTL.

Nevertheless, NTL does not prevent developers from writing consistent model transformations. However, guidelines are necessary to developers when to use which concept. It is subject of further evaluation how many different concepts would be used for a typical model transformation task. In fact, some programming guidelines are contained in this thesis in section 7.4.

8.7. Conciseness

Being an internal DSL, NTL does hamper the conciseness of the resulting model transformations in that way that it expects certain syntactic elements, i.e. has some syntactic noise. As an example, NTL expects the declaration of dependencies within the `RegisterDependencies` method that must be overridden. Thus, when a transformation developer wants to specify that a transformation rule has a dependency, he has to write down the complete signature of the `RegisterDependencies` plus the keyword `override` to specify that the contents of this methods should be used to set the dependencies to other transformation rules. In an external DSL, the language could specify a single new keyword to achieve the same semantics. Thus, repeating the signature of the overridden method is a clear point of syntactic noise that hampers the conciseness of the resulting model transformation.

However, this syntactic noise helps developers not used to model transformation too much to understand what is actually going on. As a consequence, they do not need to wonder how the magic underneath could be realized, but simply recognize the fact that they need to override a certain method much similar that they would need a certain keyword in an external language. Furthermore, the tool support allows it that the developers do not have to remember the entire signature but simply have to remember the name of the method they need to override. As soon as they write the `override` keyword, Visual Studio actually suggests a list of methods that can be overridden and by using the code completion support, developers just have to choose the method they wish to override and press the tab key (also the `override` is subject to code completion as it usually suffices to type ”ov” and press the tab key). Furthermore, by repeating the entire signature of the overridden method, the developer knows exactly which variables he can access and what type these variables are (provided that he has a principle understanding of the C# language) whereas in some external languages, it is not always obvious to the developer which variables he has access to or their type. However, expressing the semantics in a single concise keyword instead of whole method signature also has advantages for the

understandability and thus maintenance as the single keyword attracts more attention to developers, especially because keywords are mostly highlighted by syntax-highlighting.

Another point where a concise declaration of a model transformation hampers the understandability of model transformations specifically in NTL is for example the usage of named parameters. The compiler is actually able to choose the correct overload without specifying the name of any parameters name. Thus, semantic in the name of a parameter is completely redundant, resulting in a less concise specification of the model transformation. However, by writing the name of the parameters, developers can greatly enhance the understandability of the model transformation. Anyone who reads the code does not have to look up the meaning of a specific parameter as this meaning is encoded in meaningful parameter names. However, although C# does support named parameters, many other languages as Java do not and hence, this statement cannot be generalized for all internal DSLs.

Thus, if rich tool support as for example provided by Visual Studio is present, a concise syntax is not as important to maintainability. The verbosity of languages like C# in terms of how the language is suitable for model transformation can be balanced by the tool support, especially in terms of writing the code. But it is not only code completion. Alternative visualizations of the code can also serve to enhance the understandability of a solution in a similar way as a concise solution does. Tool support may indicate the outline of a code file, thus cutting the time to navigate to a specific piece of code.

On the other hand, conciseness does not only consist of a concise model transformation language, but also in the conciseness in which the model transformation can be expressed using the model transformation language. This means whether the language contains appropriate abstractions.

9. TTC Flowgraphs case study

In this chapter, the Flowgraphs case of the TTC 2013 and a solution using NMF Transformations are presented. First, an overview on the case description is given in section 9.1. Section 9.2 explains the planned validation for this case study with the validation criteria and how these criteria are evaluated. The next section 9.3 covers the solution of the Flowgraphs case with NMF Transformations. Next, section 9.4 briefly introduces the other solutions at the TTC. Section 9.6 then presents the results from the TTC. The actual evaluation of this case study is performed in section 9.7, before section 9.8 concludes this chapter.

The solution of the Flowgraphs case using NMF is based on [HGH13a] meanwhile the case description is based on [Hor13].

9.1. Case Overview

The Flowgraphs case of the TTC [Hor13] is about creating both control flow graph and data flow graph for a source code written in Java. For this purpose, this Java source code is turned into a JaMoPP model using the JaMoPP parser [HJSW09]. The JaMoPP metamodel is an extensive representation of a Java source code. However, to simplify the transformation a bit, only a subset of this metamodel was promised to be used in the test models. This subset of 39 metaclasses was presented in the case description. Furthermore, the models used for testing would only contain a single class with a single method.

In a first initialization step, this extensive representation is simplified by transforming it into a separate metamodel for flow graphs provided in a GitHub repository[1]. The part of this metamodel related to the structure of a method is presented in figure 9.1. Since the Flowgraphs metamodel only contains a single class for expressions, whereas the JaMoPP metamodel contains a fine structure of expressions, this initialization task involves an embedded Model-to-Text transformation that represent a complex expression as simple text.

The second task consists of deriving the control flow within the Flowgraphs model derived in the first step. This is an in-place model transformation that populates the *cfNext* and *cfPrev* references of flow instructions (the according part of the Flowgraphs metamodel is presented in figure 9.2).

[1] `https://github.com/tsdh/ttc-2013-flowgraphs-case`

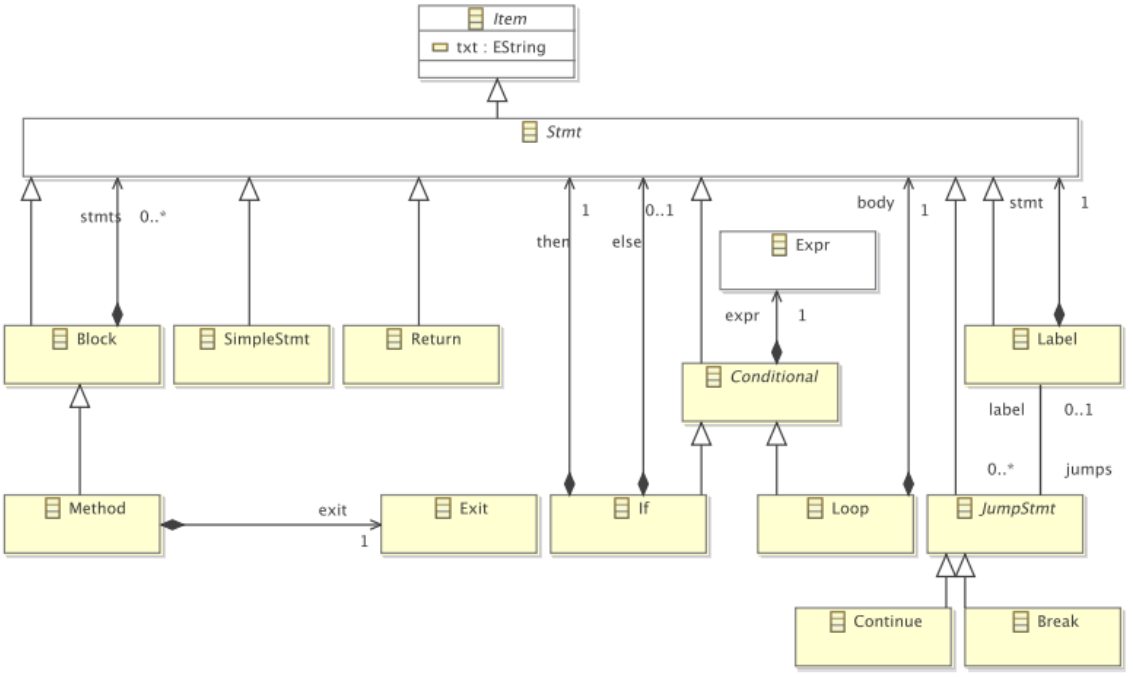

Figure 9.1.: The metaclasses of the Flowgraph metamodel describing the structure of a method [Hor13]

The third task is about deriving the data flow within these methods. For this purpose, the Flowgraphs metamodel contains further references that allow to specify the data flow in a method (see figure 9.3).

The third task has actually been divided into two subtasks. The first subtask consisted of extending the initialization to allow to further populate the definitions and usages of variables. These variables can either be local variables or parameters, where parameters represented by the *Param* class is considered to be a special case of a local variable represented by the *Var* metaclass. The second subtask is to use these references to compute the data flow.

In a last task, solutions for the Flowgraphs case are to provide means for experienced Java developers to check assertions on the created Flowgraph models. Thus, a simple DSL has to be provided that allows developers to check whether assertions on the resulting flow graphs hold. These assertions can be either on control flow or data flow. As an example,

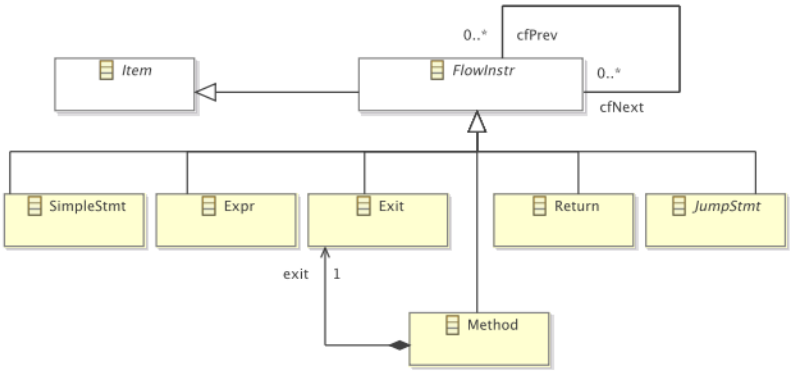

Figure 9.2.: The metamodel classes of the Flowgraph metamodel related to control flow [Hor13]

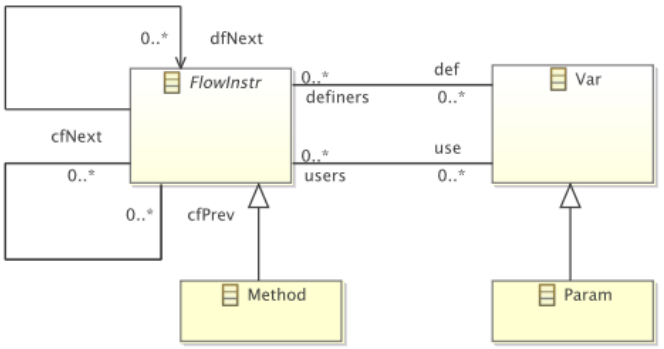

Figure 9.3.: The Flowgraphs metamodel elements related to data flow [Hor13]

the case description suggests a language like the language shown below in listing 9.1.

```
1  cfNext: "testMethod()" --> "int a = 1;"
2  cfNext: "int a = 1;" --> "int b = 2;"
3  cfNext: "int b = 2;" --> "int c = a + b;"
4  cfNext: "int c = a + b;" --> "a = c;"
5  cfNext: "a = c;" --> "b = a;"
6  cfNext: "b = a;" --> "c = a / b;"
7  cfNext: "c = a / b;" --> "b = a - b;"
8  cfNext: "b = a - b;" --> "return b * c;"
9  cfNext: "return b * c;" --> "Exit"
10
11 dfNext: "int a = 1;" --> "int c = a + b;"
12 dfNext: "int b = 2;" --> "int c = a + b;"
13 dfNext: "int c = a + b;" --> "a = c;"
14 dfNext: "a = c;" --> "b = a;"
15 dfNext: "a = c;" --> "c = a / b;"
16 dfNext: "a = c;" --> "b = a - b;"
17 dfNext: "b = a;" --> "c = a / b;"
18 dfNext: "b = a;" --> "b = a - b;"
19 dfNext: "c = a / b;" --> "return b * c;"
20 dfNext: "b = a - b;" --> "return b * c;"
```

Listing 9.1: The example validation DSL from the case description

The resulting solutions for this case are then to be evaluated by an evaluation scheme. Solutions are generally evaluated per task. The biggest influence on the evaluation of the tasks has the factor *Completeness & correctness* which is weighted with 50%. The factor *Conciseness & understandability* is weighted with 30% and the *Performance* of the solutions is weighted with 20%. Furthermore, the tasks have different weightings. While the first task is weighted with 30%, task 2 as presumably the most difficult is rated with 40%. The two subtasks of task 3 and task 4 get weights of 10% each.

9.2. Planned validation

In this section, the validation for this case study is prepared. Section 9.2.1 introduces the criteria for this validation while section 9.2.2 describes how the validation is done for these validation goals.

9.2.1. Validation criteria

The huge advantage of the TTC is that this case study is not only solved by NTL, but also by several other teams, employing several other transformation languages. As every team tried its best to compete with the other teams, one can expect that the differences of the solutions in terms of model transformation quality attributes largely depend on the used model transformation tool and language rather than the enthusiasm of the teams. This yields the great possibility to compare NTL as a transformation language with other state-of-the-art model transformation languages.

However, as maintainability is very difficult to grasp, we concentrate on several specific points that will be evaluated with this case study. The validation goals for the TTC Flowgraphs case are as follows:

- **Applicability:** By solving the Flowgraphs case, it is evaluated that NMF Transformations is an applicable technology to solve a wide range of model transformation tasks including typical mappings as well as embedded M2T-transformations as well as in-place refinement transformations.
- **Understandability:** The evaluation sheet for the solution presentations at the TTC conference included a question for the combined assessment of understandability and conciseness. As the conciseness can be evaluated in terms of the LOC metric, conclusions may be drawn from this data to the understandability of the NMF solution and thus to the understandability of model transformations with NMF Transformations.
- **Modifiability:** As discussed in section 8.4, the discoverability of the model transformation languages takes a big proportion of the modifiability of a model transformation. Thus, it is an evaluation goal to compare the support for discoverability of the different transformation languages.
  Furthermore, as the Flowgraphs case contained a restriction on the input metamodel, it also yields a clear maintenance scenario. In the future, requirements may change to support other parts of the input metamodel previously not supported by the model transformation. In fact, some of the later test models of the Flowgraphs case that were published after the initial solutions had to be handed in actually contained instances of metaclasses not on the list of used metaclasses provided by the case description. Such perfective changes in requirements are the most common type of changed requirements [LS81] and it is therefore crucial for the maintainability of a model transformation to support this scenario. Thus, the change impact of such a perfective change is evaluated in this case study, i.e. how many places of the model transformation code have to be changed to support new metaclasses from the JaMoPP metamodel. The most typical changes in this scenario are introductions of new kinds of statements or expressions.
  The debugging, testing and refactoring support are not validated in this case study as the other case studies better suit these evaluation disciplines.
- **Consistency:** The consistency of a model transformation is not reviewed in the open peer reviews. Instead, the consistency of all solutions is evaluated through discussion. However, the results may be debatable. As discussed in section 8.6, the consistency of a model transformation defined with NTL greatly depends on the way transformation developers stick to programming guidelines. Whether or not developers stick to programming guidelines greatly depends on whether these guidelines are clearly documented. But as the solution to the Flowgraphs case has been made by myself that wrote these guidelines, it cannot be evaluated how these guidelines affect the consistency of the resulting model transformations.

- **Conciseness:** The conciseness can be measured in terms of the *Lines of Code* metric. However, there exist different ways to catch up this metric. Thus, the evaluation must not only rely on the plain results of *Lines of Code*. Instead, there is further evaluation required to account for e.g. syntactic noise but also blank lines.

The performance of the solutions is not taken as a validation criteria as it is unimportant for the maintenance.

9.2.2. Validation procedure

The understandability is the quality attribute that is the hardest to compare as it is hard to evaluate other than through its perception. However, the TTC evaluation scheme did not involve questions only on the understandability of the solutions but there were questions in the open peer reviews and in the TTC conference to ask for the combination of understandability and conciseness. With the combination of this data and the results of the *Lines of Code* (LOC) metric to measure the conciseness, it may be possible to draw conclusions for the understandability of the solutions. However, these statements on the understandability must then be used with care.

The discoverability support is analyzed in a discussion. All of the solutions are available in SHARE([VGM11]) demos online[2]. These demos have been used to analyze the discoverability support of these languages.

For the size of the change impacts, we simulate the following scenarios:

- Add support for a new kind of statements.
- Add support for a new kind of expressions.

The validation then evaluates how many places in the transformation must be changed. Especially, it is desirable that the change impact on existing code is as low as possible.

The consistency is evaluated in a discussion based on the solutions available in SHARE. The definition of consistency is not exactly clear where to speak of a different programming style. Furthermore, if multiple programming styles are used within a solution, there may be a continuous pass between them. After all, it is unclear how to exactly define a programming style. However, one can definitely assume different programming styles, if multiple programming languages are used. Thus, the consistency is discussed based on the amount of different programming styles used within a solution.

The conciseness of the solutions is measured in LOC. However, when the LOC metric is compared across language boundaries, one has to account for the influence of different coding-styles. As an example, some languages like C# are typically used putting braces on separate lines. While this dramatically increases LOC, it is not really less concise. A similar factor is the different impact of aspects of conciseness. While syntactic noise indeed hampers the conciseness, it may not be as bad as unsuitable abstractions. Thus, a further discussion is required for the evaluation of conciseness.

9.3. NMF solution

All of the tasks have been tackled. The final solution consists of two separate C# console projects. The first console application reads a JaMoPP file, transforms it to a Structure Graph (Task 1 and Task 3.1, see sections 9.3.1 and 9.3.3.1), derives the control flow (Task 2, see section 9.3.2) and sets the data flow (Task 3.2, see section 9.3.3.2). Each of the transformations operates in memory, only. After all transformations have been

[2] `http://goo.gl/rgGBJ`

applied, the resulting data flow model is persisted using XMI into an output file specified by command line parameters. Furthermore, deriving the control flow graph and setting the data flow can be switched on or off using command line arguments. The second console application validates the links (see section 9.3.4).

9.3.1. Task 1: Initialization

The inner structure of expressions in the JaMoPP has to be mapped to strings. The task description suggested to embed a M2T-transformation. However, NMF currently does not support M2T-transformations and thus, the transformation is implemented as a M2M-transformation where the target model is strings. However, this has an impact on the development. Strings are immutable in .NET and thus it cannot be modified during transformation stage. Hence, we have to do the main work within the *CreateOutput*-method. Alternatively, we could have used string buffers.

```
1  public class Method2Method : TransformationRule <JaMoPP.Members.
       ClassMethod, Flowgraph.Method>
2  {
3   public override Flowgraph.Method CreateOutput(JaMoPP.Members.
       ClassMethod input, ITransformationContext context)
4   {
5    return new Flowgraph.MethodImplementation();
6   }
7
8   public override void Transform(JaMoPP.Members.ClassMethod input,
       Flowgraph.Method output, ITransformationContext context)
9   {
10   output.Txt = input.Name + "()";
11
12   output.Exit = new Flowgraph.Exit()
13   {
14    Txt = "Exit"
15   };
16  }
17
18  public override void RegisterDependencies()
19  {
20   RequireMany(Rule<Statement2Stmt>(),
21    selector: method => method.Statements,
22    persistor: (method, statements) => method.Stmts.AddRange(
         statements));
23  }
24 }
```

Listing 9.2: The transformation rule to transform a method

Let us begin with the code of transforming a method. This code is shown in listing 9.2. The code snippet specifies a transformation rule that transforms a `ClassMethod` from JaMoPP into a `Method` in the Flowgraphs model. Because the metaclass `Method` inherits from two metaclasses, it is generated as an interface, together with a default implementation. As NMF Transformation cannot instantiate an interface directly, we have to tell it to create an instance of `MethodImplementation` instead. In the `Transform`-method, the properties of the method are being set. The `RegisterDependencies`-method states

that whenever a method is transformed, all the statements also have to be transformed (using the `Statement2Stmt`-rule) and the transformed statements are to be stored in the transformed method. Other transformation rules instantiate the `Statement2Stmt`-rule to handle all possible cases where JaMoPP statements have to be transformed into *Stmt* elements.

Combining the expressions to a single string works similar. Much like `Statement2Stmt`, a transformation rule `Expression2Text` is created as base for transforming an expression into a string. As you can see in listing 9.3, the transformation rule `Expression2Text` is empty and does not know anything about the derived classes of *Expression*. This reflects that there is no common transformation of a general expression into text. Instead, the transformation is entirely specific to the specific type of the expression.

```
1 public class Expression2Text : AbstractTransformationRule <JaMoPP
    .Expressions.Expression,string> { }
```

Listing 9.3: The abstract rule to create text from expressions

This rule is then instantiated by transformation rules that can transform the derived classes of Java expressions. An example is the *AssignmentExpression*. The transformation rule `AssignmentExpression2Text` is presented in listing 9.4 as an example of such transformation rules.

```
1  public class AssignmentExpression2Text : TransformationRule<
       JaMoPP.Expressions.AssignmentExpression, string>
2  {
3   public override string CreateOutput
4    (JaMoPP.Expressions.AssignmentExpression input,
       ITransformationContext context)
5   {
6    var child = context.Trace.ResolveIn(Rule<Expression2Text>(),
       input.Child);
7    var value = context.Trace.ResolveIn(Rule<Expression2Text>(),
       input.Value);
8    var assignment = context.Trace.ResolveIn(Rule<
       AssignmentOperator2Text>(), input.AssignmentOperator);
9
10   return child + assignment + value;
11  }
12
13  public override void RegisterDependencies()
14  {
15   MarkInstantiatingFor(Rule<Expression2Text>());
16
17   Require(Rule<Expression2Text>(), expr => expr.Child);
18   Require(Rule<Expression2Text>(), expr => expr.Value);
19   Require(Rule<AssignmentOperator2Text>(), expr => expr.
       AssignmentOperator);
20  }
21 }
```

Listing 9.4: The transformation of assignment expressions

As a consequence of the instantiation, whenever an *Expression* is transformed to text and this expression is an *AssignmentExpression*, the `AssignmentExpression2Text` rule

is called instead to create the output of the `Expression2Text`-rule. `Expression2Text` is still called but here it is empty and only serves as a hub.

With this technique, the transformation can be written without any casting operator or type check. All the type checks and castings are done by NMF Transformations and are thus properly secured. The introduction of new metaclasses (and there are a lot more of them in JaMoPP) only requires a new rule whereas the rest of the transformation may stay untouched. This yields a small change impact and therefore good maintainability.

The `Require` method calls define dependencies. In example, whenever an `AssignmentExpression` is transformed into text, also its `Child`, `Value` and `AssignmentOperator` must be transformed with the appropriate transformation rules. Usually dependencies can have persistors but as the transformation targets to strings, these persistors cannot do anything as strings are immutable. Thus, we need to call the results of these dependencies within the method that creates the output of the transformation rule, which is `CreateOutput`. There, we can use the transformation context instance for tracing.

9.3.2. Task 2: Deriving Control Flow

To derive the control flow graph, semantical information like the first flow instruction within a statement has to be added to existing model elements. But it is not only data, it is also the behavior of how to set the control flow that is important for this transformation. Thus, we first define the behavior that we need from a flow instruction to be able to set the control flow. This definition is implemented as an interface which is presented in listing 9.5.

```
1 public interface IControlFlowInformation
2 {
3  FlowInstr First { get; }
4  FlowInstr Successor { get; set; }
5  void SetControlFlow(Stack<Stmt> callHierarchie,
       ITransformationContext context);
6 }
```

Listing 9.5: The interface of what is interesting regarding control flow

At first, we need the first flow instruction since the entrance in a statement (`First`). However, in some cases like empty blocks, this first flow instruction is not part of the current statement. Therefore, we need to tell what the next flow instruction after the current statement is (`Successor`). Finally, the procedure of how to set the control flow for a statement also depends on a statement. A simple statement only sets the CfNext reference to the first inner flow instruction of the successor statement, whereas a jump statement sets a CfNext reference to the target jump label. However, some statements like `break` and `continue` cannot set their control flow predecessor without context. I.e., the predecessor of a continue statement is the test expression of the innermost loop that the continue is contained in. Additionally, the method also has the transformation context as parameter for tracing purposes.

NMF Transformations does not draw a difference between objects that are part of the model and objects that are just helpers. Thus, we can just use an abstract transformation rule that returns an instance of the above interface for any statement. The nature of `Successor` and `First` are quite different. Whereas `First` can only be derived in a bottom-up manner, `Successor` must be set in top-down fashion. Thus, we use the above interface to first set the `Successor`. As soon as this is done, the `First` reference is available and we can derive the control flow. Unfortunately, an interface declaration is not expressive

enough to specify such a protocol. As an example, this procedure is demonstrated by the `SetControlFlow` implementation for Blocks in listing 9.6.

```
1  public void SetControlFlow(Stack<Stmt> callHierarchie,
       ITransformationContext context)
2  {
3   //Set Successors
4   Children[Children.Count-1].Successor = Successor;
5   for (int i = Children.Count - 2; i >= 0; i--)
6   {
7    Children[i].Successor = Children[i + 1].First;
8   }
9
10  //Set Contriol Flow
11  callHierarchie.Push(Parent);
12  for (int i = 0; i < Children.Count; i++)
13  {
14   Children[i].SetControlFlow(callHierarchie, context);
15  }
16  callHierarchie.Pop();
17 }
```

Listing 9.6: SetControlFlow-method for Blocks

Finally, the control flow of the method is set by simply calling the `SetControlFlow`-method of the methods control flow interface implementation and thereby utilizing the helper model induced by the that interface. At the method level, the method also knows the successor of the statements contained in the method which is the methods *Exit* element. The code is demonstrated in listing 9.7. However, most of the work is done in the `SetControlFlow` of the individual interface implementations for each of the statement types.

```
1  public class DeriveMethodControlInfo : InPlaceTransformationRule<
       Method>
2  {
3   public override void RegisterDependencies()
4   {
5    Require(Rule<Block2CFInfo>());
6   }
7
8   public override void Transform(Method input,
       ITransformationContext context)
9   {
10   var cfInfo = context.Trace.ResolveIn
11    (Rule<Block2CFInfo>(), input);
12
13   cfInfo.Successor = input.Exit;
14   cfInfo.SetControlFlow(new Stack<Stmt>(), context);
15   input.CfNext.Add(cfInfo.First);
16  }
17 }
```

Listing 9.7: Deriving the control flow for a method

Again, we only used the transformation rule instantiation for pattern matching. As a consequence, deriving the control flow for any additional metaclass is just as easy as

defining a new transformation rule that instantiates one of the existing transformation rules.

9.3.3. Task 3: Deriving Data Flow

As the task description of the third task is divided into two subtasks, so is the solution that are presented in the subsequent paragraphs of this section. The extended initialization is described in section 9.3.3.1 while the data flow generation is presented in section 9.3.3.2.

9.3.3.1. Task 3.1: Extended Initialization

The deduction of the variable declarations and usage changes the whole part of the transformation that deals with expressions. From now on, it is not only the string expression that is interesting for an expression. There is also the information which variables are used. Although this might seem easy to do, there are some issues with that. The problem is that the information is split up to several parts across the model. An `IdentifierReferenceExpression` knows which variable is accessed but it does not know whether it is written or read.

As this task is very similar to task 2, the solution is also quite similar. Again, we first define an interface of what is interesting for us from an *Expression*. The resulting interface is presented in listing 9.8. The `Expression` property is just to return the expression string from the *Expression* instance. The enumeration `UsedVariables` tells the used variables in the *Expression* and `SetDefs` sets the defined variables by that Expression. The `LastVariable` reference tells the last variable within an expression.

```
1 public interface IExpressionDFInfo
2 {
3  string Expression { get; }
4  IEnumerable<Flowgraph.Var> UsedVariables { get; }
5  Flowgraph.Var LastVariable { get; }
6  void SetDefs(Flowgraph.FlowInstr parentFlow);
7 }
```

Listing 9.8: The interesting attributes for an Expression

In this transformation, however, the information on the order of read and write accesses is lost. The metamodel of the FlowGraph just does not provide this detail in this precision. In an expression like

```
i = i++ + i++;
```

the variable i is read two times and is defined three times. However, the order of read and write accesses is lost. However, we can hope that these cases rarely occur.

We use a `LastVariable` property to get the variable that has been accessed in the most recent expression. This is necessary to prevent an expression like

```
i + 2 = i;
```

to produce definition of the variable i. We could also claim that expressions like these are not allowed. However, if we imagine a as a more complex object like an array, we still want to know that the statement

```
a[i] = 0;
```

defines the variable a (to be more precisely, it defines the contents of the array a) but uses the variable i. Thus, we cannot, in general, just use the `UsedVariables` property of the left side of an expression to get the variables that are defined by this assignment statement. For this purpose, the `LastVariable` property is used that returns the variable that is a candidate for a definition. The variables within `UsedVariables` of an assignment statements left side are read, except for the last variable that is written to.

9.3.3.2. Task 3.2: Deriving Data Flow

Data flow is usually derived from the control flow through an iterative procedure. However, in NMF Transformations, it has been a principle design decision that transformation rules may only be executed once per transformation context and input. Thus, NMF Transformations is not well suited for this in-place transformation and thus, we solve this transformation task in general purpose code. In the first versions (including the version handed in to the TTC, the algorithm simply went through all flow statements and looked for the defined variables. For each flow instruction that defines a variable, it walked all possible ways through the control flow graph and set DfNext edges whenever it found some expression using that variable until this variable is redefined.

However, it turned out that it is more efficient to perform this task the other way around. For every flow instruction within the method, we iterate through the used variables and look for all possible definers of that variable by following the control flow backwards. As NMF Transformations operates on POCOs ("Plain Old C# Objects"), integrating this general purpose code in the transformation is just as easy as calling the algorithm. There is no conversion that has to be done.

```
1  public static void SetDataFlow(Method method)
2  {
3   var instructions = method.Closure<FlowInstr>(m => m.CfNext);
4   foreach (var instr in instructions)
5   {
6    foreach (var use in instr.Use)
7    {
8     SetDefinitions(instr, instr, use, new HashSet<FlowInstr>());
9    }
10  }
11 }
```

Listing 9.9: The algorithm to set the control flow

The implementation is presented in listing 9.9. The algorithm makes use of the `Closure` operation offered by NMF Utilities. This collects all expressions that are reachable through `CfNext` references. For any used variable of any expression, we then set the possible definers. The code for this operation is shown in listing 9.10.

```
1  private static void SetDefinitions(FlowInstr current, FlowInstr
       root, Var variable, HashSet<FlowInstr> visited)
2  {
3   foreach (var prev in current.CfPrev)
4   {
5    if (!visited.Contains(prev))
6    {
7     visited.Add(prev);
8
9     if (prev.Def.Contains(variable))
```

```
10      {
11        prev.DfNext.Add(root);
12      }
13      else
14      {
15        SetDefinitions(prev, root, variable, visited);
16      }
17    }
18  }
19 }
```

Listing 9.10: The initialization algorithm for deriving the data flow

The `SetDefinitions` function is recursive. It goes back the control flow. If the visisted flow instruction is not already marked visited, the algorithm marks this instruction visited. If the instruction defines the variable that is looked for, the function terminates. Otherwise, it recursively looks backwards in the control flow graph. The hash set is used to remember which flow instructions already have been visited in order to prevent endless loops.

9.3.4. Task 4: Validiation

As the proposed query language is very simple, it suffices to solve the problem with regular expressions. If the language became more complex, there would be some parser generators such as Irony[3] that can parse strings based on EBNF grammars. But as mentioned, in this case regular expressions suffice. To parse a command, we use the following regular expression:

```
(?<command>(cfNext|cfPrev|dfNext)):\s*
"(?<source>[^"]*)"\s*-->\s*"(?<target>[^"]*)"(;)?
```

Note that this regular expression uses the syntax of .NET regular expressions. This means it introduces character classes like whitespaces (`\s`) and catches parts of the expression in groups for later reference, as in `(?<groupname>)`. The validation application now just reads a target model and creates an internal hashtable with all the instructions that are contained in the model. Any validation string is then parsed with the above pattern and the application simply checks whether the asserted condition holds.

9.4. Other solutions

This section briefly introduces the other solutions to the Flowgraphs case at the TTC. However, this introduction only covers the tasks 1, 2 and 3.1 as the tasks 3.2 and 4 are out of the scope of this thesis. All solution descriptions will be published in brief form in the formal post-proceedings of the TTC. More detailed descriptions are available on the TTC homepage under `http://planet-sl.org/community/index.php?option=com_community&view=groups&task=viewgroup&groupid=24`. Demos of the solutions can be found on SHARE[4].

For the comparison to the NMF solution, only those code snippets were used that also appeared in the solution papers, as these code snippets are typically selected to demonstrate the strengths of these approaches.

[3] `http://irony.codeplex.com/`
[4] `http://goo.gl/rgGBJ`

9.4.1. FunnyQT

Like NTL, FunnyQT[5] is a shallow internal DSL for model transformation. However, unlike NTL, FunnyQT uses the host language Clojure which is a dialect of Lisp that compiles to Java byte-code. The flexible syntax of Clojure yields a very high conciseness of the FunnyQT solution. The FunnyQT solution is quite fast in terms of execution speed. As a reason, being a Lisp dialect, FunnyQT can make use of semantic macros that get expanded at compile time and thus, the transformation gets optimized when the transformation is compiled to Java bytecode. Furthermore, FunnyQT exposes several language features of Clojure that are optimized during compilation.

However, from a maintainability perspective, FunnyQT yields some problems. As also the review of Sanchez[6] stated, FunnyQT lacks of an abstraction layer that makes it useful to non-Clojure programmers. As maintainability depends on the users background, this is harmful to the maintainability. Furthermore, Clojure is not statically typed which has huge impacts on the IDE regarding productivity features like code completion.

```
1 (^:top method2method [m]
2  :from 'members.ClassMethod
3  :when-let [mstmts (seq (eget m :statements))]
4  :to [fgm 'flowgraph.Method, ex 'flowgraph.Exit]
5  (eset! fgm :txt (stmt2str m))
6  (eset! ex :txt "Exit")
7  (eset! fgm :stmts (map stmt2item mstmts))
8  (eset! fgm :exit ex)
9  (eset! fgm :def (map param2param (eget m :parameters))))
```

Listing 9.11: Transforming a method in FunnyQT

Listing 9.11 shows how a method is transformed in FunnyQT. The syntactic noise is reduced to a minimum so that the code for a method transformation fits into just 9 lines of code. However, this listing also shows some vulnerabilities to the maintainability of the resulting transformation. Although a method is also a block, the semantics that the statements inside a block must be transformed by transforming the statements inside the block is repeated. In the small scenario of the Flowgraphs case, this may not be a problem but such duplication of code is always unwanted as it may cause maintenance problems as soon as the transformation grows large.

Furthermore, although the code is very concise, it is not really understandable for developers not used to programming with Lisp dialects. Without an understanding of what functions like *eset!* or even just the colon operator mean, it is hard to understand the code. Further, it is an open question, how many developers are actually able to *write* such code.

```
1 (stmt2item [stmt]
2  :generalizes [local-var-stmt2simple-stmt condition2if
     block2block
3  return2return while-loop2loop break2break continue2continue
4  label2label stmt2simple-stmt])
```

Listing 9.12: The stmt2item rule in FunnyQT

These maintenance problems on the transformation of inheritance hierarchies continue if we look at the definition of the *stmt2item* rule in FunnyQT, see listing 9.12. Same as a

[5] `https://github.com/jgralab/funnyqt`
[6] Available under `http://goo.gl/vdgL7`

disjunct mapping in QVT-O, the transformation rule for the base class has to be aware of the all derived classes that need to be transformed. This yields the same maintainability issues as described in section 7.4.5.

9.4.2. Epsilon

Rather than a single transformation language, the EPSILON[7] solution used multiple languages, the EPSILON OBJECTS LANGUAGE (EOL) and the EPSILON TRANSFORMATIONS LANGUAGE (ETL). As EOL is used by multiple other languages inside the EPSILON language family, this has the advantage that code specified in EOL can easily be reused in other transformations.

Listing 9.13 demonstrates how the model transformation looks like by showing the transformation of conditionals. The listing shows the conciseness in which the model transformation can be specified.

```
1  rule Condition2If
2   transform s:JaMoPP!statements::Condition
3   to t: FlowGraph!If
4   extends JavaElement2Item {
5
6   t.expr = s.condition.equivalent();
7   t.then = s.statement.equivalent();
8   if (s.elseStatement.isDefined()) {
9    t.'else' = s.elseStatement.equivalent();
10  }
11 }
```

Listing 9.13: The rules to transform a condition in ETL

The embedded M2T-transformation as well as the control flow generation are solved by applying the visitor pattern in the essence described in [PJ98] and in section 6.3.

```
1 operation LocalVariable toString() {
2  var type = self.typeReference.toString();
3  var initialValue = self.initialValue.toString();
4  return type + " " + self.name + " = " + initialValue ;
5 }
```

Listing 9.14: The M2T-transformation to transform a local variable to string in EOL

However, what the EPSILON solution basically does to achieve the control flow generation is to use EOL as an ordinary general purpose language. The only feature that the whole solution is really based on is late binding, i.e. to choose the correct overload of a method depending on an objects runtime type rather than by its static type. However, EOL is by far not the only language that supports late binding. As a consequence, the EPSILON solution can be more or less directly translated into C# code without using any framework (except for the model representation) that is just as concise as the EPSILON solution. This solution is presented in section 9.5. This is remarkable as the EPSILON solution won an award for the most concise solution. This leads to the question whether in-place transformations as in task 2 are really an appropriate task for model transformation tasks.

Using these late binding mechanisms does also have a downside. Although the different implementations of the `toString` operation are used together, this dependency only exists implicit. As a result, as soon as transformation developers type the name `toString`

[7] `http://www.eclipse.org/epsilon`

wrongly, an implementation of the method may be ignored to set up the text. The transformation tool cannot support the transformation developer in this as the tool cannot know that a `toSting` operation should belong to the `toString` operations group but is just spelled incorrectly. This is specifically harmful in refactoring scenarios when the name of an operation is to be changed.

For the concise specification of a solution to the Flowgraphs case, the EPSILON solution has been awarded for being the best overall solution of the Flowgraphs case.

9.4.3. eMoflon

The EMOFLON solution is a solution based on advanced Triple Graph Grammars (see section 2.2). Advanced means that EMOFLON is able to incorporate extensions that act as value converters. This allows transformation developers to match attributes of different model elements in a more flexible way than matching on identity or equality. An important example of this is adding a constant suffix to a string value which is for example required in the transformation of methods as the name of a method in the Flowgraphs model is supposed to consist of the original methods name followed with a pair of parenthesis. Such value converters can be parametrized to allow reusability. E.g. adding a suffix to a string is a common task that might be required in multiple model transformations. In fact, EMOFLON incorporates a library of such value converters.

Being based on TGGs, EMOFLON offers a bidirectional transformation. However, the solution only implements support for a subset of model elements (*while*, *if/else*, *break*) in a proof-of-concept manner. Although bidirectionality support has not been a core requirement in the case description, it is indeed a feature that is nice to have. The solution paper especially highlights that such a bidirectional model transformation enhances support for refactorings, i.e. refactorings made on the Flowgraphs model could be translated back to the source Java model. As the EMOFLON solution extended the Flowgraphs metamodel with a *CompileUnit* class, the solution can persist these changes back to the Java files. Further to the changed input metamodel, the solution also uses a different parser based on ANTLR.

Figure 9.4 shows an example how these TGG rules look like in Enterprise Architect which has been used by the authors of the solution. The black elements at the top represent the axiom of this rule, i.e. the elements that determine where the rule is applicable. The green boxes below represent the elements that participate in a rule application. The rectangles represent model elements whereas the hexagonals represent correspondings. The rule basically specifies that for any *AssignmentNode* that is a child of *StatementsNode* of the axiom, a corresponding *SimpleStmt* element (referred to as `declStatement`) should be created. The dashed lines specify these extensions to put constraints on these elements.

Such transformation rules are not self-explanatory. However, as soon as one gets used to these specifications, they are quite understandable and somehow quite concise.

9.4.4. ATLAS Transformation Language (ATL)

ATL is an external transformation language that represent model transformations as sets of transformation rules. These transformation rules can have multiple input and output elements and specify the attributes of the resulting model elements via OCL expressions. Compared to all other transformation tools and languages, ATL is the oldest one (from 2006). It has been designed as a hybrid language (being both imperative and declarative) to combine the advantages of both QVT-O and QVT-R.

As an example, listing 9.15 shows the transformation rule contained in the ATL solution to transform *WhileLoop* elements.

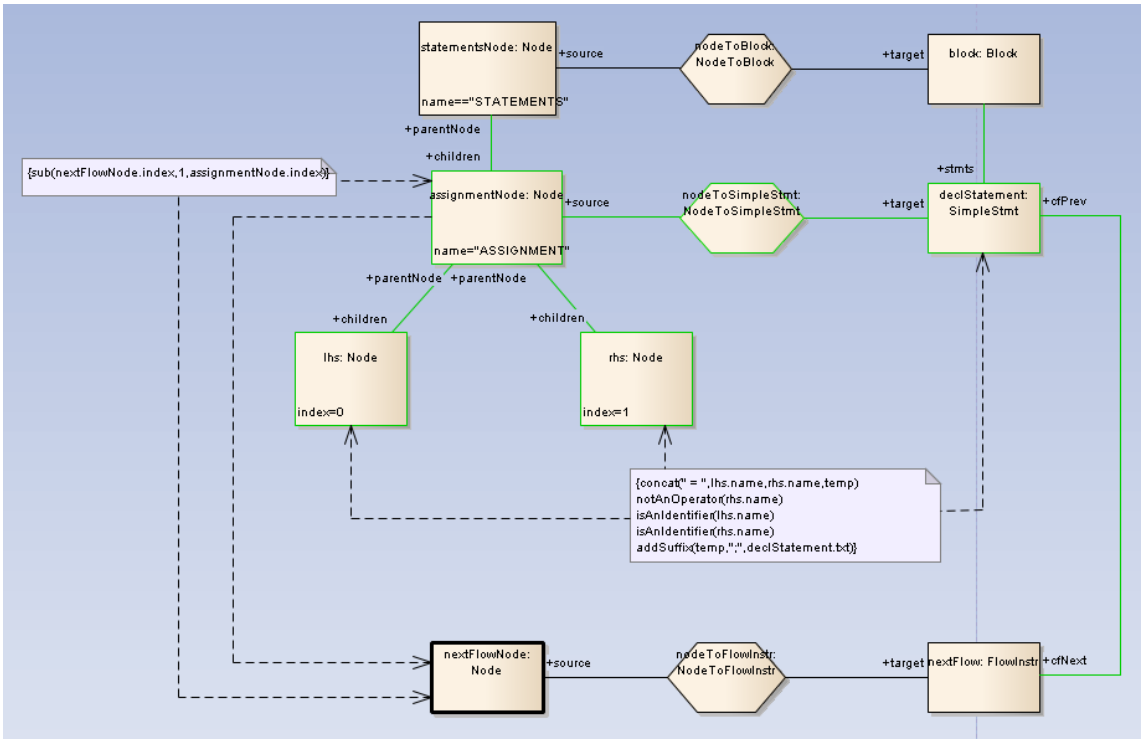

Figure 9.4.: The TGG rule for an assignment in eMoflon

```
1  rule WhileLoop2Loop {
2   from
3    s : Java!WhileLoop
4   to
5    t : GRP!Loop (
6            expr <- s.condition,
7            body <- s.statement,
8            txt <- s.getText
9           )
10 }
```

Listing 9.15: A transformation rule to transform a *WhileLoop* element in ATL

The language automatically infers that the *expr* reference has to be obtained from the trace functionality by tracing the condition of the input conditional element. As the textual representation of an input model is not directly represented in the input model, this functionality is implemented using helpers. Listing 9.16 shows as example of such a helper method.

```
1 helper context JAVA!AssignmentExpression def : getText : String =
2  self.child.getText + ' ' + self.assignmentOperator.getText + ' '
3  + self.value.getText;
```

Listing 9.16: The helper to obtain the text of an assignment expression in ATL

These helpers in ATL may only consist of OCL statements, which is a side-effect free query language. To apply these results of these queries, the ATL solution uses a transformation run in refinement mode. This transformation consists of transformation rules for each metaclass that does nothing but calling a helper method and assign the results. This is required to enable ATL to update references as the helpers itself must have no side-effects.

9.4.5. Eclectic

Similar to Epsilon, also Eclectic uses a whole family of model transformation languages. The initialization can be done using the concept of mappings. Listing 9.17 demonstrates how such mappings look like.

```
1 from src : in!WhileLoop
2  to tgt : out!Loop
3   tgt.expr <  src.condition
4   tgt.body <  src.statement
5   tgt.txt = task1_attribution!text[src]
6 end
```

Listing 9.17: Transforming a *WhileLoop* element in Eclectic

As the *txt* reference of expressions needs to generated in a bottom-up manner, Eclectic declares an attribution on the transformation rules where the `text` attribute is marked as a synthesized attribute. For this, Eclectic has an attribute modifier keyword `syn`. The attribution further specifies the rules how to compute the attributes. Listing 9.18 shows an example how this looks like for a *LocalVariableExpression*.

```
1 rule in!LocalVariableStatement
2  init text = text[self.variable.initialValue]
3  type ref = text[self.variable.typeReference]
4  text[self] <- type ref.concat(' ').concat(
5      self.variable.name.concat(' = ').
6      concat(init text)).concat(';')
7 end
```

Listing 9.18: Retrieving the attribution of a local variable statement in Eclectic

9.5. General purpose solution

The Flowgraphs case seems like a typical use case for model transformations. However, to be more confident in the evaluation of this case study, the transformation task has also been implemented using general purpose code without using any framework (except for serialization and model representation). To do this, the model representation of NMF was reused and the transformation tasks of the Flowgraphs case have been implemented in plain C#. The solution has been inspired by the Epsilon solution of the Flowgraphs case and was only implemented after the TTC conference. As a reason, Epsilon takes huge advantage of its late binding feature. However, late binding is also available in C#. Thus, the model transformation can be implemented much in the same way as Epsilon is. Furthermore, as the JaMoPP parser constructs a model that is almost a tree, cyclic references can almost never occur. As a consequence, checking whether an element is already transformed is not necessary in this case.

To get an impression how the transformation case can be solved via general purpose code without the support of any framework, the same code snippets as in the Epsilon solution are reflected here to better compare how these solutions differ in conciseness. Thus, listing 9.19 shows the method to transform a conditional statement.

```
1  public Flowgraph.If TransformStatement(JaMoPP.Statements.
       Condition condition)
2  {
3   var ifStmt = new Flowgraph.If();
4   ifStmt.Txt = "if";
5   ifStmt.Expr = TransformExpression(condition.Condition);
6   ifStmt.Then = TransformStatement((dynamic)condition.Statement);
7   if (condition.ElseStatement != null)
8   {
9    ifStmt.Else = TransformStatement((dynamic)condition.
         ElseStatement);
10  }
11  return ifStmt;
12 }
```

Listing 9.19: The method to transform a condition statement in plain C#

The code actually looks pretty similar to the code of the Epsilon solution. As a difference, method calls in C# are resolved by static typing by default and thus, developers have to make it explicit if they wish to use late binding for a method. However, other .NET languages like Visual Basic.NET offer the ability to switch the default to late binding.

But unlike ETL, these plain methods obviously do not check whether an element is already transformed, as no trace is maintained. Thus, developers face the issues discussed in sections 6.1 and 6.2. However, the initialization only requires a trace access for the creation of jump labels and variables. The rest of the input models conforms to a tree structure. However, as discussed in section 6.1/6.2, future changes of e.g. usage scenarios may introduce new cycles and therefore have the transformation behave unexpected.

9.6. Results on the TTC

From the open peer reviews, the NMF solution only achieved a fourth place in the ranking of the overall evaluation. However, as these reviews have been peer reviews, not everybody was reviewing everybody else, making the results less reliable. The fact that NMF has got the lowest evaluation confidence may serve to support that the reviewers have not been sure how to rate NMF Transformations. In the evaluation of the case, an important criteria has been conciseness. As the syntax of C# is sometimes quite verbose, the NMF solution actually achieved the worst conciseness that was entirely rated by the Lines of Code metric. On the other hand, the biggest advantage of NMF Transformations, its tool support, has not been asked for and has also not been focussed on in the solution paper.

The plain evaluation data from the open peer reviews can be reviewed online at `http://goo.gl/jAo4T`. However, they are also contained in the appendix in section D.2 as it is unclear how long the link will be valid.

The overall evaluation (see figure 9.5) is not beneficial for the NMF solution. As a reason, most reviewers are developers of external transformation languages. Most of them have a focus on the transformations conciseness. However, this is the biggest advantage of external DSLs. As they invented external DSLs, the improved conciseness seems to have a high priority for them. However, as tool support is present, conciseness gets less valuable (as discussed in section 8.7).

Consequently, the attendees of the TTC conference found NMF Transformations also not so useful. The rating of the usefulness of the tool is quite interesting. Visual Studio,

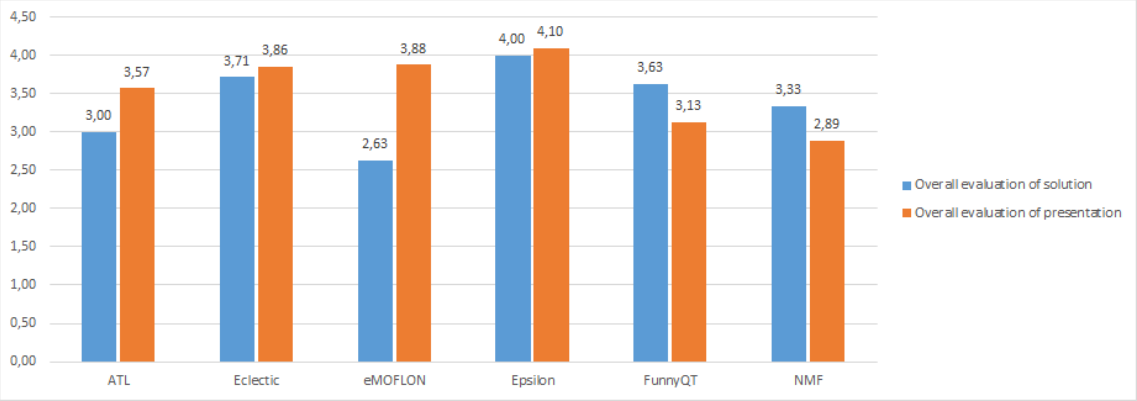

Figure 9.5.: The overall evaluation of the Flowgraphs case

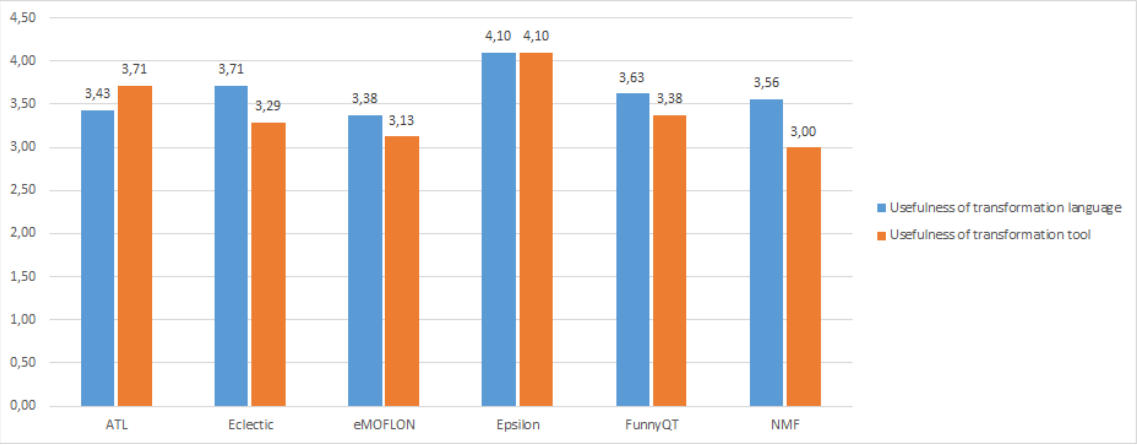

Figure 9.6.: The usefulness results of the Flowgraphs case

the tool for NMF Transformations but also for general C# development, has been rated as the most useless tool, even more useless than the simple editors with only syntax highlighting support that is offered by other transformation languages. As Visual Studio has proven its usefulness in practice, this may highlight the fact that the background of the TTC attendees is so much different to the target audience of NMF Transformations (C# developers) that value Visual Studio. This is also confirmed by the fact that NMF Transformations had the least familiarity with the technology.

The results data from the TTC is presented in more detail in the appendix, see section D.2.

9.7. Validation

In this section, the NMF solution of the Flowgraphs case is validated for to the validation criteria defined in section 9.2.1 in comparison to the other solutions. As few of these validation criteria were covered in the TTC, most of the validation is done through a discussion here. The following subsections include the validation discussion for each of the validation criteria defined in section 9.2.1.

As the understandability discussion is based on the conciseness evaluation, it is moved to the end of this section.

9.7.1. Modifiability

9.7.1.1. Discoverability

The discoverability of NTL already has been discussed in section 8.2. Thus, we concentrate on the discoverability support of the other tools and languages here.

FunnyQT

Much like NTL, FunnyQT is a shallow internal DSL. Its author Tassilo Horn even speaks of a framework for model transformation with Clojure rather than a language on its own. Thus, FunnyQT also inherits the tool support for discoverability from the tool. There are a row of tools supporting development with Clojure and the community does not seem to have found a consensus on it. Here, the Eclipse plug-in Counterclockwise[8] (CCW) is taken into consideration. CCW is a more sophisticated IDE for Clojure recommended by the FunnyQT author[9]. This Eclipse plug-in gives a code-completion feature in that way that it knows all functions and has the potential to show inline documentation. However, such inline documentation is not available for FunnyQT. Furthermore, the CCW editor does not give a hint how to use the offered functionality, the developer can only made aware of their existence.

The reason the discoverability support of FunnyQT is so much worse than the discoverability support for C# is that C# is a statically typed language. Thus, the compiler knows the methods or members available for a variable based on its static type. As a result, tool support like the IntelliSense feature in Visual Studio can access this information and provide the developer with the members that are valid on this specific type. This greatly improves the discoverability for the transformation developer.

[8] `http://updatesite.ccw-ide.org/stable/`
[9] `https://github.com/jgralab/funnyqt`

Epsilon

Epsilon is a family of external DSLs, i.e. the model transformation to solve the Flowgraphs case consisted of multiple languages. In external DSLs, the features offered by the language are usually encoded in the language. In this way, the language constructs make their meaning mostly obvious to the developers. However, the developer must know these language constructs.

The Epsilon plug-in for Eclipse has little support to let the developers discover the language constructs of ETL. The editor provides a couple of snippets that can be discovered by the control+space shortcut. The most important snippet for ETL, the snippet for a model transformation rule, is shown in listing 9.20.

```
1 rule rulename
2  transform s : sourcemodel!sourcetype
3  to t : targetmodel!targettype {
4
5 }
```

Listing 9.20: A snippet for a transformation rule in Epsilon

There is no documentation available what these snippets represent. The developer must consult documentation to see that the snippet is actually incomplete as it hides that one may also define guards or extend other transformation rules, for instance. Furthermore, if a transformation developer has already started creating a transformation rule, the Eclipse plug-in does not support him in the way that it shows the possible keywords that could apply at a certain position in code. As an example, the Eclipse plug-in does not support developers in that the rule name may be followed by a `transform` keyword. For external MTLs, Epsilon is one of the more important ones and exists after all for five years now. The community behind Epsilon is relatively big, but still none of such tool support has been accomplished. This shows the principle problem of external DSLs that it is appearently too much effort to support an external DSL with rich tool support.

The only thing that is offered by the Eclipse plug-in besides these snippets is that when pressing control+space, the IDE lists all available primitive types and all properties that were used anywhere in the transformation. However, the latter feature does not help for discoverability as developers first have to know that a property exists in order to use it.

The imperative EOL language is supported in the same way. Snippets exist that show the definition of an operation or function, but no further tool support is offered for discoverability.

eMoflon

As eMoflon follows an entirely different approach to model transformation, it is hard to compare it to the other model transformation languages. As a TGG, eMoflon has a graphical syntax. This graphical syntax prevents some features that are common to textual codes. However, with the graphical syntax, eMoflon can make use of the different properties to allow developers to discover how they can modify an existing model transformation. The Enterprise Architect allows to add a rule in the context menu of the rules element in the project explorer. In this way, transformation developers can discover the functionality of eMoflon.

There is only the Enterprise Architect Lite installed on SHARE, it has been impossible to review how to modify the solution as every modification functionality is disabled in the Lite version. However, as a developer not experienced with eMoflon, it is indeed possible to find the functionality as it is discoverable through wizards of the graphical user interface.

ATL

The discoverability support of ATL is very similar to the support of Epsilon. The tool (again the Eclipse plug-in is considered) only offers support in creating snippets. It does not offer the developer with contextual support with respect to the language constructs. However, similar to Epsilon the tool supports the discoverability of the input and target metamodels.

```
1 rule name {
2  from
3   input_name : input_element
4  to
5   output_name : output_element (
6
7   )
8 }
```

Also for ATL, no context-sensitive discoverability support is provided.

Eclectic

Eclectic could not be reviewed in terms of discoverability as the SHARE demo was not accessible and no further documentation exist how to set up the IDE for Eclectic. However, as Eclectic is an external language developed only by Jesús Sánchez Cuadrado and does not inherit rich tool support from any host language, the support for Eclectic is unlikely to be better than for ATL or Epsilon that exist for years now and are maintained by a team consisting of more than a single developer (although much less developers than e.g. in the teams evolving mainstream languages such as Java or C#).

Results

In the model transformation languages with textual representation, NTL is much more discoverable than any other model transformation language as it inherits the tool support provided by Visual Studio. In FunnyQT, the existing tool support of the IDE is not used entirely as no inline documentation is provided for CCW. For external languages, implementing the necessary tool support for discoverability seems to be a barrier that is yet too big to manage. The only transformation tool that has a discoverability support that may challenge the support in NTL is eMoflon. However, as these languages differ greatly in their nature, it is hard to compare them with respect to discoverability. To discover eMoflon, developers have to understand what the diagrams as in figure 9.4 mean. On the other hand, the tool support can exhibit the full functionality. But the fact that a feature is available in a graphical user interface (GUI) does not necessarily mean that developer will find it. User studies have to be performed to evaluate whether the functionality in a GUI is found by the users of that application. In contrast, the discoverability support of textual languages like NTL can be offered by few tools. In case of NTL, the discoverability support of Visual Studio is mainly offered by the IntelliSense feature. Thus, further validation would be necessary to better compare NTL and eMoflon in terms of discoverability. However, such in-depth comparison is not worth the effort as the GUI of tools often change.

Put in a nutshell, it can be said that NTL and eMoflon have the best tool support for discoverability of all the MTLs that participated in the Flowgraphs case.

9.7.1.2. Change Impact

NMF

The transformation instantiation feature allows NTL to easily extend the model transformation for later introduction of additional metaclasses. However, NTL currently only allows a transformation rule to instantiate only one other transformation rule. As a result, three additional transformation rules had to be created for two additionally used metaclasses:

1. A rule to transform unary expressions to temporary transformation objects
2. A rule to transform unary operators to strings
3. A rule to transform a *Subtraction* element when used as a unary operator

Furthermore, no existing code had to be modified to support unary expressions. However, the NTL solution requires an explicit hub for *UnaryExpression*s. Furthermore, because of the restriction to a single base rule for transformation rule instantiation, a separate transformation rule has to be created for the situation that a *Subtraction* is used as a *UnaryExpression*.

At the time of the initial submissions to the TTC, NMF Transformations did only throw a plain *InvalidOperationException* whenever an abstract transformation rule was called without an appropriate instantiating transformation rule. However, in the meantime there is a meaningful error message to tell transformation developers what is wrong and how they can fix such issues.

FunnyQT

As presented in listing 9.12, FunnyQT uses a similar technique like the disjunct mappings from QVT-O with the generalization of other transformation rules. As a result, whenever a new metaclass of the input metamodel is supported, the rule for this new metaclass has to be referenced in every rule that generalizes these rules, e.g. rules that transform any kind of specific statements to *Stmt* elements.

However, the extension scenario that actually appeared included additional expression classes. The FunnyQT solution has split the usages of these expressions all across the model transformation. The easy part is to create a new overload to specify how this new expression type (the example included a *UnaryExpression*) has to be transformed to a string representation. The more complex part is to determine the impact of these new statements to the *def* and *use* links. As a reason, the FunnyQT solution creates these links within the statements. Luckily, the unary expressions do not have an influence on *def* and *use* links. But if we had introduced new expression types that influenced the used or written variables (like for example lambda expressions, although they are not yet available in Java), this would have badly affected the change impact size.

As FunnyQT uses a single polymorphic method to transform any object to string, there is no specific function required to model the string representation for a *UnaryOperator*.

Epsilon and ATL

The Epsilon and ATL solutions also use polymorphic operations to turn expressions to strings and thus, for the introduction of a *UnaryExpression*, only a further overload of these polymorphic methods has to be created. In the Epsilon solution, also setting the definers and users of a variable is done via polymorphic operations, so only new overloads have to be created. In ATL, this is accomplished via a helper method that looks through

the elements by using the `allInstancies` feature provided by EMF. Using this feature often leads to concise code, but the scope is often unclear. It is not intuitive what *all* in *allInstancies* really means.

For the introduction of new statement classes, only new transformation rules had to be added that would specify how instances of these classes would have to be transformed to Flowgraph elements. The transformation engines for ATL and ETL work quite similar in that they match the input elements and the guards. The tracing in both of these language is done implicitly. In ETL, this tracing functionality is inferred by the `equivalent` operation. ATL even does the tracing implicitly as it automatically detects that assignments of types that do not match have to be resolved with the trace.

As the general purpose solution is also entirely based on polymorphic operations, these propositions also hold for the general purpose solution. The only difference is that the general purpose solution does not involve a trace, but explicitly calls the rule methods for descendent elements.

eMoflon

The eMoflon solution did not include support for unary expressions as it only is a proof-of-concept solution. The existing solutions show that transforming a single element in TGGs sometimes requires much more than a single transformation rule.

For its proof-of-concept nature, eMoflon is excluded from the comparison here.

Eclectic

The mapping language of Eclectic works pretty much in a similar way as ATL and ETL in that the engine automatically matches the inputs and thus executes the rules. Thus, the propositions for adding new statement metaclasses in ATL and ETL also hold for Eclectic.

The huge difference in comparison to ATL and Epsilon is where Eclectic uses attribution similar to attribute grammars. However, although the underlying concepts differ, the results do not as additional metaclasses also require an additional rule how to obtain the attribution for instances of this new metaclass, only.

Conclusion

The NTL solution fundamentally differs to most other solutions in that it requires an explicit structure of model transformations whereas the other transformation languages infer the structure. In this way, NTL is more imperative than its opponents. This has an influence on the size of changes due to updated requirements. In the extension scenario, the introduction of *Subtraction* as a unary operator did not have an influence on most of the solutions, as they do not draw a difference in how an operator is used. Thus, they allow to specify how an element, e.g. a *Subtraction*, has to be transformed regardless of the context this element is used in. The NTL solution, however, specifies a context when elements have to be transformed, by declaring abstract transformation rules, e.g. the *UnaryOperator2Text* rule. Instantiating transformation rules have to specify exactly in which context they are to be applied instead of just ”any context”.

This may change when transformation rules in NTL may be allowed to instantiate multiple other transformation rules. However, for the present, they must not. Thus, it is necessary to e.g. create an entire new rule for *Subtraction* elements when transformed as a unary operator rather than a binary operator.

In terms of the size of the change impact, the solutions also differ. However, except for FunnyQT that seems to use a disjunct mapping like feature for polymorphic tracing, all solutions could be written in a way to have perfective changes not affecting existing code (leaving eMoflon unchecked). Besides ATL, all these solutions in fact did not require changes of existing code to support new metaclasses but required straight forward additions at few places.

NTL has the advantage that support of a new metaclass is really bundled in a single place, i.e. only a single new transformation rule has to be created (possibly together with a helper element that the transformation rule targets to). However, the fact that NTL requires model transformations to have an explicit structure also increases the change impact as new metaclasses (as the unary expressions in the example) may require new structures. The fact that NTL currently disallows transformation rules to instantiate multiple other transformation rules further increases this change impact.

9.7.2. Consistency

NMF

The NTL solution uses dependencies as in section 7.4 whenever possible. If it is not possible to do so as e.g. the target model elements for expressions in task 1 (strings) are immutable, it uses the tracing functionality. Also the in-place model transformations are implemented as usual M2M-transformations where the target model this time consists of helper models that are weaved into the model transformation. This makes the NTL-solution very consistent as it uses a uniform programming style.

However, this consistency is mainly achieved through a consistent usage of NTL. As it is possible to mix up multiple programming styles in NTL, the consistency is a property of the solution rather than a property of the transformation language.

As NTL uses an explicit structure of the model transformation, represented by abstract transformation rules acting as hubs, the transformation rules must explicitly refer to this structure. As a consequence, the compiler already detect many mistakes in the structure of the transformation.

FunnyQT

For the transformation of statements, FunnyQT uses model transformation rules pretty much comparable to NTL, ATL or ETL. To derive the text statements, polymorphic operations are used. However, to derive the control flow, the FunnyQT solution uses explicit type switches to base the operations behavior on the currents element type. This results in three different programming styles within the solution and clearly counts towards an inconsistent solution.

To use a polymorphic method, the FunnyQT solution explicitly declares this method as polymorphic. This is somehow similar to the abstract transformation rules in NTL as it also results in explicitness. As a result, overloads of this polymorphic method can be checked by the system that a declaration of this polymorphic method exists, in the same way as the C# compiler can check that the transformation rule that another rule is instantiating exists.

Epsilon

The Epsilon solution consists even of multiple file types. However, ETL is only an extension of EOL for a very specific purpose, namely to specify mappings between model elements. The solution exactly uses the language in this way as all mapping tasks are

accomplished in ETL and everything else is done with operations in EOL. This also speaks for a uniform programming style and a consistent solution.

However, EOL does not draw a difference between polymorphic methods and those who are not. As a result, a typing mistake would result in a bug in the model transformation, as no compiler can detect the incorrect spelling (possibly except for the fact that this misspelled operation is never used). This may hamper the consistence of the solutions during maintenance operations. However, an incorrect spelling of an overloaded operation can be found through tests.

eMoflon

The reason that eMoflon extended the concept of TGGs was that TGGs alone would be inappropriate for some model transformation tasks. Thus, they basically extended the idea of TGGs by converters. These converters are specifically for simple operations such as adding suffixes. As the eMoflon does not rely on external converters but only on those included in the library, this library can be considered as similar to a part of the transformation language, especially because of the simple nature of these converters. As a consequence, the eMoflon specifies the biggest part of the model transformation as a TGG and only relies on library elements, making the solution quite consistent.

However, as eMoflon is a graphical language, it is hard to indentify programming styles.

ATL

The ATL solution also uses mappings for mapping tasks. However, the refinement transformation uses mappings not for real mapping tasks but to execute the control flow mechanism implemented in helper functions. Thus, the same concept is used entirely different. This is specifically bad as invoking specific methods is not a true mapping task. Thus, using mappings feels a bit inappropriate. Furthermore, ATL uses polymorphic methods (helpers in ATL) but uses a single helper accessing the *allInstancies* feature of EMF which represents a different programming style. Thus, the solution is also quite inconsistent.

The same propositions on polymorphic methods made for Epsilon regarding inconsistency problems during maintenance operations also hold for ATL.

Eclectic

Eclectic consequently uses mappings and attribution. Much like in the Epsilon solution, Eclectic uses mappings where appropriate and solves the other queries with its attribution features which leads to a very consistent solution. As the attribution is declared once, the compiler is able to check any inconsistency that may occur during maintenance.

General purpose

The general purpose solution is entirely based on polymorphic methods and dictionaries for bookkeeping as there is no trace support available. However, dictionaries are used not for all operations but only for those where cycles may occur. Furthermore, the same remarks on inconsistencies in maintenance scenarios also hold for ETL. As the solution uses C# as a dynamic language, no static type checking can be performed by the compiler and thus no compiler errors are given when an operator is spelled incorrectly. Thus, the solution is quite consistent.

Conclusions

The comparison shows that NTL was written in a very consistent way but this is also made possible by other languages. Solutions like e.g. Eclectic or Epsilon very clearly separate the purpose of their language constructs, thus making it very clear when to use what language construct. However, the NTL solution in the way it is implemented in the case study can keep up with the opponent solutions in terms of consistency, while other solutions like FunnyQT or ATL lose ground.

However, whereas transformation languages like Eclectic imply a specific programming style, the consistency of the NMF solution cannot be generalized to the transformation language. Instead, it is only possible to state that it is possible to use NTLconsistently.

During maintenance, the transformation rule instantiation feature of NTL keeps up the consistence of a model transformation as the transformation rules have to explicitly mark themselves instantiating for a rule where the compiler can check that it exists. This differs from the polymorphic methods that are sometimes only based on naming conventions.

9.7.3. Conciseness

For the conciseness, the eMoflon solution is not considered as it is only a proof-of-concept solution. Furthermore, the visual nature of the language leads to limited comparability. The eMoflon solution can only be compared to the other solutions by evaluating the perceived conciseness with questionnaires. To some extend, this has been done in the open peer reviews. However, eMoflon has received poor ratings for the later tasks for being only a proof-of-concept.

The values of the *Lines of Code* (LOC) metric are depicted in table 9.1. These numbers were counted excluding blank lines and comments by the case submitter Tassilo Horn. However, for NTL, also the LOC as counted by Visual Studio is listed. The result differ as Visual Studio counts the lines of IL code and only estimates the lines of source code. The version of Visual Studio has the huge advantage that it is independent of the programming style, i.e. where developers set line breaks. In theory, languages like C# allow to put everything in a single line of code, but IL does not. The *LOC* metric abstracts from this.

Solution name	Task 1/3.1	Task 2	Task 3.2	Total
NMF (based on IL)	299	134	19	452
NMF (code lines)	600	295	45	940
FunnyQT	223	57	19	299
Epsilon	320	90	29	439
ATL	397	268	66	731
Eclectic	290	97	29	416

Table 9.1.: Implementation size of the Flowgraphs case

As one can see in table 9.1, the NTL solution is the least concise solution in terms of the actual file length (except comments and blank lines). Based on IL, the solution is very close to the Epsilon and Eclectic solution. Only the FunnyQT solution is more concise. As a reason, the FunnyQT solution used some shortcuts to e.g. shorten the text generation of operators. The ATL solution loses in the comparison mainly because of the refinement transformation that has to include a rule for every metaclass to call the operations that perform the actual control flow transformation.

The huge difference in the manual measurement of code lines compared to the outcome of Visual Studio has a simple reason. Visual Studio only counts the actual code lines, i.e.

lines that specify execution semantics. This does not include class, interface or method declarations, i.e. lines that make up the metadata. Only the executable code is measured. It also does not include lines that are to comply with coding guidelines such as the rule to put braces in a separate line. The NTL solution sticks to these common coding guidelines to improve understandability but of course, additional braces hamper the conciseness of the solution in terms of LOC. On the other hand, chains of commands are not counted as one, although they might fit into a single line. Thus, the difference can be interpreted in a way such that both the verbose metadata declarations as also the coding styles common in C# lead to a quite verbose solution. A similar coding style (putting paranthesis in separate lines) common in ATL is responsible for its relatively high LOC metric value in the first task.

On the other hand, the metadata is evaluated by the NTL classes and thus indeed carries execution semantics as a declaration of a nested class carries the semantics that the transformation consists of that rule. Furthermore, it specifies the signature of the transformation rule as well as its name. This is also specified in other transformation languages. Thus, the NTL solution is also less concise when leaving aside syntactic noise. The reason for that is that most transformation languages do not explicitly define the structure of the model transformation but NTL requires transformation developers to do so, e.g. via abstract transformation rules.

The worse conciseness is also reflected in the results of the open peer review in terms of combined understandability and conciseness (see tables D.2, D.3 and D.4 in the appendix). Whereas the average evaluation for the conciseness of the other solutions (besides eMoflon that does not count) is rated roughly equal, the NMF solution is rated significantly worse. This may also be due to the fact that NTL has a lot of syntactic noise. Where languages like ATL or ETL have a keyword, NTL usually has a whole method signature. Instead of a `transform` keyword, the whole signature `public override void Transform(Type1 input, Type2 output, ITransformationContext context)` is repeated. Both expressions carry exactly the same semantics but the NTL-version is much more verbose. As these lines appear for each transformation rule, the perceived conciseness is degraded.

9.7.4. Understandability

The results of the combined assessment of understandability and conciseness are shown in figure 9.7. The results are based on 49 responses to the Flowgraphs evaluation sheet. These responses are distributed equally among the solutions. The results show that the NMF solution was rated on third rank with respect to understandability and conciseness. The NMF solution has even bet the ATL solution although the difference is small as can be. However, the gap between Epsilon and Eclectic to NMF is very large.

This is due to the fact that both Epsilon and Eclectic are external languages that can concentrate on conciseness and thus understandability. Both of them do not have any syntactic noise that hampers the understandability of the solution. An internal language always has the restrictions from its host language and thus can never be as concise as an external language can. The results seem all the more astonishing with respect to the fact that the NMF solution achieved a score as good as ATL in terms of understandability and conciseness. ATL also is an external language that may use specialized language features to improve its understandability and conciseness. A reason for that may have been that the ATL solution lacks of conciseness in task 2 due to the refinement transformation that only serves to execute some helper functions.

The other internal DSL in the Flowgraphs case, FunnyQT achieved the lowest ranking. This is remarkable as FunnyQT has been the most concise solution in terms of LOC.

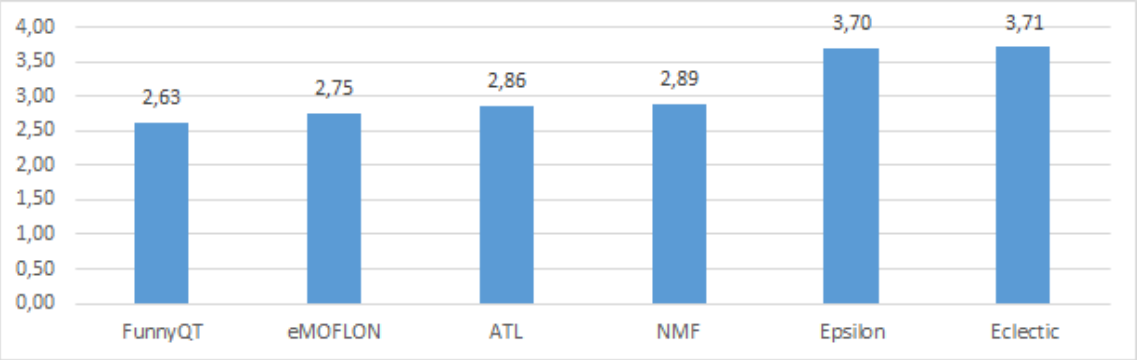

Figure 9.7.: The combined assessment of understandability and conciseness at the TTC conference

Thus, it seems like most of the attendees of the TTC conference found FunnyQT difficult to understand. A reason for this may be what already has been remarked in the open peer reviews. The functional paradigm of Clojure hampers the understandability for many of the attendees of the TTC. As NMF Transformations is based on the imperative language C#, it is more understandable.

Hence, the results indicate that the NMF solution was very understandable. It does not reach the understandability of Epsilon or Eclectic, but this is not suprising as these languages enable the transformation developers to specify the transformations more concisely. This can also affect understandability. However, the fact that NMF took the third place despite its verbosity (it is the least concise solution) is a clear indication that model transformations with NMF Transformations are very understandable.

9.8. Conclusions

The Flowgraphs case at the TTC is a complex scenario that involves a row of typical model transformation tasks. This includes a M2M-transformation, a refinement transformation and an embedded M2T-transformation. The case shows that these model transformation tasks can be dealt with using NMF Transformations. The case can be solved in a very consistent way. However, the NMF solution fundamentally differs in its implementation from other solutions in that NMF Transformations requires an explicit transformation structure where this structure is inferred in most other MTLs. This yields a less concise model transformation and thus larger change impact when new input model elements are added. However, the additional structure helps to keep the model transformation consistent in maintenance scenarios and may also improve the solutions understandability. Unfortunately, the results for the understandability and overall evaluation from the TTC were not published early enough to have them presented in this thesis.

However, despite its verbosity, NMF has achieved a third place (practically together with ATL) in terms of the combined assessment of understandability and conciseness. This indicates that model transformations with NMF Transformations are very understandable.

On the other side, for the very most concepts used in the other solutions (besides the eMoflon solution), equivalent concepts in NMF Transformations exist. The concepts provided by NMF Transformations fit naturally to solve the Flowgraphs case. But unlike the other solutions based on textual MTLs, NMF Transformations has great tool support. Specifically, this tool support has been reviewed in terms of discoverability support. A good discoverability support eases the learnability and thus helps to improve the productivity in maintenance scenarios.

10. TTC Petri Nets to State Charts case study

In this chapter, the Petri Nets to State Charts (PN2SC) case of the TTC is used as a case study to evaluate the usefulness of NMF Transformations. First, an overview on the case description is given in section 10.1. Section 10.2 explains the planned validation for this case study evaluation. The next section 10.3 covers the solution of the subtasks involved in solving the Petri Nets to State Charts case with NMF. Section 10.4 briefly introduces the solutions from the opponents at the TTC. Section 10.5 then presents the results from the TTC. Section 10.6 performs the validation of the PN2SC case with respect to the evaluation criteria from section 10.2.1, before finally section 10.7 concludes this chapter.

The solution description of the Petri Nets to State Charts case is based on [HGH13b] meanwhile the case description is based on [GR13].

10.1. Case Overview

Solutions of the PN2SC case have to solve the algorithm to transform Petri Nets to hierarchical state charts that has been originally described in [Esh05]. This transformation essentially creates a hierarchical state chart that represents the Petri Net. However, the Petri Net is destroyed during this process which is why the transformation is referred to as being *input-destructive*.

Figure 10.1 shows the metamodels of Petri Nets and State Charts that is used for this transformation task. Petri Nets have the structure of places and transitions where each transition can have arbitrary many source and target places.

The state chart model in PN2SC is a usual metamodel for state charts with states and transitions buth with the special property that it is hierachical. The hierarchic nature arises from the introduction of compound states that can either be *OR* states or *AND* states. These compounds can contain other states and describe the hierarchical structure of the state machine. As there may be multiple tokens in a Petri Net, a hierarchical state chart can have multiple current (active) states. An *AND* compound means that all of the subsequent child states must be active meanwhile an *OR* compound expresses that any of the child states must be active. The model further contains *Basic* and *HyperEdge* elements to represent the places and transitions of the original Petri Net.

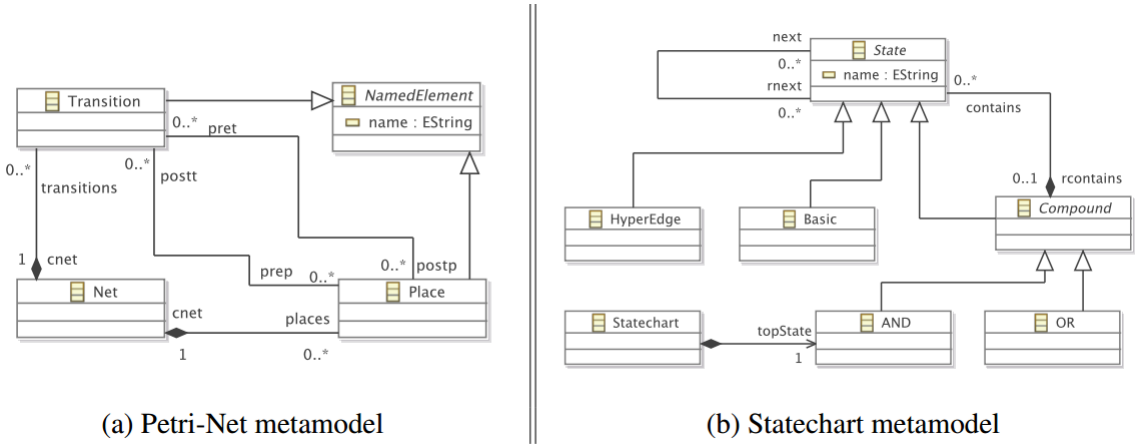

Figure 10.1.: The metamodels for Petri Nets and State Charts [GR13]

10.1.1. Initialization

In a first initialization step, the transformation creates a corresponding State Chart model. For every place, a corresponding *Basic* and *OR* element has to be created where the *OR* state contains the *Basic* state and both have their names set to the corresponding places name. Each transition is also transformed into a *HyperEdge* state. These *Basic* and *HyperEdge* elements are to be connected according to the *pret/postp/prep/postt* links in the Petri Net.

10.1.2. Reduction

The transformation now aims to simplify (and thus destruct) the Petri Net step by step in order to retrieve the hierarchic structure of the resulting state chart by applying two reduction rules. These rules aim to deduct *OR* or *AND* compounds and are thus named accordingly.

The *AND* rule

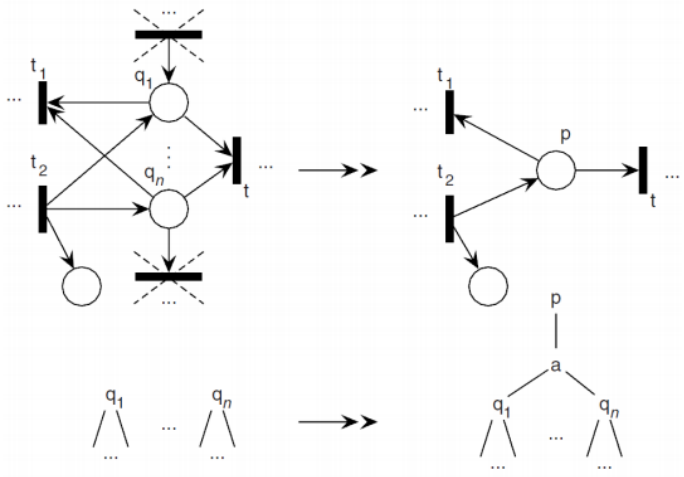

Figure 10.2.: The impact of the *AND* rule to the Petri Net and the State Chart model [GR13]

The *AND*-rule is applicable whenever a set of places $q_1, \ldots, q_n$ has only common incoming and outgoing transitions. It is not allowed that any transition may be an incoming transition for only some of these places. In this case, these places are merged into a single place.

As these places are represented in the resulting state chart by compound states, a new *AND* compound a is created to contain these compounds. Furthermore, a new compound state p is created to contain this state a. Figure 10.2 shows this procedure.

The *OR* rule

To perform the *OR* rule to places q and r, no transition t may be a common incoming or outgoing transition for either of these places.

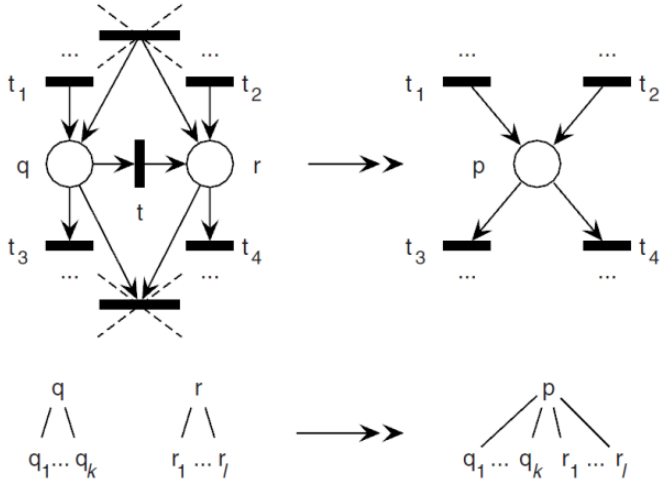

Figure 10.3.: The impact of the *OR* rule to the Petri Net and State Chart model [GR13]

In this case, these places can be merged into a new place p_{PN} and so can the corresponding compound *OR* elements in the State Chart model. This newly created place p has the incoming/outgoing transitions of both q and r. In the State Chart Model, a new *OR* p_{SC} is created that contains the elements of both *OR* elements corresponding to q and r. Alternatively, either q or r can be reused for this purpose.

For more details to the transformation or its purpose, see [Esh05, GR13].

10.1.3. Extensions

The case description of the Petri Nets to State Charts case included some extensions, mainly to specify the resulting state charts more precisely in order to have them validated automatically by the validator written by Tassilo Horn. An important extension was to set the correct containment position for hyper edges. This means that a hyper edge should be contained in the least common compound state that (transitively) contains all incoming and outgoing *Basic* elements of the hyper edge.

Furthermore, the case description defines several bonus criteria including transformation language support for change propagation, debugging support, refactoring support, support for bidirectionality, verification and simulation. Few solution papers of the solutions to these cases included descriptions in which way they offered means to provide these additional features but the support has been asked for in the open peer reviews.

10.1.4. Evaluation

The evaluation scheme of this case is divided into basic and bonus criteria. The basic criteria includes that the correctness of the presented solutions, secondly their performance, their understandability and reproducibility. Understandability and reproducibility are reviewed by publishing all artifacts to SHARE [VGM11] to enable others to review these artifacts and run the transformation.

10.2. Planned validation

This section presents the planned validation for this case study. Section 10.2.1 introduces the validation criteria for this case study before section 10.2.2 describes the procedure how the validation is done.

10.2.1. Validation criteria

Similar to the Flowgraphs study, the Petri Nets to State Charts case at the TTC offers great ways to compare NMF with other transformation tools and languages. However, unlike the Flowgraphs case, the Petri Nets to State Charts case did not offer such typical model transformation tasks. As a result, the transformation tools that participated were not as typical transformation languages as those who participated in the Flowgraphs case. As a reason, the model transformation constructs the State Chart model by destructing the Petri Net, which is somehow an unusual behavior for a model transformation. Thus, we can identify the following validation goals:

- **Applicability & Integration:** By solving the Petri Nets to State Charts case, it is evaluated that NMF Transformations is an applicable technology to transform cyclic models. Furthermore, it is evaluated how NMF Transformations integrates with general purpose code.
- **Understandability:** Much like in the Flowgraphs case, the TTC also here collects data on the perceived conciseness and understandability. Together with the metrics results of the *Lines of Code* metric, it may be possible to draw conclusions to the understandability of NMF Transformations, especially compared to other solutions. However, as the Petri Nets to State Charts case has been solved by other developers than in the Flowgraphs case that largely have other background, the results may differ.
- **Modifiability:** Unlike the Flowgraphs case, the PN2SC case does not contain such a clear extension scenario. Thus, it is hardly possible to validate the modifiability regarding to perfective changes. Instead, we concentrate on the support of the competing transformation languages and tools for debugging and refactoring the model transformations. We may take advantage of the perceived debugging and refactoring support that has been collected during the TTC open peer reviews.
  Unlike the Flowgraphs case, the discoverability support is not evaluated once more to keep the master thesis compact. Furthermore, the PN2SC case has been solved with a couple of languages where discoverability support is hard to measure. Support for testing is also not evaluated.
- **Consistency:** As the PN2SC involves an unusual model transformation, many solutions may get to the boundary of their expressiveness (including NMF Transformations). Thus, it is important to validate how the consistency can be preserved.
- **Conciseness:** The mixture of languages that has been applied in the PN2SC case makes it interesting to validate the solutions in terms of conciseness. Especially, the difference of this case study to the Flowgraphs case is interesting.

Again, the performance of the solution is not a validation criteria as it is not related to maintenance.

10.2.2. Validation procedure

As the Petri Nets to State Charts case coming from the TTC has the same origin as the Flowgraphs case, the surrounding conditions for the validation are basically the same. That is, all solutions are available via SHARE[1] and there is a combined assessment of conciseness

[1] `http://goo.gl/rgGBJ`

and understandability that can be used to draw conclusions for the understandability of the solutions.

However, further to the validation criteria for the Flowgraphs case, also debugging and refactoring support is evaluated in this case study. The evaluation for both of these criteria can take advantage of the fact that both the debugging and the refactoring support were asked for to evaluate in the open peer reviews. However, many solution papers (including the NMF solution paper) did not include any statements on debugging or refactoring support of the used tools. As a consequence, the data from the open peer reviews might be less reliable. Thus, the validation will also focus on reviewing the solution demos for debugging and refactoring support.

10.3. NMF Solution

This section presents the solution of the Petri Nets to State Charts case using NMF Transformations. The solution description is divided into two subsections that describe the solution to the initialization (section 10.3.1) and the reduction part (section 10.3.2).

10.3.1. Initialization

The task of the initialization is to create an initial structure for the state chart model. Although it is not mentioned in the description, the first rule for this initialization is that for each Petri Net, a state chart has to be created with a top-state which is an *AND*.

```
1  public class PetriNet2StateChart : TransformationRule<Net,
       Statechart>
2  {
3   public override void RegisterDependencies()
4   {
5    Require(Rule<PetriNet2TopState>(), (chart, topState) => chart.
       TopState = topState);
6   }
7  }
8
9  public class PetriNet2TopState : TransformationRule<Net, AND>
10 {
11  public override void RegisterDependencies()
12  {
13   RequireMany(Rule<Place2Or>(),
14    selector: net => net.Places,
15    persistor: (and, places) => and.Contains.AddRange(places));
16
17   RequireMany(Rule<Transition2HyperEdge>(),
18    selector: net => net.Transitions,
19    persistor: (and, transitions) => and.Contains.AddRange(
       transitions));
20  }
21 }
```

Listing 10.1: The transformation rules to transform the Petri Net

The first rule in listing 10.1, `PetriNet2StateChart`, is a rule that transforms a Petri Net into a State Chart. The only thing that happens here is that a top state for the Petri Net is required and the top state is saved in the `TopState` reference of the *StateChart*.

The second rule, `PetriNet2TopState` creates the top state for the Petri Net. This rule already contains information for the next two rules: For every *Place* within the Petri Net, a corresponding *OR* element should be created and added to the initial *AND* top state. Furthermore, any transition should be transformed to a *HyperEdge*. Note that within this rule, there is no information about how the places are to be transformed, it is only required that they are transformed using the `Place2OR`-rule. The same applies for the transitions, it is only required that they are transformed with the `Transition2HyperEdge`-rule.

Furthermore, for every Place p in the PetriNet

- an instance of *Basic*, b (with the name set accordingly to p.name) and
- an instance of *OR* o, such that $o.contains = \{b\}$

have to be created.

```
1  public class Place2Basic : TransformationRule<Place, Basic>
2  {
3   public override void Transform(Place input, Basic output,
       ITransformationContext context)
4   {
5    output.Name = input.Name;
6   }
7  }
8
9  public class Place2Or : TransformationRule<Place, OR>
10 {
11  public override void Transform(Place input, OR output,
       ITransformationContext context)
12  {
13   output.Name = input.Name;
14  }
15
16  public override void RegisterDependencies()
17  {
18   Require(Rule<Place2Basic>(),
19    persistor: (or, basic) => or.Contains.Add(basic));
20  }
21 }
```

Listing 10.2: The transformation rules for a place

For the implementation, two rules were created to do exactly what was in the description. The first rule in listing 10.2 is `Place2Basic` that fulfills the first part of the requirement: A new Basic state is created and the name is set accordingly. The second rule, `Place2Or`, fulfills the second requirement and fills its Contains reference (which is for the .NET naming conventions now in Pascal Case, i.e. start with an upper case letter) with the Basic element created for the Place that is also the input for this rule. Furthermore, we also set the name of the OR element for better evaluation.

The description requires a solution for every transition within the Petri Net to create a HyperEdge with a corresponding name and furthermore transform all the pret/postp and post/prep links accordingly. Since we already accessed this rule in listing 10.1, we have to name this rule accordingly as `Transition2HyperEdge` (see fig 10.3).

```
1  public class Transition2HyperEdge : TransformationRule<Transition
       , HyperEdge>
2  {
3   public override void Transform(Transition input, HyperEdge
       output, ITransformationContext context)
4   {
5    output.Name = input.Name;
6   }
7
8   public override void RegisterDependencies()
9   {
10   RequireMany(Rule<Place2Basic>(),
11    selector: transition => transition.Prep,
12    persistor: (transition, preps) => transition.Rnext.AddRange(
         preps));
13
14   RequireMany(Rule<Place2Basic>(),
15    selector: transition => transition.Postp,
16    persistor: (transition, posts) => transition.Next.AddRange(
         posts));
17  }
18 }
```

Listing 10.3: The transformation of a transition

The `Transition2HyperEdge`-rule again requires that the places in the Prep and Postp references are transformed. However, the transformation engine is responsible that any place is transformed at most once per context and rule.

With these five transformation rules, the initialization task is completed. The demanded equivalence function is implicitly stored in the transformation context, as it contains a trace where we just need to trace the transformation output, where a place has been transformed by the `Place2Basic` or `Place2OR` rule. However, this trace functionality is not serialized by default. If we wanted to serialize the trace into a file, we would have to do this on our own.

10.3.2. Reduction

By the time of the TTC, NMF did not support optimization by reduction. NMF Transformations indeed has a pattern matching system to execute transformation rules based on patterns but the system to trigger the required checks does not work in this situation as NMF Transformations only allows to check a pattern once. Furthermore, NMF Transformations currently does not allow to match sets of objects of any size (only fixed sizes). Thus, we implemented the reduction in general purpose code in C#. However, this general purpose code is embedded in the transformation and makes use of the transformation engine, especially of the provided trace.

Therefore, we extend the transformation with the given reduction rules and include the reduction code. As the rules apply on transitions, we write this code in the *Transform*-method of the `Transition2HyperEdge`-rule. We can just call the reduction rule from within the *Transform*-method as both the reduction rules and the transformation is written in C#. We can even put the helper methods that we need into the class representing the `Transition2HyperEdge`-rule. The following two paragraphs describe the implementation of each reduction rules in more detail.

Applying the *AND* rule

The AND rules are implemented in a single method which takes a set of places and a transformation context as parameters. The form to take a set of places as arguments makes this reduction rule more flexible. For example, it can be called for both the predecessors and the predecessors of a transition.

```
1 if (places == null || places.Count < 2) return;
2 var place = places[0];
3 foreach (var other in places.Skip(1))
4 {
5  if (!other.Postt.SetEquals(place.Postt) || !other.Pret.SetEquals
      (place.Pret)) return;
6 }
```

Listing 10.4: Code to check whether the AND rule is applicable

At first, we need to check whether the AND rule is applicable at all. This procedure is presented in listing 10.4.

As soon as a list of candidate places passes these checks, the places and transitions are saved copied into arrays named *_places*, *_postt* and *_pret*. Copying these items into an array is necessary because .NET throws exceptions if a collection is edited while it is enumerated. As the rest of this method will eventually change these collections, we have to copy these into new arrays.

```
1  var or = new Place() { Name = "q" + (created++).ToString()};
2  or.Cnet = place.Cnet;
3  or.Cnet.Places.RemoveRange(_places);
4  foreach (var t in _postt)
5  {
6   t.Prep.RemoveRange(_places);
7   t.Prep.Add(or);
8  }
9  foreach (var t in _pret)
10 {
11  t.Postp.RemoveRange(_places);
12  t.Postp.Add(or);
13 }
```

Listing 10.5: Code to apply the AND rule to the PetriNet

After saving the arguments, we have to apply the AND rule to the Petri Net. The code for this procedure is presented in listing 10.5. Afterward, we create a new place replacing the ones that we are now removing and assign it to the PetriNet. The trace entry created for the abandoned place remains although we could also remove it. We will not access this place anymore and thus, the trace entry may reside in the transformation context. At some point, the garbage collector will take it away, anyway. On the other hand, we remove the original collection of places. Furthermore, for every transition that uses any of the places, we configure it to not use any of the places anymore and use the newly created place instead.

```
1 var or_transformed = context.CallTransformation
2     (andRulePlace2Or, or).Output as OR;
3 var and = new AND();
4 or_transformed.Contains.Add(and);
5 var places_transformed = context.Trace.ResolveMany<Place, OR>(
      _places);
6 or_transformed.Rcontains = places_transformed.First().Rcontains;
7 and.Contains.AddRange(places_transformed);
```

Listing 10.6: Code to apply the AND rule to the statechart model

Now that we applied the *AND* rule to the Petri Net, we also have to apply it to the state charts. The code for this is presented in listing 10.6. We do not just create a new *OR* element. Instead, we call a transformation rule to do this. As a consequence, there is a trace entry created for the transformation. In this way, the algorithm can derive that the newly created place corresponds to the new *OR* model element. This is important especially for the *OR* rule implementation. The variable *andRulePlace2Or* is private variable that caches the instance of the `AndRulePlace2Or` rule. The implementation of this rule is omitted here as it only copies the name of the corresponding place. We just need this rule to exist in order to have a nice way to create a trace entry. Then we register the newly created OR element at the parent compound by setting the opposite reference. The code generated for the StateChart metamodel is aware of the opposite relations and sets the opposite references automatically.

```
1  foreach (var t in _postt)
2  {
3   ApplyOrRule(t, context.Trace.ResolveIn(hypeEdgeRule, t),
4      context);
5  }
6  foreach (var t in _pret)
7  {
8   ApplyOrRule(t, context.Trace.ResolveIn(hypeEdgeRule, t),
9      context);
10 }
```

Listing 10.7: Code to check whether any OR rule is applicable now

Finally, the application of the *AND* rule may have the consequence that it is now possible to apply *OR* rules where the preconditions were not met before because any of the transitions using all the places of this *AND* rule application are not replaced by a single place. This might cause a transition to comply with the *OR* rule prerequisites. However, the OR rule requires not just a transition but also the corresponding *HyperEdge*. Thus, we need to use the transformation context and trace the corresponding *HyperEdge* first. To do this, we need to tell NMF Transformations the input and the transformation rule. In this way, the general purpose C# code can easily make use of the trace functionality of NMF Transformations.

Applying the *OR* rules

The basic structure of the *OR* rule implementation is much the same like the implementation of the *AND* rules. First, we check, whether the rule is applicable. Then, we apply it to the Petri Net and the State Chart model. Afterward, we check if there is any *AND* rule now applicable.

```
1  if (input == null || output == null || input.Postp.Count != 1 ||
       input.Prep.Count != 1) return;
2  var q = input.Prep.First();
3  var r = input.Postp.First();
4  foreach (var qPret in q.Pret)
5  {
6   if (r.Pret.Contains(qPret)) return;
7  }
8  foreach (var qPostt in q.Postt)
9  {
10  if (r.Postt.Contains(qPostt)) return;
11 }
```

Listing 10.8: The code to check whether the OR rule is applicable

In listing 10.8, the code to check the prerequisites for the OR rule is presented. We basically do all the checks that are described in [GR13].

```
1  q.Postt.Remove(input);
2  r.Pret.Remove(input);
3  if (q != r)
4  {
5   r.Cnet = null;
6   q.Pret.AddRange(r.Pret);
7   q.Postt.AddRange(r.Postt);
8   r.Pret.Clear();
9   r.Postt.Clear();
10  q.Name = q.Name + "␣or␣" + r.Name;
11  input.Cnet = null;
12 }
```

Listing 10.9: The code to apply the OR rule to the PetriNet

In listing 10.9, the effect of the *OR* rule to the Petri Net is specified. We use the proposed version from listing 10.9 to merge the *OR* elements instead of creating a new one. In this implementation we use the names of the two places p and q to rename the place q. The place p is discarded.

```
1  var q_or = context.Trace.Resolve<Place, OR>(q);
2  var r_or = context.Trace.Resolve<Place, OR>(r);
3  output.Rcontains = q_or;
4  if (q_or != r_or)
5  {
6   r_or.Rcontains = null;
7   q_or.Contains.AddRange(r_or.Contains.ToArray());
8   q_or.Next.AddRange(r_or.Next);
9   q_or.Rnext.AddRange(r_or.Rnext);
10  r_or.Next.Clear();
11  r_or.Rnext.Clear();
12  q_or.Name = q.Name;
13 }
```

Listing 10.10: The code to apply the OR rule to the state chart model

The code to apply the effect of the *OR* rule to the state charts model is shown in listing 10.10. Again, this is basically the same as before on the PetriNet, we merge the *OR* elements instead of creating a new one and just abandon the old one. To do this, we need the *OR* element that corresponds to *q*. This could be an OR element initially created for a place or an OR element created by the explicit call to the transformation context in the *AND* rule (see listing 10.6). Thus, we use a different version of the trace functionality and do not specify the transformation rule that has been used to transform the place into an OR element.

```
1  foreach (var t in q.Pret.ToArray())
2  {
3   ApplyAndRule(t.Postp, context);
4   ApplyAndRule(t.Prep, context);
5  }
6  foreach (var t in q.Postt.ToArray())
7  {
8   ApplyAndRule(t.Postp, context);
9   ApplyAndRule(t.Prep, context);
10 }
```

Listing 10.11: The code to check whether any AND rule is now applicable

Much like in the *AND* rule, we now check, whether any new OR rule is applicable with the code in listing 10.11. We must copy the items into an array prior to iteration as .NET otherwise claims that we attempt to remove items from a collection we are currently iterating. A reversed `for`-loop is a viable alternative but less concise.

10.4. Other solutions

In this section, we go through the other competing solutions of the Petri Nets to State Charts case and briefly introduce them. Unlike for the Flowgraphs case, there is no additional general purpose solution as the reduction is already implemented in general purpose code by the NMF solution. As the initialization step includes a model transformation of models exposing a high degree of cycles, an implementation of the initialization task would incorporate a lot of bookkeeping (see section 6.2). All solution papers can be obtained online under `http://planet-sl.org/community/index.php?option=com_community&view=groups&task=viewgroup&groupid=24`. Furthermore, demos can be found on SHARE[2].

10.4.1. FunnyQT

For the Petri Nets to State Charts case, FunnyQT uses its model transformation API for the initialization and its plain model modification API for the reduction part. As the code for the initialization is very similar to the code presented in listing 9.11, it is not repeated here. The *AND* rule and *OR* rule are implemented in general purpose Clojure code. To give an impression how this code looks like, the implementation of the *OR* rule is shown in listing 10.12.

[2] `http://goo.gl/rgGBJ`

```
(defn or-rule [pn sc place2or]
 (loop [ts (vec (eallobjects pn 'Transition)), applied false]
  (if (seq ts)
   (let [t (first ts), preps (prep t), postps (postp t)]
    (if (= 1 (count preps) (count postps))
     (let [q (first preps), r (first postps)]
      (if (or (identical? q r)
        (and (not (member? r (adjs q :pret :postp)))
             (not (member? r (adjs q :postt :prep)))))
       (let [merger (@place2or q), mergee (@place2or r)]
        (when-not (identical? q r)
         (eaddall! q :pret (eget-raw r :pret))
         (eaddall! q :postt (eget-raw r :postt))
         (edelete! r)
         (eaddall! merger :contains (eget-raw mergee :contains))
         (edelete! mergee))
        (edelete! t)
        (recur (rest ts) true))
       (recur (rest ts) applied)))
     (recur (rest ts) applied)))
   applied)))
```

Listing 10.12: The implementation of the OR rule in Clojure

Some reviews claimed that this code following fully functional programming is very hard to read and barely understandable at all. However, it definitely is a very concise solution. In the open peer reviews, the FunnyQT solution has also been the fastest solution by far, now beaten only by the NMF solution. The FunnyQT solution was awarded for the best performance as well as the best overall solution.

10.4.2. UML-RSDS

UML-RSDS is a transformation language entirely different to any other solution. Model transformations are defined entirely by a set of rules defined with pre- and post-conditions. From these pre- and postconditions, UML-RSDS generates Java or C# code that executes the transformation. As an example, the transformation rule that for each place a *Basic* element as well as a *OR* element is created is defined using the following post-condition:

$$Basic \rightarrow exists(b|b.name = name \& OR \rightarrow exists(o|o.name = name \& b : o.contains)).$$

The reduction is performed in a similar way. The following gives an example of the pre- and postcondition, whose enforcement performs the OR-rule.

$$
\begin{aligned}
&prep.size = 1 \& postp.size = 1 \& \\
&q : prep \& r : postp \& \\
&(q.pret \cap r.pret) \rightarrow size() = 0 \& \\
&(q.postt \cap r.postt) \rightarrow size() = 0 \Rightarrow \\
&\quad OR \rightarrow exists(p | p.name = q.name + "_O R" + r.name \& \\
&\qquad p.contains = OR[q.name].contains \cup OR[r.name].contains \& \\
&\qquad q.name = p.name) \& \\
&\qquad q.pret \rightarrow includesAll(r.pret) \& \\
&\qquad q.postt \rightarrow includesAll(r.postt) \& \\
&\qquad r \rightarrow isDeleted() \& \\
&\qquad self \rightarrow isDeleted()
\end{aligned}
$$

The authors note that this definition closely follows the specification of the desired transformation. However, in logic, the above post-condition can never hold as the equations $p.name = q.name + \ldots$ and $q.name = p.name$ may never hold at the same time. As the specification works, the engine treats the logical statements slightly differently than one would expect from propositional calculus. As a result, the specification may be easy to read but is difficult to modify as one has to clearly understand what the engine is doing with the specification. This is an error-prone procedure, as the specification relies on the engine implementation.

The UML-RSDS solution also supports bidirectionality, e.g. the Petri Nets to State Charts transformation can be reversed.

10.4.3. Story Driven Modeling Library (SDMLib)

SDMLib is an internal DSL based on Java for domain modeling with a curious history. It was originally developed to show students complaining on the usability of visual tool Fujaba that a textual tool would be much worse [ZÏ3]. Surprisingly for the authors, the textual syntax gained popularity and thus has been further developed. SDMLib also creates a classes to implement model transformations.

For model transformation purposes, SDMLib utilizes the Fujaba tool, as it directly accesses its internal tool API. As a consequence, SDMLib is in some sort similar to graph grammars like TGGs. Thus, the solution creates instances of strongly typed pattern objects as demonstrated in listing 10.13.

```
1  P2SModelPattern preAndPattern = new P2SModelPattern();
2  Transition preAndT = preAndPattern.hasElementTransitionPO(null);
3  preAndT.hasPrepCard(2, Integer.MAX_VALUE);
4  // all pre-places have the same incoming transitions
5  TransitionPO preTransition = preAndT.startNAC().hasPrep().hasPret
       ()
6     .withPatternObjectName("preTransition");
7  PlacePO prePlaceMissingPreTransition = preAndT.hasPrep()
8     .withPatternObjectName("prePlaceMissingPreTransition");
9  prePlaceMissingPreTransition.startNAC().hasPret(preTransition).
       endNAC
10    ().endNAC();
```

Listing 10.13: Strongly typed pattern objects in SDMLib

As the code is a bit hard, SDMLib also offers to automatically visualize the pattern structure of the transformation. Such a visualization is shown in figure 10.4. The patterns created in listing 10.13 are in the upper left corner.

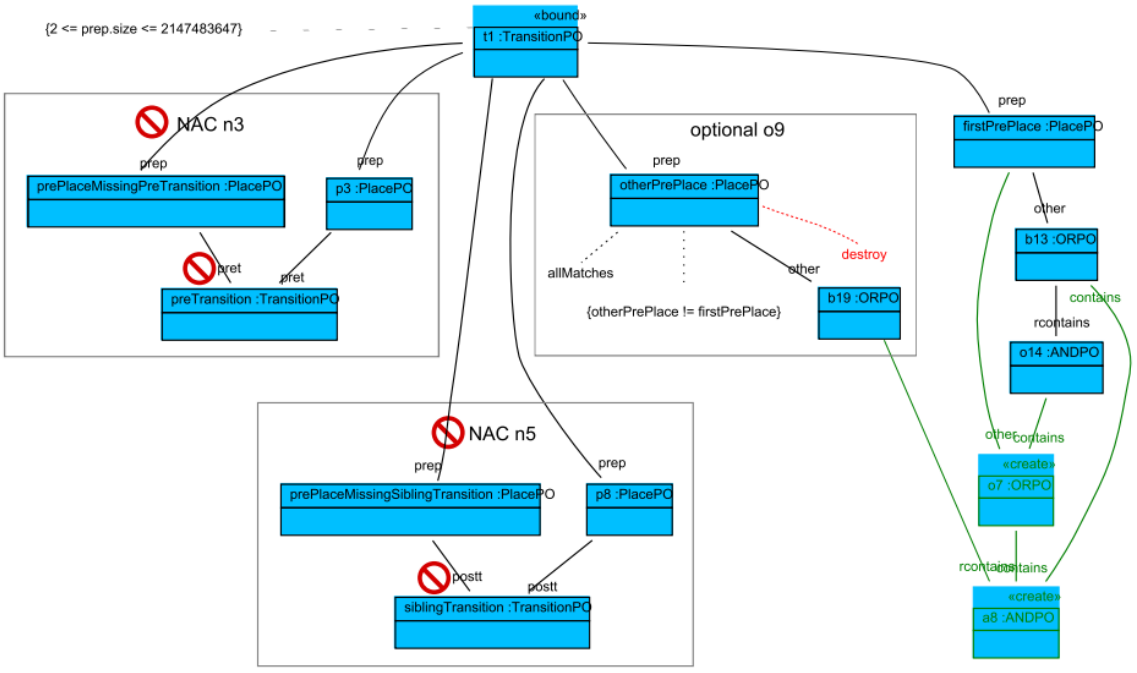

Figure 10.4.: Visualization of the transformation patterns in the SDMLib solution [ZÏ3]

The pattern defined in listing 10.13 uses a *negative application pattern* (NAC), i.e. a pattern that describes when *not* to apply a transformation pattern. What the solution does is to nest these NACs. Thus, to have the pattern match a transition, the transition must have at least two pre-places (line 3). Furthermore, there must not be a pre-place that has an incoming transition (line 5) when there is a pre-place that does not have the same incoming transition (line 9).

The initialization is also done through such patterns. The pattern objects further have methods generated for the models that allow to create child objects in a similar way to the sub-patterns and NACs. I.e., there are functions `startCreate()` and `destroy()` that allow to add and remove objects.

Furthermore, every object in SDMLib has the ability to dump it into a dot-file that can be further processed into an svg-file. This enhancement to debugging brought SDMLib an award for the best debugging support.

10.4.4. EMF-IncQuery

The EMF-IncQuery solution consists of two parts. In the first part, EMF-IncQuery is used to specify patterns when to execute certain transformation rules. The imperative parts of these rules are then specified in Xtend. The orchestration is done with Java. Listing 10.14 shows how the declaration of such patterns look like in EMF-IncQuery (for the AND rule).

```
1  pattern andPrecond(P:Place, T:Transition) {
2   Transition.prep(T,P);
3   countPrePlaces == count find postT(_PX, T);
4   check(countPrePlaces >= 2);
5   neg find nonCommonTPost(T);
6  } or {
7   Transition.postp(T, P);
8   countPostPlaces == count find preT(_PX, T);
9   check(countPostPlaces >= 2);
10  neg find nonCommonTPre(T);
11 }
```

Listing 10.14: The AND rule pattern in EMF-IncQuery

EMF-IncQuery is an incremental technology and therefore supports change propagation in the sense that as soon as new places are added to the Petri Net, the EMF-IncQuery processor can react on that.

10.4.5. AToMPM

AToMPM is a model transformation language based on TGGs. However, AToMPM allows to explicitly set the control flow to arrange in which order the graph transformation rules are to be executed. Figure 10.5 shows simplified representations how such rules look like. In the tool, it is necessary to specify which type the elements on the left and right hand side should be and which references are represented by the conjunction lines. The purple lines represent the tracing functionality of AToMPM.

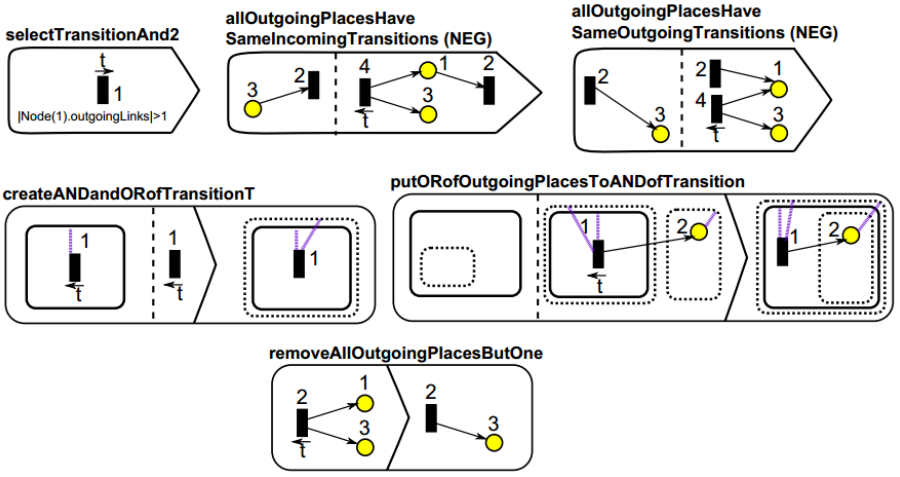

Figure 10.5.: The graph transformation rules to perform the AND rule with AToMPM

Furthermore, the AToMPM solution also consisted of additional rules to simulate a PetriNet which has been awarded with the best simulation support.

10.5. Results on the TTC

The NMF solution to the Petri Nets to State Charts case has been awarded for having the best refactoring support. This can be seen as a consequence of the language nature of NTL to be an internal DSL and thus inherit the great tool support that is provided

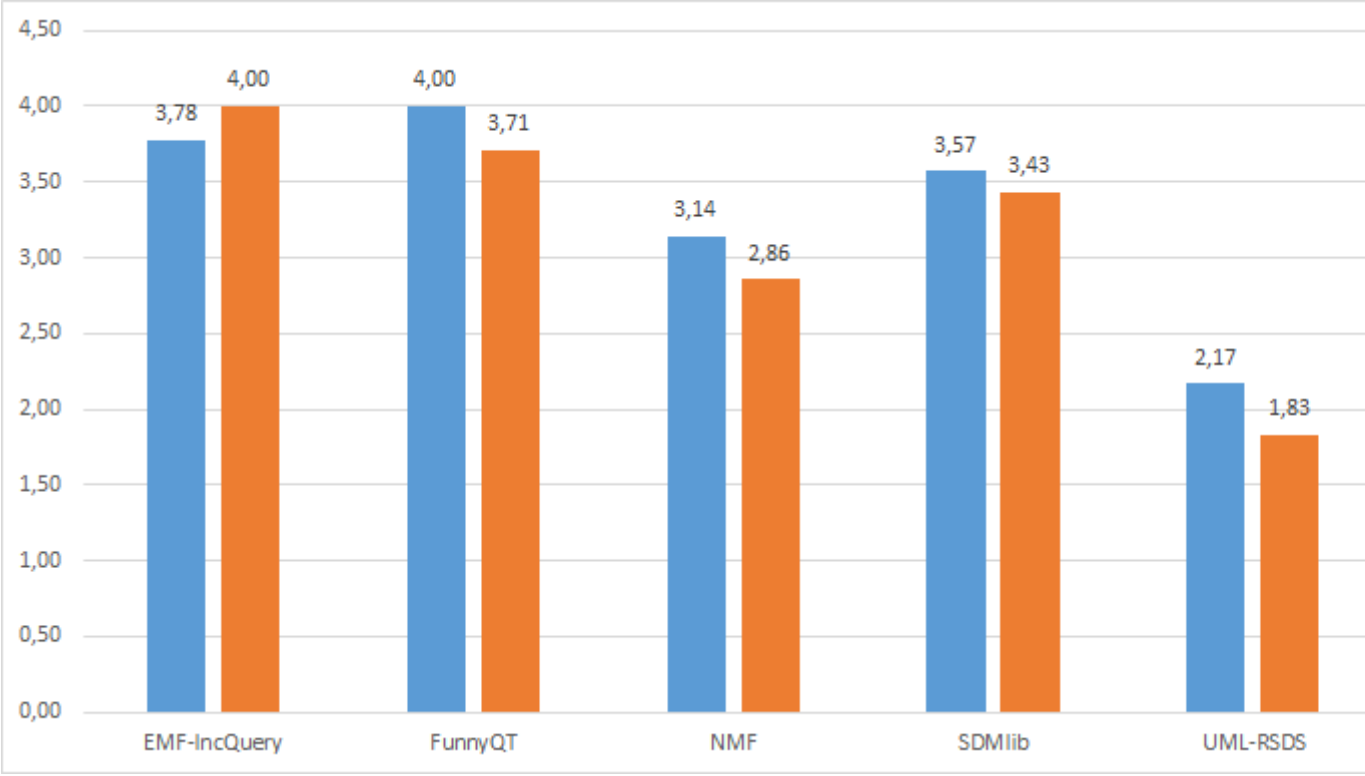

Figure 10.6.: The overall evaluation results from the TTC conference for the PN2SC case

by Visual Studio and other productivity enhancement tools like ReSharper[3] that offer support for various refactorings such as described in [FB99] out of the box. These tools also include a great debugging support, as Visual Studio supports debugging with the "Edit & Continue" feature that allows developers to define breakpoints, edit the code while the execution remains on the breakpoint and continue execution, provided the changes make this possible.

However, NTL was not the only internal DSL and other internal DSLs such as SDMLib [ZÏ3] provided a better support to display a model to the user. Given that NMF Transformations does not rely on a specific metamodeling foundation like EMF, such operations are hard to implement by the transformation engine. Thus, the debugging support award was given to the SDMLib solution.

In the overall evaluation, the NMF Transformations solution achieved the second last place in the ranking of the overall evaluation. As the overall evaluation closely correlated to the evaluation of the presentation, this may also be due to a worse presentation.

The evaluation data is presented in more detail in the appendix, see D.3.

10.6. Validation

In this section, the NMF solution of the Petri Nets to State Charts case is validated for to the validation criteria defined in section 10.2.1 in comparison to the other solutions. The validation is done by evaluating the validation goals stepwise in the following subsections.

In the validation, the AToMPM solution is ignored as both the SHARE demo is meaningless and the solution author did not attend at the TTC conference. As an example, attempting to load the test models, AToMPM returns with the error message that one

[3] http://www.jetbrains.com/resharper/

was trying to load a file with an invalid extension. This error message is just wrong, as the test model files also has the file extension as specified as correct in the error message (`*.model`). The solution authors obviously assumed that the SHARE demo would initialize just as they left it when terminating the VM - which is not true.

10.6.1. Modifiability

10.6.1.1. Debugging support

The results of the perceived debugging support as collected in the questionnaires at the TTC conference are shown in figure 10.7. From this figure, SDMLib has the best debugging support, closely followed by NMF and EMF-IncQuery.

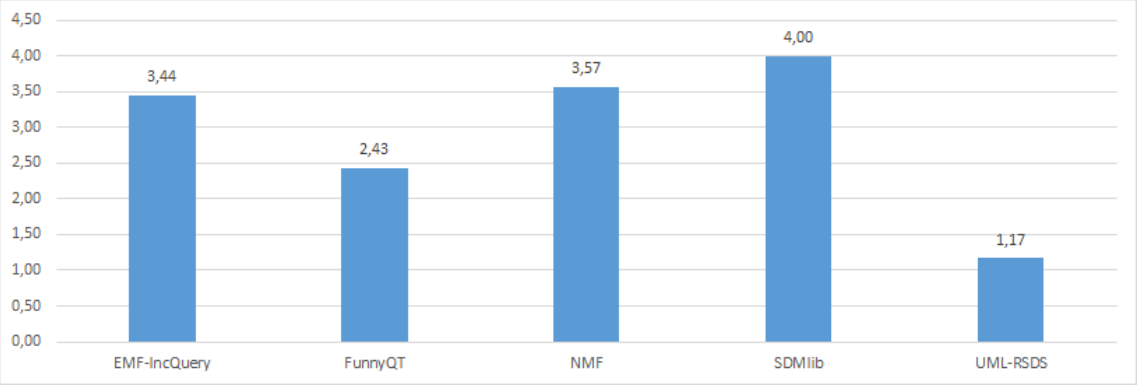

Figure 10.7.: The results for the perceived debugging support for the PN2SC case

The debugging support for NTL has already been explained in section 8.2. However, the debugging support was not a focus of the solution paper at the TTC. This may be a result for the relatively poor rating in the open peer reviews (see section D.3). In fact, most solution papers did not make a statement on debugging and refactoring support of the transformation tool.

A further problem for NTL in the open peer review was that most of the reviewers were not so familiar with the debugging (and refactoring) support of Visual Studio, as most of the developers in the TTC are rather working with Eclipse.

FunnyQT

Being an internal DSL, FunnyQT inherits the tool support from the Clojure tool. As in section 9.7.1, we assume the Eclipse plug-in Counterclockwise as the IDE to use FunnyQT. There, it is possible to set breakpoints in the code but as Tassilo Horn (the author of the FunnyQT solution) admits, the debugging support is not yet ready for prime-time. The language paradigm of Clojure prevents proper debugging. Clojure as a functional programming language passes functions as parameters that are executed elsewhere. As a consequence, the reference to a function and the place where this function is executed can easily differ, such that the position where a fault appears usually is not the place where the bug is in the code. This is similar to NTL where the persistors and selectors can be specified through lambda expressions. Visual Studio 2012 does not allow to set breakpoints in lambda expressions (although it allows to step through lambda expressions). But unlike C#, Clojure is an entirely functional programming language and thus, such effects are much more common than in C#.

However, FunnyQT has built-in model visualization support that makes it possible to discover the elements that are matched by a given pattern.

UML-RSDS

In the solution paper, the authors stated that UML-RSDS does not provide tool support for debugging.

SDMLib

SDMLib is an internal DSL for Java and thus inherits the debugging support of Java. In Eclipse, the debugging for Java is almost as good as the debugging support for C# in Visual Studio. Eclipse also offers an equivalent functionality to the *Edit & Continue* feature from Visual Studio which is called *Hotswap Bug Fixing*. The differences in the debugging support mainly relate to parallel programming scenarios and are therefore mostly unimportant for model transformations.

However, reviewing an objects structure via queries is not always a good way of understanding an objects structure. Thus, SDMLib also supports a generic dump operation that operates on model objects and the transformation patterns and it can dump the models and the transformation files into dot files that can be rendered by GraphViz[4], a tool to visualize graphs. As this offers a visual representation of the model transformation, such visual representations can help the developer to debug the problem as potential flaws are uncovered more easily. The same applies to models, so developers can set a breakpoint and create a dump for a model at a specific time in the model transformation. This helps to inspect the state of the model during debugging more easily.

This visualization support of the model transformation and the models brought SDMLib the award for the best debugging support.

EMF-IncQuery

The EMF-IncQuery solution is divided among three different transformation languages. EMF-IncQuery is responsible to specify the patterns for the reduction process declaratively. This part is impossible to debug for its declarative nature. The declarative way of programming does not have a clear execution semantic attached that could be observed via a debugger. However, the orchestration that is done with Java as well as the actual code executed for these patterns specified in Xtend can be debugged with the full tool support that is provided by the Eclipse IDE.

However, EMF-IncQuery allows to take snapshots of a model and load these into a query explorer, so that transformation developers can review where pattern would be applicable on this particular snapshot. This can be seen as a debugging support for a declarative language.

Conclusion

The comparison shows that NTL has a strong debugging support in terms of following the execution flow. However, it lacks the ability to visualize the models and the model transformation. Unlike many of the other model transformation languages, NTL cannot benefit from the rich tool support built around the modeling framework, such as available for EMF. Model visualizations are more closely related to the modeling environment but they also have a string impact to model transformations built on top of these modeling frameworks. However, this yields a nice portion of future work to enhance the model visualization abilities of NMF.

A visualization of the model transformation is actually even possible with the existing tools, by utilizing the Visual Studio Code Map feature (see chapter 11). However, this possible application of the tool support was only discovered after the TTC.

[4] `http://www.graphviz.org/`

10.6.1.2. Refactoring support

The refactoring support of NTL has been discussed in section 8.2, already. Same as for the refactoring support, the solution paper did not contain any notes on refactoring support. For readers not familiar with C#, this probably resulted in a quite low ranking for the NMF solution.

Fortunately, this could be clarified at the TTC conference and brought the NMF solution an award for the best refactoring support. The results for the perceived refactoring support of the solutions participating in the PN2SC case are shown in figure 10.8. The results show that NMF has the best refactoring support, closely followed by SDMLib.

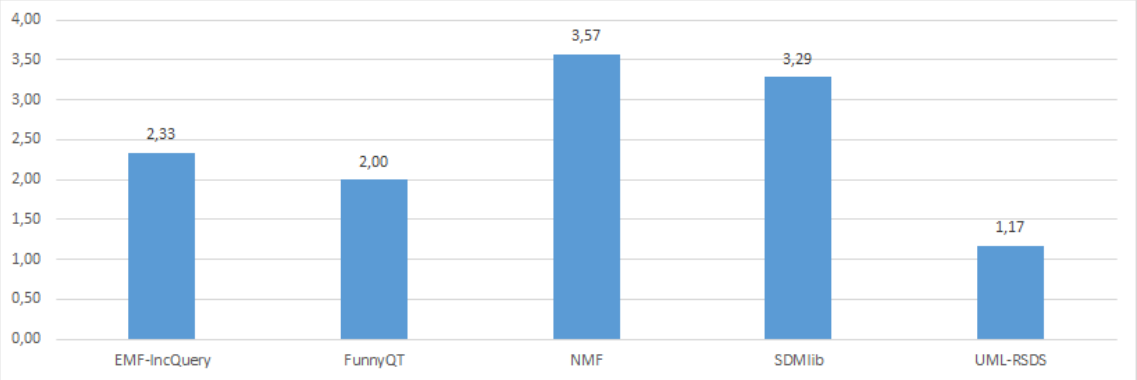

Figure 10.8.: The perceived refactoring support as collected at the TTC conference for the PN2SC case

FunnyQT

As admitted on the TTC conference, FunnyQT and also its host language Clojure do not have any refactoring support. However, the Clojure editor has syntax-highlighting that can ease refactoring a bit. This is probably the reason that FunnyQT has been rated with better refactoring support than e.g. UML-RSDS.

UML-RSDS

In the solution paper, the authors state that UML-RSDS does not have a tool support for refactoring operations. The presentation also did not mention any refactoring support. Indeed, the presentation did not show how the transformation rules are set.

SDMLib

As SDMLib is an internal DSL for Java, it inherits the refactoring support from Java. However, some parts of SDMLib make the usual refactoring operations inappropriate. As an example, a local variable can easily be renamed by Eclipse, but Eclipse does not know that the name of a pattern object is also encoded in the `WithPatternObjectName` parameter that also has to be changed. Thus, the refactoring support inherited from Eclipse is limited through the internal DSL.

EMF-IncQuery

Much like the debugging support, the refactoring support of the EMF-IncQuery solution is dependent on the language. The Java part has the refactoring support built in Eclipse for any Java code. Xtend and EMF-IncQuery inherit the refactoring support of XText, only, which is very limited. However, the basic but very important refactoring operation to rename artifacts is supported in XText (and thus in Xtend and EMF-IncQuery).

Conclusion

The NMF solution actually convinced the committee of the TTC to provide the best refactoring support. This is a consequence of the way how NTL is built on top of C# and the rich refactoring support provided for C# by Visual Studio and further add-ins, such as ReSharper. As C# is a mainstream language and statically typed, the refactoring support is possible (because of the static typing) and maintained by a large development team.

On the other hand, it is also NTL whose design allows to take advantage of these refactorings. This is in contrast to e.g. SDMLib that cannot properly take advantage of e.g. the rename refactoring provided for Java by Eclipse because the name is also encoded as a string parameter to the `WithPatternObjectName` method.

10.6.2. Consistency

NMF

The NTL solution mainly consists of the initialization created with NTL and a reduction part implemented in general purpose code. However, the integration of the general purpose part is seamless. Furthermore, the tasks that are implemented with general purpose code are clearly separated from the initialization. Thus, the solution is very consistent.

FunnyQT

Similar to the NMF solution, the FunnyQT solution also consists of of an initialization part and a reduction part that are both implemented in FunnyQT but with different parts of it. The initialization uses mappings to transform the initial Petri Net into an initial State Chart. Here again, the separation is very clear as the purpose for mappings is strictly restricted to initialization. Thus, the solution is also very consistent.

However, the difference between the mapping API of FunnyQT and the general purpose solution is not as big as in the NMF solution. This is due to the more concise mapping language introduced by FunnyQT that does not differ too much from the general purpose part. In the NMF solution, the general purpose part syntactically looks very different to the general purpose part. A reason could be that any functions in Clojure are compiled into classes by default, which is not true for C# method (that are compiled to methods). As the transformation rules in NTL are also compiled to classes, the syntax for them must differ from the usual syntax for methods. In Clojure, the mapping API is not too much different compared to the general purpose part, as also the general purpose part is compiled to classes.

UML-RSDS

As discussed in section 10.4.2, the definition of the post-condition is not consistent with propositional logic. This is specifically important as the solution authors stress the easy and concise way to specify the model transformation through predicate logic. The fact that the UML-RDSD solution uses this inconsistency for their transformation (as this is the way how the name of a resulting name State Chart state is set for an OR-rule), the term of consistence has to be reconsidered for this case. As a reason, the UML-RDSD solution is consistently using this inconsistency. However, as the solution is based on an inconsistency, the solution is rated as having a poor consistency.

SDMLib

The SDMLib solution uses its story patterns for both initial transformation as well as reduction. However, the nested sub-patterns look quite different, as the story patterns for the reduction require nested negative application patterns, whereas the initialization patterns are rather simple. As the very same technology is used throughout the whole model transformation, the SDMLib solution is the most consistent one.

EMF-IncQuery

The EMF-IncQuery solution uses three different languages for the Petri Nets to State Charts case: EMF-IncQuery itself, Xtend and Java. In the solution, EMF-IncQuery is used to specify the patterns when a rule is applicable. The engine then identifies occurrences of the patterns and trigger other reduction parts written in Xtend to perform the actual task involved in the rule. Java is used to wire up these triggers and orchestrate the solution.

While it is very clear to use EMF-IncQuery to specify patterns (as the language is built for this purpose), the imperative parts of the rule could have also be written in Java. Although there are reasons to use Xtend (e.g. better conciseness), the usage of a further language introduces a further programming style and thus yields a less consistent solution. Furthermore, the separation between Xtend and Java is not obvious. Thus, the solution is relatively inconsistent, as three different programming styles (in three different programming languages) are used.

Conclusion

The most consistent solution is the SDMLib solution as it uses the same programming style for both initialization and reduction rules. It is followed up by the FunnyQT and NMF solution that each apply two different programming styles. However, FunnyQT is a bit more consistent, as the syntax of these programming styles is closer to each other. EMF-IncQuery is relatively inconsistent, as it incorporates three different programming languages where the separation is not so clear. The UML-RSDS solution can be seen as the most inconsistent solution, as it is based on inconsistencies in way how the engine interprets the predicate logic from the transformation rules.

10.6.3. Conciseness

Solution	NLOC
FunnyQT	91
UML-RSDS	96
NMF (source code)	250
NMF (based on IL)	180
SDMLib	220
EMF-IncQuery	578

Table 10.1.: NLOCs of the Petri Nets to State Charts case solutions

Unfortunately, when the conciseness analysis was written, the SHARE demos were unavailable. Thus, the evaluation can only be done based on the conciseness evaluation during the open peer reviews, counted by Tassilo Horn (see table 10.1). According to this data, the most concise solutions were FunnyQT and UML-RSDS with about 90 NLOC each. The second most concise solutions were NMF and SDMLib with about 230 NLOC each. The least concise solution was EMF-IncQuery with 578 NLOCs.

Similar to the Flowgraphs case, the LOCs computed by Visual Studio are also included in the table. As a much bigger proportion of the code is usual general purpose code, the numbers for NMF are much closer together than in the Flowgraphs case.

This may be surprising as the patterns specified in EMF-IncQuery are very concise. However, the orchestration efforts in this solution seem to overweigh these advantages also in terms of the solution conciseness.

The NMF solution again suffers from the verbosity of NTL. Other mapping languages are more concise. However, the general purpose part is quite concise - even remarked in the peer reviews. As a result, the difference between NMF and FunnyQT is a little less than in the Flowgraphs case where the solutions were apart by a factor of 3.

10.6.4. Understandability

The results for the understandability of the solutions of the PN2SC case are shown in figure 10.9. The results are based on 36 responses to the questionnaire of the PN2SC case at the TTC conference[5].

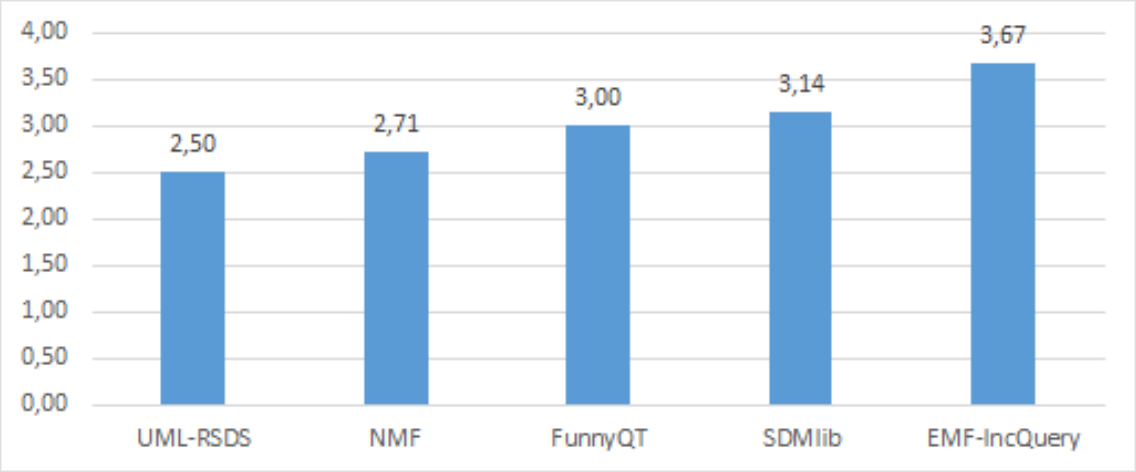

Figure 10.9.: The results of understandability and conciseness for the PN2SC case

The results indicate that the NMF solution has the second-worst understandability and conciseness. EMF-IncQuery is rated for having the best combined understandability and conciseness. As a reason, the presentation of the EMF-IncQuery solution focussed on the pattern specifications that are indeed written very concisely. As the questionnaire collect the *peceived* conciseness and understandability, the solution appearently got very good marks for its conciseness. As a reason, the EMF-IncQuery solution applies an external language that is tailored to such pattern specifications while the NMF solution applies general purpose code for the patterns that are no way tailored to the solution. On the other side, UML-RSDS also has a biased rating as it possibly got downgraded as the presentation failed to introduce how the concise transformation rules are specified.

However, the fact that SDMLib and FunnyQT were given better ratings than NMF is suprising. The SDMLib solution may have gained some points from the visualization of its transformation rules. Appearently such a visualization improves not only the debugging but also the understandability of such a solution. The code in SDMLib is rather verbose and hard to read and thus is unlikely to have caused a good understandability. A roughly similar tool support is available for NMF Transformations via the Code Map feature of Visual Studio but this support has only been detected after the TTC conference.

[5] `http://goo.gl/yU3as`

However, the transformation languages of NMF Transformations and SDMLib are also quite different. Where NMF Transformations rather specifies mappings, SDMLib like EMF-IncQuery specifies patterns that may occur multiple times. After all, the inappropriate abstractions of transformation rules that may only be executed once made the NMF solution solve the reduction in general purpose code. Thus, the abstractions from SDMLib seem to be better suited for this case and may therefore deserve the better understandability and conciseness ranking.

FunnyQT may have taken most profit from its conciseness. Like NMF Transformations, FunnyQT does not have suitable abstractions for pattern specification and thus solves the reduction with general purpose code. However, the built-in mechanisms of the Clojure language seem to suffice that the lack of suitable abstractions is concealed by the concise general purpose specification of these patterns. What FunnyQT does is to iterate functions that perform the pattern tasks until they return `false` and thus no more patterns are applicable. It may also have impressed the attendees of the TTC with its boosting performance. At the TTC conference, the FunnyQT solution was the fastest solution by far.

Another factor that may serve to support the assumption of the tool support helping to imropove the understandability is that a similar tool support has recently been presented for QVT [RNHR13] where it indeed helped to improve the maintenance of model transformations.

Summarizing the discussion, NMF Transformations seems to lack in conciseness and understandability of the pattern specification due to a lack of suitable abstractions. Furthermore, the NMF did not specify the patterns in general purpose code as concisely as the FunnyQT solution and suffers from the lack of tool support to improve the understandability. However, a similar tool support is available but has not been detected by the time of the TTC conference. Although this biases the results a bit, NMF Transformations still seems to lose the understandability due to inappropriate abstractions. A solution may be to allow transformation rules to be executed multiple times and use the relational extensions to specify these patterns. However, this is a portion of future work.

10.7. Conclusions

The Petri Nets to State Charts case contained a rather unusual model transformation task, an input-destructive model transformation that consists of an initialization and a reduction part. The abstractions of NMF Transformations do not fit these requirements. As a result, the reduction part has been solved by general purpose code. This solution demonstrates how arbitrary general purpose code can easily be included in a model transformation written with NMF Transformations. The approach of integrating general purpose code whenever the model transformation abstractions provided by NTL make it a very flexible approach.

Similar to the Flowgraph case, the Petri Nets to State Charts case also showed the good tool support provided for NTL. Although also other internal DSLs could benefit from the tool support of the IDE, examples like SDMLib show that not all internal MTLs can make use of the provided refactoring support as NTL does. On the other side, also SDMLib extended the tool support from Eclipse by providing a custom model visualization. This is a useful improvement for debugging. Possibly a similar tool support also served to improve the understandability of SDMLib.

After all, the goal of the different solutions seems to be quite different. While the goal of the NMF solution was to demonstrate how general purpose code could be integrated, other transformation languages mostly concentrated on their pattern matching abilities

and wanted to show how these abilities also apply on unusual model transformation tasks. The lack of such abstractions lead to a low understandability of the NMF Transformations solution. Thus, the case study shows a clear limit of the usage of NMF Transformations.

11. Code generator for OPC UA

This chapter introduces the case study of a code generator for OPC UA, conducted at ABB Corporate Research, Ladenburg, Germany. At first, section 11.1 briefly introduces OPC UA, specifically its Address Space Model. Next, section 11.2 introduces the code generating mechanism that is to be implemented, before section 11.3 explains the planned validation for this case study. Section 11.4 presents the solution of this case study using NMF. Section 11.6 evaluates the solution with regard to the evaluation criteria from section 11.3.1. This validation is partially done against general purpose solutions. Finally, section 11.7 concludes this chapter.

11.1. OPC UA

In the automation domain, it is of great importance to exchange data with the devices that analyze or handle an industrial process (e.g. reading a temperature or pressure values, position valves or start motors). For this purpose, the OPC Foundation[1] (Open Platform Communication) has released the OPC Unified Architecture standard (OPC UA[2]). An introduction can be given by [MLD09]. One of the biggest improvements to its predecessor Classic OPC (besides the platform independence) is the introduction of what OPC UA calls the Address Space Model.

This Address Space Model is a way to describe a syste and its metadata. For this purpose, the device data is represented in a full meshed network of nodes with different types. The model is structured by object type nodes that define the minimum structure for other instance nodes. It can be considered as a loosely coupled type system. However, unlike most type systems built into programming languages that exactly define the layout of objects in memory, objects in OPC UA are represented in nodes with references (although these nodes eventually have a memory representation). Instead of specifying the exact layout of these nodes, object types in OPC UA only specify constraints to these nodes, such as they must have certain references to other nodes.

An example of such types is depicted in figure 11.1. The *ObjectType MotorType* defines the structure of its instances. Although the only instance *Motor1* in this example has exactly the same structure, its structure could be slightly different. Other than fields of an object that explicitly belong to this object, nodes in OPC UA may be referenced by

[1] `https://www.opcfoundation.org/`
[2] `https://www.opcfoundation.org/UA/`

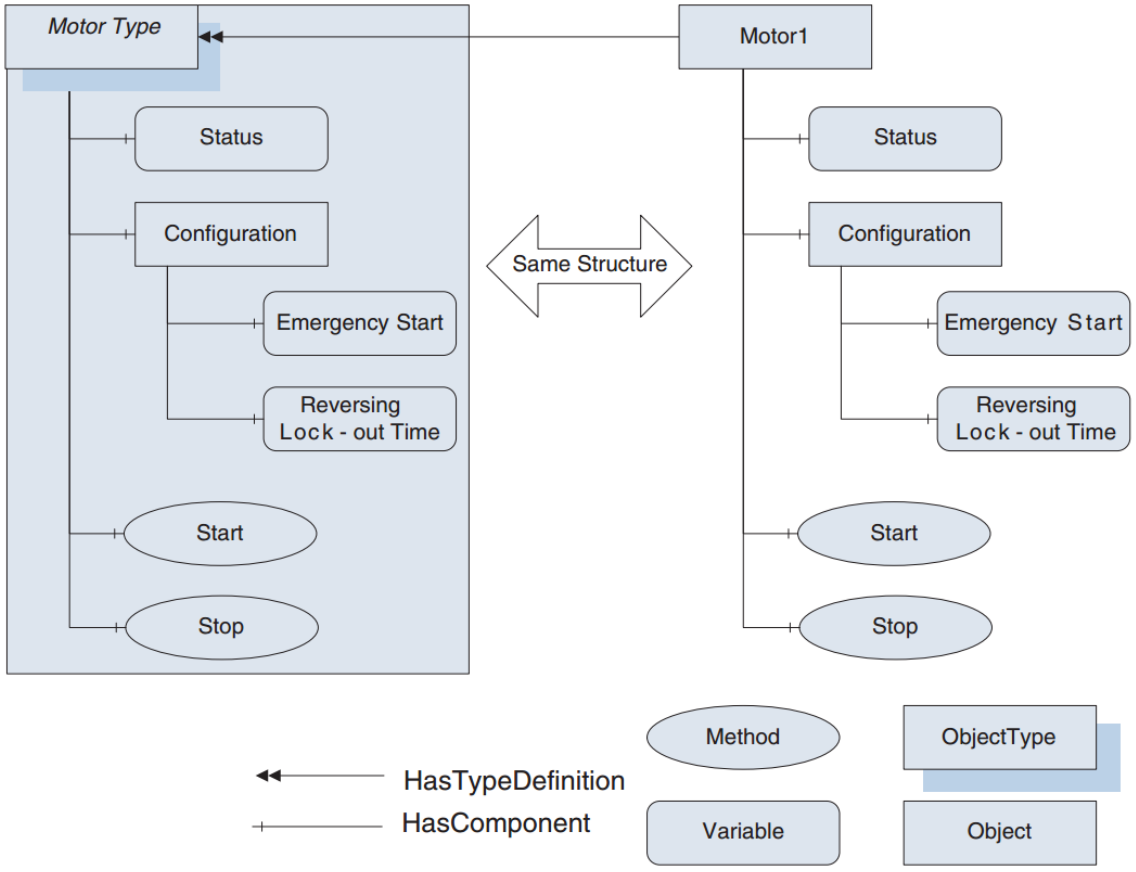

Figure 11.1.: A motor type in the OPC UA Address Space Model [MLD09]

arbitrarily many references. These references are typed with *ReferenceTypes*. It is also possible to create own reference types.

The object type only defines constraints to the structure of its instances. Thus, unless the motor type explicitly disallowed this, a motor instance may have several configurations assigned to it. The OPC UA standard also provides means to declare new variable types or data types. Data types can have an inheritance hierarchy and can be arrays of arbitrary dimensions (including "at least one") or scalars. Furthermore, OPC UA also supports to specify enumerations. Instances can be marked as event sources of an object.

With this Address SPace Model, OPC UA defines a flexible way to communicate between devices. A device can identify any object on another device via its node id. The necessary observations or operations as e.g. measuring a temperature or starting a motor are simply done by requesting the value for a specific node or invoking a method node. Such a method node represents an operation of a certain object, e.g. to start a certain motor. The standard also allows to exchange the metadata of this device types.

11.2. The model transformation in theory

Due to its flexibility, the Address Space Model of OPC UA is great to specify the data that can be obtained from a server. However, if developers want to use this interface for their applications, the flexible object layout is not desirable. Instead, developers usually want to fix the object layout by defining classes in an object-oriented design with a fixed object layout that they can work with. Code generators that can do this task already exist. However, there are a couple of different SDKs for OPC UA and all these different SDKs have their own code generators that generate code in one language that is specific to this particular SDK.

To reduce the duplication of code, it is desirable to have a code generator that can be easily adopted to the case-specific requirement, e.g. the used SDK and creates code in multiple languages. In a first step, a basic code generator is created that is not specific to any SDK. This code generator is not intended to be used directly. Instead, it is designed to allow easy extensibility for the case-specific requirements.

Fortunately, .NET provides a code metamodel independent from the programming language. This code metamodel is represented by the classes of the `System.CodeDOM`[3] namespace. Various code generators exist that generate code from such code models in different languages such as C#, Visual Basic.NET, C++ and J#. Being a Java port, J# provides a similar syntax to Java, so that this code generator may also be used to emit Java code. However, in case of Java, the code generator should behave differently, as Java uses different naming conventions and does not have concepts like properties or events. Thus, in this case, the code generator must create a model with separate getter and setter methods and avoid creating events. As NMF Transformations operates on POCOs, it is possible to use the `System.CodeDOM` classes as the target metamodel for a M2M-transformation written with NTL.

The intended scenario is that the code generator that is currently agnostic of the used SDK is extended by another code generator that is specific to the used SDK and adds the functionality specific to the SDK. The code generator creates a model of the `System.CodeDOM` namespace which is then generated to code by the code providers for the intended language. This procedure is illustrated in figure 11.2.

Some SDKs might require the generated types to eventually inherit from a specific base class. With other SDKs it might be possible to retrieve the references of an object directly

[3] `http://msdn.microsoft.com/en-us/library/system.codedom.aspx`

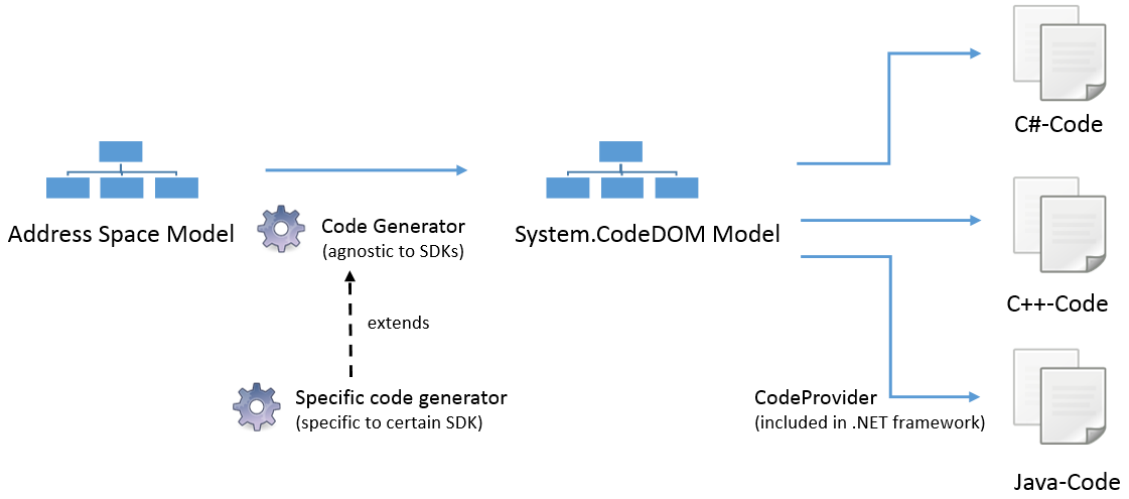

Figure 11.2.: The intended code generator at a glance

from the server and possibly caching them locally. This can be achieved by downloading these referenced objects the first time a reference is accessed (lazy loading). Such an access is easy to track via property getter methods. Thus, the code generator must provide easy means to extend the code generated for a referenced object.

However, designing the code generator, it is yet unclear in which scenarios the code generator might be used. Instead, the idea is to provide a basic implementation that is to be adopted for the scenario. The problem here is that it is unclear where the the code generator is going to be extended, i.e. it is unclear where to set extension points.

11.3. Planned validation

This section presents the planned validation for the ABB case study.

11.3.1. Evaluation criteria

For the code generator case study, the validation goals are as follows:

- **Understandability:** Unlike the first two case studies, the code generator for OPC UA is an example of a case where most model transformation languages are infeasible as they cannot operate on plain CLR objects. As a consequence, the resulting code model of other model transformations would have to be converted to `System.CodeDOM` or otherwise separate code generators would have to be written per language that is to be supported. Thus, this scenario would otherwise be dealt most likely with general purpose code, i.e. plain C#. As the prospected users of the code generator are familiar with C# but not necessarily with model transformation languages, it is important to evaluate the understandability of the solution. This consists of the understandability of NMF Transformations as well as the understandability of the model transformation. This is done by using questionnaires among the prospected users of the code generator.
- **Modifiability:** The code generator case includes a clear extensibility scenario, namely to support new SDKs to generate code that makes use of the features provided by this new SDK which is why the extensibility of the code generator is so important. Existing code generators for OPC only have support for a specific SDK and have been shown that they are difficult to extend. Thus, they lack usability in many scenarios. Hence, it is very important to evaluate the extensibility of the

code generator utilizing NMF Transformations. As extensibility is also an aspect that is hard to measure, this evaluation is also done with questionnaires among the prospected users of the code generator.
Furthermore, as the code generator is created for usage in industry, it is particularly important to achieve a good test code coverage of the model transformation. As a consequence, an important evaluation goal is to observe how the model transformation can be tested by means of unit tests. Multiple metrics exist that measure the coverage of a source code by test code such as *Covered Blocks* or *Covered Lines*.

- **Reusability:** The extension scenarios also include that the code generator is even specifically designed to be reused. In the implementation, the code generator can show that it is possible to write reusable model transformations with NTL. However, as there are no solutions with other transformation languages, no assumptions can be made to compare the reusability support of the model transformation when compared to other transformation languages.

However, as the case study lacks of other concrete solutions to compare to, it is not meaningful to validate the solution in terms of conciseness or consistency. The debugging and refactoring support has already been validated in comparison to other model transformation languages and thus, there is no point in reviewing the support again, especially because the support cannot be compared with other solutions in this case study, as the case study is not solved by any other transformation language.

11.3.2. Evaluation procedure

The testability is evaluated by covering the solution with an as good as possible test code coverage and a reflection how the schematic this test code coverage could have been achieved.

The remainder criteria are evaluated using questionnaires among prospective users of the code generator. As a reason, these criteria are only measurable through perception. In the remainder of this section, the questions of the evaluation sheet are presented together with their intention. The original evaluation sheet can be found in the appendix (see figure E.1).

How much experience do you have with model-driven software development?

This question is used to gather information on the background of the respondents. Unlike the attendees of the TTC that typically have long lasting experience with MDE, the prospected users of the code generator are unlikely to have such a strong background in MDE.

However, the background is important as a strong background in MDE is likely to make it easier to understand NMF Transformations. It uses concepts that exist in similar form in other transformation languages as well. Thus, knowing the background is important to understand the responses of the later questions.

Which transformation languages were you using before?

Again, this question is closely related to understand the responses to the understandability questions that follow.

How understandable was the presentation of the code generator?

As the users only have the solution presentation to understand the code generator, this implicitly collects the understandability of the model transformation in a given amount of time.

How understandable was the presentation of NMF Transformations?

Like the last question, the evaluation assumes that the users only know NMF Transformations from the presentation of the code generator. Thus, the question also implicitly collects the understandability of NMF Transformations in a given amount of time. It can be suspected that NMF Transformations is more understandable for developers used to MDE and also used to other transformation languages like QVT or ATL. On the other hand, it will be interesting to see how the understandability of NMF Transformations correlates with the understandability of the code generator.

Are you planning to use the code generator?

The goal of this question is to determine whether the prospective users really plan to use the code generator presented in this case study. Answers to this question can be interpreted as how convincing the presentation and thus how convincing NMF Transformations is to prospected users.

If so, please rate the effort to adopt the code generator for your scenario!

The requirements for a code generator are different for different business units, which has been accounted for with the explicit requirement for extensibility. However, this question asks how deep the necessary adoptions would be for a usage in the respective business unit.

Please estimate the efforts for the adoptions from the previous question in relation to a general purpose solution!

This question is to gather estimates on the effort of such adoptions. Thus, it collects data on the extensibility of the code generator because of its structure, which implicitly yields a proposition on the reusability of NMF Transformations.

How does NMF Transformations affect the following quality attributes compared with a general purpose solution?

This second last question aims for estimations on the quality attributes. The quality attributes that are asked to rate are

- Understandability
- Modifiability
- Conciseness
- Consistency
- Reusability.

This is probably the most difficult question to answer. Thus, the answers are assumed to be less reliable. As a consequence, the results of this question are only used to doublecheck the whole evaluation and to show trends. A statistical analysis is not meaningful.

Please feel free to add any comment on the solution or the presentation!

This free-form question is to collect more general remarks that do not suit to any of the other questions. The answers to this questions will not be presented in this thesis.

11.4. Generating code with NMF Transformations

From the large, the code generation seems like a typical model transformation task. The model transformation would transform UA type nodes into type declarations from `System.CodeDOM`. To reduce the dependency to any SDK, the code representation for the UA node set has entirely been generated from the XML Schema definition by using the *xsd.exe* tool included in the .NET SDK. Thus, the code generator has been designed as a M2M-transformation with 24 transformation rules where the models are plain CLR objects (POCOs). Most of these rules are visualized in figure 11.3.

What the diagram shows are the connections between the members of the `CodeGenerator` class that implements the code generator. Furthermore, the helper methods from the `CodeGeneratorHelper` class have been added to the diagram. However, to simplify the diagram a bit, some members like constructors have been removed while others have been moved hiding the separation between `CodeGenerator` and `CodeGeneratorHelper`.

As usual in NTL, the transformation rules used by the `CodeGenerator` are public nested classes. An arc between two rules is created as soon as a transformation rule uses another transformation rule either for dependencies (also reversed ones) or for tracing. However, some dependencies are created indirectly (moved to helper methods) and thus not visible in the diagram. Improved tool support that takes this into account would be desirable.

Diagrams as figure 11.3 can automatically be generated by Visual Studio by using the Code Map feature. This feature shows how the members (including nested classes) of the type `CodeGenerator` interact as an interactive DGML file (Directed Graph Modeling Language). Visual Studio also includes a row of automatic layout algorithms that operate on DGML. The diagram in figure 11.3 has only been modified by deleting some members like the constructor from the diagram and improving the layout for the master thesis manually.

The diagram also has further functionality that may turn out particularly useful for model transformation development. As an example, circular references can be detected and highlighted in red, or hubs can be identified. The code map also allows to jump to the code where an element is defined and can analyze all references (possibly loading these objects into the graph).

An example of these model transformation rules can be seen in listing 11.1 where the transformation rule to transform an UA object into a member field is presented. The functionality of the transformation rule is further broken down to smaller units that are handled in smaller methods. In this way, NMF Transformations serves to have to modularize the code. This structure can be queried with the tracing functionality. As in listing 11.1, the transformation rule *ObjectNode2Field* uses the trace functionality to obtain any model element that is not one of the input arguments.

Furthermore, the *ObjectNode2Field* rule declares the methods containing the broken down functionality as virtual methods. This yields an extension point as derived code generator classes can easily override the behavior e.g. that all fields are rendered as collections by default.

```
1 public class ObjectNode2Field : TransformationRule<UAObject,
      CodeTypeDeclaration, CodeMemberField>
2 {
3  public override void Transform(UAObject objectNode,
       CodeTypeDeclaration declaringType, CodeMemberField field,
       ITransformationContext context)
4  {
```

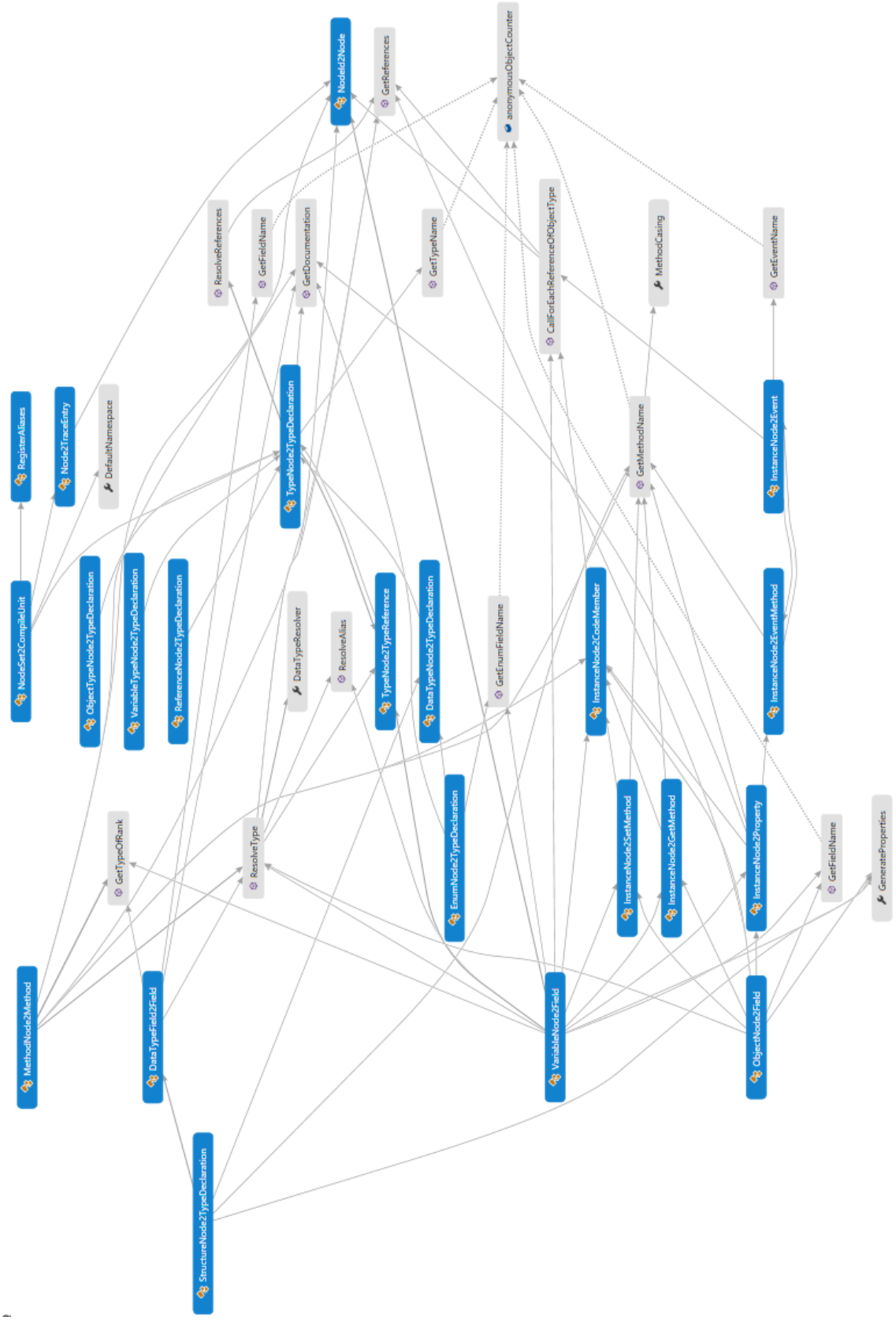

Figure 11.3.: The rules of the code generator and their interactions

```
5    var gen = Transformation as CodeGenerator;
6    field.Name = gen.GetFieldName(objectNode);
7    var typeDef = CodeGeneratorHelper.GetReferences(objectNode,
         KnownReferences.HasTypeDefinition, true, context).
         FirstOrDefault();
8    var type = CodeGeneratorHelper.ResolveType(typeDef, context);
9    if (IsCollection(objectNode))
10   {
11    SetCollectionType(objectNode, declaringType, field, context,
          type);
12   }
13   else
14   {
15    field.Type = type;
16   }
17   if (declaringType != null && !declaringType.IsInterface)
18   {
19    declaringType.Members.Add(field);
20   }
21  }
22
23  private void SetCollectionType(UAObject objectNode,
       CodeTypeDeclaration declaringType, CodeMemberField field,
       ITransformationContext context, CodeTypeReference type)
24  {
25   field.Type = GetCollectionInterfaceType(objectNode, type);
26   var initExpr = GetCollectionInitializationExpression(objectNode
         , type);
27   if (initExpr != null)
28   {
29    var constructor = context.Trace.ResolveIn(Rule<
          TypeNode2DefaultConstructor>(), declaringType);
30    if (constructor != null)
31    {
32    constructor.Statements.Add(new CodeAssignStatement(new
          CodeFieldReferenceExpression(new
          CodeThisReferenceExpression(), field.Name), initExpr));
33    CodeGeneratorHelper.SetCollectionField(field);
34    }
35   }
36  }
37
38  protected virtual CodeTypeReference GetCollectionInterfaceType(
       UAObject objectNode, CodeTypeReference baseType)
39  {
40   return new CodeTypeReference(typeof(ICollection<>).FullName,
         baseType);
41  }
42
43  protected virtual CodeExpression
       GetCollectionInitializationExpression(UAObject objectNode,
       CodeTypeReference baseType)
```

```
44   {
45     return new CodeObjectCreateExpression(new CodeTypeReference(
           typeof(List<>).FullName, baseType));
46   }
47
48   public virtual bool IsCollection(UAObject objectNode)
49   {
50     return true;
51   }
52
53   public override void RegisterDependencies()
54   {
55     MarkInstantiatingFor(Rule<InstanceNode2CodeMember>());
56
57     var gen = Transformation as CodeGenerator;
58     if (gen.GenerateProperties)
59     {
60       Call(Rule<InstanceNode2Property>());
61     }
62     else
63     {
64       Call(Rule<InstanceNode2GetMethod>());
65       Call(Rule<InstanceNode2SetMethod>());
66     }
67   }
68 }
```

Listing 11.1: The rule to transform object nodes into fields

At some points, the transformation tends to be quite complicated. One prominent problem is that the object nodes in a UA node set can be used multiple times in different contexts. As an instance node can be referenced from arbitrarily many other nodes, it may represent properties of arbitrarily many types. However, the code generation for these types may differ. As an example, an object type may have a *HasEventSource* reference to this instance node while other object type nodes referencing that same instance node only have the *HasComponent* reference set. In the first case, an event has to be created that informs clients when the property changes. As a result, the code generation of the property depends on the existence of such an event to inform possible clients. As a consequence, members must be generated from tuples of instance nodes and their type node contexts or their resulting type declarations, respectively. To simplify the syntax in the transformation rules for the members, the latter version is used in the presented code generator.

A further problem is that the UA node set model is only loosely coupled, i.e. a node does not have a true reference to its referenced nodes as the reference only knows the id of the referenced node. To resolve this reference, one must look up in the node set to search for the node with the same node id. As the UA nodes do not have a reference to the node set they reside in, this requires the transformation context. To simplify this procedure, a separate transformation rule is called for each node and creates a trace entry for the `NodeId2Node` rule to identify a node with its node id via the trace. This is done in the `Node2TraceEntry` rule depicted in figure 11.2.

```
1 public sealed class Node2TraceEntry : InPlaceTransformationRule<
      UANode>
2 {
```

```
3   public override void RegisterDependencies()
4   {
5     TraceInput(Rule<NodeId2Node>(), n => n.NodeId);
6   }
7 }
```

Listing 11.2: The Node2TraceEntry rule that creates a trace entry for each transformation

However, this leads to sophisticated dependencies for transformation rules transforming instance nodes to members. As these dependencies are required at multiple rules, this functionality is moved to a helper class `CodeGeneratorHelper`. The code for this helper method is presented in listing 11.3.

```
1  public static void CallForEachReferenceOfObjectType<TNode, TCode
       >(rule, referenceId)
2    where TNode : UANode
3    where TCode : CodeObject
4  {
5    var nodeId2Node = rule.Transformation.GetRuleForRuleType(...);
6    var objectTypeNode2TypeDeclaration = rule.Transformation.
       GetRuleForRuleType(...);
7
8    rule.CallForEach(objectTypeNode2TypeDeclaration,
9      selector: (Computation c) => c.Context.Trace.ResolveManyIn(
         nodeId2Node,
10       CodeGeneratorHelper.GetReferences(c.GetInput(0) as UANode,
           referenceId, true, c.Context))
11       .OfType<TNode>()
12       .PairWithConstant(c.Output as CodeTypeDeclaration),
13     needOutput: true);
14 }
```

Listing 11.3: A helper method to set more sophisticated dependencies

Instead of specifying the selection method based on the inputs, this dependency is specified directly using the plain computation. As a consequence, the dependency must specify whether the selection method requires the output of the computation to be set. In this case, the member transformation rule is to be called based on the computations of the `TypeNode2TypeDeclaration` rule. These computations are aware of their context and thus, this context can be used to acquire the dependent nodes with the specified reference.

However, the fact these dependency specification is moved to a separate class, the usage of this helper functionality hides the dependency on both transformation rules `NodeId2Node` and `TypeNode2TypeDeclaration`. Thus, the diagram from figure 11.3 is actually wrong in the sense that it does not show the dependencies created with this helper functionality. This affects all rules that use the helper functionality, i.e. all transformation rules that transform an instance node to a member object.

As an example of how the transformation rules work, let look on the `InstanceNode2Property` rule. By expanding the node representing the rule in figure 11.3, we can obtain the inner structure of the transformation rule as shown in figure 11.4.

The structure shows that the `InstanceNode2Property` rule consists of further methods that split the work done by this rule further. This is done to override this behavior separately in extensions. As a result, developers can override the behavior how code is

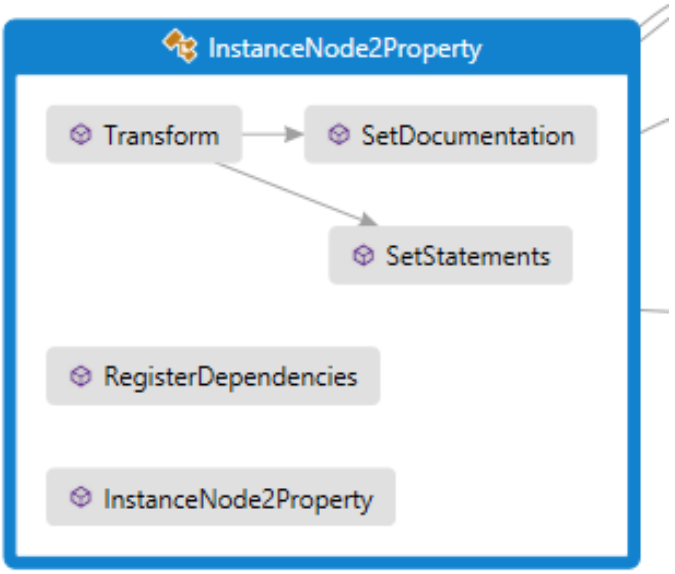

Figure 11.4.: The inner structure of the InstanceNode2Property rule

generated for the getter and setter methods but leave everything else as it. Listing 11.4 shows an example of how this can be accomplished by the introduction of a new transformation rule `NewInstanceNode2Property` that overrides the `InstanceNode2Property` rule.

```
1  public class ExtendedCodeGenerator : CodeGenerator
2  {
3    [OverrideRule]
4    public class NewInstanceNode2Property : InstanceNode2Property
5    {
6      protected override void SetStatements(...)
7          {
8            ...
9          }
10   }
11 }
```

Listing 11.4: An example how a code generator extension can override the code generation for properties

However, by overriding the `RegisterDependencies` method, an extension cannot only override the code generation for a property, but also when a property is created at all. For example, if a developer wished to create a property also for every *Organizes* reference, this could easily be done by overriding the `RegisterDependencies` method and call the above discussed helper function with the reference id of the *Organizes* reference (which is "i=35"). However, overriding the `RegisterDependencies` method must be done with care, especially if the base method is not called (inheritance is still a white-box technology!).

Overriding an existing transformation rule may not always be sufficient to fulfill a changed set of requirements. However, it is still possible to add new transformation rules by including them again as public nested classes of an inherited code generator class. Using reversed dependencies, these new transformation rules may hook in anywhere in the transformation.

Together with the simple extensibility mechanism of overriding the member methods of the *CodeGenerator* class, this yields a three-stage-extensibility of the code generator:

1. **Override member methods:** The easiest way of extensibility is to override the member methods of the *CodeGenerator* class. This is of course limited to the virtual member methods of this class. These methods mainly serve to implement the naming conventions for the code generator.
2. **Override transformation rules:** For adoptions beyond the (quite narrow) limits of overriding member methods, one can override individual transformation rules. In this way, the behavior of these rules can be changed. As both the `Transform` and the `RegisterDependencies` method are not sealed in any transformation rule, the entire transformation rule (besides its signature) can be changed. However, most transformation rules also have virtual methods that allow a more fine grained extensibility, by e.g. only overriding the procedure of how to set the statements for a method.
3. textbfNew transformation rules: For further adoptions, entirely new transformation rules can be added. These transformation rules can be wired up by either overriding the `RegisterDependencies` method or by using reversed dependencies. These new transformation rules of course can take advantage of the tracing functionality of all transformation rules.

11.5. Testing

In many cases, testing for model transformations is achieved by a black-box procedure. Thus, models must be derived that test a transformation rule as isolated as possible. However, in many cases, the input metamodels have a high degree of cohesion, making it impossible to transform tiny models just large enough to trigger a certain model transformation rule.

However, NTL has dedicated support for model transformation testing. This support is achieved by a mock context that does not automatically execute dependencies. Other than the usual transformation context, the mock context also provides a simplified access to the computation list. As a result, this mock object can be used to simulate that a certain object has been transformed with a given transformation output (see section 7.4.8 for details).

With this procedure, it was possible to achieve a full test case coverage (in Release mode; in Debug mode, there are some blocks not coverable by test code). MSTest has been used as testing framework. However, the testing support of NTL is not limited to this unit test framework, but can also be used with other test frameworks.

Visual Studio provides a good tool support for testing. All tests can be run at once through the test explorer that shows the results. Projects can be configured such that the tests run after any build. Furthermore, the Continuous Integration features of the Team Foundation Server allow to use tests for code check-in policies, i.e. a code commit may be refused if the tests fail. Alternatively, the person in charge of the tests may be notified automatically if a code check-in causes the tests to fail.

The unit testing for model transformation rule is demonstrated by the unit tests for the `ObjectNode2Field` rule. This rule is used to transform an OPC UA object and a class context into a member field. The implementation of this transformation rule (excluding comments) is shown in listing 11.1.

The transformation rule has been chosen as it contains examples for most of the concepts included in NTL.

The ways how to create test cases for the solution fundamentally differs for the `Transform` and `RegisterDependencies` method. The following two sections introduce how these

methods are tested for the `ObjectNode2Field` rule and thereby show the schematic procedure.

11.5.1. Transform

Testing the `Transform` method with unit tests yields several problems:

- The method relies on the `Transformation` property to be set to an instance of the `CodeGenerator` class as it uses this reference to obtain the name of the newly created field. Thus, it is not possible to use mock transformation that only consist of few transformation rules. Thus, an instance of `CodeGenerator` must be used as the transformation rule container.
- The calls to the helper methods `GetReferences` and `ResolveType` include an access to the trace functionality of NTL. Thus, the trace must be filled with meaningful trace entries to allow these helper methods to work correctly.

Furthermore, the test code frequently uses the rule instances within a given `CodeGenerator` object. Thus, these references can also be saved as field of the test class. Next, the code helper methods require that the aliases from the node set are registered. This is usually done by the `RegisterAliases` rule, but as we want to execute as less other transformation rules as possible, the aliases are set in the initialization of the test class using the same helper method as the `RegisterAliases` rule does. The complete code for the test initialization is depicted in listing 11.5.

```
1  [TestInitialize]
2  public void Initialize()
3  {
4   codeGenerator = new CodeGenerator();
5   codeGenerator.Initialize();
6
7   objectNode2Field = codeGenerator.Rule<CodeGenerator.
        ObjectNode2Field>();
8   instanceNode2Field = codeGenerator.Rule<CodeGenerator.
        InstanceNode2CodeMember>();
9   typeNode2Reference = codeGenerator.Rule<CodeGenerator.
        TypeNode2TypeReference>();
10  nodeId2Node = codeGenerator.Rule<CodeGenerator.NodeId2Node>();
11
12  context = new MockContext(codeGenerator);
13
14  CodeGeneratorHelper.RegisterAliases(null, context);
15 }
```

Listing 11.5: The test initialization for the unit tests for the Transform method

With this initialization, a test case can be set up as depicted in listing 11.6. This test case tests that the transformation rule sets the correct type when the object references to an object type definition included in the node set (which is the most common scenarios for objects).

```
1  [TestMethod]
2  public void ObjectNode2Field_Transform_KnownType()
3  {
4   var node = new UAObject() { BrowseName = "1:TestName"};
```

```
5  var type = new CodeTypeDeclaration();
6  var field = new CodeMemberField();
7  var typeNode = new UAObjectType() { NodeId = "TestID" };
8  var typeReference = new CodeTypeReference();
9  node.References = new Reference[] {
10  new Reference() {
11    IsForward = true,
12    ReferenceType = KnownReferences.HasTypeDefinition,
13    Value = typeNode.NodeId
14  }
15 };
16
17 context.Computations.Add(nodeId2Node, typeNode.NodeId, typeNode)
       ;
18 context.Computations.Add(typeNode2Reference, typeNode,
       typeReference);
19
20 objectNode2Field.Transform(node, type, field, context);
21
22 Assert.AreEqual("testName", field.Name);
23 Assert.AreEqual(1, field.Type.TypeArguments.Count);
24 Assert.AreEqual(typeReference, field.Type.TypeArguments[0]);
25 Assert.IsTrue(type.Members.Contains(field));
26 }
```

Listing 11.6: Testing the Transform method of the ObjectNode2Field rule

The lines $4 - 15$ set up a minimum test model for this scenario. Lines $17 - 18$ simulate that the node id of the type node has been transformed to the type node and that this type node has been transformed to the type reference from the appropriately named variable. These statements complete the test scenario, so that the `Transform` method of the `ObjectNode2Field` rule can be run (line 20). The lines $22 - 25$ finally make some assertions on the expected outcome of the method.

11.5.2. RegisterDependencies

In the `ObjectNode2Field` rule, the dependencies of the transformation rule depend on the configuration of the parent `CodeGenerator` instance. If the code generator is configured to generate properties, the `ObjectNode2Field` rule has a dependency to also create a property for this field. However, if the code generator is configured to not generate properties, a pair of getter and setter methods is created instead (each using different transformation rules).

A possible way to test the behavior of the `RegisterDependencies` method would be to let the `CodeGenerator` instance register the rules incompletely and then call the `RegisterDependencies` method in the test. However, this approach is infeasible as it is difficult to review the dependencies of the transformation rule. Thus, we rather test this behavior in a black box manner by simulating a computation of the `ObjectNode2Field` rule and execute its dependencies explicitly.

Listing 11.7 shows how this is achieved. As the dependencies of the `ObjectNode2Field` rule depend on the configuration of the code generator, we cannot use a test initialize this time but have to insert the initialization code in the test method (lines $4 - 10$). Lines $12 - 14$ then set up the test environment. Line 16 explicitly executes the dependencies of this

computation that are to be executed after the computation itself (since the dependencies are call dependencies). In the normal transformation context, these dependencies would be executed automatically. However, in the mock context, they are not and must instead be executed separately. Next, lines $18-20$ fetch the property, getter and setter method created for this object. Lines $22-24$ assert that the computation created for the property exist, but no getter and setter methods have been created.

```
1  [TestMethod]
2  public void
       ObjectNode2Field_RegisterRequirements_GenerateProperties()
3  {
4   var codeGenerator = new CodeGenerator() { GenerateProperties =
       true };
5   codeGenerator.Initialize();
6
7   var objectNode2Field = codeGenerator.Rule<CodeGenerator.
       ObjectNode2Field>();
8   var instanceNode2Property = codeGenerator.Rule<CodeGenerator.
       InstanceNode2Property>();
9   var instanceNode2GetMethod = codeGenerator.Rule<CodeGenerator.
       InstanceNode2GetMethod>();
10  var instanceNode2SetMethod = codeGenerator.Rule<CodeGenerator.
       InstanceNode2SetMethod>();
11
12  var context = new MockContext(codeGenerator);
13
14  var node = new UAObject();
15  var type = new CodeTypeDeclaration();
16  var c = context.Computations.Add(objectNode2Field, node, type);
17
18  context.ExecuteDependencies(c, false);
19
20  var property = context.Trace.TraceIn(instanceNode2Property, node
       , type).FirstOrDefault();
21  var getMethod = context.Trace.TraceIn(instanceNode2GetMethod,
       node, type).FirstOrDefault();
22  var setMethod = context.Trace.TraceIn(instanceNode2SetMethod,
       node, type).FirstOrDefault();
23
24  Assert.IsNotNull(property);
25  Assert.IsNull(getMethod);
26  Assert.IsNull(setMethod);
27 }
```

Listing 11.7: Testing the call dependencies of the ObjectNode2Field rule

The instantiation can be tested more easily as a transformation rule has a property to its base rule and an additional method `IsInstantiating` to determine whether it will instantiate a specific computation.

11.6. Validation

In this section, the results of the empirical study among prospected users of the code generator from this case study are presented. At first, section 11.6.1 presents the results from

the evaluation sheets. The testing support has already been evaluated in section 11.5. Section 11.6.2 will evaluate the understandability of the solution and NMF TRANSFORMATIONS while section 11.6.3 evaluates the perceived extensibility of the solution.

In total, only five responses for the evaluation sheet were collected. Thus, the assertions cannot be validated by means of statistical analysis. Furthermore, the data can only serve to show trends.

11.6.1. Evaluation sheet results

In this section, the results from the evaluation sheets are presented. There were few responses so that further statistical analysis is infeasible.

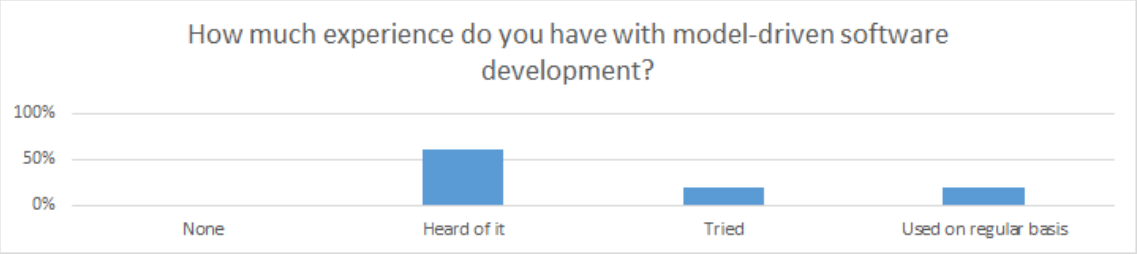

Figure 11.5.: The results for question 1

The results of question 1 are shown in figure 11.5. Five responses for this question could be collected. The results show that most of the attendees barely have experience with model-driven techniques.

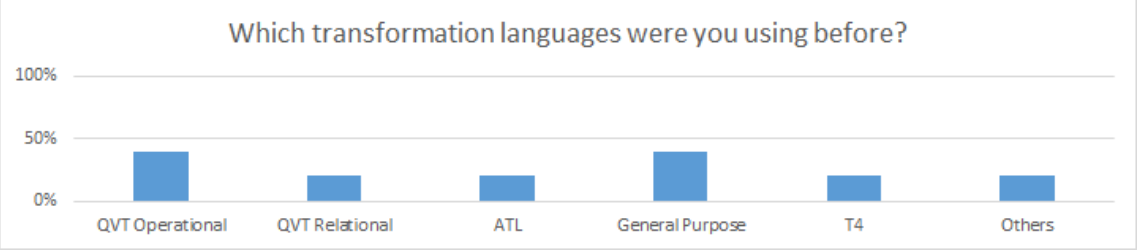

Figure 11.6.: The results for question 2

The results for question 2 (see figure 11.6) confirm the results of question 1. Most users do not have any experience with model transformation languages. Five responses were collected, although some of them did not have any answer ticked. However, this may be due to the fact that some people might not have experience with any of the given transformation languages.

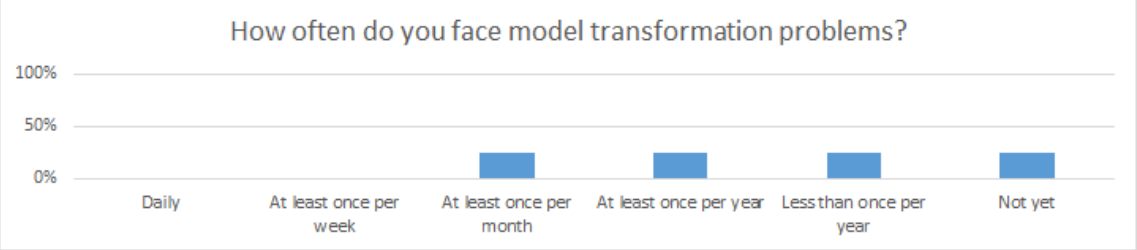

Figure 11.7.: The results for question 3

The results of question 3 (see figure 11.7) shows that most attendees do not frequently face model transformation problems. Most responses came from people that face model transformations less than once in a month. Again, five responses were collected.

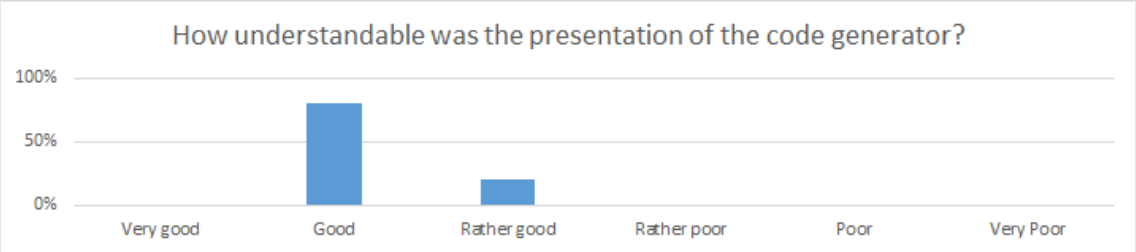

Figure 11.8.: The results for question 4

Question 4 has been the first question on the topic rather than on the responders background. The results shown in figure 11.8 show that the respondents understood the code generator quite good. In particular, almost all responses voted for a good understandability. As the code generator is an example of a model transformation with NTL, we may conclude that NTL can be used to write understandable model transformations. Five responses could be collected.

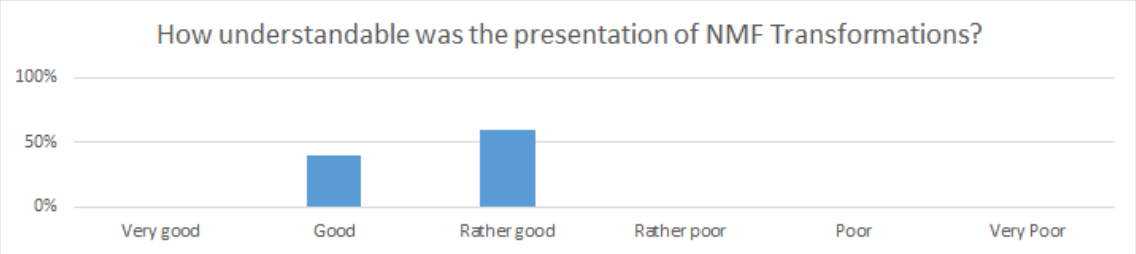

Figure 11.9.: The results for question 5

Unlike the code generator, the underlying framework was not as understandable. At least, this is indicated by the results of question 5 (see figure 11.9). Still, most responses reported a at least rather good understandability of NMF Transformations but the responses are worse than those given for question 4. However, in combination with the results from question 4, this speaks that one does not necessarily has to understand the underlying framework to understand what is going on in the code generator - in scenarios like this where most users do not face model transformations on a regularly basis, this is a good thing. Five responses could be collected.

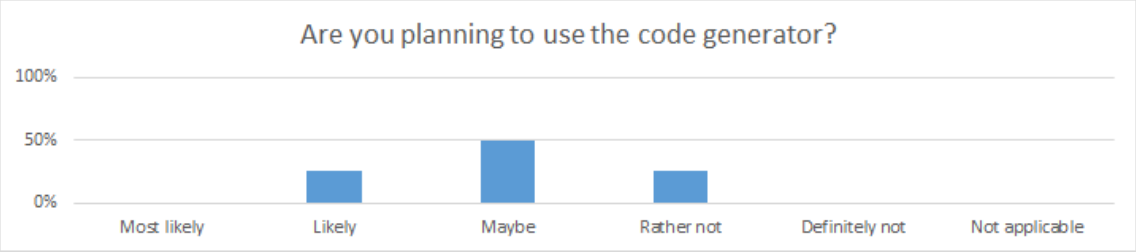

Figure 11.10.: The results for question 6

The responses regarding whether the attendees would use the code generator in production code are rather reluctant (see figure 11.10). Although a ”Not applicable” answer was provided, only four responses were collected. The other responses indicated that the

respondents were unsure whether they will use the code generator as the responses are centered arround ”Maybe”.

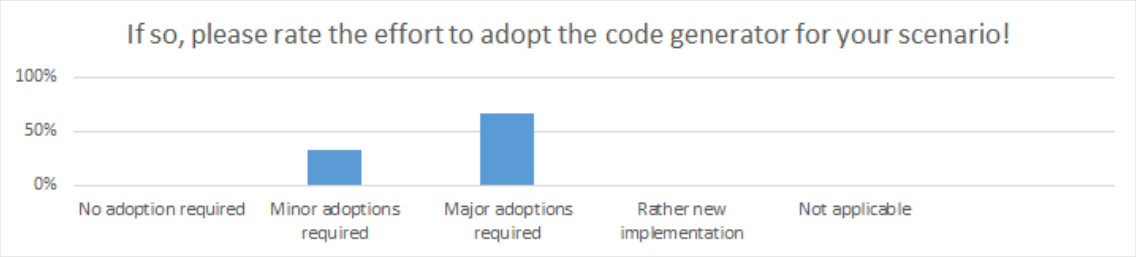

Figure 11.11.: The results for question 7

However, the results from question 7 show that indeed, in case of a use in a production environment, the code generator would be a subject to adoption. This again underlines the importance of the extensibility.

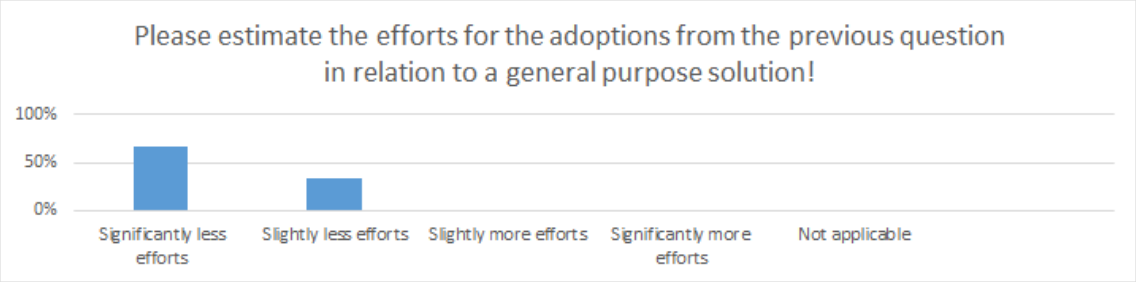

Figure 11.12.: The results for question 8

In case of such adoptions, the results of question 8 (see 11.12) show that the structure of the code generator help to reduce the efforts. Most responses even stated that the efforts could be reduced significantly.

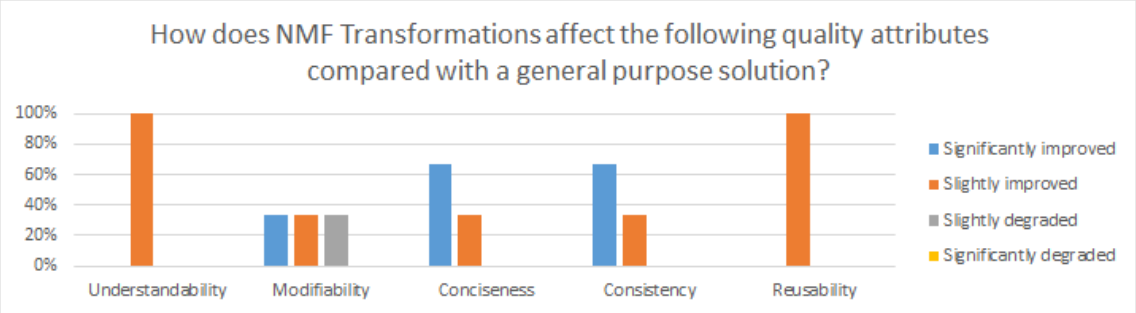

Figure 11.13.: The results for question 9

The results of question 9 in figure 11.13 indicate that most quality attributes of model transformations are improved using NMF Transformations. The clearest improvements compared to a general purpose solution were expected for the consistency and conciseness where NMF Transformations is expected to achieve significant improvements. As a reason, what NMF Transformations mainly does is to bring the abstractions known for model transformations to the C# language. As a result, the code generator gains some shares of the conciseness and consistency of model transformations written in other MTLs.

Other quality attributes like modifiability were not expected to improve significantly. One

response even reported that the modifiability of the model transformations is slightly degraded.

From all quality attributes, the modifiability is the one that the respondents seem to be most uncertain with as the responses are most discordant. The reason for this may be that modifiability has multiple aspects that each of the respondents may have weighted differently. On the one hand, modifiability incorporates consistency, conciseness and understandability. If a developer wants to modify code, he has to understand it. If the model transformation is written in a consistent manner, the programming style for the modification is clear. A better conciseness result in less code to modify. Following these arguments, one could argue that the modifiability is improved with NMF Transformations, as seemingly understandability, conciseness and consistency are improved by NMF Transformations. On the other hand, modifiability is probably the quality attribute that is most affected by tool support. This existing tool support in tools like Visual Studio was created for general purpose code. Without knowing NMF Transformations too much, one could be doubtful about whether this tool support is also applicable on NTL. This may be the reason for the responses expecting the modifiability to decline. However, the previous case studies indicate that the tool support provided by Visual Studio are still valid for NTL.

11.6.2. Understandability

The results from the understandability questions indicate a good understandability of the code generator (see figures 11.8 and 11.9) and NMF Transformations. The few responses do not allow any analysis on correlations. However, the code generator was even rated more understandable than NMF Transformations, indicating that the code generator as a concrete model transformation example can be understood without an in-depth knowledge of the transformation language. As a reason, for anyone knowing C#, the code is still understandable as a general purpose code. Although NTL adds a flavor of an internal DSL, the code within the transformation rules read pretty much like usual general purpose code that is understandable without any further knowledge of NMF Transformations. As the very first questions unveil that most developers do not have a strong background in model-driven techniques (unlike the participants at the TTC), this is an important finding.

The results of question 9 show that most people told that the understandability was improved by NMF Transformations, although only slightly. As a possible reason, to fully understand a model transformation, one has to understand the underlying model transformation language or framework. The results of question 5 show that the model transformation language and framework used for the code generator, NMF Transformations, was not understood really well. This hampers the impact of NMF Transformations on the understandability of a model transformation language. On the other hand, both conciseness and consistency are improved, some even say significantly. From the results, these improvements probably prevail the drawbacks of a further framework or language to master. In combination, this argumentation leads to slight improvements for the understandability which is confirmed by the responses.

11.6.3. Extensibility

The results of question 7 indicate that indeed, the code generator is a clear subject of adoptions. Most respondents report that major adoptions would be required to apply the code generator in a production scenario. As a reason, the code generator does only produce very basic code that is not tied to a specific SDK, yet. In a use case, users will have to add support that the code generator generates code tied to the SDK that they want to work

with. The results of question 6 indicate that these adoptions look to big to master, as they tended to rather not use the code generator of this case study. However, the majority of responses indicate that extensions of the code generator would viable alternatives.

According to the extensibility and adaptability of the code generator, the results of question 8 clearly show the trend that the efforts for the adoptions are expected to decline with the NMF Transformations code generator. Most responses tell that the efforts for the adoptions are significantly reduced in the code generator implementation. This may be seen a confirmation of the discussion made in section 7.4.9. Furthermore, the code generator combines the possibilities of NMF Transformations with the features offered by the C# host language. Furthermore, as reversed dependencies may also serve as an extension mechanism, this yields the three-stage-extensibility described in section 11.4.

11.7. Conclusions

The ABB case study showed the applicability of NMF Transformations in an industry setting. As a basic prerequisite, NMF Transformations operates on plain objects and thus can result in an object model of the `System.CodeDOM` namespace that can be used for language-independent code generation.

The resulting model transformation could be entirely covered by unit test code. Almost more important, the model transformation could be shown to have an excellent extensibility. This is achieved by combining the extensibility features of NMF Transformations with those from usual C# to a three-stage-extensibility of the model transformation. Results of questionnaires show that the efforts for adopting the model transformation are significantly less than for a general purpose solution.

Thus, in a situation where other MTLs are inappropriate for technical restrictions, NMF Transformations serves to introduce main model transformation abstractions. As a result, the code generator is more concise, more consistent, more understandable and last but not least more extensible than a general purpose solution would be.

12. Validation Summary

The previous three case study were used to validate NMF Transformations against both other model transformation languages and plain general purpose solutions. The validation has been done regarding to the model transformation quality attributes from van Amstel (see chapter 3.4). Table 12.1 lists the validation that has been performed in the previous chapters.

Quality attribute	**Chapter 9**	**Chapter 10**	**Chapter 11**
Understandability	✓	✓	✓
Modifiability	✓[a]	✓[b]	✓[c]
Reusability	–	–	–
Modularity	–	–	–
Completeness	–	–	–
Consistency	✓	✓	✓
Conciseness	✓	✓	✓

[a] Focus on discoverability support and change impact of extension scenario
[b] Focus on debugging and refactoring support
[c] Focus on testing and extensibility

Table 12.1.: Performed validation for NMF Transformations

This validation accompanies the discussion from chapter 8 as all quality attributes that needed further research have been investigated with case studies. The following sections summarize the findings from the case studies regarding to these quality attributes. First, section 12.1 compares these case studies in order to classify their results. Next, section 12.2 summarizes the findings for the understandability of model transformations created with NMF Transformations. Section 12.3 summarizes the findings for modifiability, section 12.4 for consistency and section 12.5 for the the conciseness of the model transformations.

12.1. Comparison of the case studies

The case studies used for the validation of NMF Transformations differ in the background of the developers involved in it. Whereas the attendees of the TTC mostly also attended the International Conference on Model Transformations (ICMT, co-located on the STAF), one can assume that these people have a strong background in model transformation. On the other hand, the attendees at the code generator presentation at ABB

were prospected users of the code generator that did not necessarily have a background in model-driven engineering. Indeed, the evaluation sheet results show that few of them have background knowledge in MDE.

This has various consequences. First and foremost, the evaluation criteria for the responses differ. Where the most important metric for the TTC has been the conciseness and understandability, the tool support is more important in industry. As NMF Transformations can provide this tool support at the cost of a worse conciseness, the acceptance of NMF Transformations in academia (at the TTC) is not as good as in industry. Especially the full test code coverage has been an important argument for the industry. However, neither the test code coverage nor the debugging or refactoring support were remarked in the PN2SC solution paper. As a reason, the test support was not ready for prime-time at the time of the TTC conference. The debugging support was not described as the debugging support of a mainstream language seemed obviously good. This was not perceived by the reviewers. The fact that the debugging support of NMF Transformations was rated so bad can be seen as an evidence of the mismatch of the TTC reviewers and the targeted audience for NMF Transformations as NMF Transformations aims to experienced C# developers.

It has also consequences on the understandability of NMF Transformations. The .NET platform is *very* uncommon among the TTC participants. Thus, valuable space and time during the solution presentation had to be spent to clarify some aspects that have nothing to do with the transformation language itself. On the other hand, concepts such as transformation rules are common to these people where most of them designed a transformation language on their own. The situation at ABB was exactly the other way round. While everybody was able to understand C#-code, common abstractions of model transformation languages were mostly unknown. This affects how the attendees rate the understandability of the model transformations.

Another difference is the state of NMF Transformations during these case studies. While the ABB case study has been conducted almost at the end of this master thesis, the TTC conference was already in late June. The call for solutions ended even before that date. As a consequence, NMF Transformations was not as matured. Some useful contributions of the existing tool support (such as the visualization of transformation rules with Visual Studios Code Map feature) have only been detected long after the TTC conference. Thus, the solution presentation did not make a clear focus on the biggest advantage of NMF Transformations, which is its outstanding tool support. Thus, the results of the TTC are biased to the disadvantage of NMF Transformations.

But also the case studies at the TTC have a large difference between them. Where the tasks of the Flowgraphs case were typical model transformation tasks, the Petri Nets to State Charts case was a rather seldom case. As a consequence, these cases attracted very different model transformation languages. The Flowgraphs case mostly attracted typical external MTLs. Some of them such as ATL and Epsilon have gained quite some popularity. The Petri Nets to State Charts case rather attracted more exotic transformation languages not as popular. This is because the nature of the PN2SC transformation as an input-destructive transformation is rather seldom. While other transformation languages had built-in support for such transformations, we only used the case for a demonstration how to integrate general purpose code into a model transformation with NMF Transformations. As a result, the results for the PN2SC case are not as meaningful as the results from the Flowgraphs case (which is also why the Flowgraphs case is described in more detail).

However, as the TTC cases were validated with opinions from model transformation experts whereas the ABB case study was evaluated by developers at ABB, the results of

the ABB case study are also more important than the results of the TTC Flowgraphs case. As a reason, the responses that were collected in the ABB case study came from industry from developers that are quite close to the intended target audience for NMF Transformations, C# developers with little experience in MDE.

12.2. Understandability

The Flowgraphs case showed that NMF Transformations yields quite understandable model transformations. The understandability (and also conciseness) of NMF Transformations is worse than for external languages, but the difference (probably especially in the understandability) is not too big. In comparison to other internal DSLs like FunnyQT, the model transformations with NMF Transformations are more understandable.

The PN2SC case indicates that the understandability can be further improved by appropriate tool support. Such tool support is available but has only been detected after the TTC conference. However, the abstractions of NMF Transformations may be unsuitable for the PN2SC case, thereby dragging down the understandability of the solution.

The ABB case study indicates that model transformations written with NMF Transformations are more understandable than general purpose solutions. This is due to the lack of proper abstractions for model transformation in mainstream general purpose languages like Java or C# [SK03a]. Furthermore, the Code Map feature of Visual Studio can be used for a visualization of the transformation structure. Such a visualization is shown to be beneficial for maintenance tasks [RNHR13]. However, whereas such tool support for QVT took a lot of effort, a very much similar tool support for NMF Transformations is simply inherited from the tool support offered by Visual Studio for C#.

12.3. Modifiability

Besides the understandability that has also has a great impact on the modifiability of a model transformation, the modifiability of model transformations written with NMF can characterized as follows:

Discoverability

The analysis of the discoverability support of the transformation languages that solved the Flowgraphs case showed that NTL has a very good discoverability support that enhances the learnability of the transformation language. The discoverability support also has the consequence that developers do not necessarily need to know the full language as they can discover it while performing maintenance tasks. The discoverability support is only hampered by the fact that the discoverability does not cover the way how a model transformation is composed of transformation rules.

As general purpose solutions do not have any discoverability restriction, the discoverability support of these solutions is better than for NMF Transformations. However, as these restrictions only appeared in the composition of transformations of transformation rules, the differences can be considered negligible. Thus, basically the full discoverability support of Visual Studio is provided for model transformations with NMF Transformations.

However, this statement is only valid for the aspects of NMF Transformations that have been covered by the Flowgraphs case. This does not include transformation composition or the relational extensions. The discoverability support for these parts of NMF Transformations has to be investigated in further research. However, besides some minor restrictions, it can be prospected that the discoverability support is still very good.

Extensibility

As a result from the analysis of the extension scenario in chapter 9, NTL can be used to write extensible model transformations in the sense that they have a limited change impact for perfective changes. No existing code had to be changed to implement support for additional metaclasses in the input metamodel. However, this proposition holds for the most transformation languages that participated in the Flowgraphs case. Furthermore, the analysis also shows that the fact that NMF Transformations requires an explicit transformation structure also leads to an increased change impact when compared to other solutions.

However, compared to a general purpose solution, the analysis of section 11.6.3 suggests that model transformations with NTL are much more extensible than those written with general purpose code. The code generator created in the ABB case study demonstrates a three-stage-extensibility that is possible to specify with NMF Transformations.

The results from the ABB case study indicate that the efforts for adopting the code generator are significantly reduced compared to a general purpose solution.

Debugging

The analysis of the Petri Net to State Chart case showed that NMF Transformations has a relatively good debugging support. This debugging support is inherited from the C# host language. However, it could be improved by means of model visualization and improved techniques to visualize the model transformation. The Flowgraphs case has not been analyzed for debugging support but the debugging support in these languages is rather poor as most of the transformation languages in the Flowgraphs case are external DSLs that do not inherit any debugging support from their host language (as they do not have any host language).

As the debugging support from NMF Transformations is entirely inherited from Visual Studio for just any C# projects, the debugging support of NTL is not improved compared with the debugging support of a general purpose solution. The general purpose solution also lacks of suitable model visualization techniques. On the other hand, if suitable model visualizations could be used in general purpose code, this support would also be applicable for NMF Transformations.

Thus, a suitable model visualization technology would be a very important contribution.

Testing

The case study from ABB Corporate Research shows that NTL has a good built-in support for unit tests of model transformations. It was possible to achieve a full test code coverage. These tests could be written applying a schematic procedure.

Refactorings

Similar to the debugging support, the refactoring support for NTL is also inherited from its host language C#. However, this tool support is remarkably good and resulted in an award for the best refactoring support at the Petri Nets to State Charts case at the TTC.

On the other side, not all refactorings may be suitable for NMF Transformations. It is a subject of further research to investigate which refactoring operations can be applied to NMF Transformations. However, as there is a plethora of refactoring operations available, this would blow the limitations of this master thesis. But as these refactoring are designed while not having NMF Transformations in mind, one can conclude that refactoring support for NMF Transformations is not as good as the support for general purpose solutions, as it is likely that not all refactoring operations can be used for NMF Transformations.

Overall Modifiability

It would be desirable to combine the results of the disciplines within the modifiability to something like an overall score for modifiability. However, this is not possible as it unclear how these factors influence the overall modifiability. We experienced this problem in section 11.6.1, already. The analysis of this section also suggests that the different aspects of modifiability are perceived individually important. From the previous paragraphs, NMF Transformations has a very strong tool support for a MTL in terms of discoverability, debugging and refactoring support. Furthermore, model transformations with NTL allow a full test code coverage where the transformation code can be covered in a schematic way.

On the other hand, the worse conciseness and understandability in comparison with other MTLs lile Epsilon hamper the modifiability of model transformations written with NTL. In comparison to general purpose languages, however, the situation is exactly reversed. The conciseness and understandability are improved whereas the tool support is hampered. NMF Transformations provides a compromise between other (typical especially external) MTLs and general purpose solutions. However, as most of the tool support for C# is still valid for NTL, it provides much of the strength of MTLs while still preserving the advantages of general purpose solutions.

12.4. Consistency

The case studies showed that NMF Transformations enables transformation developers to specify very consistent model transformations. However, although NTL allows developers to easily integrate arbitrary general purpose code, this integration lowers the solution consistency more than it does for other model transformation languages as for example FunnyQT. As a reason, the difference between the transformation language and the host language is larger for NTLas C# does not allow a syntax as flexible as Clojure. In some cases, external languages like in this case EMF-IncQuery may experience that the language is not capable for all requirements. In such a scenario, NMF Transformations can preserve the consistency much better as it allows to integrate arbitrary general purpose code much easier.

In comparison to general purpose solutions, the ABB case study indicates that the consistency of the model transformations is significantly improved. This is mainly due to the existence of appropriate abstractions like e.g. transformation rules.

Thus, the internal DSL nature of NMF Transformations makes it possible to combine the strength of both general purpose solutions (easy integration of arbitrary code) and external MTLs (their conciseness).

12.5. Conciseness

As could be expected, the conciseness of the NTL solutions to the TTC cases are rather worse in comparison to other MTLs. As a reason, there is a lot of syntactic noise in NTL. Unlike its opponents in the Flowgraphs case, NTL requires the transformation developer to specify the transformation structure. As this can be omitted in other languages, this additional specification lowers the conciseness. However, it is unclear whether omitting the structure of a model transformation is a such a good idea as an explicit structure helps in the understandability of the code in maintenance scenarios. Furthermore, it is unclear whether the conciseness is so important as NTL also inherits great tool support, including code completion.

However, when compared to general purpose solutions, the ABB case study indicates that the conciseness of the solutions is significantly improved. This is also a consequence from the introduction of appropriate high-level abstractions.

The combination of these findings may explain the need for MTLs as they dramatically improve the conciseness of model transformations. Although NTL cannot completely keep up with other MTLs, it does provide appropriate abstractions to improve a model transformations conciseness.

A further aspect of the conciseness discussion is the scale of the transformation. Even the most concise solution of the Flowgraphs case took 300 LOC. Given that this transformation was cut down in functionality to reduce the effort for the solutions and thus attract more participants, many transformation scenarios easily exceed the border where transformations fit to one or two screen sizes. Thus, the transformation is too big so that the transformation developer can only look at parts of it at a time. Instead, it is necessary to navigate through the solutions. But if only parts can be considered at a time, the conciseness gets less important. Instead, the tool support helping the developer to find the necessary parts of the transformation as quick as possible gain importance. This effect is not catched up by the TTC asking mainly for conciseness but not for the tool support that may make the conciseness less important because the model transformations are small enough that this tool support is mostly unimportant.

This is an intrinsic problem of academia. If one conducted a study with model transformations large enough to observe such effects and compare them across multiple transformation languages, the efforts for the participants handing in solutions would be much too large for anybody to participate. Furthermore, analyzing such solutions gets difficult, as it is hard to present the characteristics of a large transformation in a reasonable small amount of time. The Flowgraphs case was already at the borderline, as the space and time limitations did not allow anybody to present their model transformation completely.

13. Conclusion

What remains in this thesis is to summarize what has actually been done. Section 13.1 wraps up the results of this master thesis. Section 13.2 shows up the assumptions that were made to obtain these results and thus their possible threats to validity. Section 13.3 draws future work to be made.

13.1. Results

In this master thesis, NMF Transformations has been presented, a model transformation framework and internal DSL for .NET languages, specifically C#. Its main contribution is to introduce main model transformation abstractions in a mainstream general purpose language. On the other hand, NMF Transformations inherits the matured tool support for C#. This promises to lower the maintenance efforts especially for model transformations maintained by developers less frequently faced with model transformation problems. Furthermore, unlike most other model transformation languages, NMF Transformations operates on plain objects in memory, so no modeling framework is required.

A further result of this thesis is a comprehensive validation of NMF Transformations, particularly against several other MTLs in the TTC case studies and against a general purpose version in the ABB case study. The validation shows that NMF Transformations has outstanding tool support in terms of discoverability, debugging and refactoring. This is because NMF Transformations can make use of the tool support offered by Visual Studio. Unlike other tools, this tool support is also actively maintained. Especially important in industry scenarios, NTL allows for full test code coverage of model transformations where the test code can be retrieved in a schematic procedure. Furthermore, NTL leads to very consistent solutions. On the other hand, model transformations with NTL are not as concise as solutions of other MTLs. This is due to the amount of syntactic noise of NTL in C# but also due to the fact that NTL requires transformation developers to specify the transformation structure explicitly. However, compared tpo general purpose solutions, the validation also indicates that the conciseness of model transformations with NTL is still much better. The understandability of model transformations is improved by NMF Transformations compared to general purpose solutions but also with model transformations written in other internal DSLs.

The abstractions from NMF Transformations are not as appropriate for the input-destructive model transformation of the PN2SC case. However, the case also shows that

NMF Transformations allows an easy integration of general purpose code. Thus, it is possible to take advantage of the abstractions provided by NMF Transformations where they are appropriate and integrate general purpose code for the remaining tasks.

Put in a nutshell, NTL provides a compromise between state-of-the-art MTLs and mainstream languages in that it provides the tool support from these mainstream languages to model transformation where this tool support is outstanding. On the other hand, NTL preserves much of the conciseness and consistency that is connected to most model transformation languages. While the tool support can be inherited almost entirely and the model transformations are very consistent, the conciseness cannot be preserved equally well.

For the overall problem of the acceptance of MDE in an industrial scenario, the master thesis shows a way how tool support can be obtained for model transformation languages by inheriting it from the tool support for the host language. The remarkable result here is that it is possible to design an internal DSL that is nearly as concise as external MTLs but inherits the tool support for general purpose languages.

13.2. Assumptions & Limitations

The validation is based on some assumptions that limit its validity. These limitations are as follows:

- **Relational Extensions:** The relational extensions to NTL have not been subject of validation as the cases could be solved without it. Thus, the solutions did not make use of it to increase the solutions consistency. However, these extensions are meant for more sophisticated patterns that did not appear in the case studies. Further evaluation may show the applicability.
- **Quality attributes:** The validation is based on the model transformation quality attributes defined by van Amstel [vA11]. Basically, the thesis assumes that these quality attributes are meaningful for the perceived quality of a model transformation. Furthermore, these quality attributes were evaluated with a focus on maintainability. Although originally defined for development and maintenance, the focus of the quality attributes may be biased by this focus.
- **Evaluation of transformation language with single cases:** The thesis uses the solutions of the TTC to validate NTL against opponents from the TTC. Although the purpose of the TTC is to compare the transformation tools, the solutions might have not put a focus on the quality attributes from van Amstel. However, some of these quality attributes were also used to evaluate the solutions in the TTC. However, the thesis basically assumes that the solution authors did their best to write as good as possible solutions to the TTC.
- **Evaluating the quality attributes:** These quality attributes are impossible to measure directly and therefore hard to compare. In this thesis, we mostly based the comparison on questionnaires from the TTC, metrics (mainly LOC for the conciseness) and simple discussion. While the results of questionnaires may be biased for several reasons, discussions yield the threat that important arguments are missing. The most important reason for a bias in the questionnaires is the background of the people that fill out these questionnaires. This factor is dampened by the fact that all of the case studies include questionnaires on e.g. the understandability. The risk of missing arguments in the discussion cannot be excluded.

- **Breakdown of modifiability:** For evaluation purposes, the quality attribute modifiability has been broken down into several aspects. However, this breakdown itself is also subject to evaluation.
- **Understandability of the TTC case studies:** The evaluation of the understandability in the TTC case studies is likely biased by the fact that the questionnaires originally only contained a combined assessment for conciseness and understandability. As one does not know how the attendees of the TTC set their priorities regarding understandability *or* conciseness, a validation of the understandability with this data is slightly unreliable.
- **Difficult evaluation questions:** The quality attributes are hard to grasp. For this reason, one usually just evaluates the perception of these quality attributes. However, it is also hard for developers to put this perception in numbers, i.e. fill out a questionnaire. As a result, the data on these perceived quality attributes is not so entirely reliable. However, the data is much better than no data of these perceived quality attributes. However, this assumption limits the expressiveness of the evaluation.
- **Limited amount of questionnaires:** The small amount of responses to the questionnaires both of the TTC and the ABB case study prohibits proper statistical evaluation of this data. As a consequence, the data may only be used to indicate a trend. As a consequence, the results of the evaluation sheets may only be interpreted as showing how it might be. Further research must be taken to consolidate the results of this master thesis. However, it is difficult to conduct a survey on a topic as special as model transformations with a particular model transformation tool such that enough responses can be collected for statistical validation.

As a consequence, the validation within this master thesis may only be used to see trends. It cannot provide proper empirical evidence of the propositions made on the various aspects of model transformation languages.

13.3. Future Work

There are two principal ways of future work, either to improve NMF Transformations or to improve its validation. Improving the validation would be necessary to obtain more reliable results. However, the benefit taken from such an improved validation is unclear. Many of the threats to validity of the validation in this master thesis also apply to further validation, e.g. the fact that the validated quality attributes are mostly hard to grasp and can only be measured by their perception. Thus, the validation would still have to be based on questionnaires. However, it would be hard to gather enough responses to allow for statistical evidence of the propositions made from this validation.

On the other side, the model transformation community has still not reached consensus about MTLs. This is indicated by the fact that most of the languages participating in the TTC 2013 used entirely new model transformation languages. Only ATL and Epsilon were MTLs that exist for a relatively long time (at least five years). The other very young MTLs may fail to gather a reasonable large user base and thus may be dropped in near future.

Thus, the future work is likely to concentrate on the improvements to NMF Transformations as this promises to be more worth the efforts. The possible improvements to NMF Transformations have already been discussed in section 7.5. From this list, the most promising portion of future work is the extension for change propagation and eventually the model synchronization. As a reason, with these aspects, NMF Transformations can

benefit most from the probably biggest difference to other MTLs: While other MTLs are based to transform files into other files, NMF Transformations works to transform in-memory representations into other in-memory representations. Thus, NMF Transformations can abstract from the current idea of model transformations that only run once but rather install a maintaining correspondence link. Such a model synchronization could be used for model servers that maintain multiple models in parallel.

Final Note

The master thesis showed how the problem of immature tool support for model transformations can be avoided by inheriting the existing tool support from general purpose languages. This may help MDE to become accepted in idustry, as NMF Transformations gained also gained acceptance among the attendees of the code generator presentation of ABB.

But even if NMF Transformations does not become accepted as a model transformation language on its own, I think it is still a valuable contribution as it can be used as a target of code generation. The performance of the TTC case solutions was very good (though not discussed in this thesis as performance is not related to maintainability), so it may make sense to generate NTL-code for a model transformation originally specified in e.g. ATL and execute this code instead of using the (potentially slower) ATL engine.

However, the improvements of NMF Transformations look even more promising, especially the model synchronization aspect.

Bibliography

[BBG+06] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow, "Model transformations? transformation models!" in *Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 440–453.

[BBM96] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *Software Engineering, IEEE Transactions on*, vol. 22, no. 10, pp. 751–761, 1996.

[BH11] H. Barringer and K. Havelund, *TraceContract: A Scala DSL for trace analysis*. Springer, 2011.

[BKR09] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *JSS*, vol. 82, pp. 3–22, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2008.03.066

[CA08] K. Cwalina and B. Abrams, *Framework design guidelines: conventions, idioms, and patterns for reusable. net libraries*. Addison-Wesley Professional, 2008.

[CFM10] A. Ciancone, A. Filieri, and R. Mirandola, "Mantra: Towards model transformation testing," in *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*. IEEE, 2010, pp. 97–105.

[CH03] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, no. 3, 2003, pp. 1–17.

[CH06] ——, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.

[Chu36] A. Church, "An unsolvable problem of elementary number theory," *American journal of mathematics*, vol. 58, no. 2, pp. 345–363, 1936.

[Chu40] ——, "A formulation of the simple theory of types," *The journal of symbolic logic*, vol. 5, no. 2, pp. 56–68, 1940.

[CK91] S. Chidamber and C. Kemerer, *Towards a metrics suite for object oriented design*. ACM, 1991, vol. 26, no. 11.

[CK94] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476–493, 1994.

[CM08] J. S. Cuadrado and J. G. Molina, "Approaches for model transformation reuse: Factorization and composition," in *Theory and Practice of Model Transformations*. Springer, 2008, pp. 168–182.

[CM09] ——, "Modularization of model transformations through a phasing mechanism," *Software & Systems Modeling*, vol. 8, no. 3, pp. 325–345, 2009.

[CMT96] G. Canfora, L. Mancini, and M. Tortorella, "A workbench for program comprehension during software maintenance," in *Program Comprehension, 1996, Proceedings., Fourth Workshop on.* IEEE, 1996, pp. 30–39.

[CMT06] J. S. Cuadrado, J. G. Molina, and M. M. Tortosa, "Rubytl: A practical, extensible transformation language," in *Model Driven Architecture–Foundations and Applications.* Springer, 2006, pp. 158–172.

[ERRJ95] G. Erich, H. Richard, J. Ralph, and V. John, "Design patterns: elements of reusable object-oriented software," *Reading: Addison Wesley Publishing Company*, 1995.

[Esh05] R. Eshuis, *Statecharting petri nets.* Beta, Research School for Operations Management and Logistics, 2005.

[EV06] S. Efftinge and M. Völter, "oaw xtext: A framework for textual dsls," in *Workshop on Modeling Symposium at Eclipse Summit*, vol. 32, 2006.

[FB99] M. Fowler and K. Beck, *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

[FMSA13] O. Finot, J.-M. Mottu, G. Sunyé, and C. Attiogbé, "Partial test oracle in model transformation testing," in *Theory and Practice of Model Transformations.* Springer, 2013, pp. 189–204.

[Fow10] M. Fowler, *Domain-specific languages.* Addison-Wesley Professional, 2010.

[FSB04] F. Fleurey, J. Steel, and B. Baudry, "Validation in model-driven engineering: testing model transformations," in *Model, Design and Validation, 2004. Proceedings. 2004 First International Workshop on.* IEEE, 2004, pp. 29–40.

[GDG08] T. Gelhausen, B. Derre, and R. Geiß, "Customizing grgen. net for model transformation," in *Proceedings of the third international workshop on Graph and model transformations.* ACM, 2008, pp. 17–24.

[GGL05] L. Grunske, L. Geiger, and M. Lawley, "A graphical specification of model transformations with triple graph grammars," in *Model Driven Architecture–Foundations and Applications.* Springer, 2005, pp. 284–298.

[GK08] R. Geiß and M. Kroll, "Grgen. net: A fast, expressive, and general purpose graph rewrite tool," in *Applications of Graph Transformations with Industrial Relevance.* Springer, 2008, pp. 568–569.

[GR13] P. V. Gorp and L. Rose, "The petri-nets to statecharts transformation case," in *Sixth Transformation Tool Contest (TTC 2013)*, ser. EPTCS, 2013, to appear.

[Gue12] E. Guerra, "Specification-driven test generation for model transformations," in *Theory and Practice of Model Transformations.* Springer, 2012, pp. 40–55.

[GW08] T. Goldschmidt and G. Wachsmuth, "Refinement transformation support for qvt relational transformations," in *3rd Workshop on Model Driven Software Engineering, MDSE*, 2008.

[GWS12] L. George, A. Wider, and M. Scheidgen, "Type-safe model transformation languages as internal dsls in scala," in *Theory and Practice of Model Transformations.* Springer, 2012, pp. 160–175.

[Hal77] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[HGH13a] G. Hinkel, T. Goldschmidt, and L. Happe, "An NMF Solution for the Flowgraphs case study at the TTC 2013," in *Sixth Transformation Tool Contest (TTC 2013)*, ser. EPTCS, 2013, to appear.

[HGH13b] ——, "An NMF Solution for the Petri Nets to State Charts case study at the TTC 2013," in *Sixth Transformation Tool Contest (TTC 2013)*, ser. EPTCS, 2013, to appear.

[Hin13] G. Hinkel, "An approach to domain-specific model optimization," http://www.codeplex.com/Download?ProjectName=nmf&DownloadId=722482, Karlsruhe Institute of Technology, Tech. Rep., 2013.

[HJSW09] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, *Jamopp: The java model parser and printer*. Techn. Univ., Fakultät Informatik, 2009.

[HLG13] S. Hildebrandt, L. Lambers, and H. Giese, "Complete specification coverage in automatically generated conformance test cases for tgg implementations," in *Theory and Practice of Model Transformations*. Springer, 2013, pp. 174–188.

[Hor13] T. Horn, "The TTC 2013 flowgraphs case," in *Sixth Transformation Tool Contest (TTC 2013)*, ser. EPTCS, 2013, to appear.

[KB03] S. Kounev and A. Buchmann, "Performance modelling of distributed e-business applications using queuing petri nets," in *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*. IEEE, 2003, pp. 143–155.

[KBL13] M. E. Kramer, E. Burger, and M. Langhammer, "View-centric engineering with synchronized heterogeneous models," in *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. ACM, 2013, p. 5.

[KGBH10] L. Kapová, T. Goldschmidt, S. Becker, and J. Henss, "Evaluating maintainability with code metrics for model-to-model transformations," in *Research into Practice–Reality and Gaps*. Springer, 2010, pp. 151–166.

[Kle06] A. Kleppe, "Mcc: A model transformation environment," in *Model Driven Architecture–Foundations and Applications*. Springer, 2006, pp. 173–187.

[Kön05] A. Königs, "Model transformation with triple graph grammars," in *Model Transformations in Practice Satellite Workshop of MODELS*, 2005, p. 166.

[KP09] S. Kelly and R. Pohjonen, "Worst practices for domain-specific modeling," *Software, IEEE*, vol. 26, no. 4, pp. 22–29, 2009.

[KS06] A. Königs and A. Schürr, "Tool integration with triple graph grammars-a survey," *Electronic Notes in Theoretical Computer Science*, vol. 148, no. 1, pp. 113–150, 2006.

[Leh74] M. Lehman, *Programs, Cities, Students: Limits to Growth?* Imperial College of Science and Technology, University of London, 1974.

[LS81] B. P. Lientz and E. B. Swanson, "Problems in application software maintenance," *Communications of the ACM*, vol. 24, no. 11, pp. 763–769, 1981.

[McC76] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

[MEG+03] E. Merks, R. Eliersick, T. Grose, F. Budinsky, and D. Steinberg, "The eclipse modeling framework," *retrieved from, total*, p. 37, 2003.

[MFM+09] P. Mohagheghi, M. A. Fernandez, J. A. Martell, M. Fritzsche, and W. Gilani, "Mde adoption in industry: challenges and success criteria," in *Models in Software Engineering*. Springer, 2009, pp. 54–59.

[MGSF13] P. Mohagheghi, W. Gilani, A. Stefanescu, and M. A. Fernandez, "An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases," *Empirical Software Engineering*, vol. 18, no. 1, pp. 89–116, 2013.

[MLD09] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC unified architecture*. Springer, 2009.

[Obj11] Object Management Group, "Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification," http://www.omg.org/spec/QVT/1.1/PDF/, 2011.

[OH92] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," in *Software Maintenance, 1992. Proceerdings., Conference on*. IEEE, 1992, pp. 337–344.

[OHA92] P. Oman, J. Hagemeister, and D. Ash, "A definition and taxonomy for software maintainability," *Moscow, ID, USA, Tech. Rep*, pp. 91–08, 1992.

[Pic08] C. Picard, "Model transformation with scala," 2008.

[PJ98] J. Palsberg and C. B. Jay, "The essence of the visitor pattern," in *Computer Software and Applications Conference, 1998. COMPSAC'98. Proceedings. The Twenty-Second Annual International*. IEEE, 1998, pp. 9–15.

[RD11] M. P. Robillard and R. Deline, "A field study of api learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[RNHR13] A. Rentschler, Q. Noorshams, L. Happe, and R. Reussner, "Interactive visual analytics for efficient maintenance of model transformations," in *Theory and Practice of Model Transformations*. Springer, 2013, pp. 141–157.

[RRMB08] R. Romeikat, S. Roser, P. Müllender, and B. Bauer, "Translation of qvt relations into qvt operational mappings," in *Theory and Practice of Model Transformations*. Springer, 2008, pp. 137–151.

[S+00] R. Soley *et al.*, "Model driven architecture," *OMG white paper*, vol. 308, p. 308, 2000.

[SBM08] S. Sen, B. Baudry, and J.-M. Mottu, "On combining multi-formalism knowledge to select models for model transformation testing," in *Software Testing, Verification, and Validation, 2008 1st International Conference on*. IEEE, 2008, pp. 328–337.

[Sch95] A. Schürr, "Specification of graph translators with triple graph grammars," in *Graph-Theoretic Concepts in Computer Science*. Springer, 1995, pp. 151–163.

[Sha10] R. Shatnawi, "A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems," *Software Engineering, IEEE Transactions on*, vol. 36, no. 2, pp. 216–225, 2010.

[SK03a] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *Software, IEEE*, vol. 20, no. 5, pp. 42–45, 2003.

[SK03b] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *Software Engineering, IEEE Transactions on*, vol. 29, no. 4, pp. 297–310, 2003.

[Slo08] T. Sloane, "Experiences with domain-specific language embedding in scala," *Domain-Specific Program Development*, 2008.

[SS09] M. Stephan and A. Stevenson, "A comparative look at model transformation languages," *Software Technology Laboratory at Queens University*, 2009.

[Sta06] M. Staron, "Adopting model driven software development in industry–a case study at two companies," in *Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 57–72.

[SV05] T. Stahl and M. Völter, *Modellgetriebene Softwareentwicklung*. dpunkt-Verlag, 2005.

[SWM97] M.-A. Storey, K. Wong, and H. A. Muller, "How do program understanding tools affect how programmers understand programs?" in *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*. IEEE, 1997, pp. 12–21.

[TDH08] N. Tillmann and J. De Halleux, "Pex–white box test generation for. net," in *Tests and Proofs*. Springer, 2008, pp. 134–153.

[TJF+09] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, "On the use of higher-order model transformations," in *Model Driven Architecture-Foundations and Applications*. Springer, 2009, pp. 18–33.

[vA11] M. F. van Amstel, "Assessing and improving the quality of model transformations," Ph.D. dissertation, PhD thesis, Eindhoven University of Technology, 2011.

[vAvdB10] M. van Amstel and M. van den Brand, "Quality assessment of atl model transformations using metrics," in *Second International Workshop on Model Transformation with ATL*, 2010.

[vAVDB11a] M. F. van Amstel and M. G. Van Den Brand, "Model transformation analysis: staying ahead of the maintenance nightmare," in *Theory and Practice of Model Transformations*. Springer, 2011, pp. 108–122.

[vAvdB11b] M. van Amstel and M. van den Brand, "Using metrics for assessing the quality of atl model transformations," in *Proceedings of the Third International Workshop on Model Transformation with ATL (MtATL 2011)*, vol. 742, 2011, pp. 20–34.

[VGM11] P. Van Gorp and S. Mazanek, "Share: a web portal for creating and sharing executable research papers," *Procedia Computer Science*, vol. 4, pp. 589–597, 2011.

[Wag08] D. Wagelaar, "Composition techniques for rule-based model transformation languages," in *Theory and Practice of Model Transformations*. Springer, 2008, pp. 152–167.

[WKC08] J. Wang, S.-K. Kim, and D. Carrington, "Automatic generation of test models for model transformations," in *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*. IEEE, 2008, pp. 432–440.

[WKK+12] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger, "Fact or fiction–reuse in rule-based model-to-model transformation languages," in *Theory and Practice of Model Transformations*. Springer, 2012, pp. 280–295.

[WS13] M. Wieber and A. Schürr, "Systematic testing of graph transformations: A practical approach based on graph patterns," in *Theory and Practice of Model Transformations*. Springer, 2013, pp. 205–220.

[WVDSD10] D. Wagelaar, R. Van Der Straeten, and D. Deridder, "Module superimposition: a composition technique for rule-based model transformation languages," *Software & Systems Modeling*, vol. 9, no. 3, pp. 285–309, 2010.

[yWL12] B. T. y Widemann and M. Lepper, "Paisley: pattern matching à la carte," in *Theory and Practice of Model Transformations*. Springer, 2012, pp. 240–247.

[ZÏ3] A. Zündorf, "Story Driven Modeling Libary (SDMLib): an Inline DSL for modeling and model transformations, the Petrinet - Statechart case," in *Sixth Transformation Tool Contest (TTC 2013)*, ser. EPTCS, 2013, to appear.

A. NMF

This chapter briefly introduces NMF, an open-source project to provide support for model-driven techniques on the .NET platform. The abbreviation NMFstands for .NET Modeling Framework. It is an open-source framework initiated in July 2012 and now hosted on Codeplex at `http://nmf.codeplex.com/`. The following sections explain the purpose of the projects that NMFconsists of.

- Transformations for M2M transformations presented in section A.1
- Optimizations for domain-specific optimizations presented in section A.2 or [Hin13]
- EcoreInterop for EMF interopability presented in section A.3
- Serializations for XMI serialization presented in section A.4
- AnyText, a parser based on language descriptions similar to xText. However, this project retired.
- Several others projects, including projects for collections and utilities

A.1. Transformations

The Transformations project exists since July 2012. However, it has been subject to various extensions, improvements, refactorings during this master thesis. Furthermore, test code has been written as an important contribution to this project that now has a test code coverage of 80% in terms of covered blocks. It is the project that this thesis is based on. It will be described in great detail in chapter 7.

A.2. Optimizations

The Optimizations project is a framework to support domain-specific model optimization. It has been developed as contribution of this master thesis, however, it is only described in a separate technical report [Hin13] due to space limitations.

A.3. EcoreInterop

The EcoreInterop is a project to provide interopability support with EMF, especially the Ecore meta-metamodel. In this way, it provides a code generator that can create

classes representing the metaclasses of an Ecore-metamodel. These classes are further decorated with attributes that enable the Serializations project to load and save these models from and to XMI. This interopability was needed to participate in the TTC2013, as the resulting models were in the Ecore XMI serialization format. The code generation for Ecore packages is an important use case of NMF Transformations, as it is a model transformation that targets the metamodel defined in the `System.CodeDOM` namespace, much like the ABB case study from chapter 11.

A.4. Serializations

The Serializations project is a framework to easily serialize objects from and to XML. Unlike the XML-serializer that is contained in the .NET framework, the XML-serializer from the Serializations project is capable of serializing models with cyclic references. However, it is still possible to influence the format of the resulting XML files. Furthermore, the XML document is not kept in memory, but the model is created on-the-fly.

Compared to the EMF builtin serializer, this serializer has some drawbacks, as it is not capable to load models split among multiple files. Neither is it possible to serialize a model to multiple files.

B. Implementation details on NMF Transformations

In this chapter, some details on the implementation of NMF Transformations are presented.

B.1. Test coverage

The whole NMF project currently consists of 1056 test cases. Both NMF Transformations Core and NTL are covered with 722 unit tests and integration tests. Further 43 test cases test the utilities library NMF Utilities that is also massively used by NMF Transformations. The other test cases test the NMF Collections and NMF Optimizations project.

NMF Transformations Core has a test code coverage of 90.22% in terms of covered blocks. NTL has a code coverage of 68.70% in terms of covered blocks. In terms of the LOC metric computed by Visual Studio, the tests (5.214 lines) only top the implementation by roughly a third (3.251 lines for NTL plus 666 lines for NMF Transformations Core), as tests can be shared across different implementations of an interface.

B.2. More close architecture diagram of NMF Transformations

The class diagram in figure B.1 shows a more complete overview of the implementation of NMF Transformations than the diagram presented in section 7.2. However, there are still some classes missing, including the whole Relational extensions.

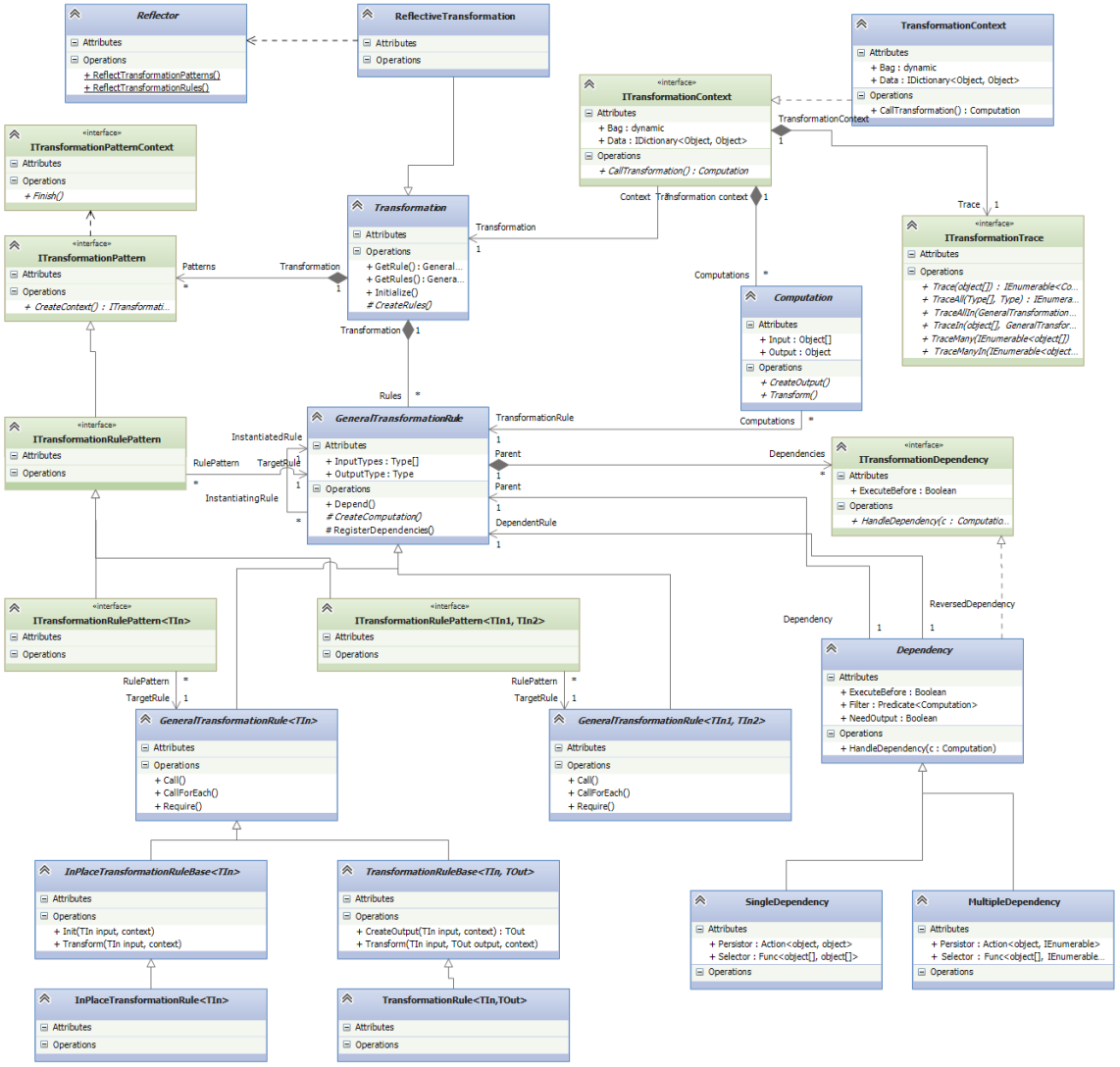

Figure B.1.: A more complete class diagram

C. Implementations of the example transformation problems using NMF

In this section, implementations using NMFare provided for the example transformation problems.

C.1. Finite State Machines to Petri Nets

A complete implementation (though without inline code documentation) is presented in listing C.1 below.

```
1  using System;
2  using System.Linq;
3  using NMF.Transformations;
4  using NMF.Transformations.Simples;
5
6  namespace NMF.Transformations.Example
7  {
8   public class FSM2PN : ReflectiveTransformation
9   {
10  public class AutomataToNet : SimpleTransformationRule<FSM.
        FiniteStateMachine, PN.PetriNet>
11  {
12   public override void Transform(FSM.FiniteStateMachine input, PN
        .PetriNet output, ITransformationContext context)
13   {
14    output.ID = input.ID;
15   }
16  }
17
18  public class StateToPlace : SimpleTransformationRule<FSM.State,
       PN.Place>
19  {
20   public override void Transform(FSM.State input, PN.Place output
        , ITransformationContext context)
21   {
```

```
22      output.ID = input.Name;
23      if (input.IsStartState)
24      {
25       output.TokenCount = 1;
26      }
27      else
28      {
29       output.TokenCount = 0;
30      }
31     }
32
33     public override void RegisterRequirements()
34     {
35      CallForEach<FSM.FiniteStateMachine, PN.PetriNet>
36       (fsm => fsm.States,
37       (pn, places) => pn.Places.AddRange(places));
38     }
39    }
40
41    public class TransitionToTransition : SimpleTransformationRule<
        FSM.Transition, PN.Transition>
42    {
43     public override void Transform(FSM.Transition input, PN.
          Transition output, ITransformationContext context)
44     {
45      output.Input = input.Input;
46     }
47
48     public override void RegisterRequirements()
49     {
50      CallForEach<FSM.FiniteStateMachine, PN.PetriNet>
51       (fsm => fsm.Transitions,
52       (pn, transitions) => pn.Transitions.AddRange(transitions));
53
54      Require(Rule<StateToPlace>(),
55       t => t.StartState, (t, p) =>
56       {
57        t.From.Add(p);
58        p.Outgoing.Add(t);
59       });
60
61      Require(Rule<StateToPlace>(),
62       t => t.EndState, (t, p) =>
63       {
64        t.To.Add(p);
65        p.Incoming.Add(t);
66       });
67     }
68    }
69
70    public class EndStateToTransition : SimpleTransformationRule<FSM
        .State, PN.Transition>
```

```
71   {
72     public override void Transform(FSM.State input, PN.Transition
           output, ITransformationContext context)
73     {
74       var from = context.Trace.TraceTransformationOutput(Rule<
             StateToPlace>(), input);
75       output.From.Add(from);
76       from.Outgoing.Add(output);
77       output.Input = "";
78     }
79
80     public override void RegisterRequirements()
81     {
82       CallForEach<FSM.FiniteStateMachine, PN.PetriNet>
83         (fsm => fsm.States.Where(s => s.IsEndState),
84         (pn, endTransitions) => pn.Transitions.AddRange(
             endTransitions));
85     }
86   }
87   }
88 }
```

Listing C.1: An example implementation to transform finite state machines to Petri Nets

C.2. Persons to Family Relations

```
1  using System;
2  using System.Linq;
3
4  using NMF.Transformations;
5  using NMF.Transformations.Simples;
6  using NMF.Utilities;
7
8  using Ps = NMF.Transformations.Example.Persons;
9  using Fam = NMF.Transformations.Example.FamilyRelations;
10
11 namespace Transformations.Sample.Persons2Families
12 {
13   class Persons2FamilyRelations : ReflectiveTransformation
14   {
15     public class Root2Root : SimpleTransformationRule<Ps.Root, Fam.
         Root> { }
16
17     public class Person2Female : SimpleTransformationRule<Ps.Person
         , Fam.Female>
18     {
19       public override void Transform(Ps.Person input, Fam.Female
           output, ITransformationContext context)
20       {
21         output.Husband = context.Trace.TraceTransformationOutput(
             Rule<Person2Male>(), input.Spouse);
22
```

```
23      foreach (var child in context.Trace.
            TraceAllTransformationOutputs(Rule<Person2Person>(),
            input.Children))
24      {
25       child.Mother = output;
26      }
27    }
28
29    public override void RegisterRequirements()
30    {
31      MarkInstantiatingFor(Rule<Person2Person>(), p => p.Gender ==
            Ps.Gender.Female);
32    }
33  }
34
35  public class Person2Male : SimpleTransformationRule<Ps.Person,
        Fam.Male>
36  {
37    public override void Transform(Ps.Person input, Fam.Male
          output, ITransformationContext context)
38    {
39      output.Wife = context.Trace.TraceTransformationOutput(Rule<
            Person2Female>(), input.Spouse);
40
41      foreach (var child in context.Trace.
            TraceAllTransformationOutputs(Rule<Person2Person>(),
            input.Children))
42      {
43       child.Father = output;
44      }
45    }
46
47    public override void RegisterRequirements()
48    {
49      MarkInstantiatingFor(Rule<Person2Person>(), p => p.Gender ==
            Ps.Gender.Male);
50    }
51  }
52
53  public class Person2Person : AbstractTransformationRule<Ps.
        Person, Fam.Person>
54  {
55    public override void Transform(Ps.Person input, Fam.Person
          output, ITransformationContext context)
56    {
57      output.LastName = input.Name;
58      output.FirstName = input.FirstName;
59
60      var daughters = context.Trace.TraceAllTransformationOutputs(
            Rule<Person2Female>(), input.Children.Where(child =>
            child.Gender == Ps.Gender.Female));
```

```
61          var sons = context.Trace.TraceAllTransformationOutputs(Rule<
                Person2Male>(), input.Children.Where(child => child.
                Gender == Ps.Gender.Male));
62
63          foreach (var daughter in daughters)
64          {
65           daughter.Sisters.AddRange(daughters.Except(daughter));
66           daughter.Brothers.AddRange(sons);
67          }
68
69          foreach (var son in sons)
70          {
71           son.Sisters.AddRange(daughters);
72           son.Brothers.AddRange(sons.Except(son));
73          }
74      }
75
76      public override void RegisterRequirements()
77      {
78          CallForEach<Ps.Root, Fam.Root>(
79           root => root.Persons,
80           (root, people) => root.People.AddRange(people));
81      }
82    }
83  }
84 }
```

Listing C.2: An example implementation to transform people

D. Feedback from the TTC 2013

In this section, the plain feedback from the TTC 2013 is reflected.

D.1. Remarks to NMF Transformations

The first thing I want to express is my thanks for all the people I had very interesting discussions with at the STAF 2013. As sadly, the TTC did not provide much time for questions, the remarks were often made in the breaks. However, some remarks indeed changed my way of recognition, but I cannot remember them. Thus, the following presents an incomplete list:

- The Turing completeness requirement that much of this thesis is based on, is debatable. As Sendall and Kozaczynski just presumed this requirement, I did not question it too much. But indeed, it is hard to tell, whether this is a necessary requirement. In the meantime I have come to the opinion that it very much depends on how one defines model transformations. The way I do, basically to also cover transformations like the Petri Net to State Charts case from section 10, I am pretty sure it is a valid requirement.
- People asked me, whether I would like to port NMF Transformations to Scala. While I take this as a compliment for NMF Transformations being worth ported, I will rather concentrate on the extensions roughly described in section 7.5.
- A good pointer was to try NTL with other .NET languages than C#. Indeed, this is also recommended by [CA08]. To me it is pretty clear that a framework designed to be used with C# also works fine with Visual Basic. However, as I am writing these lines, I tried a bit with F#, but although it actually worked, it did not work nicely.

D.2. Evaluation data for the Flowgraphs case

Table D.1 shows the plain evaluation data from the open peer review stage for the Flowgraphs case. However, this table is still incomplete, as the review form included several more fields. The full review data can be accessed under the following url: `http://goo.gl/jAo4T`. Furthermore, the review stage was before the presentation of the solutions at the TTC.

The average scores are printed to the charts in figure D.1. The detailed rankings for the tasks $1-4$ are shown in the tables D.2, D.3, D.4, D.5 and D.6.

Table D.1.: Results of the open peer reviews for the Flowgraphs case

Solution Name	Reviewer name	Overall evaluation	Overall evaluation confidence	Paper readability	Faithful solution description	Usefulness of transformation language	Ease to use transformation language	Usefulness of transformation tool	Ease to use transformation tool
ATL	Anthony Anjorin	4	3	3		5	3	5	3
ATL	Jesùs Sànchez Cuadrado	4	4	3	3	4	4	4	4
ATL	Tassilo Horn	3	4	4	4	5	4	5	4
Eclectic	Georg Hinkel	4	4	5	4	4	4	3	4
Eclectic	Tassilo Horn	2	4	3	3	3	3	3	3
eMoflon	Fabian Büttner	3	3	3	4	3	3	3	3
eMoflon	R Wei and Dimitris Kolovos	3	4	3	4	3	2	3	2
eMoflon	Tassilo Horn	1	3	3	4	3	2	3	2
Epsilon	Georg Hinkel	4	4	4	4	5	3	3	3
Epsilon	Tassilo Horn	4	4	4	4	5	5	5	5
FunnyQT	Fabian Büttner	4	4	4	4	4	4	4	4
FunnyQT	Jesùs Sànchez Cuadrado	4	4	4	5	4	3	3	3
NMF	Anthony Anjorin	4	3	4	3	3	2	3	2
NMF	Babajide Ogunyomi and Louis Rose	3	3	4	3	3	2	3	2
NMF	Tassilo Horn	3	4	4	4	4	3	4	3

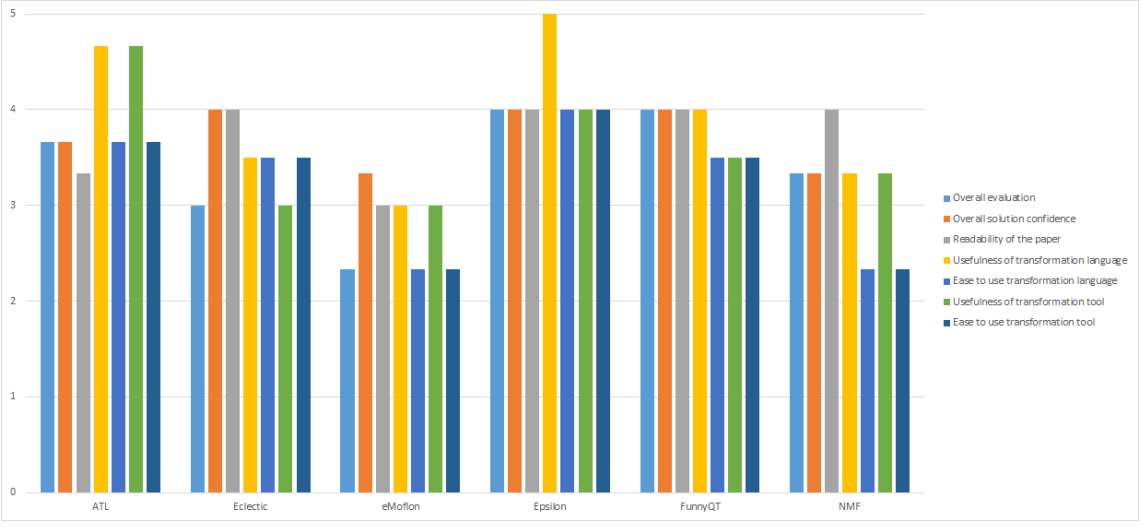

Figure D.1.: Open peer review results of the Flowgraph case

Table D.2.: Results of the open peer reviews for task 1

Solution	Completeness & Correctness	Conciseness & Understandability	Efficiency
ATL	5.0	4.3	3.5
Eclectic	4.5	4.5	1.0
eMoflon	3.0	2.5	2.5
Epsilon	5.0	4.5	1.0
FunnyQT	5.0	4.5	5.0
NMF	5.0	2.7	5.0
Grand Total	4.58	3.83	3.00

Table D.3.: Results of the open peer reviews for task 2

Solution	Completeness & Correctness	Conciseness & Understandability	Efficiency
ATL	5.0	2.7	3.0
Eclectic	4.0	4.0	1.0
eMoflon	2.5	2.5	2.5
Epsilon	4.0	4.0	1.0
FunnyQT	5.0	4.5	5.0
NMF	5.0	2.7	4.0
Grand Total	4.25	3.39	2.75

Table D.4.: Results of the open peer reviews for task 3.1

Solution	Completeness & Correctness	Conciseness & Understandability	Efficiency
ATL	5.0	3.7	3.5
Eclectic	4.0	4.0	1.0
eMoflon	1.0	1.0	1.0
Epsilon	5.0	4.5	1.0
FunnyQT	5.0	4.0	5.0
NMF	5.0	2.7	3.0
Grand Total	4.17	3.31	2.42

Table D.5.: Results of the open peer reviews for task 3.2

Solution	Completeness & Correctness	Conciseness & Understandability	Efficiency
ATL	5.0	3.0	2.5
Eclectic	4.5	3.5	1.0
eMoflon	1.0	1.0	1.0
Epsilon	4.5	4.0	1.0
FunnyQT	5.0	4.5	5.0
NMF	3.5	3.0	3.0
Grand Total	3.92	3.17	2.25

Table D.6.: Results of the open peer reviews for task 4

Solution	Completeness & Correctness	Conciseness & Understandability	Efficiency
ATL	5.0	3.7	3.5
Eclectic	4.0	3.0	3.0
eMoflon	1.0	1.0	1.0
Epsilon	5.0	4.0	3.0
FunnyQT	5.0	4.0	5.0
NMF	4.0	3.0	3.0
Grand Total	4.00	3.11	3.08

The results from the TTC conference are printed in table D.7. The data is also presented as a graph in figure D.2. The results are based on 49 responses to the questionnaire at the TTC conference[1].

Table D.7.: The results from the TTC conference for the Flowgraphs case

Solution name	Overall evaluation of solution	Overall evaluation of presentation	Familiarity with technologies	Understandability and Conciseness	Ease to use transformation language	Usefulness of transformation language	Ease to use transformation tool	Usefulness of transformation tool
ATL	3.00	3.57	3.86	2.86	3.57	3.43	3.71	3.71
Eclectic	3.71	3.86	3.14	3.71	3.14	3.71	3.57	3.29
eMOFLON	2.63	3.88	3.50	2.75	2.75	3.38	3.00	3.13
Epsilon	4.00	4.10	3.50	3.70	3.70	4.10	3.90	4.10
FunnyQT	3.63	3.13	2.88	2.63	2.50	3.63	2.50	3.38
NMF	3.33	2.89	2.89	2.89	2.78	3.56	2.89	3.00
Average	3.41	3.57	3.29	3.10	3.08	3.65	3.27	3.45

D.3. Evaluation data for the Petri Nets case

Table D.8 shows the plain evaluation data from the open peer review stage for the Flowgraphs case. However, this table is still incomplete, as the review form included several more fields. The full review data can be accessed under the following url: `http://goo.gl/5Wihy`. Furthermore, the review stage was before the presentation of the solutions at the TTC.

The average scores for each solution are printed in figure D.3. Furthermore, table D.9 shows the results of the open peer reviews in terms of the tool capabilities. However, only few solution papers put notes on these capabilities, as most of the tools do not support these capabilities. Thus, the evaluation of the tool capabilities is in many cases based on guessing of the reviewers that hardly know the entire transformation tool. The tool capabilities were clarified during the TTC itself.

Furthermore, table D.10 shows the performance results for some of the performance models. These performance figures show the results of the initial solutions. In case of NMF, major performance improvements were accomplished after the TTC by switching the collection implementation to unordered hash sets. Thus, the table D.10 also includes a row for the improved solution of NMF. The TTC version used ordered sets as the input models were given in a format that used index-based referencing scheme due to a performance bug in EMF. NMF Serializations then expects the model to have an index-based access.

[1] `http://goo.gl/tu9jY`

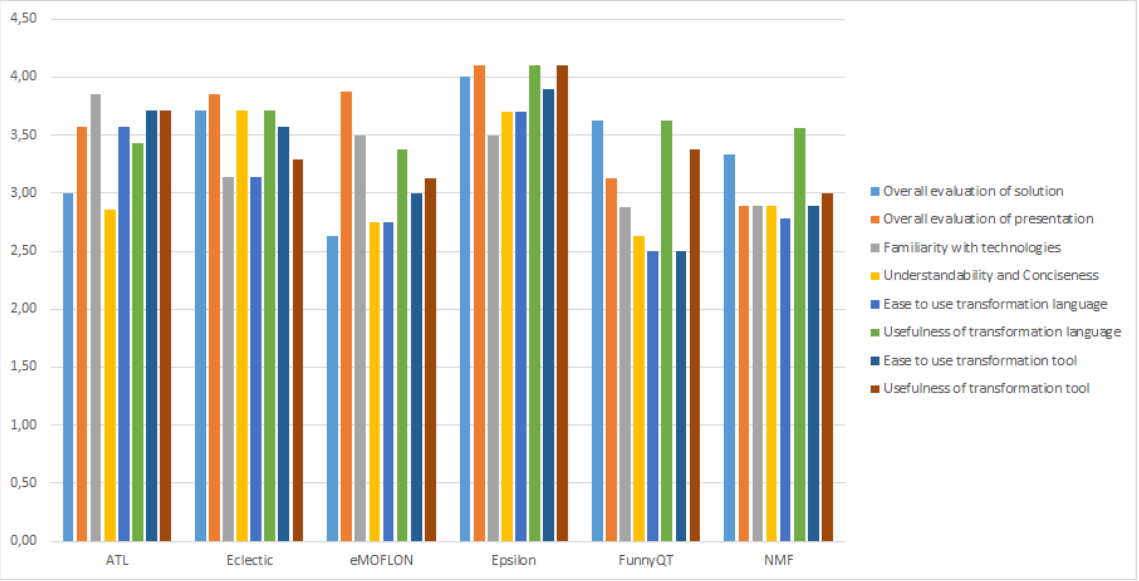

Figure D.2.: The results from the TTC conference for the Flowgraphs case

Table D.8.: Results of the open peer reviews for the Petri Nets to State Charts case

Solution Name	Reviewer name	Overall evaluation	Overall evaluation confidence	Paper readability	Faithful solution description	Usefulness of transformation language	Ease to use transformation language	Usefulness of transformation tool	Ease to use transformation tool
AToMPM	Albert Zündorf	4	5	4	4	4	4	4	4
FunnyQT	Albert Zündorf	5	5	4	4	4	2	4	2
FunnyQT	Benedek Izsò	5	4	5	5	4	4	5	5
FunnyQT	Georg Hinkel	5	4	4	5	5	3	4	4
IncQuery	Albert Zündorf	4	5	3	3	3	3	3	3
IncQuery	Tassilo Horn	2	4	4	4	4	2	4	2
NMF	Albert Zündorf	4	5	3	3	3	3	3	3
NMF	Kevin Lano	4	3	3	4	3	3	3	3
NMF	Tassilo Horn	3	4	4	4	4	4	4	4
SDMlib	Benedek Izsò	2	4	4	4	2	1	1	1
SDMlib	Georg Hinkel	4	3	2	3	4	3	3	5
SDMlib	Tassilo Horn	3	4	2	3	3	2	3	2
UML-RSDS	Albert Zündorf	4	5	3	4	4	3	4	3
UML-RSDS	Tassilo Horn	3	4	4	3	4	4	2	1

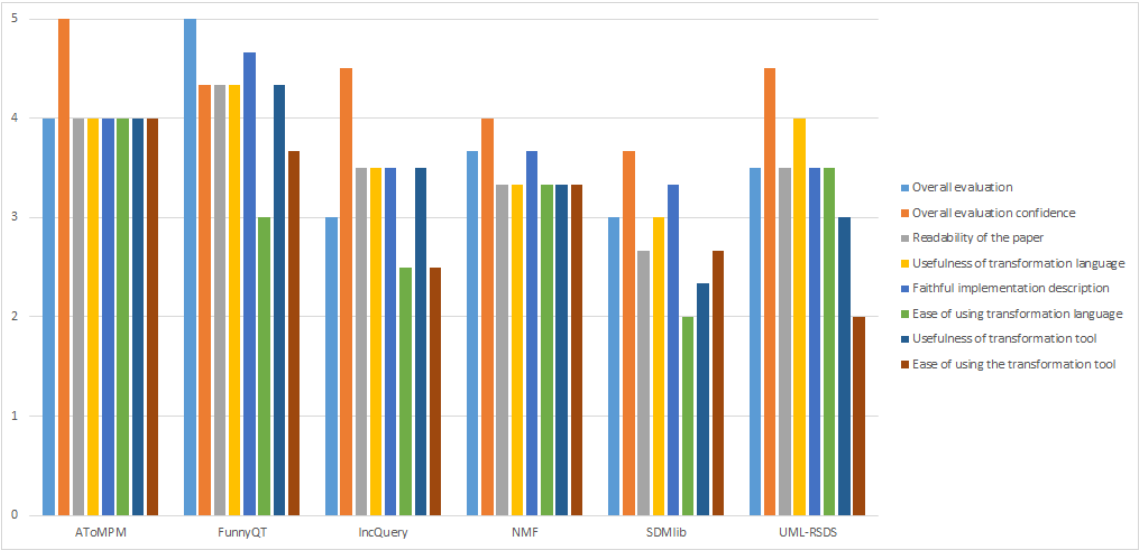

Figure D.3.: The results from the open peer reviews for the Petri Net to State Charts case

Table D.9.: Results of the open peer reviews in terms of tool capabilities

Solution Name	Reversability	Change Propagation	Refactoring Support	Debugging Support	Simulation Support
AToMPM	3.0	3.0	3.0	3.0	3.0
FunnyQT	1.7	1.7	2.3	2.3	1.5
IncQuery		4.0		3.0	
NMF	1.0	2.0	2.3	2.7	1.5
SDMlib	1.0	1.5	2.5	3.5	1.5
UML-RSDS	5.0	2.0	3.0	3.0	1.0
Grand Total	2.33	2.36	2.63	2.92	1.70

After the TTC, the initial model versions were used instead that had an id-based referencing scheme in the XMI files and thus could be supported by model collections that do not provide an index-based access. However, the id-based referencing scheme still has a worse performance for deserialization. From the $521s$ to transform the largest test case, $488.7s$ were used to load the model and further $18.3s$ were used to save the target model back to a file. Thus, the transformation only took as less as $13.5s$.

Table D.10.: The performance results from the initial solutions

Solution Name	sp200	sp1000	sp40000	sp200000
Epsilon	2.020s	19.019s		
FunnyQT	138ms	212ms	12.010s	114s
GrGen.NET	383ms	597ms	76.000s	2,498s
IncQuery	907ms	3.637s		
NMF	119ms	184ms	96.182s	4,584s
NMF (updated)	30ms	169ms	27.212s	521s
SDMlib	113ms	140ms	18.562s	1,440s
UML-RSDS	60ms	360ms		

The results from the TTC conference are shown in table D.11. The data is also presented in figure D.4 in graphical form. The results are based on 36 responses to the questionnaire from the TTC conference[2].

Table D.11.: The results from the TTC conference

Solution name	**Overall evaluation of the solution**	**Overall evaluation of the presentation**	**Familiarity with the technologies**	**Understandability and Conciseness**	**Usefulness of transformation language**	**Ease to use transformation language**	**Usefulness of transformation tool**	**Ease to use transformation tool**
EMF-IncQuery	3.78	4.00	3.44	3.67	3.89	3.44	3.78	3.78
FunnyQT	4.00	3.71	3.00	3.00	3.43	2.43	3.29	2.57
NMF	3.14	2.86	2.86	2.71	3.29	2.86	3.00	3.14
SDMlib	3.57	3.43	3.57	3.14	3.71	2.71	3.43	2.71
UML-RSDS	2.17	1.83	2.67	2.50	2.50	2.00	1.83	1.33
Average	3.39	3.25	3.14	3.06	3.42	2.75	3.14	2.81

The results for the PN2SC solutions tool support in terms of debugging and refactoring support are presented in figure D.5.

[2] `http://goo.gl/yU3as`

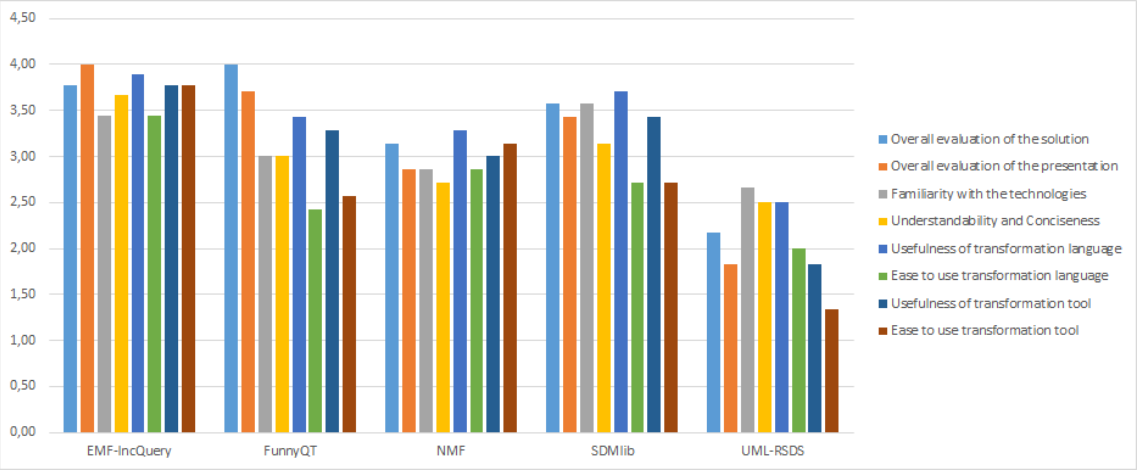

Figure D.4.: The results of the TTC conference for the Petri Nets to State Charts case

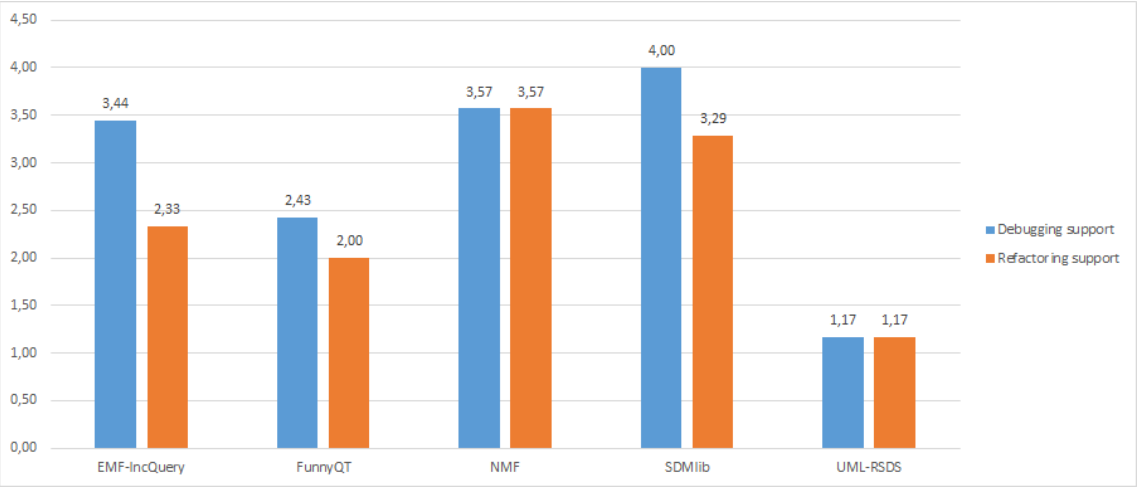

Figure D.5.: The results of the TTC in terms of tool support for the PN2SC case

E. Evaluation data from the ABB case study

The evaluation sheet is shown in figure E.1.

Evaluation sheet for the OPC UA Code generator presentation

Please answer the following questions. The questionnaire is evaluated anonymously and published only through aggregates.

1. How much experience do you have with model-driven software development?

☐ None ☐ Heard of it ☐ Tried ☐ Used on regular basis

2. Which transformation languages were you using before?

☐ QVT Operational ☐ QVT Relational ☐ ATL ☐ General Purpose ☐ T4 ☐ Others

3. How often do you face model transformation problems?

☐ Daily ☐ At least once per week ☐ At least once per month ☐ At least once per year ☐ Less than once per year ☐ Not yet

4. How understandable was the presentation of the code generator?

☐ Very good ☐ Good ☐ Rather good ☐ Rather poor ☐ Poor ☐ Very Poor

5. How understandable was the presentation of NMF Transformations?

☐ Very good ☐ Good ☐ Rather good ☐ Rather poor ☐ Poor ☐ Very Poor

6. Are you planning to use the code generator?

☐ Most likely ☐ Likely ☐ Maybe ☐ Rather not ☐ Definitely not ☐ Not applicable

7. If so, please rate the effort to adopt the code generator for your scenario!

☐ No adoption required ☐ Minor adoptions required ☐ Major adoptions required ☐ Rather new implementation ☐ Not applicable

8. Please estimate the efforts for the adoptions from the previous question in relation to general purpose solution!

☐ Significantly less efforts ☐ Slightly less efforts ☐ Slightly more efforts ☐ Significantly more efforts ☐ Not applicable

9. How does NMF Transformations affect the following quality attributes compared with a general purpose solution?

Understandability	improved	○○○○	degraded
Modifiability	improved	○○○○	degraded
Conciseness	improved	○○○○	degraded
Consistency	improved	○○○○	degraded
Reusability	improved	○○○○	degraded

10. Please feel free to add any comment on the solution or the presentation.

Thank you!

Figure E.1.: The evaluation sheet for the ABB case study

F. Metrics for transformations written in NMF Transformations

This chapter discusses how existing maintance metrics apply for transformations written in NMF Transformations. At first, several metrics measuring the maintainability of an object-oriented design are reviewed in section F.1. Afterwards, we will review some metrics originally created for model transformation languages like ATL or QVT in section F.2.

F.1. Metrics for object-oriented design

NMF Transformations is an internal DSL for programming languages that follows the e.g. object-oriented programming paradigm (leaving aside that C# and Visual Basic nowadays support multiple programming paradigms). This makes model transformations written in NMF Transformations candidates for metrics specified for object-oriented code. Some of these metrics will be evaluated in this section for their usage when writing model transformations with NTL. Especially, the `Transform` method of the transformation rules is a target for traditional object-oriented code metrics, as implementations of this method use the benefits arising from the alternative computational model of NMF Transformations, but essentially, they are represented by traditional imperative object-oriented code and thus candidates for code metrics designed for this paradigm.

There are a lot of metrics available for object-oriented imperative code. Some metrics focus on the imperative nature of method implementations and concentrate on concepts from procedural programming, while others focus on the rather specific properties of object-oriented design, such as inheritance. Many of these object-oriented metrics have been originally defined or evaluated for object-oriented progarmming in [CK91, CK94] and many of them have gained popularity among developers. In this section, we will talk about the applicability of these metrics for model transformations using NMF Transformations. We will concentrate on the metrics that are implemented within Visual Studio[1] for three reasons: The first reason is that obviously there is a tooling available that measures them. As second reason, these metrics have been implemented within Visual Studio mainly because of their wide acceptance and popularity among developers. Finally, any developer that will use NMF Transformations to write model transformations will usually use Visual Studio and thus eventually stumble on these metrics. The results of these metrics

[1] `http://msdn.microsoft.com/en-us/library/bb385914.aspx`

can be combined with the static code analysis with several code analysis rules. Combined with the continuous integration abilities of Visual Studio and the Team Foundation Server, static code analysis can be performed each time a developer checks some code files, possibly rejecting the check-in because of poor code metric results. Thus, it is important to review the validity of these results when creating model transformations.

The impacts of these metrics on the `RegisterDependencies` operation and `Transform` method for the solutions of the example transformations from chapter 4 (the complete solutions can be found in the appendix section C) are summarized in the tables F.1 and F.2.

Metric	Min	Max	Mean	Standard error
Maintainability Index	58	94	80.54	14.00
Cyclomatic Complexity	1	13	3.92	3.48
Class Coupling	2	19	6.83	5.27
Lines of Code	1	10	3.30	2.73

Table F.1.: The code metrics measured for the `RegisterDependencies` method

Metric	Min	Max	Mean	Standard error
Maintainability Index	57	95	86.34	10.96
Cyclomatic Complexity	1	2	1.17	0.38
Class Coupling	3	18	5.55	3.32
Lines of Code	1	13	2.31	2.83

Table F.2.: The code metrics measured for the `Transform` method

The following sections will analyze the impacts of NTL to these metrics in depth.

F.1.1. Depth of Inheritance

Originally proposed in [CK91], the Depth of Inheritance (DIT) of a class measures the longest path from a class to its ancestor. This metric is extended to a DIT of a whole assembly or project by taking the maximum DIT of the classes contained in that project or assembly. It has been shown that the DIT correlates with the probability of fault detection [BBM96, SK03b].

The problem with finding a proper threshold for the DIT is that the DIT exceeds a threshold of 5 or 6 only in rare occasions. Thus, even evalutaing the data of Eclipse 2.0, Shatnawi was unable to provide a threshold for the acceptable risk level for the DIT metric, as this metric simply lacks of variance [Sha10]. However, several articles apart from academia suggest a threshold of 5 to 6. Indeed, Microsoft released a code analysis rule CA1501[2] warning developers to "Avoid excessive use of inheritance" when the DIT reaches a value of 6.

The problem with the DIT in the context of NMF Transformations is that the class `SimpleTransformationRule` already has a DIT of 4. This means that a transformation rule specified using this class as base class already has a DIT of 5, which is the maximum allowed DIT without getting the CA1501 validation exception. However, the class `AbstractTransformationRule` that simply represents that a transformation rule must be instantiated and must not create an output inherits from `SimpleTransformationRule` so that any transformation rule using this class (which is intended for abstract transformation

[2] `http://msdn.microsoft.com/en-us/library/ms182213.aspx`

rules) automatically has a DIT of 6, so that Visual Studio will suggest the transformation developer to "Avoid excessive use of inheritance".

Indeed, the NMF Transformations-solution to the Flowgraphs case that will be described in 9 includes several transformation rules that expose a DIT of 6. All of these classes represent transformation rules that act as hubs, i.e. the bodies of these classes is empty and they only consist of the compiler-generated constructor. As the transformation involved in the Petri Nets to State Charts case does not employ instantiation, but instead uses transformation rules that inherit from `SimpleTransformationRule`, it does not contain a class with a DIT of 6.

Hence, the DIT is not applicable to model transformations written in NMF Transformations, as using the intended way to specify an abstract transformation rule already dominates the DIT for the whole project. This even has the consequence that the according analysis rule CA1501 is better turned off for projects incorporating NMF Transformations.

F.1.2. Cyclomatic Complexity

The cyclomatic complexity originally proposed by McCabe in 1976 [McC76] for graphs measures the structural complexity by calculating the amount of different paths in a graph. This is used as a code metric by considering the flow graph (V, E) of a method. The cyclomatic complexity of that method is then defined as

$$CC := |E| - |V| + 1,$$

where V denotes the nodes of the flow graph and E denotes its edges. Details on how to create flow graphs can be found in section 9, as this was the task of a transformation case of the TTC2013.

Again, there is a threshold for the Cyclomatic Complexity given in form of a predefined rule CA1502[3] to "Avoid excessive complexity" that is triggered if a method has a Cyclomatic Complexity above 25.

Cyclomatic complexity is mainly triggered by conditional statements and loops, since these language constructs make up large varieties of possible paths through a method. In NMF Transformations, the main methods to specify the behaviour of a model transformation is the `Transform` method, which after all is a method like any method in an OO program. Thus, the Cyclomatic complexity also measures the structural complexity of such a `Transform` method similar to any other method that developers would usually write. Also the intention behind this metric - a method with a higher structural complexity is more complex to test - als applies to `Transform` method implementations and thus, this metric helps transformation developers to prevent them from writing methods that are difficult to test.

If we consider the `RegisterDependencies` method, this method is to be used to specify dependencies. Setting these dependencies may depend on the configuration of the parent transformation, but in most cases, it is independent from any configuration and thus, the Cyclomatic Complexity of these methods should be 1. As the Cyclomatic Complexity of a class, namespace or module is considered the sum of the included methods, the Cyclomatic Complexity of these `RegisterDependencies` should not change the overall result, notably.

However, Visual Studio 2012 actually measures the code metrics on IL-code (an intermediate representation as a stack machine) instead of C# code. This has a consequence for

[3] `http://msdn.microsoft.com/en-us/library/ms182212.aspx`

lambda expressions, as they are compiled into ternary operators that cache the underlying delegate objects. As a consequence, the control flow is forked, inducing the cyclomatic complexity to rise. As a consequence, Visual Studio actually computes a too high Cyclomatic Complexity for methods that incorporate lambda expressions. As the API of NMF Transformations is based on lambda expressions to be used especially in the `RegisterDependencies`-method, these methods are affected by this bug. However, this bug may be fixed in the future independently from the further development of NMF Transformations and other tools exist that correctly measure the Cyclomatic Complexity.

In the case studies from the sections 9 and 10, the Cyclomatic Complexity reached values of up to 13 where the correct result would have been 1. However, as the threshold for a warning of the static code analysis is set to a value of 25, this bug is unlikely to raise a warning or error for the `RegisterDependencies` method. The Cyclomatic Complexity of the `Transform`-method ranged only between 1 and 2 and thus was uncritical.

F.1.3. Maintainability Index

The maintainability is a heuristic metric trying to express the essence of maintainability in one number. Unlike the original proposal in [OHA92, OH92] by Oman and Hagemeister, the definition within Visual Studio[4] rescales this index to return values between 0 and 100.

$$MI = \max(0, (171 - 5.2 * \ln(HV) - 0.23 * (CC) - 16.2 * \ln(LOC)) * 100/171),$$

MI = Maintainability Index HV = Halstead Volume
CC = Cyclomatic Complexity LOC = Lines Of Code.

The Halstead Volume originally proposed by Halstead in 1977 [Hal77] basically meaures the amount of information that must be understood when reading the code and mainly consists of the length of the program multiplied by the logarithm of the size of the language (the amount of used vocabulary). As this metric has originally been proposed long before object oriented design was known, it aims at procedural code rather than object-oriented design.

Whereas the original proposal of the Maintainability Index (MI) comes in two versions, with or without considering comments, Visual Studio only integrates the version without the consideration of inline comments. As a reason, Visual Studio computes these metrics on the compiled assemblies rather than on source code to keep the metrics comparable across the borders of .NET languages.

The threshold for the maintainability index lies as low as 20. Values above are considered to reflect a good maintainability, whereas values below 10 reflect a poor maintainability. Values below cause to trigger the code analysis rule CA1505 to "Avoid unmaintainable code".

Being an entirely heuristic metric, it is difficult to have an idea of how this metric will actually impact on anything else than it was intended for. As the Maintainability Index utilizes the values from the Cyclomatic Complexity metric, it also inherits its problems. Furthermore, the heuristic nature of the metric makes it fixed upon the area where it is usually applied, which is imperative methods. It is hard to apply it on any other code artifact, like declarative methods as `RegisterDependencies`.

On the other hand, we can also consider the main specification of the transformation, the `Transform` method, as an imperative method in an object-oriented design. This makes

[4] `http://msdn.microsoft.com/en-us/library/ms182213.aspx`

the Maintainability Index as much applicable to these methods than to any other imperative method and hence, the Maintainability Index following this argument had the same expressiveness as with any OO code.

In the case studies from the TTC (sections 9 and 10), the Maintainability Index values ranged from 57 and 95. These values are far above the threshold for a warning. This can be explained by the fact that these methods also do not perform too complex operations. The `Transform` method that achieved the MI of 57 was actually the `Transform` method of the transformation rule to transform local variables in the Flowgraphs case, that creates a string buffer and fills this buffer with the name as well as some static elements and, if any, the initialization of that variable - which is indeed far more than most other `Transform` methods do.

F.1.4. Class Coupling

Class Coupling, originally proposed as Coupling between objects (CBO) in [CK91, CK94] counts the references from a class to other classes. It counts the different references to other classes in local variables, parameters, return types, method calls, generic or template instantiations, base classes, interface implementations, fields defined on external types, and attribute decorations. Different references mean that all references to a certain class are counted as one. The rationale behind this metric is that classes ought to have a high cohesion, but low coupling.

There again is a Microsoft code analysis rule CA1506[5] to "Avoid excessive class coupling". The threshold used by this rule is a Class Coupling of 80 for a method. Any values below are considered representing a maintainable method. Values above 95 are considered to cause a poor maintainability.

There again, the `Transform`-method and the `RegisterDependencies`-method can be treated like any other object-oriented method and thus, the metric has its original expressiveness.

The values for the class coupling in the TTC case studies ranged from values as low as 2 to values up to 18, so there there is a lot more coupling possible before a warning arises.

F.1.5. Lines of Code

The Lines of Code (LOC) metric as implemented in Visual Studio measures the lines of IL-code that is produced by a piece of code and estimate the lines of C#-code that is necessary to produce such IL-code. Thus, this metric mainly measures the size of the project, where it does not count additional line breaks or comments. The advantage of measuring the IL-code representation is that this measure is independent from line breaks. Unlike IL-code, languages like C# or Java allow arbitrary many statements to be written in a single line. However, as metadata in general do not count as IL code, an interface definition has not a single line of code. In fact, the definition of the trace interface of NMF Transformations is written in a C#-file with 625 lines (including comments), but does not count towards the LOC metric in Visual Studio.

As writing a transformation with NMF Transformationsaims to modularize a model transformation by including a structure made up of transformation rules, the code that is executed inside the structure tends to get smaller. However, transformations can get very complex and the structure of transformation rules might not always be appropriate, as for example in the Petri Nets to State Charts case study, where only the initialization can be properly supported by NMF Transformations. In such cases, it is important to keep the transformation code maintainable.

[5] `http://msdn.microsoft.com/en-us/library/bb397994.aspx`

Unlike the other metrics, there is no predefined rule for static code analysis to warn developers of methods that exceed a certain threshold in their length. However, a method that performs too many actions is likely to have a high Cyclomatic Complexity or a high Class Coupling.

In the case studies, most of the `RegisterDependencies` methods achieved a LOC of 1, as especially in the Flowgraphs case they often only instantiate other rules. The maximum LOC in the observed `RegisterDependencies` and `Transform` methods within the TTC case studies was 13, indicating reasonable short method sizes.

F.2. Metrics for transformation languages

Besides metrics for object-oriented code, NMF Transformations also introduces higher-level abstractions to write model transformations. In fact, most of these abstractions were introduced in other transformation languages like e.g. QVT-O and QVT-R that in turn inherited these abstractions from earlier transformation languages. As the QVT languages matured in the recent years, there are metrics available that try to measure some maintainability aspects of model transformations written in either of these languages. As the underlying abstractions are similar, some of these metrics are also applicable for NMF Transformations.

In his PhD thesis, Marinus van Amstel presented a set of 66 metrics for ATL and another set of 28 metrics for ASF+SDF[6] to assess the quality of model transformation written in either of these model transformation languages [vA11, vAvdB10, vAvdB11b]. These metrics, especially the ones for ATL may be candidates for adoption to transformations written in NMF Transformations. Furthermore, Kapova et al. presented a set of another 21 metrics suitable for QVT-R. As checking every single metric for its applicability for model transformations written with NMF Transformations would certainly blow the limitations for this master thesis, these metrics are not presented in this thesis.

Of course, many of these 115 metrics measure very similar code aspects, event within one language. For example, the number of rules in ATL is just the sum of the number of the numbers of rules of each rule kind. Furthermore, NMF Transformations can again make use of its host language, as there already are metrics to measure the quality of the implementations within the important methods, e.g. `RegisterDependencies` and `Transform`. Thus, here we only depict metrics that measure what makes model transformations with NMF Transformations different to those written entirely in ordinary general purpose code. Most of these difference arise from the alternative computational model that is the backend of NMF Transformations. Thus, metrics are depicted that measure important aspects of this computational model.

However, these metrics will only be described here and applied to the case studies from the chapters 9 and 10. Proper evaluation is required to prove that these metrics uncover potential maintainability problems. In addition, the metrics presented here will be defined in a way such that they are pretty much orthogonal, i.e., instead of using dozens of metrics that catch every detail, only a few metrics are described that try to cover as much maintainability issues as possible. As a reason, most developers do not really want to review a set of 66 metrics to find maintainance issues, as it is hard to review such a great amount of metrics at once.

A general problem of metrics for NMF Transformations is its nature as an internal DSL for a Turing-complete general purpose language. The semantic model of NMF Transformations is built by calling certain methods like `Require`, `Call` or `MarkInstantiatingFor`

[6] `http://www.meta-environment.org/`

and all their different versions. However, static analysis can detect such method calls, but it cannot review the parameter values when such a method is called. This is an immediate consequence of the Halting problem. However, as C# is a type safe language, static code analysis can at least make some assertions on the type of the parameters. In many cases, this is sufficient to compute the necessary metrics, but this might not always be the case. In a similar way, the transformation rules inside a transformation are created within a method call. As a result, metrics that incorporate any of these features cannot be done through static code analysis only. Many metrics will require dynamic code analysis, i.e. executing the code. However, this is only possible on certain assumptions and hence, many metrics will need user interaction in some circumstances and may only be computed automatically under certain circumstances.

As the availability of software code metrics is an important factor to maintainability, the design decision becomes crucial at this point, as code metrics largely cannot be guaranteed to be computable automatically.

F.2.1. Size metrics

Usually the starting point for metrics is size metrics, such as the number of rules or the number of patterns. However, in object-oriented design, this metric is mostly ignored as no conclusions on the maintainability can be drawn. A high number of classes does usually stand for a large project. However, the whole project can also be put all in one class, meaning that the software is actually created in a procedural manner rather than utilizing an object-oriented design - which results in a low number of classes, but is considered harmful to the maintainance, as it does not employ advantages of object-oriented design, such as data encapsulation. However, if one assumes a certain way of splitting code into classes, for example multiple code projects from a single developer, then *Number of Classes* indeed gives a good overview on the size of a project.

The size of a project is rather measured in general code metrics like lines of code. However, as explained in section F.1.5, some implementations of this metric only measure the code, which does not include interface declarations as they only define metadata. Again, as transformation developers are likely to use Visual Studio as their IDE, it is important to note that Visual Studio also does not count metadata towards the LOC metric.

The equivalent of a *Number of Classes* metric in NMF Transformations would be something like *Number of Rules* and *Number of Patterns*. These metrics suffer from the same effect as the *Number of Classes*. When solving a transformation problem with a model transformation specified in NMF Transformations, it is an important design question which requirements are implemented with transformation rules - and which of them are not, but solved in general purpose code instead, e.g. inside the `Transform` method. It is yet unclear, whether a good design decision is dependent on the transformation problem or rather a question of personal flavour. To answer this question, user studies have to be conducted where several transformation problems are implemented by several transformation developers. Such a user study is not part of this thesis. As a consequence, the applicability of size metrics like *Number of Rules* or *Number of Patterns* cannot be reviewed as important research questions so far must be left unanswered.

F.2.2. Rule coupling

A factor that predictably impacts the maintainability of a model transformations even without a large user study across different transformation problems is the coupling of the transformation rules, i.e. how many transformation rules are referenced by a certain transformation rule. This is reflected by the dependencies of a transformation rule. However,

there are some more question on how to measure the coupling between transformation rules. Most importantly, it is important to decide where to count inversed dependencies. As such dependencies are to be maintained in the transformation rule that defines these dependencies, it is more logical to count them for the transformation rules that define them, insetad of the transformation rules that actually own the resulting dependency. Furthermore, NMF Transformations allows to specify dependencies to transformation rules that are not known to the developer by just specifying the type signature of these transformation rules. These depencies may result in arbitrary many dependencies. As a consequence, a reusable transformation rule might have different values of rule coupling in different transformations. If a transformation can be instantiated without parameters (using a constructor that does not require parameters), these possible multiple dependencies can be resolved and the exact number of dependent transformation rules can be obtained. If however this is not possible, we can only assume there is only one dependent transformation rule. A similar problem is that transformation rules may define dependencies based on some configuration properties. In such cases, we can simply assume that every actual call to any version of `Require` or `Call` counts as a dependency, regardless of whether this dependency is set for a specific instance of the transformation.

Another way in which a transformation rule can use another transformation rule is via the trace functionality of NMF Transformations. There again, we can hope that static code analysis can detect the transformation that is actually called, but this is not always possible.

F.2.3. Depth of Instantiation Tree

Similar to object-oriented design, where deep inheritance trees have been shown to be harmful for the maintainance of a software project [BBM96], also transformation rules form instantiation trees. As transformation rule instantiation is a concept to overcome problems when transforming inheritance hierarchies, the tree tends to reflect the inheritance hierarchy in either source or target metamodel. However, the instantiation tree can also represent a mixture of both inheritance hierarchies. Consider for example the second exaple transformation case as in section 4.2. In the NMF Transformations solution of this scenario, the instantiation tree of the transformation rules transforming persons is actually higher than the inheritance tree in the source metamodel (where no inheritance relations are used at all). Consider the case where *Person* had some base class in the source metamodel that is not represented in the target metamodel and at some point of the transformation, it is required to trace instances of this base class. The easiest way to accomplish this is to introduce a new abstract transformation rule that transforms that base class into objects and let the transformation rule *Person2Person* instantiate that rule. As a result, the instantiation tree of the resulting transformation rules is higher than each of the inheritance hierarchies in the source and target metamodels.

Similar to the Rule Coupling metric, this metric is only automatically computable where the transformation rule instantiations can be inferred by static code analysis.

Like the discussion for the *Depth of Inheritance*, it can be argued that values of the *Depth of Instantiation Tree* that exceed a certain threshold must be considered as being harmful to the maintainability of a model transformation. As transformation rule instantiation is a concept so similar to inheritance, a good starting point would probably be the threshold for the *Depth of Inheritance*, which is 6 when following the Microsoft static code anaylsis rules. However, the fixation of this threshold requires further research.

List of Figures

List of Tables

Listings

Abbreviations

.NET	A software development platform provided by Microsoft (actually not an abbreviation).
3GPL	3rd Generation General Purpose Language. An object-oriented general purpose language like C# or Java.
ABB	Asea Brown Boveri Ltd. A large multinational corporation operating in power and automation technologies.
API	Application Programming Interface.
ATL	ATLAS Transformation Language. A prominent transformation language implemented as external DSL.
CLR	Common Language Runtime. The virtual machine that executes code written in managed languages like C# or Visual Basic.NET
CLS	Common Language Specification. The specification of the minimum feature set for .NET languages. A library where the public API conforms to the CLS can be consumed by any .NET language.
DSL	Domain Specific Language. A language specific to a domain with limited expressiveness [Fow10]
EMF	Eclipse Modeling Framework. A framework to support MDE on the Eclipse platform [MEG+03].
EOL	Epsilon Object Language. The general purpose modification language within the Epsilon language family.
ETL	Epsilon Transformation Language. The transformation language within the Epsilon language family.
GUI	Graphical User Interface.
ICMT	International Conference on Model Transformations.
IDE	Integrated Development Environment. An editor with rich tool support for development such as Visual Studio or Eclipse.
IL	Intermediate Language. A byte code to the CLR stack machine.
LINQ	Language Integrated Query. An internal DSL for .NET languages that is implemented as a monad, see section 3.3.5 for details.
LOC	Lines of Code. A language independent code metric that measures the size of an implementation.
M2M	Model-to-Model Transformations. Model transformations that take one or multiple models as inputs as create one or many models as outputs
M2T	Model-to-Text Transformations. Model transformations that create arbitrary text out of models.
MDE	Model Driven Engineering. See section 3.1
MDSD	Model Driven Software Development. See section 3.1
MOF	Meta Object Facility. A meta-metamodel by the OMG

MTL	Model Transformation Language. A DSL written specifically for model transformations.
NMF	.NET Modeling Framework. An open-source project to provide support for model-driven techniques on the .NET framework
NTL	NMF Transformations Language. An internal DSL for easier use of NMF Transformations
OCL	Object Constraint Language. A side-effect free query language.
OMG	Object Management Group. A standardization organization, famous for e.g. the MOF standard.
PN2SC	The Petri Nets to State Charts case at the TTC 2013.
POCO	Plain Old CLR Object. An object loaded to the CLR that does not use features of a specific SDK, but only relies on features of the language.
QVT	Query-View-Transformation. See section 3.2.
SDMLib	Story Driven Modeling Library. An internal DSL to work on the Fujaba tool.
SDK	Software Development Kit. A framework to allow developers work with an existing application.
STAF	Software Technologies: Applications and Foundations. A federated event that hosted the ICMT and the TTC 2013.
T4	Text-to-Text-Transformation. The M2T implementation of Microsoft that is included in Visual Studio.
TTC	Transformation Tool Contest. A contest for MTLs to investigate their advantages and disadvantages.