FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

ATL4pros: Introducing Native UML Profile Support into the ATLAS Transformation Language

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Andrea Randak
Matrikelnummer 0625409

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: O.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Mitwirkung: Mag. Dr. Manuel Wimmer

Wien, 12.10.2011

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

ATL4pros: Introducing Native UML Profile Support into the ATLAS Transformation Language

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Business Informatics

by

Andrea Randak
Registration Number 0625409

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: O.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Assistance: Mag. Dr. Manuel Wimmer

Vienna, 12.10.2011

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Andrea Randak
Josef-Sirowy-Straße 10, 2231 Strasshof

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

(Ort, Datum) (Unterschrift Verfasserin)

Acknowledgements

First of all, I would like to thank my family and my boyfriend Martin for their great and endless support during the entire duration of my study. Thank you for motivating me in the course of this thesis and for your help and patience in stressful and trying times.

Further, I would like to express my dearest gratitude to my advisor, Gerti Kappel, and my co-advisor, Manuel Wimmer, who encouraged me to write the research paper *Extending ATL for Native UML Profile Support: An Experience Report*. Thank you for your constant support and your great assistance in every respect. The possibility to present the research paper and my ideas at an international workshop was a great and memorable experience.

Finally, I would also like to thank the AtlanMod team, especially Salvador Martínez and Jordi Cabot, for their kind support and all the helpful discussions during my research stay in Nantes.

Abstract

The software engineering process has significantly changed over the last decade. Whereas in the past models were only used for communication and brainstorming purposes, this philosophy has shifted drastically. Model-Driven Engineering (MDE) is the keyword that is guiding the current engineering direction. Models are the key artifact and all development steps are aligned to these models. Sophisticated modeling techniques have been invented to ensure a consistent and comprehensive technological basis. The Unified Modeling Language (UML) was introduced by the Object Management Group (OMG) to standardize and support different modeling aspects like structural, behavioral, and architectural models. The huge success of UML is not only due to the versatility of the language but also because of the highly-developed language extension mechanism in form of UML profiles. UML profiles may be defined for tailoring UML to specific domains and technological platforms.

Apart from modeling languages, the technique of model transformation plays a crucial role for the model-driven approach. Model transformations aim at transforming an existing source model into some desired target model. Different model transformation languages were invented that support various kinds of model transformations. One of them is the ATLAS Transformation Language (ATL), which is currently one of the most widely used transformation languages. While modeling languages defined by metamodels are directly supported in an ATL transformation, the use of UML profiles demands for a complex work-around. It would be desirable, however, to simplify the handling of UML profiles in such a way that profiles are, like metamodels, represented as language definitions.

The contribution of this master's thesis is to extend ATL for a native UML profile support. New language constructs are integrated into the abstract and concrete syntax of ATL to ease the use of profile-specific information within a model transformation. Apart from the extension of the ATL syntax, also an operational semantics for the new constructs is defined by translating the extended ATL to standard ATL. The goal of this work is to enrich the ATL language with new language-inherent constructs and keywords. During the implementation phase several technological possibilities as well as limitations were encountered and discussed. This collection of lessons learned can be seen as a guideline for future extensions of ATL, and as a starting point for a critical discussion about the extensibility of ATL.

Kurzfassung

Softwaremodelle, anfänglich für reine Kommunikations- und Entwurfszwecke genutzt, sind mit der Etablierung der Modellgetriebenen Softwareentwicklung (englisch Model-Driven Engineering, MDE) zum Kernstück des gesamten Entwicklungsprozesses avanciert. Verschiedenste Modellierungstechniken und -sprachen wurden geschaffen, um eine fundierte Basis für MDE zu gewährleisten. Die Unified Modeling Language (UML), ein Standard der Object Management Group (OMG), ist der prominenteste Vertreter unter den Modellierungssprachen. Der Erfolg von UML liegt nicht nur in den vielfältigen Einsatzmöglichkeiten begründet, sondern ist auch auf die elegante Erweiterungsmöglichkeit der Sprache zurückzuführen. Mit Hilfe von UML Profilen kann die Sprache für verschiedene Zwecke und Plattformen angepasst werden.

Abgesehen von Modellierungssprachen spielen Modelltransformationen eine entscheidende Rolle in MDE. Modelltransformationen haben zum Ziel, ein bestimmtes Quellmodell in ein gewünschtes Zielmodell zu transformieren. Die ATLAS Transformation Language (ATL) ist die am häufigsten eingesetzte Modelltransformationssprache. Während Modellierungssprachen, die durch Metamodelle definiert sind, direkt in ATL unterstützt werden, muss man für die Verwendung von UML Profilen auf komplizierte Behelfe zurückgreifen. Es wäre jedoch wünschenswert, Modellelemente des UML Profils genauso als Sprachdefinitionen nutzen zu können, wie es für Metamodelle möglich ist.

Ziel dieser Diplomarbeit ist es, ATL dahingehend zu erweitern, dass UML Profile direkt unterstützt werden. Die abstrakte sowie die konkrete Syntax von ATL werden um neue Sprachkonstrukte ergänzt, um damit die Verwendung von UML Profilen zu vereinfachen. Neben der Syntaxerweiterung muss auch eine operationale Semantik definiert werden, um die modifizierte ATL Version mittels Präprozessor zur regulären ATL Version umwandeln zu können. Die ATL Sprache wird dadurch um neue, sprachinhärente Bestandteile und Schlüsselwörter bereichert. Im Zuge der Umsetzungsphase wurden technische Möglichkeiten sowie existierende Einschränkungen identifiziert und aufbereitet. Diese Zusammenfassung kann als Richtlinie für zukünftige Erweiterungen gesehen werden sowie als Ausgangspunkt für eine kritische Auseinandersetzung mit der Erweiterbarkeit von ATL dienen.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In times of Model Driven Engineering (MDE) [5], the Unified Modeling Language (UML) [39] is one of the most widely used modeling languages. The success of UML is not least because of its mature language architecture. The sophisticated language extension mechanism of the UML in form of UML profiles allows to adapt the metamodel (i.e., the language definition of UML) for different technological platforms and modeling domains. The UML profile mechanism [14] was introduced by the Object Management Group in order to provide a lightweight extension mechanism for UML without requiring modifications of UML modeling tools. Therefore, profiles are a *language inherent extension mechanism* and allow to define arbitrary stereotypes and tagged values. Stereotypes are regarded as specialized metaclasses for either introducing new modeling concepts or restricting existing metaclasses of the UML metamodel. In the latter case, a restriction means that the value range of a given metaclass feature may only be restricted but the feature as such may not be deleted or deactivated. Tagged values are used for introducing new features for existing modeling concepts. Stereotypes and tagged values, structured within a UML profile, are a powerful tool for a controlled extension of any UML model.

When talking about MDE, the technique of model transformation is a key concept for bridging design and implementation of software systems. Various transformation languages were invented to facilitate the required transformation functionalities and to provide the transformation developer with a sophisticated toolkit. The ATLAS Transformation Language (ATL) [23] is currently the state-of-the-art transformation language in the Eclipse Modeling Framework[1] (EMF). The transformation engineer is using ATL for developing rule-based model transformations between a source metamodel representing the source modeling language and a target metamodel representing the target modeling language. An ATL transformation is then executed on a source model conforming to the source metamodel in order to produce a target model which

[1] http://www.eclipse.org/modeling/emf/

is conform to the target metamodel. The ATL technology offers a wide range of useful features and a comprehensive collection of available ATL examples that help to specify a model transformation.

With the ongoing importance of MDE, also the application field of model transformation has broadened drastically. Not only transformations between metamodels for bridging the gap between design and implementation of software systems are needed, but different model management tasks, model formats, metamodeling techniques, and model extension techniques need to be supported. Thus, it is inevitable for transformation languages such as ATL to align to these new challenges. In this respect, a huge limitation of the ATL transformation approach is the fact that currently only metamodels are supported as first class language definitions, whereas UML profiles are not natively supported in the ATL language.

1.2 Problem Statement

At present, metamodels are well supported in the ATL transformation language since both the source and the target model of a model transformation conform to a source or a target metamodel. Metamodel elements may be accessed and used in order to define ATL transformation rules. However, as more and more transformation engineers take advantage of the powerful UML profile mechanism, it is desirable to use stereotypes and tagged values like any other metamodel element. Thus, a language-inherent support for handling UML profiles would be of great value.

The problem with the application of UML profiles in an ATL transformation is that currently UML profiles are not natively supported. The use of UML profiles demands for a complex and laborious work-around. This means that operation calls to an external Java UML2 API must be performed in order to apply profile-specific elements and information. Apparently, this leads to various shortcomings for an ATL engineer. Firstly, the transformation code becomes more verbose due to complex statements. As a result, the readability of the transformation code becomes diminished. Additionally, maintaining the code at a later stage is much more difficult. Secondly, precise knowledge about the underlying UML2 API is needed to invoke the Java methods correctly. By this, the definition of a model transformation becomes an error-prone task. Thirdly, the definition of complicated attribute helpers and operation helper is needed for the execution of the transformation code. As these ATL helpers are different for every metamodel and profile, they have to be defined separately for each new model transformation.

Apart from the application of UML profiles, also other transformation scenarios require transformation language extensions as has been reported in several publications (cf. [15, 36, 46] for concrete extension examples and [49] for a survey). The problem with ATL is that it is not designed for introducing any desired extension. There are no interfaces which may be used for implementing additional language components. Therefore, any domain-specific extension of the ATL language requires for a direct modification of the abstract and the concrete syntax of ATL.

Such an extension could, for instance, include the introduction of new keywords for the ATL language as well as the reuse of existing language components put into a new context.

1.3 Aim of the Thesis

In short, the aim of this master's thesis is as follows:

Extending the ATLAS Transformation Language for a native UML profile support.

The goal of this work is to implement an extension of ATL which makes it possible to use UML profiles natively within an ATL transformation. More precisely, the result of this thesis is the implementation of an extended version of the standard ATL framework, called ATL4pros. The benefits of ATL4pros are not only shorter ATL transformations due to newly introduced keywords but also the abstraction from the Java UML2 API. No further technical knowledge is needed for handling UML profiles within ATL rules. Moreover, stereotypes and tagged values are assigned explicitly instead of implicitly which allows for sophisticated code completion as well as static type checking. In particular, these benefits are realized by the following three envisioned subgoals:

- **Abstract Syntax** extension for the integration of new keywords
- **Concrete Syntax** extension for reflecting the modifications of the abstract syntax
- **Operational Semantics** definition for the syntactical extension

ATL itself is described by a metamodel (abstract syntax) and by a Textual Concrete Syntax (TCS). Therefore, the metamodel of the standard ATL version has to be extended by new metaclasses in order to integrate new keywords. Thus, these keywords expand the abstract syntax of ATL. In order to make this extended ATL version work in the ATL editor, also the TCS needs to be adapted. As a result, the newly introduced keywords can be used by the transformation engineer for handling and applying profile-specific information such as stereotypes and tagged values within transformation rules.

As Figure 1.1 illustrates, ATL4pros ② resides on top of the standard ATL infrastructure, which is depicted in Figure 1.1-①. The ATL4pros transformation rules including the new keywords have to be transformed to standard ATL code in order to be interpreted by the ATL virtual machine. Therefore, a transformation between ATL4pros and standard ATL has to be implemented, see Figure 1.1-③. This transformation adds semantics to the keywords and is designed as a Higher-Order Transformation (HOT) [49]. The transformation is responsible for generating standard ATL rules out of ATL4pros rules and is therefore an essential part of this work.

1.4 Methodological Approach

The methodological procedure of this master's thesis is based on the design science paradigm [19]:

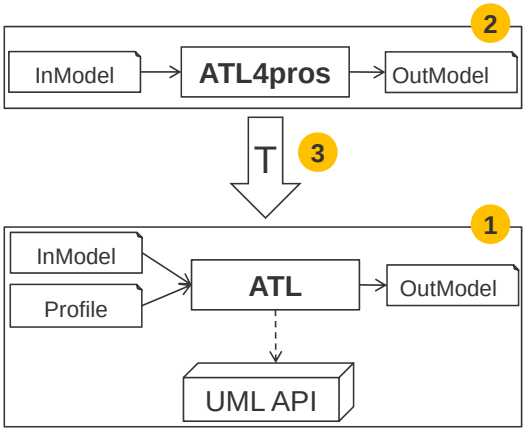

Figure 1.1: ATL extension architecture

1. **Literature analysis.** First of all, a comprehensive analysis of the current state-of-the-art and also of related work has to be conducted. This includes a literature survey in order to get a better understanding of the involved technologies.

2. **Analysis of the ATL architecture.** For being able to conduct a language extension of ATL, the ATL infrastructure has to be analyzed. The connection between abstract and concrete syntax needs to be studied to get a better idea of the technical interactions. Moreover, also the existing operational semantics of ATL needs to be analyzed in order to understand the involved concepts.

3. **Design of the extension.** Preparatory work in terms of an elaborate extension design has to be done before the actual implementation may start. New metamodel elements have to be defined and located in the ATL metamodel. Also, respective metaattributes have to be specified and associations to predefined metaclasses have to be taken into consideration.

4. **Implementation.** After completing the design phase, the implementation phase consists of three successive steps: (1) extension of the abstract syntax, (2) extension of the concrete syntax, and (3) definition of the operational semantics.

5. **Evaluation.** Finally, an example-driven evaluation of ATL4pros is performed to check its functionality, provide examples for a better understanding, and to analyze its improvement over standard ATL transformations. The comparison between standard and extended ATL transformations will be based on metrics and performance analysis.

1.5 Running Example

To clarify and explain the most important technological concepts and to provide a better understanding for the reader, this master's thesis contains a great number of examples. All examples are small excerpts of one primary running example. The running example is in turn based on

a real-world case study which was conducted by members of my research group in the course of the ModelCVS project[2]. This case study had two important purposes for the implemented ATL4pros extension:

1. At the beginning of this work, the case study was used to design the ATL4pros extension. An existing ATL transformation was used for abstracting and planning the structure of the realized ATL extension.

2. After completing the implementation work, the transformation example of the case study was taken and rewritten to the extended ATL version. By this, the correctness of the extension could be evaluated by comparing target models generated by the original transformation with target models derived from the extended transformation.

Case Study

The original goal of the case study in the ModelCVS project was a migration from the CASE[3] tool AllFusion Gen (AFG) to a UML tool. More precisely, the transformation of AFG data models to corresponding UML class diagrams should be facilitated. The problem concerning this migration was the huge gap between the AFG metamodel for the data models and the UML metamodel for class diagrams. Hence, the majority of the platform-specific information in the AFG data models had no counterpart in UML. The proposed solution to this problem was the use of UML profiles: With the help of a domain-specific profile, all additional information could be stored that would otherwise have been lost. The case study showed that finally, 50 percent of the resulting UML model was available as profile information.
The following case study artifacts were used for creating the ATL4pros extension:

- **DSL_2_UML transformation**
  This large and complex ATL transformation was the starting point for designing the ATL4pros extension. The DSL_2_UML transformation is used to transform DSL source models to UML target models. Moreover, this transformation was later rewritten to an ATL4pros transformation.

- **DSL metamodel**
  This metamodel contains all classes used in an AFG data model. Selected excerpts of this metamodel will be repeatedly illustrated in this thesis.

- **UML profile**
  The UML profile of the case study was used to store additional model information. Specific parts of this profile are chosen for explaining the basic concepts of UML profiles.

- **Source models**
  The available source models were used for executing and testing the implemented ATL extension.

[2] http://www.modelcvs.org/
[3] CASE is the abbreviation of Computer-Aided Software Engineering

- **Target models**
  The target models of the case study were compared with the target models generated by ATL4pros. Thereby, the correctness of the implemented extension was evaluated.

1.6 Structure of the Thesis

This master's thesis is organized as follows.

Chapter 2 gives a general introduction to the topics of Model Driven Engineering (MDE) and Model-Driven Architecture (MDA). The concepts and goals of MDA are discussed. Moreover, the topics language engineering and transformation engineering are explained in more detail. Language engineering focuses on the concepts of metamodels as well as UML profiles. The section about transformation engineering discusses the term model transformation and sets the focus on the ATLAS Transformation Language (ATL).

A motivating example that illustrates the use of UML profiles in the ATLAS Transformation Language is introduced in Chapter 3. The problems and limitations of the current profile support in ATL are shown. Subsequently, different improvement approaches and their respective benefits and drawbacks are discussed in more detail.

Chapter 4 focuses on the realized ATL extension named ATL4pros. This chapter is based on a publication that I presented at the *3rd International Workshop on Model Transformation with ATL*[4]. The first part of this chapter explains the overall ATL extension methodology. Subsequently, the involved process steps that are needed for the extension are depicted and discussed in detail.

Chapter 5 focuses on the evaluation of the implemented ATL4pros extension. An analysis of code quality and performance is followed by a collection of lessons learned, summarizing important observations that were made during the implementation work. Finally, current limitations of the developed ATL extension conclude this chapter.

Related work about other available model transformation approaches as well as different applications of UML profiles in combination with model transformations are discussed in Chapter 6. The final Chapter 7 of this master's thesis gives a conclusion and an outlook on future work.

[4] http://www.emn.fr/z-info/atlanmod/index.php/MtATL2011

CHAPTER 2

Model Driven Engineering in a Nutshell

2.1 Introduction to Model Driven Engineering

The notion of Model Driven Engineering [30] (MDE) has become an integral part of todays modern software engineering discipline. With MDE, models are now at the center of attention since they act as the key artifact of the entire engineering process. By definition, a model is regarded as an abstraction from reality, including only the relevant aspects of a system. It is therefore only natural to represent the details of a complex software system by means of a model. Models are no longer restricted to documentation and representation purposes but they can be used for a variety of important engineering tasks:

- **Model transformation**
  In general, models have diverse fields of application, different levels of abstraction and may be defined based on different modeling languages. The aim of a model transformation is to automatically transform one model into another model.
- **Model Testing**
  Errors at the source code level are hard to find and to eliminate. The functionality of model testing eases the detection of errors and provides testing facilities at a higher level of abstraction.
- **Code generation**
  The (semi-)automatic generation of programming code is an essential part of the model-based approach. Instead of using models only for communication purpose, they are directly transformed into code fragments.
- **Documentation**
  After all, models may still be the primary source for the documentation of a software system.

As the complexity of software systems is constantly rising and the integration of different technologies is becoming more important, models yield a good option for specifying such comprehensive systems. Like most technical innovations, also the model-driven approach is in need of a sound basis by means of standardized technologies. The next subsection gives an overview of the leading initiative of MDE called Model-Driven Architecture.

Model-Driven Architecture

The Object Management Group[1] (OMG) introduced the Model Driven Architecture[2] (MDA) in 2001. Basically, MDA is an integrated framework that both promotes and supports the concepts of model-driven engineering. Different modeling standards and specifications constitute the core of MDA. The Unified Modeling Language (UML) is amongst the most popular modeling standards published by the OMG. But also other predefined metamodels exist, e.g., the Meta-Object Facility[3] (MOF).
The overall goal of the MDA framework is the strict separation between the functionality of a software system and its platform-specific implementation. This separation allows for the reuse of one software system on different platforms, like for example on Java- or .NET-based frameworks. To achieve this aim, different levels of abstraction are required, made available by different models. The most important abstraction levels are explained in the following section.
The terms MDE and MDA are repeatedly confused as a clear separation of these two concepts is hard to identify. As stated in [20], MDA is one of many approaches to deal with the requirements of model-driven software development. In contrast, MDE has a more general meaning and is not necessarily restricted to the concepts of the Model-Driven Architecture.

Concepts of MDA

The reuse of existing software models on different platforms is an important aspect in relation to MDA. A computation independent model is used to understand and present the system functionality from an external point of view. Apart from this business model, it is essential to have a platform independent model that can, at a later stage, be converted into a platform specific model that is applicable for different software platforms. Having this platform specific model, the last step is to automatically generate application code that may be executed. This structured approach is illustrated in Figure 2.1.

Computation Independent Model (CIM). The computation independent model captures the business requirements of a system under development. The main characteristic of this model is that no implementation details like underlying data structures are present. The model only focuses on the later use of the system and on the systems environment. Business experts use this model to document the requirements of a system.

[1] http://www.omg.org/
[2] http://www.omg.org/mda/
[3] http://www.omg.org/mof/

Platform Independent Model (PIM). This model is a complete representation of the entire system with all required details and the necessary abstractions. Platform-specific characteristics are totally omitted in this type of model. By this, a possible change regarding the implementation platform has no effect on the model. The construction of a platform-independent model requires for some suitable metamodeling language, which is more deeply discussed in Section 2.3.

Platform Specific Model (PSM). Adding platform-specific information to a PIM instance leads to a new model called platform-specific model. This type of model is limited to one specific target platform considering all platform specifics and details.

Application Code. The (semi-)automatic generation of the final application code as well as the generation of other required artifacts is based on a platform specific model. Different code generation tools are available to support the user with the production of the final implementation code.

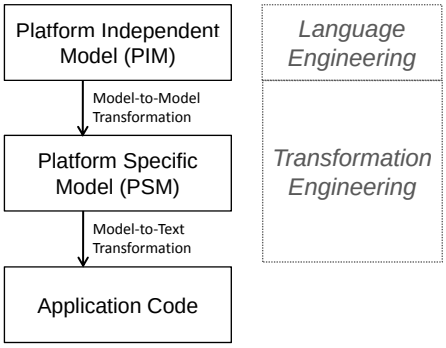

Figure 2.1: Concepts of MDA

As depicted in Figure 2.1, the terms language engineering and transformation engineering play an important role within the MDA approach. *Language engineering* is required for the specification of a modeling language like UML and is therefore the basis for any model definition. *Transformation engineering* on the other hand covers all aspects of a model-to-model or a model-to-text transformation. The important PIM-to-PSM translation is accomplished by the definition of a model-to-model transformation. To generate executable application source code, a PSM-to-Code mapping, that is, a model-to-text transformation, is needed. A more detailed introduction to language and transformation engineering is given in Section 2.3 and Section 2.4 respectively.

MDA Concepts in Practice

The OMG recommends its own technologies for the specification of the three aforementioned models. Both CIM and PIM may be defined using UML. The definition of a PIM is normally done with the help of UML profiles. As the use of these two techniques is only a suggestion, also other generic modeling languages may be chosen.

The different abstraction levels and the respective models, i.e., CIM, PIM and PSM, are basically only a conceptual design of the MDA initiative. In many cases it is not really possible to clearly distinguish between a CIM and a PIM. Moreover, current model transformation tools mostly provide standardized solutions for transforming a PIM into a PSM but no mechanisms for translating a CIM into a PIM, cf. [29,57]. Therefore, a computation independent model is of less importance and is usually not supported by MDA technologies. Due to the fact that the PIM is not automatically generated from the CIM but has to be defined separately, this procedure results in an additional effort and is therefore not very efficient.

The (semi)-automatic code generation is another issue that needs to be addressed and optimized in the future. The use of UML and UML profiles for defining the PIM and the PSM often leads to incomplete code parts or even errors in the code. Incomplete method bodies are only one of many examples for weak points of the generated code. The result is a manual revision and refinement of the programming artifacts.

2.2 Goals of Model Driven Engineering

After having explained the basic ideas and principles of MDE and MDA, it is time to focus on the pursued goals [6]:

- **Higher level of abstraction**
  The use of models as the key artifact in Model-Driven Engineering in general and within the Model-Driven Architecture in particular guarantees for a higher level of abstraction. With the help of platform independent models, system components can be described independently from their actual implementation. Models also help in understanding and capturing all aspects of complex systems.

- **Portability**
  Due to the clear separation of modeling concerns by PIMs and PSMs it is easy to migrate an existing software system from one software environment to another one. This migration may become necessary due to small technological adaptations of the current system or the radical change to another platform.

- **Interoperability**
  Building a software system on top of diverse technologies is the common standard in todays software development. The simultaneous use of object-oriented programming languages and relational databases is only one of many examples. MDA supports interoperability and the integration of different and possibly changing technologies since the software system is defined without specifying the later implementation platform.

2.3 Language Engineering

Language engineering is a sophisticated branch of modern computer science since we are confronted with a large number of different languages, divided into various language classes. We differentiate between programming languages like Java or C++ and modeling languages like

the Unified Modeling Language (UML) or the Business Process Modeling Notation[4] (BPMN). Other classifications distinguish between General Purpose Languages (GPL) like the UML and Domain-Specific Languages (DSL) like KM3 [24].
Language Engineering in the context of Model-Driven Engineering is concerned with different scopes of application. This specific engineering discipline comprises all necessary methods and concepts that are needed for inventing any kind of language. The specification of General Purpose Languages on the one hand is important to make a solid modeling basis available for the modeling community. Formal model language specifications provide standard techniques and components which unify the way in which software models are built. The definition of Domain Specific Languages on the other hand offers completely new and versatile possibilities.
Irrespective of the type of language and its intended purpose, there are certain characteristics that all languages have in common. The structure of a language is determined by different components and these are explained in the following paragraph.

Language Characteristics

In the area of software engineering there are different types of languages like modeling or programming languages. What all languages have in common and what is required for the specification of a language are the following three distinct dimensions [32]:

- **Abstract syntax**
  The abstract syntax defines the language elements and their connection between each other, in other words, the grammar of a language. By defining the abstract syntax of some arbitrary language, the language designer determines the phrases that are valid when using this language.

- **Concrete syntax**
  The concrete syntax constitutes the notation of the language elements, i.e., how certain elements are illustrated. This can either be done in a graphical or in a textual form. The former is often achieved by means of simple graphical icons. The latter aims at specifying the keywords that represent the elements of the abstract syntax. A language is not restricted to one single concrete syntax, i.e., it is possible for a language to have more than one concrete syntax descriptions.

- **Semantics**
  The semantics specifies the meaning of the language elements. Thus, a precise semantics definition is important for understanding what the particular language elements imply. This understanding of language concepts is directed towards the users of the language.

Meta Language

Every language, whether text- or model-based, needs to have a precise definition. A language that is used for defining another language is commonly referred to as *meta language*. The first

[4] http://www.bpmn.org/

widely-used meta language for the definition of a textual language was titled EBNF – Extended Backus-Naur Form. EBNF is an extension of the Backus-Naur Form (BNF) which was invented by John Backus and Peter Naur.
EBNF consists of so-called production rules which determine the syntax of a language and are repeatedly used for the definition of programming languages. These production rules (or non-terminal symbols) consist of terminal symbols (alphanumeric characters) and non-terminal symbols which can be arranged to form a valid sentence. For a short EBNF grammar example of Mini-Java please refer to [13]. In the case of EBNF there is no strict separation of abstract and concrete syntax. The definition of a programming language in EBNF comprises both the abstract and the concrete syntax.
EBNF is only applicable for textual languages like most programming languages are. However, the model-driven engineering ideology is primarily concerned with models and these models are by definition expressed using some modeling language. The terms model, metamodel and meta-metamodel are now carefully examined in the next part of the thesis.

Models, Metamodels and Meta-Metamodels

Models are at the center of attention in the model-driven software engineering approach. Models form the basis for all steps in the development process since they act as the key artifact. The meaning of the term *model* as well as other common definitions in this context are explained in the following.

Model

The term *model* is frequently used in the area of Model-Driven Engineering. According to [34], the best definition is that a model is *„an abstraction of a system allowing predictions or inferences to be made“*. Another definition is found in [45] and reads that a model *„is a set of statements about some system under study.“*. However, there exists a large number of other definitions of the term model.
What all definitions have in common is the fact that a model, in the context of software development, is used to represent a software system. The model itself contains only the relevant parts of the real system and is therefore referred to as an abstraction. The *four-layer metamodel hierarchy* by the OMG (see Figure 2.2) illustrates the different modeling layers. The system can be found on the lowest layer M0. Consequently, the model which represents the system is placed on the next higher level, referred to as M1. Examples for specific models are any type of UML model or an ATL transformation.
A model usually consists of different model elements that together form the system under development. These elements that are present in a model are not random, but are rather formally specified in a more abstract way – the metamodel. All model elements are therefore instances of metamodel elements. The metamodel is placed on the M2 layer of the four-layer hierarchy and will be explained in the next paragraph.

Metamodel

A metamodel defines the abstract syntax of a model [31]. This means that a metamodel comprises all language concepts and constructs that can be used in a model which conforms to some metamodel. Apart from the supported concepts, the metamodel also specifies the way how these concepts are interconnected and which properties are available. A model is therefore an instance of a metamodel consisting of model elements which are in turn instantiations of metamodel elements [2]. For example, the metamodel of ATL contains all language constructs that are valid for developing an ATL transformation.
Metamodels are of high importance in the field of Model-Driven Engineering since the entire software development process is based on and guided by models. The advantages of metamodels are as follows [20]:

1. Metamodels provide a standardized and concise definition of model elements. Only those language constructs, references, and properties that are present in the metamodel are valid in the derived models. By this, a common understanding of the particular modeling domain is possible. Moreover, uncontrolled model variations can be easily prevented.

2. Metamodels are the basis for the specification of models. Therefore, models can be checked for syntactic validity as they have to conform to their respective metamodels. Metamodels are thus an efficient control mechanism which eases the task of model development.

3. A metamodel is not necessarily a rigid construct that cannot be modified at a later stage. Modeling languages like the UML offer different alternatives on how to extend or specialize a metamodel in a controlled and language-inherent manner. Nevertheless, the original metamodel should be as stable as possible in order to provide a consistent basis for all conforming models.

Metamodels are found on the M2 layer of the four-layer metamodel hierarchy. By this, metamodels are needed for the definition of a language. The UML and the ATL metamodel are popular metamodel examples that will be further studied in the remainder of this chapter.

Meta-Metamodel

Models are specified by means of metamodels, so the next logical step is to examine how the concepts of metamodels are defined. As a metamodel itself is again a model, there must exist some superior level which specifies the model elements (i.e., the abstract syntax) of the metamodel. This highest level is called meta-metamodel or M3 and is on top of the M2 layer.
As Figure 2.2 illustrates, the meta-metamodel defines a meta language. All model elements that are used within a metamodel are specified by means of meta-metamodel elements, or in other words, every metamodel conforms to some meta-metamodel. The two most prominent meta-metamodels are now briefly discussed:

- **MOF**
  The Meta-Object Facility [37] (MOF) is a specification by the OMG and is a standard for

the definition of metamodels. MOF is particularly suitable for the definition of metamodels that represent object-oriented concepts and systems, like for example the UML. As MOF was introduced by the OMG, it is the proposed meta-metamodel within the MDA approach and the basis for all modeling concepts of the OMG.

- **Ecore**
  Ecore is a meta-metamodel used in the Eclipse Modeling Framework [47]. The idea of Ecore is to have a Java-based implementation of the most important MOF components. Therefore, the basic language elements of Ecore and MOF are very similar.

As depicted in Figure 2.2, the four-layer hierarchy ends with layer M3. The reason for this is that a meta-metamodel, like for example MOF, is reflexively specified. This means that every language concept used on the M3 layer is again defined by itself. No further abstraction layer is needed which helps in keeping the hierarchy manageable.

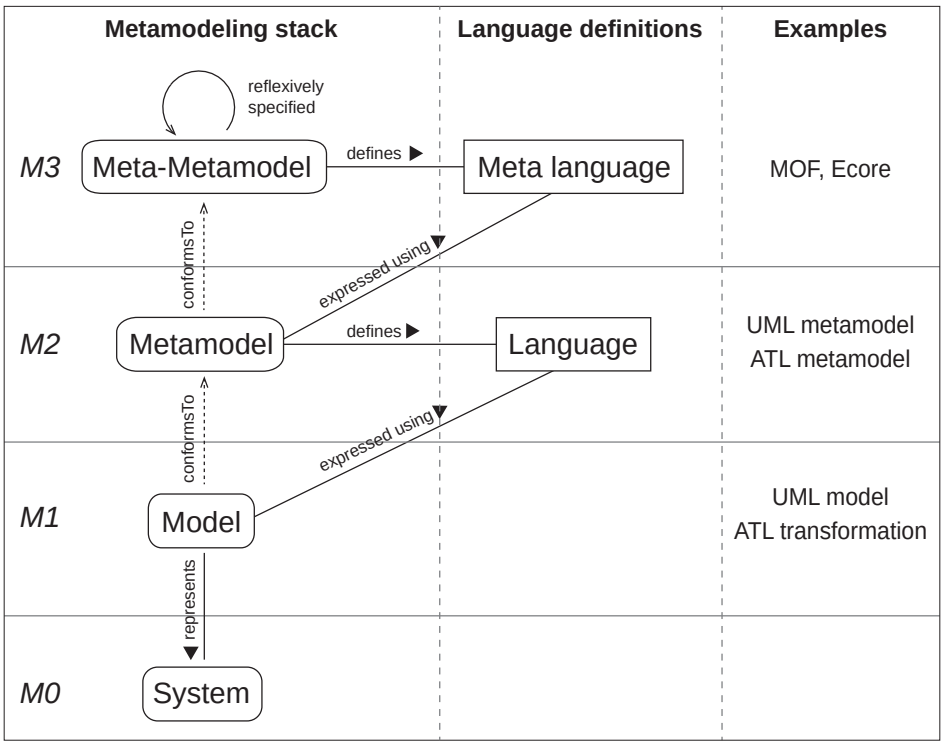

Figure 2.2: Basic structure of the four-layer metamodel hierarchy [34]

Depending on whether a language is used for a generic application or for a concrete purpose, there are two divergent language design principles, namely general-purpose languages and domain-specific languages. These two concepts along with their opposed orientation are now briefly discussed.

General-Purpose Language

The UML was introduced by the OMG as a General-Purpose Language [12] (GPL). The term *general-purpose* already indicates that such a language provides generic language features which may only be used for general problems and tasks. Another important characteristic is that the language components are not restricted to one single purpose and are therefore applicable for meeting diverse requirements.
A common problem with general-purpose languages is the fact that they are sometimes not expressive enough for certain modeling tasks. The existence of a domain-specific problem mostly requires for a domain-specific solution. Thus, a GPL may not be helpful in certain scenarios and consequently, the need for a domain-based approach arises.

Domain-Specific Language

A Domain-Specific Language (DSL) aims at providing language constructs for a specific purpose or domain. By this, it is more detailed than a GPL as it is restricted to a specified area of application. The idea of domain-specific languages has been very well received in the modeling community and is continuously used. SQL is a popular example for a DSL as it is tailored for handling diverse manipulation tasks in a relational database system. There are several alternatives for the definition of a DSL. The best way for creating a custom language including all relevant language elements is certainly by means of metamodels. Metamodels may be defined from scratch and aligned to any desired domain.

The Unified Modeling Language (UML)

The Unified Modeling Language[5] (UML) is one of the most widely-used modeling languages these days. UML was specified by the Object Management Group (OMG) and has become the de-facto standard since its first release. UML is a graphical language which may be used to model complex, object-oriented systems [41]. The UML provides different perspectives containing standardized modeling elements which allow for a holistic specification of a system under development.

UML Diagram

Basically, a UML diagram is an illustration of data and at the same time a visual representation of a UML model. The UML provides thirteen different diagram types which represent different perspectives of the system under development. A UML diagram is created by some UML modeling tool. To give the reader a better understanding of the complexity of UML, a brief summary of the most important diagram types is given below. A differentiation between structure diagrams and behavior diagrams is drawn at the highest level.

[5] http://www.uml.org/

Structure diagram. A structure diagram is needed to capture the structure of a system. For example, fixed system components and their respective properties may easily be modeled using a UML class diagram. Structure diagrams include amongst others:

- Class diagram: The logical structure of a system is captured by means of classes, relationships between classes and certain characteristics of classes in the form of features.
- Object diagram: Classes of the class diagram are instantiated and detailed in this type of diagram.
- Package diagram: Different packages may be clustered and structured within a package diagram.
- Component diagram: Components of a system including their dependencies and connections may be visualized with this diagram type.

The remaining structure diagrams are the deployment diagram and the composite structure diagram.

Behavior diagram. Apart from modeling the given structure of a system, it may also be essential to capture its behavior. Modeling different states and state transitions of system components or specifying the interaction between the user and the system are only two of many modeling possibilities. The behavior diagram type includes for example:

- Use case diagram: A use case diagram captures all functionalities that a system offers to its users.
- State diagram: State diagrams capture the internal states of certain system components including all possible state changes.
- Activity diagram: Operation sequences and other processes can be illustrated with this kind of diagram.
- Sequence diagram: Sequences of message exchanges between different actors are put in a chronological order.

Communication diagrams, interaction overview diagrams as well as timing diagrams complete the list of behavior diagrams.

The UML Metamodel

All language constructs of the UML are specified in the UML metamodel. As UML is a highly complex language containing a vast number of different components, the number of elements in the metamodel is accordingly large. Therefore, the metamodel is further divided into several packages which eases the understanding of the model and allows for a better separation of the different modeling concepts.
A very limited excerpt of the UML metamodel is depicted in Figure 2.3. Please note that a

large number of metamodel classes are defined as abstract classes. This means that those classes may not be instantiated by the UML modeler. Abstract classes are indicated by their class name written in italics, like for example the classes *NamedElement* or *Classifier*. All other classes are so-called concrete classes which can be directly instantiated in a UML diagram.
The given excerpt of the UML metamodel shows essential metamodel elements that are of great importance for the most prominent diagram type, the UML class diagram. The metamodel element *Class* inherits from the abstract class *Classifier* and represents a concrete class in the class diagram. Classes may have multiple *Properties* which are in turn subclasses of the *StructuralFeature* class. Classes may also have a number of *Operations*, an operation inherits from the abstract superclass *BehavioralFeature*. Due to space limitations only a limited set of classes and their connections between each other are shown. Other metamodel classes like for example the *Association* class are missing in Figure 2.3.
According to the four-layer hierarchy of the OMG (see Figure 2.2), the UML metamodel is defined on the M2 layer. Not surprisingly, the definition of the UML metamodel on layer M3 is done using MOF.

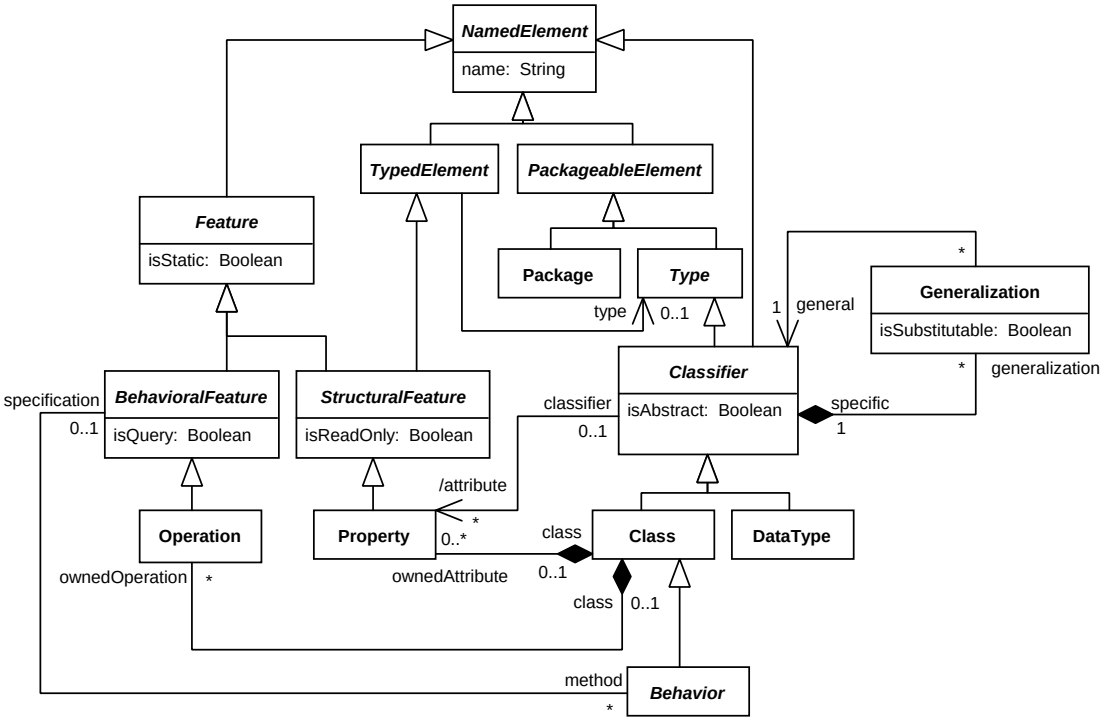

Figure 2.3: Excerpt of the UML metamodel [20]

Extensibility of the UML

UML is a general-purpose language and therefore not suitable for every possible modeling scenario. As mentioned before, some domain-specific problems may require for domain-specific

modeling constructs. However, the escape to a newly composed domain-specific language may not always be an option since the definition of such a language is a complex task. Fortunately, the UML offers some valuable extension possibilities for tailoring the predefined modeling elements to a specific purpose. Basically, three different approaches are supported:

1. Variant: **New metamodel:** In some cases it may be useful to define a completely new metamodel. This could for example be the case when the language concepts of the UML are not matching the requirements of some system. This new metamodel is designed as an instance of the meta-metamodel on layer M3 and may consist of arbitrary classes, relationships between classes and features.

2. Variant: **Heavy-weight extension:** This approach is a so-called *heavy-weight* extension. Based on the original UML metamodel, the modifications are incorporated by mechanisms like inheritance or redefinition. The introduction of new super- or subclasses is one possible aspect, but also new associations may be appended or existing element features may be overwritten.

3. Variant: **Light-weight extension:** The third alternative is referred to as *light-weight extension* as it leaves the UML metamodel unchanged [3]. An extension is defined by means of *profiles*, containing *stereotypes* and *tagged values*. This type of extension is called a language-inherent extension as all concepts of the profile are provided by the UML infrastructure. This extension variant is the most promising one and is further discussed in the next subsection.

UML Profiles

The sophisticated UML language extension mechanism in form of UML profiles allows to extend the UML metamodel for different technological platforms and modeling domains. A profile is a *language-inherent extension mechanism* and stereotypes as well as tagged values constitute the heart of the profile concept. Stereotypes may be seen as specialized metaclasses for either introducing new modeling concepts or restricting existing ones. Tagged values are used for introducing new features for existing modeling concepts.
To get a better idea of the profile concept it is best to start with a simple example:

The UML profile is titled *DataModel* and contains the two stereotypes *Identifier* and *Attribute*. The illustrated stereotypes extend the UML metaclass *Property*. This means that an instance of the metaclass *Property* may be further specialized by either the stereotype *Identifier* or the stereotype *Attribute*, or by both of them. The stereotype *Attribute* owns a tagged value named *phase* which adds additional information to a *Property* instance.
A precise definition of the terms profile, stereotype and tagged value is given in the following paragraphs.

Profile

A UML profile contains all metaclasses, stereotype definitions, tagged values and possible constraints that are needed for one specific extension of a reference metamodel. An extension is

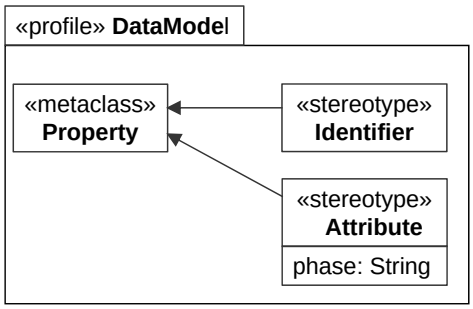

Figure 2.4: Simple UML profile example

usually covering some particular requirement or purpose and the profile is used for grouping all elements that form one domain-specific extension. A UML profile is usually indicated by the keyword *«profile»*, followed by its name (cf. profile *DataModel* in Figure 2.4).
The Profiles package in the meta-metamodel of the UML contains all required metaclasses that are needed for the definition of a profile (cf. Figure 2.9). The main class *Profile* inherits from the class *Package*. The relationship between a *Profile* instance and its contained *Stereotypes* is established via a containment relation named *ownedStereotype*. The class *ProfileApplication* that is associated with the class *Package* is used to demonstrate which profiles are applied to a package. It is important to note that a profile must always extend a reference metamodel which is in turn conforming to the MOF, e.g., the UML metamodel. This dependency is illustrated by the association named *metamodelReference*.

Stereotype

Stereotypes are the main component of a UML profile. The concept of stereotypes is used to specialize metaclasses with respect to some domain, platform or some specific functionality [4]. By applying a stereotype to some metaclass, the class gets bound to a defined purpose or a usage context. For example, an instance of the metaclass *Property* in Figure 2.4 may be assigned with the *Identifier* stereotype. This assignment indicates that the property instance is consequently considered as an identifier.
Moreover, the existing metaclasses may be extended by new metaattributes or restricted by user-defined constraints. Apart from the *Stereotype* metaclass itself, all existing UML metaclasses may be specialized by some stereotype. The metaclasses are extended on the M2 layer (metamodel layer) of the four-layer architecture, see Figure 2.2. Stereotypes may be instantiated and used like normal model elements of the M1 model layer.

Appearance and declaration of a stereotype. A stereotype is depicted as a rectangle (like classes in a class diagram) and is composed of three sections. The first section includes the keyword *«stereotype»* in a pair of guillemets, followed by the name of the stereotype. The second segment contains the attributes of a stereotype, called *tag definitions*. An instantiated tag

definition is called tagged value. The third section may be used to define possible operations of a stereotype.
The metaclass *Stereotype* inherits from the metaclass *Class*, see Figure 2.9. By this, stereotypes can form a generalization hierarchy with abstract super-stereotypes and associated sub-stereotypes. A derived stereotype inherits all extension relations, tag definitions and constraints from its super-stereotype.
Figure 2.5 shows a generalization hierarchy with the abstract stereotype *Attribute* and the derived stereotypes *UserDefinedAttribute* and *SystemDefinedAttribute*. The generalization relationship is indicated by an arrow with an empty arrowhead pointing from the specialized sub-stereotypes to the more general super-stereotype. As the figure shows, the name of an abstract stereotype is written in italics. Both sub-stereotypes inherit the tag definition *phase* as well as the required extension relation from the super-stereotype *Attribute*. The *SystemDefinedAttribute* has the additional metaattribute *attributeTestFlag*. As the defined extension is compulsory, a property instance must always be linked to an instance of either the *UserDefinedAttribute* or the *SystemDefinedAttribute* stereotype.

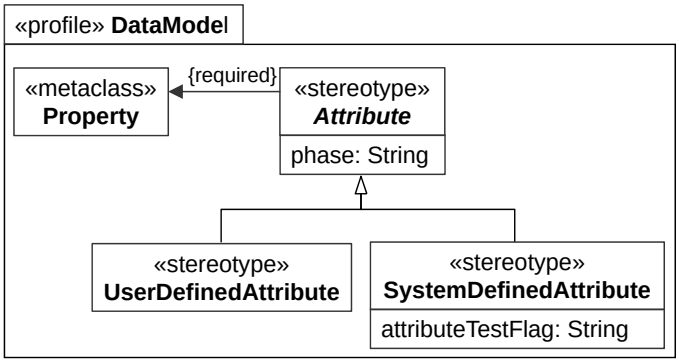

Figure 2.5: Generalization hierarchy of stereotypes

Extension of metaclasses. Stereotypes classify UML metaclasses by means of extension relations. The actual extension is represented as an arrow with a filled arrowhead pointing from a stereotype to the corresponding metaclass. The keyword *«metaclass»* on top of an extended class is optional and may also be omitted. By default, the extension relation indicates an optional extension, meaning that an instance of the extended metaclass may not necessarily be specialized by an instance of a stereotype. If the application of a stereotype needs to be enforced, the label *{required}* is printed above the arrow. Figure 2.6 exemplifies the difference between an optional and a compulsory extension.

Profile application. A stereotype can only be applied to a model element if the corresponding profile is applied to some package. This package must be created from the reference metamodel which is extended by the respective profile [39]. The profile application is realized by a dashed arrow with an open arrowhead and the associated keyword *«apply»*. The arrow is pointing from

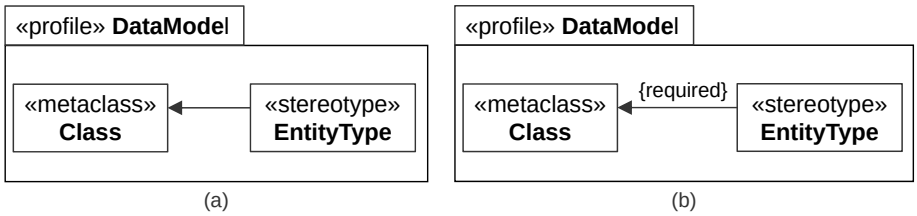

Figure 2.6: Examples for (a) optional and (b) compulsory extension relations

the package to the profile used for the extension. An example of such a profile application is illustrated in Figure 2.7. The *DataModel* profile is applied to a package which represents some *DataApplication*. In this example, only one profile is used. Note, however, that multiple profiles may be applied to one single package provided that there are no conflicting constraints. After applying a profile, all elements that are specified within this profile may be used.

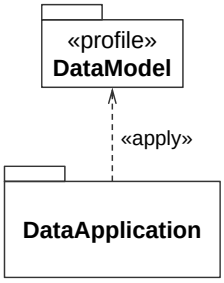

Figure 2.7: Profile application

Stereotype application. Like normal model elements, also stereotypes are instantiated when applied to a concrete model element. The usual notation to illustrate that an instance of a stereotype is applied to an instance of a metaclass is to write the name of the stereotype above the name of the class, enclosed by a pair of guillemets. If multiple stereotypes are linked to one metaclass, then the names of the stereotypes are separated by a comma, again within a pair of guillemets. Figure 2.8 illustrates a simple example.
The profile *DataModel* consists of one metaclass and two distinct stereotypes. The metaclass *Property* is extended by the two stereotypes *Identifier* and *Attribute*. The *Identifier* stereotype contains only one tag definition named *number*. The abstract stereotype *Attribute* is a generalization of the two derived stereotypes *UserDefinedAttribute* and *SystemDefinedAttribute* which both inherit the tag definition *phase*. While the *UserDefinedAttribute* has no further tag definitions, the *SystemDefinedAttribute* has an additional metaattribute named *attributeTestFlag*. The extension relationship between the metaclass *Property* and the stereotype *Attribute* is a compulsory one whereas the extension by the stereotype *Identifier* is merely optional.

Beneath the profile definition there are two examples for the application of stereotypes:
(1) The property *LocalSystemID* is specialized by the stereotype *SystemDefinedAttribute*. The values of the two tag definitions *phase* and *attributeTestFlag* are set to „phase1“ and „y“ respectively.
(2) Another property instance named *StartingPosition* is extended by two stereotypes, *UserDefinedAttribute* and *Identifier*. The *phase* attribute has the value „phase2“, the *number* of the *Identifier* is set to the integer value of 1.

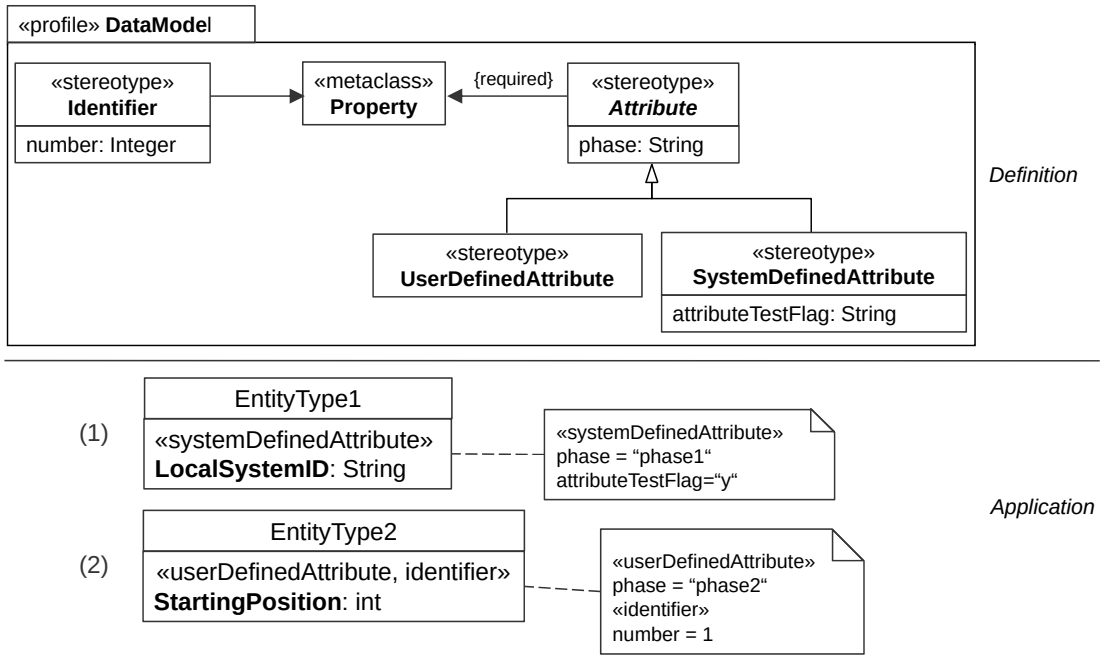

Figure 2.8: Examples of stereotype instances

Please note that the style guidelines of the OMG Superstructure recommend to write the first letter of an applied stereotype in lower case. In contrast, the first letter of a stereotype name within the profile definition is capitalized.

Tagged Value

Tagged values, or tag definitions, are used like normal properties of a class. They are composed of a name and a type definition. Possible types are for example String, Integer, Boolean or some complex type. An example of two tagged values called *phase* and *attributeTestFlag* is illustrated in Figure 2.5. There are different presentation alternatives for tagged values. The most common form is a comment symbol. This comment symbol usually contains the name of the stereotype followed by the respective tagged values. Each tagged value within the comment symbol is written as a name-value pair. A dashed line connects the comment symbol and an instance of the

extended metaclass. The comment-based representation is depicted in Figure 2.8.

The complete Profiles package from the UML Superstructure is illustrated in Figure 2.9. It contains all further metaclasses, relationships, and properties that are needed for a precise definition of a UML profile.

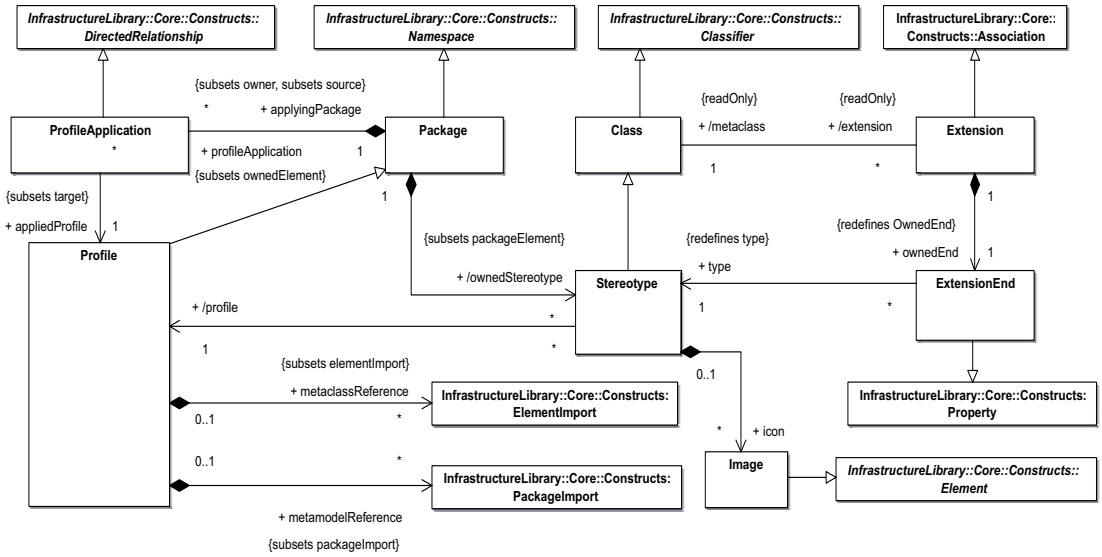

Figure 2.9: Profiles package of the UML Superstructure [39]

For more detailed information about UML profiles and stereotypes please refer to the OMG UML Superstructure Version 2.3 [39], pages 669 to 697. A brief introduction is given by [20], pages 334 to 342.

When browsing through various publication databases and searching for research articles related to the concrete application field of UML profiles, there exists a large number of results. UML profiles were specified for different application scenarios, like for example aspect-oriented software development, software product lines, multidimensional modeling in data warehouses, framework architectures, embedded system design or even for expressing GUI layout information. This is, however, only a small excerpt of the broad spectrum of applications for UML profiles.

Using UML Profiles in a Model Transformation

Even though the notion of model transformations is explained in the next section, the following table focuses on the impacts that profiles, stereotypes and tagged values have on a model transformation. Table 2.1 summarizes the impact that each concept has. Basically, this table is

inspired by the Eclipse documentation of the Java UML2 API[6].

UML element	Functionalities required in model transformations
Profile	(1) Applying a specified profile to a UML model or package (2) Removing an existing profile from a UML model or package (3) Checking if a certain profile is applied to a UML model or package (4) Querying all applied profiles of a UML model or package
Stereotype	(1) Applying a specified stereotype to a UML model element (2) Removing an existing stereotype from a UML model element (3) Checking if a certain stereotype is applied to a model element (4) Checking if a certain stereotype is applicable for a model element (5) Checking if a certain stereotype is required for a model element (6) Querying all applicable stereotypes of a UML model element
Tagged Value	(1) Setting the value of a specified tag definition and a specific stereotype (2) Querying the value of a certain tag definition and a specific stereotype (3) Checking if the value of a certain tag definition and stereotype is set

Table 2.1: The impact of UML profile elements on a model transformation

2.4 Transformation Engineering

The notion of transformation engineering plays a crucial role in the context of MDE [10]. Basically, a transformation is taking some input and subsequently generates the desired output. In the field of model-driven software engineering it is common to focus on model transformations. Based on the outcome of a model transformation, there are two fundamental types that may be distinguished. A transformation may either be a model-to-model (M2M) or a model-to-text (M2T) transformation. In the first case, both the input and the generated output are models. This type of transformation is especially useful for transforming a PIM to a PSM [16]. Another possibility is to define a model-to-text transformation that has a model as input and generates application code or other text artifacts as output. The type of transformation is chosen by the transformation designer and depends on the purpose of the particular transformation.

The Concept of Model Transformations

Model transformations aim at providing facilities and operations for converting an input model to some defined output. To be more specific: By means of model transformations it is possible

[6] `http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.uml2.doc/references/javadoc/org/eclipse/uml2/uml/package-summary.html`

to generate some target model that conforms to a target metamodel from a given source model, which conforms to a source metamodel. Designing a model transformation is thus synonymous with specifying which source elements are converted to which target elements.
This master's thesis is only concerned with model-to-model transformations and thus, the term model transformation is always referring to a model-to-model transformation.

Figure 2.10 illustrates the basic pattern and the involved artifacts of a typical model transformation[7]. A given source model M_a conforms to the source metamodel MM_a. This metamodel again conforms to some higher-level meta-metamodel MMM. The goal of a model transformation is to generate a target model M_b out of the source model M_a. All elements of the source model M_a are converted into elements of the target model M_b. This target model conforms to the target metamodel MM_b which in turn conforms to the meta-metamodel MMM. The transformation itself, referred to as M_t, conforms to the transformation metamodel MM_t. This transformation-specific metamodel is again an instance of the meta-metamodel.

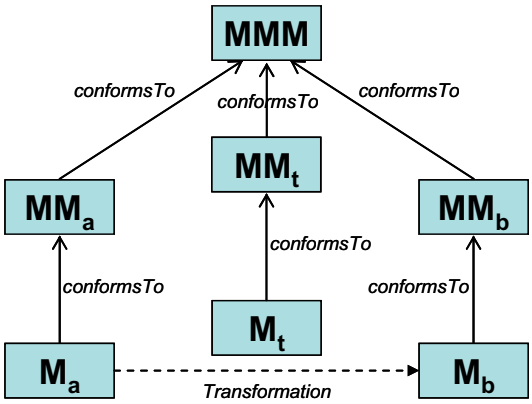

Figure 2.10: Basic pattern of model-to-model transformations

Research on a classification of existing and proposed model transformation approaches identified some commonalities that all model transformation approaches share [9]. For example, all approaches provide different variations of transformation rules, source-to-target relationships, rule organization or tracing. One model transformation approach is called QVT [38] and was specified by the OMG after issuing a Request for Proposal. An overview of different model transformation approaches is given in Section 6. The ATLAS Transformation Language (ATL)[8], the most widely-used transformation language, is introduced in the following subsection.

ATLAS Transformation Language (ATL)

The ATLAS Transformation Language (ATL) [23] is currently the state-of-the-art transformation language in the Eclipse Modeling Framework[9] (EMF). It was developed by the ATLAS group

[7] http://wiki.eclipse.org/ATL/Concepts
[8] http://eclipse.org/atl/
[9] http://www.eclipse.org/modeling/emf/

(now called AtlanMod team) and is currently maintained and further developed by OBEO[10] and AtlanMod[11]. ATL was invented as an answer to the Query/View/Transformation Request for Proposal issued by the OMG.

The transformation engineer is using ATL for developing rule-based model transformations between a source metamodel representing the source modeling language and a target metamodel representing the target modeling language. An ATL transformation is then executed on a source model conforming to the source metamodel in order to produce a target model which is conform to the target metamodel. The source model may also be referred to as input model while the target model is also called output model.

An ATL transformation is defined within an ATL file, indicated by the *.atl* file extension. Each ATL transformation is comprised of one single ATL module. The module and all other important concepts of the ATLAS Transformation Language are explained below. For a detailed introduction please refer to the ATL User Guide[12], for an in-depth look on technical details please see the ATL Developer Guide[13].

Defining an ATL Transformation

The definition of a new ATL transformation starts with the definition of a new *ATL module*. An ATL module is the highest-level unit within one ATL file and corresponds to a single model-to-model transformation. The ATL module comprises all parts of the transformation, namely (i) the compulsory header section, (ii) the optional import section, (iii) the helper section and (iv) the rule section.

Header section. The header section is used for specifying the name of the ATL module as well as the source and target models. The target models are specified after the keyword *create* while the source models are specified after the keyword *from*. The metamodels of the respective models are defined after the name of the model, separated by a colon. The names of the metamodels are subsequently used in the transformation to address specific metamodel elements. An example of an ATL header is given in Listing 2.1. The name of the module is set to *DSL_2_UML*. The target model is named *OUT* and conforms to the *UML* metamodel. The source model is called *IN* and conforms to the *DSL* metamodel.

Listing 2.1: ATL module DSL_2_UML with import section

```
1 module DSL_2_UML;
2 create OUT : UML from IN : DSL;
3
4 uses profileLibrary;
```

Import section. The import section is used to specify ATL libraries that may be used to outsource certain code parts, e.g., helpers. Libraries are reusable and may be imported into different

[10] http://www.obeo.fr/
[11] http://www.emn.fr/z-info/atlanmod
[12] http://wiki.eclipse.org/ATL/User_Guide
[13] http://wiki.eclipse.org/ATL/Developer_Guide

ATL modules. They help in keeping the transformation code short and maintainable. Listing 2.1 shows the import of an ATL library called *profileLibrary* on line 4.

Helper section. The helper section usually follows after the header section and the optional import statements and provides the possibility to declare attribute helpers or operation helpers. The term helper actually corresponds to the operation helpers that are equivalent to methods in some programming language like Java. Operation (or functional) helpers always have a name and may take parameters. These helpers may be called by the rules of a transformation or from some other helper function. Operation helpers are used to *calculate* some return value of a specified data type. Attribute helpers on the other hand may be used to store a specific constant value that may be needed at different points of the transformation. Unlike functional helpers, attribute helpers do not accept any parameter. Listing 2.2 shows the two types of helpers in ATL.

Listing 2.2: Attribute helper and operation helper

```
1  -- Attribute helper
2  helper def : stereoName : String = 'DataModel';
3
4  -- Operation helper
5  helper context UML!Element def: setVal(st:OclAny, pr:String, val:OclAny) : OclAny =
6   if(not val.oclIsUndefined())
7   then
8    self.setValue(st, pr, val)
9   else
10   OclUndefined
11  endif;
```

All helpers start with the keyword *helper*, followed by some optional context definition. The context defines for which type of metamodel element the declared helper is applicable. For example, the *setVal* helper in Listing 2.2 may only be called by an *Element* of the *UML* metamodel. The keyword *def* is written after the context definition and in front of the name of the helper. Optional parameters in brackets, the return type and the actual helper definition (introduced by an equality sign) complete the helper declaration.
The attribute helper on line 2 of Listing 2.2 is named *stereoName* and returns a String value set to the constant *DataModel*. The operation helper *setVal* on lines 5 to 11 is defined for the context of a *UML Element* and takes three parameters. The return type of this helper is set to *OclAny*, the most abstract ATL data type from which all other data types inherit. The body of the helper consists of a simple if-then-else statement which returns some value based on the evaluation of the if-part.

Rule section. The rule section is the most important part of an ATL model transformation. This section contains all transformation rules that are executed in order to generate the target model. The meaning of the term rule and the three different rule types are discussed below.

Rules

ATL is a rule-based transformation language. Therefore, the core of an ATL transformation consists of several rules that are defined by the transformation engineer. Basically, each rule is

indicated by the keyword *rule*, followed by its name and the body of the rule. ATL provides three types of rules that satisfy different requirements. Each type is briefly introduced:

Matched rule. A matched rule is the most important rule type. Its purpose is to match a distinct type of source model element and to generate the corresponding target model element. The source model elements that are to be matched are introduced after the keyword *from* whereas the target model elements are introduced after the keyword *to*. The initialization details of the resulting target model elements are also determined in the rule. Please note that matched rules are automatically executed by the ATL engine once for each match. This means that the given source model element is matched exactly once and the corresponding target model element is automatically created during the matching phase. Listing 2.3 gives an example of a simple matched rule. Source model elements of the type *DataModel* are matched and the desired target model elements of the type *Model* are generated. *DataModel* elements conform to the specified *DSL* metamodel while *Model* elements conform to the *UML* metamodel. The details of a target model element are specified using a set of bindings. A binding determines how features and references of the target element are initialized. An example can be found on line 6 of Listing 2.3. Here, the name of the UML *Model* is initialized with the value of the name attribute of the *DataModel* source element.

Listing 2.3: Matched rule example

```
1 rule DataModel_2_Model {
2  from
3   s : DSL!DataModel
4  to
5   t : UML!Model (
6    name <- s.name
7   )
8 }
```

Lazy rule. Lazy rules have the same structure as matched rules, with the difference that the keyword *lazy* is written in front of the keyword *rule*. A second difference is that lazy rules are not automatically executed but they have to be called explicitly by another rule. As a result, lazy (and called) rules are not executed during the matching phase but are evaluated during the last phase (target model elements initialization phase) of an ATL transformation execution. An example of a lazy rule is given in Listing 2.4.

Listing 2.4: Lazy rule example

```
1 lazy rule getModel {
2  from
3   s : DSL!DataModel
4  to
5   t : UML!Model (
6    name <- s.name
7      )
8 }
```

Called rule. This type of rule must also be invoked explicitly. As opposed to a matched rule, a called rule does only include the definition of the target model element. Thus, no source model element for matching purposes is required. An example of a called rule can be seen in Listing 2.5.

Listing 2.5: Called rule example

```
rule newModel (na: String) {
 to
  t : UML!Model (
   name <- na
  )
}
```

Rule inheritance. Since the release of the ATL 2006 version, also rule inheritance is provided by the concept of abstract rules. The ATL developer can define an abstract superrule and arbitrary many subrules that extend the superrule. By this, all the bindings that are defined in the abstract rule are automatically copied and combined via a union operator with the bindings of the subrule. For the sake of completeness, however, it should be mentioned that rule inheritance is also possible between concrete rules. The concept of rule inheritance is especially useful having class inheritance in the source and the target metamodel.

ActionBlock. Every ATL rule may contain one optional ActionBlock, also referred to as *do*-block. This block, introduced by the keyword *do*, contains an arbitrary number of imperative code statements that may be used for setting features and/or references of the generated target model element. The code statements defined within an ActionBlock are executed in a sequential order after the initialization of the corresponding target model element is completed. An example is illustrated in Listing 2.6. Every time the called rule *newModel* is executed, the target model element *Model* gets initialized and afterwards, the name of the *Model* is set.

Listing 2.6: Called rule with ActionBlock

```
rule newModel (na: String) {
 to
  t : UML!Model
  do {
   name <- na
  }
}
```

ATL Language Characteristics

ATL is a hybrid transformation language as it provides both declarative and imperative language features.
Imperative code: Imperative programming means that the programmer specifies the exact flow of the program instructions, i.e., the successive program steps. Imperative code includes programming constructs like for-loops and if-then-else-statements. The imperative transformation code, as for example defined inside an ActionBlock, is executed in a strict sequence.

Declarative code: Declarative programming on the other hand is not concerned with the flow of operations but rather focuses on the desired outcome.
The declarative part of ATL comprises all matched rules that match certain types of source model elements. Nevertheless, also imperative code is needed in the form of called rules, ActionBlocks, attribute helpers and operation helpers [26]. Imperative code parts are used when complex computations are not feasible by means of declarative constructs. Although imperative code is supported in ATL, the declarative programming style is the preferred one.

ATL Metamodel

An excerpt of the ATL metamodel is illustrated in Figure 2.11. It contains the most important elements that are needed for the definition of an ATL transformation. The previously discussed concepts Module, Library, Rule, Helper, and ActionBlock are represented in the metamodel as metaclasses. The rule inheritance, comprising matched, lazy, and called rules, is also illustrated. The ATL metamodel is discussed in more detail in Chapter 4.

ATL Execution Modes

ATL offers two different modes that may be set when executing an ATL model transformation, namely the normal execution mode and the refining execution mode:

Normal execution mode. This is the default execution mode of an ATL module. The given rules of the transformation are executed and, as a result, all specified target model elements are initialized and generated in the target model. Note that only those source model elements are considered for which a corresponding rule exists. All other source elements that are not matched by a rule are lost in the course of the transformation. This execution mode proves to be adequate if the involved source and target models differ fundamentally from each other. If the models are, however, similar to a large extent then the refining execution mode should be preferred.

Refining execution mode. The refining execution mode of ATL is especially useful when the target model is almost a copy of the source model, with only a small number of differences between the model elements. Using the refining mode, the transformation engineer defines rules only for those source elements that have to be modified. All other source model elements are automatically copied to the target model and remain unchanged. Thus, the refining mode assures that no model information is lost during the transformation.

ATL Architecture

The execution of an ATL model transformation requires for various components which are now briefly discussed. The ATL parser, the ATL compiler, and the ATL virtual machine together constitute the so-called ATL core.

- **Parser**
  The parser of ATL generates an ATL model conforming to the ATL metamodel from a

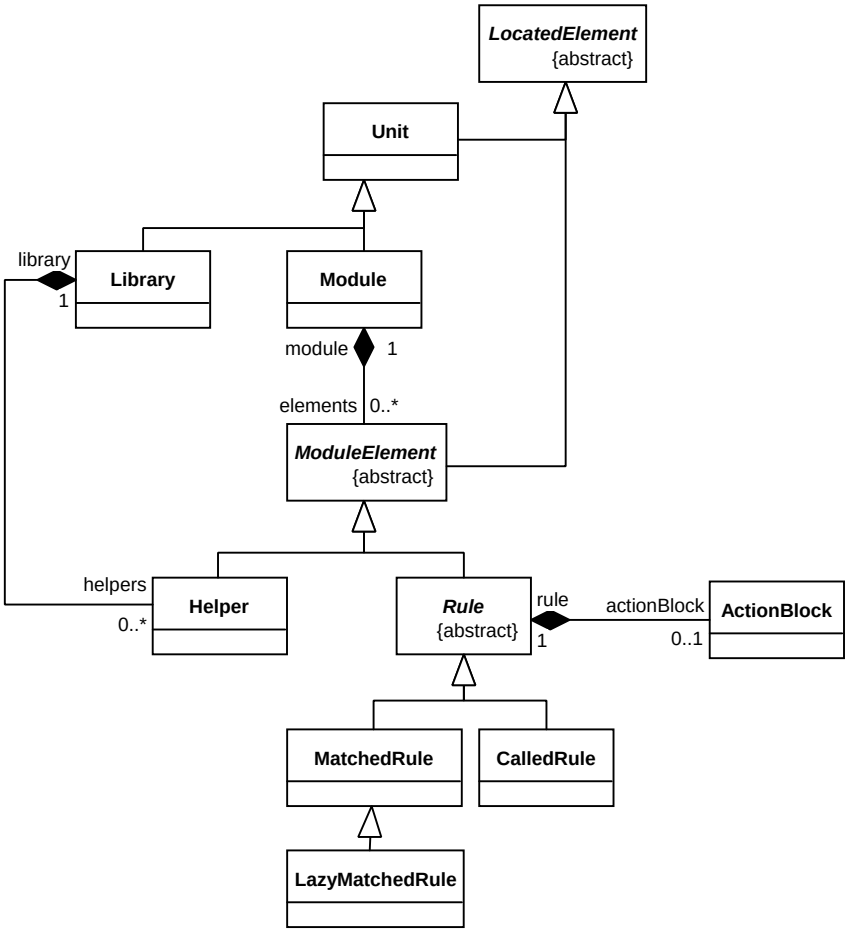

Figure 2.11: Excerpt of the ATL metamodel

given ATL transformation file. Moreover, the parser also outputs a problem model which includes all errors found in the code. The existing errors are illustrated by means of so-called markers in the editor.

- **Compiler**
  The next execution step is performed by the ATL compiler. This unit is responsible for taking the ATL model as input and creating byte code as output. Subsequently, the byte code is forwarded to the ATL virtual machine.

- **ATL Virtual Machine**
  The virtual machine (VM) of ATL interprets and finally executes the provided byte code by generating the required output model(s) from the specified input model(s).

- **Editor**
  The ATL editor provides useful features such as error markers, syntax highlighting, code

completion, occurrence highlighting, and hover information.

- **Debugger**
  ATL provides a debug mode that may be used like a debugger in any other programming language. It is possible to set breakpoints, execute a transformation step-by-step or run a transformation to the next breakpoint.

The clear separation of the three ATL core components is helpful for updates and further development done by different research groups. The modular design of the core makes it possible to focus on a small set of technologies without having to be familiar with the remaining components. For instance, modifications of the virtual machine may be accomplished independently from upgrades of the compiler or parser [8].

Limitations of ATL

ATL is a useful and comprehensive toolkit, provided that a model transformation is purely operating on models that conform to metamodels. However, a huge limitation of the ATL transformation approach is the fact that currently only metamodels are directly supported in the language, whereas UML profiles are not supported in a direct way. This means that the use of UML profiles within an ATL transformation is basically possible but the handling of stereotypes and tagged values is quite complex and requires for the escape to the external Java UML2 API.
The first part of the following chapter demonstrates a concrete example that illustrates the current problems regarding the use of UML profiles within an ATL transformation. Consequently, possible improvement approaches are discusses in the remaining part of the next chapter.

CHAPTER 3

Profile Support in Model Transformation Languages

As outlined in Section 2.4, there are several dedicated model transformation approaches. They provide their users with sophisticated functionalities for the definition as well as the execution of a model transformation. The basis for every model transformation are one or more models and as a consequence, metamodels are well supported by the different toolkits. However, a support of UML profiles is either not available or very unsatisfactory in all existing approaches. Model transformation languages like QVT or ATL do not have built-in language constructs for supporting UML profiles natively in the language. In case of ATL, the underlying UML2 API may be used as a kind of work-around. Native language structures would be needed in order to overcome this major drawback.
In this section, the extension of ATL for natively supporting UML profiles is motivated by showing a short transformation example using a specific *DataModel* profile for annotating the output model of the transformation. First, the transformation is illustrated in standard ATL code and associated shortcomings and problems are discussed. Second, an analysis of different improvement approaches concerning the use of UML profiles in ATL is conducted. Third, the chosen extension approach that has been implemented during this master's thesis is presented by showing how the original ATL code may be made more concise. Finally, the challenges of the extension process are discussed.

3.1 ATL Example using UML Profiles

The concept of UML profiles serves as a lightweight extension mechanism for UML. Arbitrary stereotypes and tagged values may be defined within a UML profile and may subsequently be applied to UML model elements. Using stereotypes and tagged values within an ATL transformation is feasible but leads to verbose transformation code. The code becomes complex and is hardly maintainable at a later stage of the transformation development process. Moreover, the

readability of the code is impaired due to difficult statements. Listing 3.1 presents an exemplary ATL code snippet. The rule matches elements of the type *SubjectArea* from the *DSL* metamodel and transforms them into *Package* elements of the *UML* metamodel.
The work-around for using a UML profile within an ATL transformation is based on providing the profile as an additional input model to the transformation. Moreover, calls to the Java UML2 API for assigning profiles and stereotypes as well as setting tagged values in the imperative *ActionBlock* are required (cf. line 10 to 27 in Listing 3.1).

Involved Modeling Artifacts

Before focusing on the ATL transformation code, the involved metamodels and the proposed UML profile are presented. Figure 3.1 illustrates an excerpt of the particular artifacts.

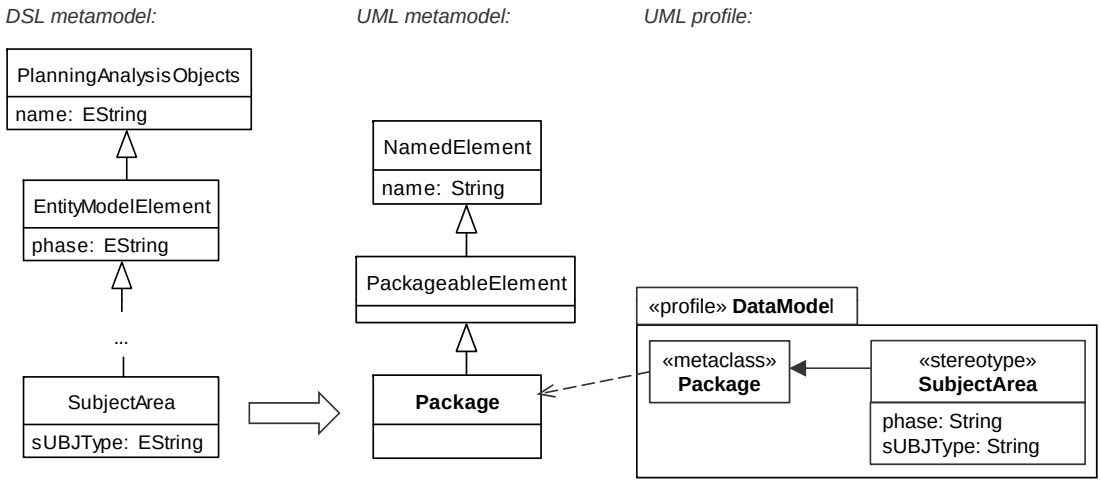

Figure 3.1: Excerpt of the DSL metamodel, the UML metamodel, and the DataModel profile

DSL metamodel. The source element that will be transformed is included in the DSL metamodel and is named *SubjectArea*. This metaclass indirectly inherits the metaattribute *phase* from the abstract metaclass *EntityModelElement*. As the *EntityModelElement* is in turn a direct subclass of the *PlanningAnalysisObjects* element, the *SubjectArea* also inherits the attribute *name*. The *SubjectArea* itself possesses one attribute named *sUBJType*.

UML metamodel. The transformation aims at matching all elements of the type *SubjectArea* in a source model and generating corresponding elements of the type *Package* in a target model. The *Package* metaclass is found in the UML metamodel. *Packages* inherit from the abstract class *PackageableElement* that again inherit from the abstract metaclass *NamedElement*. As a result of this generalization hierarchy, the *Package* class inherits the attribute *name*.

UML profile. The profile named *DataModel* is used to specialize the UML metaclass *Package* by (i) linking it to the stereotype *SubjectArea*, and (ii) adding further tag definitions to this class. As the custom-built attributes *phase* and *sUBJType* do not have a counterpart in the *Package* element of UML, these attributes are defined as tagged values in the stereotype.

Listing 3.1: ATL code excerpt for using UML profiles in standard ATL

```
1  helper def: stereo : uml!Stereotype = OclUndefined;
2
3  rule SubjectArea_2_Package  {
4   from
5    s : DSL!SubjectArea
6   to
7    t : UML!Package (
8     name <- s.name
9    )
10  do {
11    -- apply profile to target model element
12    t.applyProfile(profile!Profile.allInstances().asSequence().first());
13    -- store stereotype for later application
14    thisModule.stereo <- profile!Stereotype.allInstances()
15     -> any( e | e.name = 'SubjectArea');
16    -- apply stereotype to target model element
17    t.applyStereotype(thisModule.stereo);
18
19    -- set tagged value phase
20    if(not s.phase.oclIsUndefined()){
21     t.setValue(thisModule.stereo, 'phase',s.phase);
22    }
23    -- set tagged value sUBJType
24    if(not s.sUBJType.oclIsUndefined()){
25     t.setValue(thisModule.stereo, 'sUBJType',s.sUBJType);
26    }
27  }
28 }
```

Declarative part. Line 1 of the code example in Listing 3.1 shows the definition of an attribute helper called *stereo*. This helper is necessary for reusing the currently applied stereotype for setting the different tagged values without retrieving the stereotype from the additional input model again and again. On lines 3 to 9, the mapping between the source model element *SubjectArea* and the target model element *Package* is specified. On line 8 it is shown that the name of the *SubjectArea* may be set directly as the name of the UML *Package*.

Imperative part. Lines 10 to 27 comprise the imperative *ActionBlock* of the presented rule, indicated by the keyword *do*. On line 12, the UML profile is applied to the UML *Package* in order to enable stereotype applications. The code on lines 14 to 15 is used to query the stereotype named *SubjectArea* from the additional input model and save this stereotype to the previously mentioned helper *stereo*. Line 17 is needed for applying the recently saved stereotype to the target element. Setting the tagged value called *phase* is achieved on lines 20 to 21, the tagged value *sUBJType* is set on lines 24 to 25. The *setValue* operation on line 21 is a call to the Java UML2 API and takes the value of *s.phase* as the third parameter. It may be the case that this value is set to *OclUndefined*, which would not be a problem with normal bindings. But, as

OclUndefined is mapped to NULL in Java, this would result in an exception when invoking the *setValue* operation of the Java API. Therefore, it is necessary to provide an additional check (cf. *oclIsUndefined()* on line 20) before calling the operation. Please note that lines 12 and 17 also denote calls to the Java UML2 API, invoking the operations *applyProfile* and *applyStereotype* respectively.

Shortcomings

The things that are striking about this verbose ATL code are the following:

1. Profile related model manipulation tasks are implicitly established by means of calling Java operations. In particular, the transformation engineer needs to have knowledge concerning the Java UML2 API in order to correctly invoke operations for applying profiles, applying stereotypes and for setting tagged values. Moreover, it is not enough to know the syntax of the operations, but also to consider implicit pre- and post-conditions like e.g., valid parameter types.

2. Furthermore, all operations need to be defined within the imperative *ActionBlocks*. Even though ATL is a hybrid language supporting declarative and imperative constructs, imperative ATL code should be avoided or at least be reduced to a minimum.

3.2 Improvement Approaches for ATL

This section is mainly inspired by [55] and gives a comprehensive overview of possible improvement approaches concerning the use of UML profiles within ATL transformations. Three different alternatives are presented and their respective benefits and drawbacks are explained.
The sophisticated basic architecture as well as the modularity of the ATL toolkit (cf. Section 2.4) allow for numerous ways how an extension may be realized. Please note that this list of improvement approaches makes no claim to being complete, it should rather give an idea on different possibilities.

Approach 1: Merging the UML Metamodel with the UML profile

An intuitive idea is to combine the UML metamodel and the UML profile to produce one unified target metamodel. Generated target models would conform to this merged metamodel.

Benefits. This strategy has the advantage that no parts of the ATL framework are affected. All elements of the UML profile, that is, the stereotypes, the tagged values and the constraints, are incorporated into the metamodel of UML. Stereotypes could for instance be converted to normal metaclasses and, by means of inheritance or containment relationships, be connected to the metaclass they were originally extending. A further advantage is that no complex imperative statements are needed as the calls to the Java UML2 API are no longer necessary. Given that tag definitions are converted to standard metaattributes, setting the tagged values of a stereotype

may be done like the normal feature assignment of ATL.

Drawbacks. The problem with this procedure is that it totally contradicts the point and purpose of UML profiles. UML profiles were invented for specializing elements of the basic UML metamodel for an arbitrary domain, platform, and purpose while leaving the genuine UML metamodel untouched. By combining the UML metamodel with a UML profile, all advantages of this separation are lost. Also, the combination of the two artifacts may lead to an impedance mismatch between the profile and the generated metamodel. Another disadvantage is that the generated target model may not be processed by any kind of UML tool. A separate step would be needed in order to reverse the target model into a standard UML model so that it conforms to the standard metamodel of UML. Moreover, the merging procedure becomes more complicated as the number of involved profiles increases. An elaborate strategy would be needed for combining the different profile elements and for defining the correspondences between metamodel and profile elements. Another drawback is that a reuse of the merged metamodel is doubtful as it is tailored to one specific application.

Approach 2: Providing a Preprocessor

The second approach takes a different direction than the first one. The strategy is to actively extend ATL by incorporating new language constructs. This means that an extension of the ATL syntax is realized. This extension may include profile-specific keywords and other useful constructs in order to ease the use of UML profiles within an ATL transformation.

Benefits. A major advantage of this scenario is that the introduction of new keywords results in a more concise transformation code. A reduction of lines of code leads to an increase in both readability and maintainability. Furthermore, domain-specific keywords increase the understandability and as a result simplify the communication between ATL engineers. Moreover, complex and long statements, as often found in large-scale ATL transformations, could eventually be replaced by simple terms. Another important advantage is that static validation would be possible. As seen in Listing 3.1, the use of UML profiles within an ATL transformation requires for the use of the underlying Java UML2 API. API operations need to be invoked for the application of stereotypes and for the setting of tagged values. To accomplish this, knowledge about the UML2 API is required. An extension that is based on domain-specific new keywords and a preprocessor may fix this problem. API handling may be completely concealed from the ATL engineer by transferring the responsibility of correct operation invocations to the preprocessor.

Drawbacks. As an extended syntax may neither be recognized by the ATL compiler nor executed by the ATL virtual machine, the need for conducting an intermediate step arises. A kind of preprocessor may be needed for transforming an extended ATL transformation to an executable standard ATL transformation. This preprocessor may be implemented in various ways, either by using a high-level language like Java or by resorting to model-driven techniques like model transformations. Another restriction is concerned with the debugging functionality of ATL. Debugging may only be possible in the standard ATL transformation but not in the

extended version. Moreover, direct modifications of the ATL syntax may need the approval of the ATL community in order to be integrated into the existing ATL environment.

Approach 3: Redefinition of the ATL Compiler

The last approach is a modification of the ATL architecture by directly redefining the ATL compiler. New language constructs for an intuitive application of stereotypes and tagged values may be directly integrated into ATL.

Benefits. A modification of the ATL compiler would allow for a complete avoidance of imperative code parts. Moreover, the compilation time of an ATL model transformation may be reduced. Also, a complete tool support would be guaranteed and the definition of complex preprocessors could be avoided.

Drawbacks. The main drawback of this approach is the level of difficulty. Extending the ATL compiler demands for specific and deep knowledge concerning ACG (ATL VM Code Generator). New compiler instructions would be needed for generating the respective elements in the target model. Obviously, this is not a simple task and therefore not the first choice for the introduction of a small ATL extension. Apart from the complexity of this approach, it is also problematic due to potentially uncontrolled extensions of the compiler. If every ATL engineer adjusts the compiler to individual needs and requirements, things may start to get out of hand. As already stated in the aforementioned approach, this solution may also need the acceptance of the ATL community.

3.3 Introducing ATL4pros

The implementation work of this master's thesis is based on the preprocessor approach (cf. Section 3.2), evaluating the possibilities as well as the limitations of this approach. The approach is realized by implementing an extended ATL version named ATL4pros. ATL4pros aims at providing an extension which is tailored for the use of UML profiles within ATL transformations. The benefits of such an extension for handling UML profiles are evident:

1. The number of lines of code may be reduced due to the absence of complex statements for querying profile information from additional input models representing the UML profiles.

2. The readability of the transformation increases due to the fact that the transformation engineer may apply UML profiles without using imperative code. The imperative parts that are needed for the transformation to execute are completely hidden from the engineer.

3. Furthermore, ATL4pros eases the API handling as it is no longer necessary for the ATL engineer to have knowledge about the intricacies of the underlying Java UML2 API. All required statements that trigger an API call are automatically created by the preprocessor.

Listing 3.2: ATL code excerpt for using UML profiles in ATL4pros

```
1  rule SubjectArea_2_Package  {
2   from
3    s : DSL!SubjectArea
4   to
5    t : UML!Package (
6     name <- s.name
7    ) apply PRO!SubjectArea (
8      phase <- s.phase,
9      sUBJType <- s.sUBJType
10   )
11 }
```

Listing 3.2 (which is equivalent to Listing 3.1) illustrates the two extensions of the standard ATL syntax that have to be introduced into ATL4pros in order to allow for a more concise transformation definition:

1. The keyword *apply* (cf. line 7 in Listing 3.2) is incorporated into ATL and may be used for applying stereotypes to a UML model elements. Therefore, a new construct needs to be added to the existing ATL metamodel. In addition, a reasonable container element for the new construct needs to be identified within the existing ATL language element hierarchy.

2. The tagged values of a stereotype may be set just like normal features (cf. lines 8 and 9 in Listing 3.2) for avoiding the explicit writing of imperative code. This is achieved by reusing the existing *binding* construct of ATL and embedding it into a new context.

Challenges

There is a number of challenges that have to be met when extending ATL for supporting new language features:

Abstract syntax. The abstract syntax of ATL is defined as an Ecore-based metamodel which has to be extended with new elements. Since the metamodel is containing a large number of elements, it is crucial to find an appropriate location for the new ones. In addition, existing elements need to be altered in order to make the newly introduced elements usable within the standard transformation context.

Concrete Syntax. The extension of the concrete syntax of ATL is the second challenge. Not only is there the need to define new keywords for newly introduced elements, but also to revise the concrete syntax definitions for already existing elements by inserting references from/to the newly introduced ones.

Operational Semantics. The operational semantics determines how to transform the newly introduced language features to constructs of the standard ATL language via a higher-order transformation. To be more precise, rewriting rules need to be defined in order to produce standard ATL code and to eliminate the extended syntax elements.

All these challenges and their respective realization will be addressed in Chapter 4. First, an extension methodology in form of an extension process is presented and subsequently, the required process steps are discussed.

CHAPTER 4

Realizing ATL4pros

After having introduced the problem description and also the pursued solution strategy, it is time to focus on the actual implementation for realizing ATL4pros. The following section is dedicated to the ATL extension methodology used for building the ATL4pros extension and describes how the aforementioned challenges are tackled. In particular, the methodology at a glance is described first and subsequently, each extension step is explained in more detail. To make the methodology description more concrete, each step is exemplified by elaborating on the main artifacts of the ATL4pros extension.

Please note that Chapter 4 is based on the publication *Extending ATL for Native UML Profile Support: An Experience Report* [42] which was a result of my research stay in Nantes, France, in March 2011. This research paper was presented by me at the *3rd International Workshop on Model Transformation with ATL*[1] on the 1st of July 2011 in Zurich. The workshop was held in conjunction with the TOOLS 2011 Federated Conferences[2].

4.1 ATL Extension Methodology at a Glance

The general approach for building the ATL4pros extension is as follows. The syntactically extended ATL version ATL4pros is transformed to standard ATL via a higher-order transformation. Realizing an extension following this preprocessor approach requires for three successive steps:

1. The abstract syntax of standard ATL needs to be extended by new language elements.

2. The concrete syntax of ATL is extended according to the conducted modifications in the abstract syntax.

3. An operational semantics needs to be defined to determine how the language constructs of the extended ATL version, i.e., ATL4pros, are translated to the standard ATL constructs.

[1] http://www.emn.fr/z-info/atlanmod/index.php/MtATL2011
[2] http://tools.ethz.ch/tools2011/

Please note that this preprocessor approach is not limited to the presented example but may be applied for other domain-specific extensions as well [49]. The described approach leads to an extension process as depicted in Figure 4.1. Extending the abstract syntax is the first activity which has a direct dependency to the ATL.ecore artifact. As a next step, the concrete syntax has to be modified to reflect the performed extension of the abstract syntax. The concrete syntax of ATL is defined in the ATL.tcs artifact which has to be modified in order to use the new syntax elements in the ATL editor. In the last step, an operational semantics needs to be defined in terms of a HOT which has to be developed from scratch. In the following sections, each step is explained in more detail.

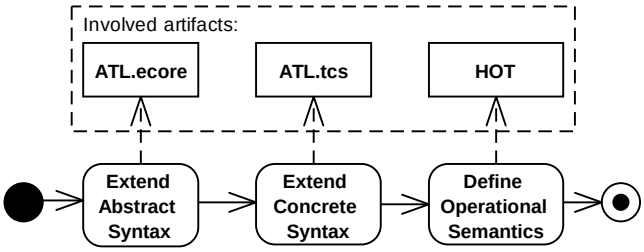

Figure 4.1: Extension process and involved artifacts

4.2 Step 1: Extending the Abstract Syntax

The abstract syntax of the ATL language is provided both as a KM3-based[3] as well as an Ecore-based metamodel. KM3 is short for Kernel Meta Meta Model and is a textual language compatible with Ecore that may be used for specifying arbitrary metamodels [24]. The KM3 version of the ATL metamodel is a textual representation whereas the Ecore-based metamodel is a graphical one. The metamodel of ATL visualizes the ATL metaclasses, their interconnections and their features in form of metaattributes. Each metaclass represents a certain language component, i.e., there are metaclasses for the most important ATL components like *Module*, *MatchedRule*, or *Helper*.

Apart from all essential ATL elements, the metamodel also contains a vast number of OCL model elements that are needed in a transformation. OCL stands for Object Constraint Language [53] and, in a transformation language like ATL, provides query language functionalities analogous to the features of SQL in a database. With OCL it is possible to process queries on metamodel elements and thereby define filtering rules. For example, if a model transformation may only consider those elements with the value of a boolean attribute set to true, the ATL engineer simply needs to specify an OCL filter. This filter is evaluated at execution time and results in the processing of only those elements that meet the filter criterion. Moreover, OCL may also be used to further restrict a collection of model elements or to impose other kinds of constraints on model elements within a model transformation [7].

[3] http://wiki.eclipse.org/index.php/KM3

Preliminary Considerations

At the beginning of the actual extension work, preliminary considerations have been examined and decisions regarding the structure of the envisioned extension and the resulting changes in the ATL metamodel have been made. To be more specific, the following questions have been answered:

1. The first question was concerned with the actual appearance of the extension within the ATL transformation code. It was important to decide how the new language constructs may fit into the existing textual structure. As outlined in Chapter 3, the final decision included the introduction of the new keyword *apply* which is always connected to one target model element. Moreover, the setting of the respective tagged values may be handled like normal feature assignments of ATL.

2. The second issue was about the required new ATL metaclasses, their metaattributes and the connections between new and existing classes. Suitable metaclasses have to be inserted into the ATL metamodel in order to implement the new constructs that have been designed and specified in the previous step. A detailed explanation about the particular new elements for the ATL4pros extension is given in the remainder of this section.

3. The third consideration related to the extension of the metamodel addressed the placement of the new metaclasses. The location of the new elements needs to fit into the overall structure of ATL transformations to ensure a smooth integration with already existing language elements. This step requires for a precise observation of the standard metamodel. An excerpt of the most important classes of the entire ATL metamodel is illustrated in Figure 4.2.

Defining an Extension

Before proceeding with the ATL4pros extension, it has to be clarified what the term *extension* in this context actually means by stating which actions are valid for producing the extended syntax, i.e., which actions are allowed for modifying the ATL metamodel during the extension process.

- **Insertion of new classes**
  Inserting new classes into the existing metamodel is of most importance. These new classes may have arbitrary features and may inherit from as well as reference to already existing and new classes.

- **Extending existing classes**
  For providing extensions, also existing elements may be extended by adding additional features. In particular, this is necessary for defining the container of newly introduced elements.

Given this definition of the term *extension*, the following operations are not allowed in the context of ATL4pros.

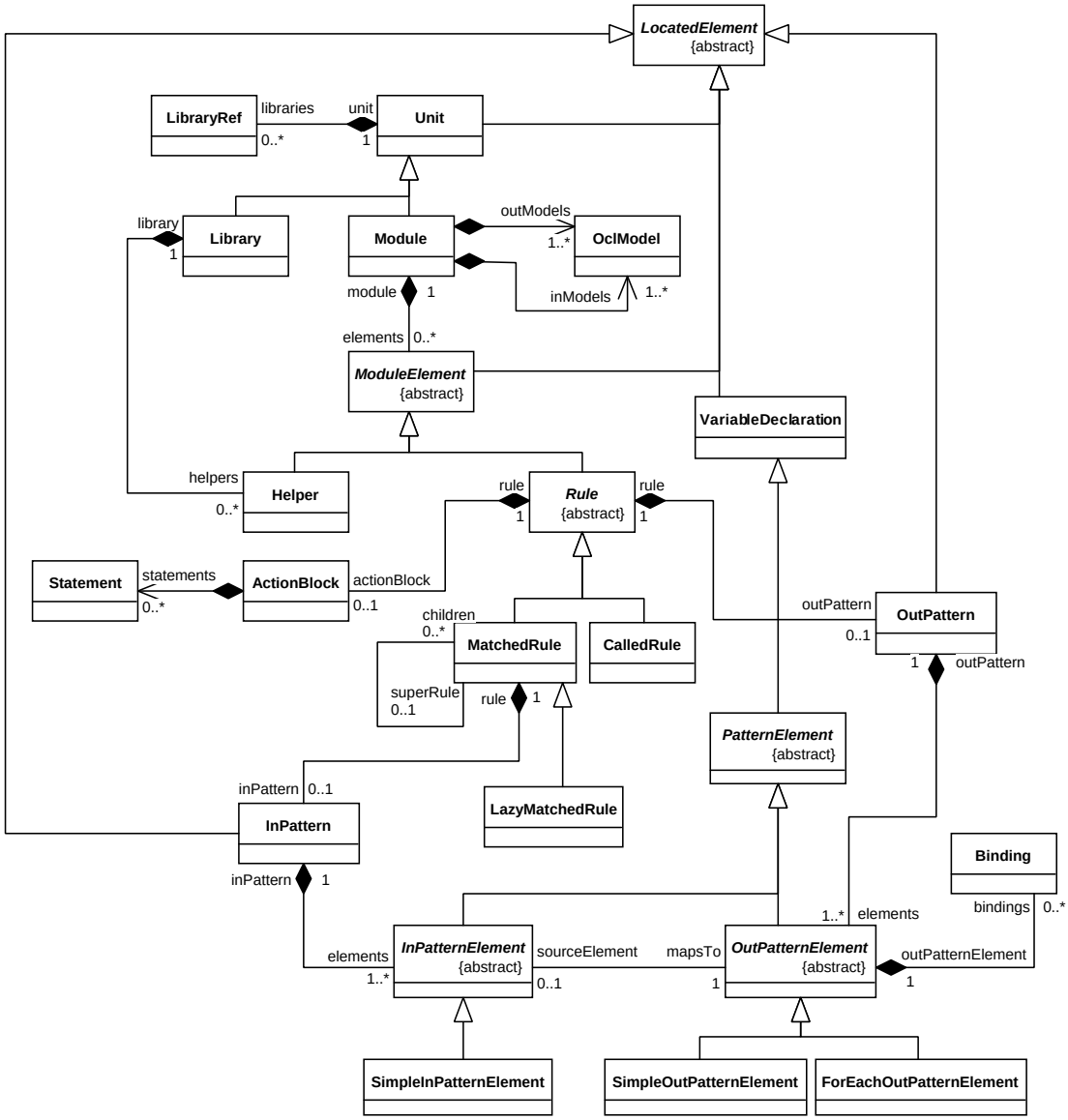

Figure 4.2: Excerpt of the standard ATL metamodel

- **Deletion of existing classes**
  All preexisting ATL metaclasses must remain in the metamodel. Removing those metaclasses may cause inconsistencies and may lead to severe problems when transforming the extended ATL4pros version to standard ATL.

- **Deletion of existing metaattributes or references**
  Due to the aforementioned problem, the removal of predefined metaattributes or references is no valid action in the extension process.

In general, deletions or updates of predefined metaclasses, metaattributes or references are of course possible. To avoid inconsistencies, however, such actions are not valid in the context of the ATL4pros extension.

New ATL Metaclasses

Three new metaclasses are inserted into the abstract syntax of ATL:

- **ApplyPattern**
- **ApplyPatternElement**
- **SimpleApplyPatternElement**

These three elements are essential for the entire ATL4pros extension. Figure 4.3 illustrates the new metaclasses along with preexisting classes of the ATL metamodel. The new constructs are depicted with gray color and blue lines while predefined elements are illustrated using white color and black lines. The *ApplyPattern*, the main component of the extension, was inserted into the ATL.ecore artifact first. Subsequently, the abstract class *ApplyPatternElement* and its concrete subclass *SimpleApplyPatternElement* followed. Note that the *SimpleApplyPatternElement* is currently the only subclass of the *ApplyPatternElement*. Given this fact, the generalization hierarchy may just as well be omitted. The reason why this inheritance still exists comes from the consideration that such a flexible hierarchy may become necessary for future work on the ATL4pros extension. For example, a *ConditionalApplyPatternElement* may be defined for the specification of further conditions.

The symbol of a filled diamond in Figure 4.3 defines a composition relationship. A composition is a special type of an association and defines a part-of relation between the involved elements. The elements on the opposite side of the diamond are the contained parts whereas the element with the attached diamond is the superior container element.

For a more detailed description about purpose and location of the three new elements please see Table 4.1.

Effects on Predefined ATL Metaclasses

A domain-specific extension of the ATL metamodel is not just about adding new metaclasses. To ensure a sound integration of the new constructs into the language, also predefined elements

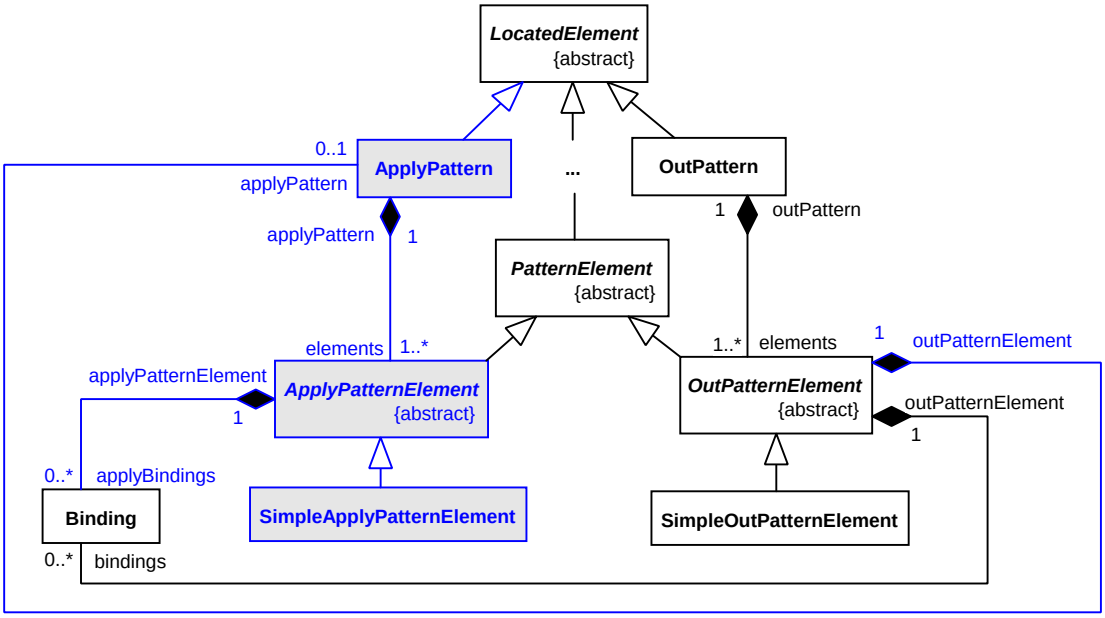

Figure 4.3: Excerpt of the ATL metamodel extended by new elements

need to be modified. The role of the most important existing ATL metaclasses regarding the ATL4pros extension and the performed adjustments are explained in the next part of the thesis.

Predefined ATL Metaclasses

The six predefined elements of the ATL metamodel excerpt illustrated in Figure 4.3 have the following purposes:

- **LocatedElement**
  The class *LocatedElement* has a significant role in the ATL metamodel as every other class directly or indirectly inherits from this element. The *LocatedElement* with its *location* feature gives every subclass the opportunity to have a fixed location within the transformation code, stating the line number(s) as well as the position within the line(s). All three newly introduced elements have the *LocatedElement* as their superclass in order to maintain the predetermined metamodel structure.

- **PatternElement**
  The *PatternElement* is an abstract metaclass and serves as a superclass for the existing classes *OutPatternElement* and *InPatternElement* (the latter is not depicted in the metamodel excerpt). Following this standard, also the *ApplyPattern* inherits from the *PatternElement*. The *PatternElement* is an indirect subclass of the *LocatedElement* metaclass.

- **OutPattern**
  The metaclass *OutPattern* represents the entire *to*-part of a model transformation (see

Listing 2.3 for an example). Therefore, the *OutPattern* is used to specify the target model elements that are generated in the course of the transformation execution. One *OutPattern* contains one to several *OutPatternElements*.

- **OutPatternElement**
  The *OutPatternElements* represent the actual target model elements to be created. For each source model element that meets the matching criteria, the corresponding target model

Name of metaclass	Description
ApplyPattern	**Purpose:** The class *ApplyPattern* is the main element of the extension. Basically, it represents the keyword *apply*, that will be available for the ATL engineer. **Function and location in the metamodel:** Like the predefined metaclasses *OutPattern* and *InPattern*, also the class *ApplyPattern* directly inherits from the abstract class *LocatedElement* (cf. Figure 4.3). A composition relationship associates one *ApplyPattern* with one *OutPatternElement*, which is the representation of a target model element. Furthermore, one *ApplyPattern* may contain one to several *ApplyPatternElements*. As an *ApplyPatternElement* represents a concrete stereotype, this connection assures that each *OutPatternElement* in the *to*-part of a transformation rule may have several stereotypes applied.
ApplyPatternElement	**Purpose:** This abstract metaclass corresponds to a stereotype that may be applied to one UML model element. **Function and location in the metamodel:** The classes *ApplyPatternElement* and *SimpleApplyPatternElement* are inspired by the hierarchical structure of the classes *OutPattern* and *InPattern* of the standard ATL metamodel. One *ApplyPatternElement* is always contained in one single *ApplyPattern*. Furthermore, the class *ApplyPatternElement* has a composition relationship to the standard *Binding* class. By this, *Bindings* may be reused to define assignments for tagged values, similarly as assignments for metamodel features are defined.
SimpleApplyPatternElement	**Purpose:** The *SimpleApplyPatternElement* represents a concrete instance of the *ApplyPatternElement* and thus, it represents a concrete stereotype. **Function and location in the metamodel:** This class is a subclass of the abstract *ApplyPatternElement*. Therefore, it inherits the two associations from the superclass, one pointing to the *Binding* class and the other one connecting it to the *ApplyPattern*.

Table 4.1: Description of the three new ATL metaclasses

element is created. The initialization of a target model element requires the specification of a type and an optional assignment of features.

- **SimpleOutPatternElement**
  The metaclass *SimpleOutPatternElement* is one concrete subclass of the abstract *OutPatternElement*. Thereby, it inherits all associations and metaattributes of the superclass.

- **Binding**
  The *Binding* metaclass is used for the ATL feature assignment. To be more precise, the *Binding* construct enables the initialization of features and/or references of a target model element with according feature and/or reference values from the source model element. A *Binding* is directly connected to an *OutPatternElement* via a composition relationship.

Modification of Predefined ATL Metaclasses

The modification of preexisting ATL metaclasses is an important part of the extension work. First of all, it is necessary to define a container element for the newly introduced classes. In the case of the ATL4pros extension, the main element *ApplyPattern* gets connected to the existing metaclass *OutPatternElement*. This connection constitutes the container relationship for the profile-specific ATL extension. Secondly, also other new elements need to be directly integrated into the metamodel by establishing relationships to predefined metaclasses. Depending on the purpose of an extension and the number of new metaclasses, this may become a time-consuming and critical activity. The extension may not work as intended if certain relationships are missing or defined incorrectly. Consequently, setting up the relationships between new and existing metaclasses deserves special attention.
The performed extension of the ATL metamodel affected only two predefined metaclasses. Therefore, only a small number of modifications concerning the default metaclasses is required (cf. Figure 4.3):

- **OutPatternElement**
  This metaclass is extended with one additional containment reference pointing to the new class *ApplyPattern*. This connection allows that each target element of a transformation rule may have one *ApplyPattern* associated. In concrete terms, the new relationship enables the application of stereotypes to UML model elements. The multiplicity value of 0..1 indicates that it is not mandatory for an *OutPatternElement* to have a relation to an *ApplyPattern*. The multiplicity value of 1 at the other end of the relation states that one *ApplyPattern* belongs to exactly one *OutPatternElement*.

- **Binding**
  The *Binding* metaclass obtains one supplementary reference which connects it to exactly one *ApplyPatternElement*. This relation assures that the binding statement, i.e., the definition of a feature assignment, may be reused in the context of an *ApplyPatternElement*. The constraint that one *Binding* is bound to exactly one *ApplyPatternElement* reflects the situation that one tagged value is always related to one stereotype. Based on the consideration that it is legitimate for a stereotype to have either several or no tagged values at all, the multiplicity value on the *Binding* side of the relation is set to 0..*.

Summarizing the extension of the abstract syntax in the context of ATL4pros, there are two major parts: (1) Three new metaclasses are incorporated into the metamodel of ATL and (2) two predefined metaclasses are modified in order to connect the new elements with the existing ones. A successful completion of this process step (cf. Figure 4.1) leads to the next activity – extending the textual concrete syntax of ATL.

4.3 Step 2: Extending the Textual Concrete Syntax (TCS)

After a successful introduction of additional abstract syntax elements in form of new metaclasses, the concrete syntax for these elements needs to be defined. The concrete syntax of ATL is available as a textual representation, also referred to as Textual Concrete Syntax (TCS) [25]. The TCS builds on the metamodel and consists of so-called *templates* which define the textual structure of the entire transformation. To be more specific, the TCS associates each element of the metamodel with a precisely defined textual counterpart. Each template in the TCS corresponds to an element of the metamodel, i.e., a new template in the textual concrete syntax needs to be inserted for each new element in the abstract syntax.
Before focusing on the available TCS templates of ATL and, subsequently, on the new templates for the ATL4pros extension, it is necessary to explain the term TCS, the purpose of this special kind of concrete syntax and the different constructs that may be used for defining the new templates.

TCS

The Textual Concrete Syntax (TCS) is a DSL that is used for converting models that are based on a metamodel to a corresponding textual representation. TCS itself is defined by a KM3-based metamodel and its syntax is specified reflexively. This means that the syntax elements of TCS are again defined in TCS. In the context of TCS, the terms *injector* and *extractor* constantly emerge. TCS provides both functionalities: The injector part is responsible for parsing the textual representation to the conforming model representation. The TCS parser itself is relying on the parser generator ANTLR [40] (short for Another Tool for Language Recognition). The extractor on the other hand may be used for serializing the information presented by a model to the associated textual format.
The metamodel of TCS contains various metaclasses that may be used for specifying a concrete syntax of an arbitrary DSL. The basic TCS constructs are explained in the following paragraph.

TCS Constructs

Basically, there are two major kinds of templates, namely the *PrimitiveTemplate* and the *ClassTemplate*. Apart from these two main constructs, there are also other syntactic elements that may be used within a template specification. For a more detailed explanation of the individual TCS constructs please refer to [25].

- **Primitive template**
  This kind of template describes the way how primitive data types of the abstract syntax, e.g., Double, Integer, or String, are textually represented. Each primitive template consists of a name and the respective data type. As more than one template may be defined for one single data type, there is always one obligatory *default* template. Lines 1 and 2 of Listing 4.1 show an example of a possible primitive template for the String data type.

- **Class template**
  Each metaclass (also called *classifier*) of the metamodel needs to be textually specified by a corresponding class template. This template determines the textual structure of the classifier and consists of a sequence of keywords, special symbols and other elements. The name of the class template has to be equivalent to the name of the metaclass. Additionally, only one class template per classifier is allowed, and exactly one class template needs to be marked with the keyword *main* to indicate that this template corresponds to the root metaclass in the metamodel. An example of a class template is given in Listing 4.3. As the name of the template is set to *Module*, this template corresponds to the *Module* metaclass of the abstract ATL syntax.

- **Keyword**
  Keywords are reserved words that have a special meaning in a model transformation. In ATL, keywords like *module*, *rule*, *helper*, or *if* allude the beginning of the particular ATL components. A good example for the definition of keywords is given by the class template of an if-expression, shown in Listing 4.1. The arrangement of the keywords *if*, *then*, *else* and *endif* determines the structure of an if-expression. The name of a keyword is written between double quotes.

- **Symbol**
  Special symbols like, e.g., opening and closing parentheses, may be used to structure the text within a model transformation. These symbols are, just like keywords, defined between double quotes. In contrast to keywords, each symbol needs a separate entry in the symbols section of the TCS definition. Each symbol in the symbols section is identified by a symbol name. For instance, the symbol names *lparen* and *rparen* identify the corresponding characters *„(„* and *„)“* respectively. Listing 4.2 illustrates the usage of the opening and closing parentheses on line 2. An excerpt of the symbols section including the two parentheses is given on lines 7 to 10.

- **Token**
  Tokens are special symbols that are not interpreted by the parser generator. The best example for a token is a comment within a code line. The token for the comment defines which characters indicate the start of a new comment.

Listing 4.1: Examples for primitive template and class template

```
primitiveTemplate identifier for String default using NAME:
 value = "%token%";

template IfExp
 : "if" condition "then" [
   thenExpression
  ] "else" [
   elseExpression
  ] "endif"
 ;
```

Listing 4.2: Symbol usage and symbols section in TCS

```
template ForStat context
 : "for" "(" iterator "in" collection ")"  "{" [
   statements
  ] "}"
 ;

symbols {
 lparen  = "(";
 rparen  = ")";
}
```

Predefined TCS Templates

The textual concrete syntax of ATL is defined in the ATL.tcs artifact (cf. Figure 4.1). The TCS strictly determines the grammar of the ATL language, i.e., which keywords and other constructs have to be used when defining an ATL model transformation. Moreover, the grammar also specifies how certain keywords have to be arranged among each other and which other text blocks are possible.
For each metaclass in the abstract syntax there is a dedicated template in the textual concrete syntax. Before focusing on the creation of the templates for the three newly integrated classes, it is necessary to examine and understand a predefined TCS template first. More precisely, the mapping between a metaclass, its features as well as relationships and the corresponding template is observed. For this purpose, the ATL metaclass *Module* and the related template serve as an example.

From an ATL Metaclass to the TCS Template

Figure 4.4 illustrates the *Module* metaclass from the ATL metamodel. To keep the example simple, all relevant metaattributes of abstract superclasses are illustrated as direct attributes of the *Module* class. The *Module* has two attributes and four composition relationships to other metaclasses.

Listing 4.3 illustrates the corresponding TCS template for the *Module* metaclass. A close look at the TCS code reveals that the two attributes *name* and *isRefining* along with the four associations *libraries*, *inModels*, *outModels* and *elements* are all present in the template.

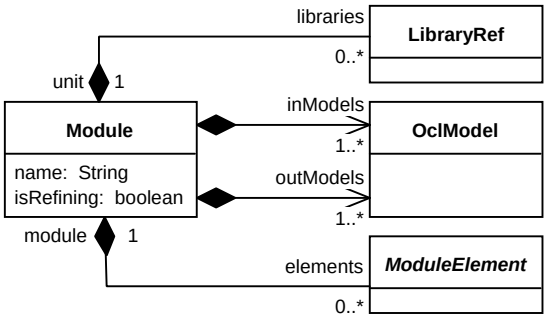

Figure 4.4: ATL metaclass Module

Attributes. The *name* that is given to a *Module* appears right behind the *module* keyword, seen on line 2 of Listing 4.3. The value of the boolean attribute *isRefining* is evaluated and, if set to true, the keyword *refining* is required between the specification of the *outModels* and the *inModels*. Otherwise, the keyword *from* is needed.

Associations. The four associations that are attached to the *Module* are pointing to other metaclasses of the metamodel. The textual representation of each of these individual classes is specified in their corresponding TCS templates and not inside the *Module* template.
The terms *outModels* on line 3 and *inModels* on line 4 of Listing 4.3 reflect the composition relationships that point to the *OclModel*. Looking at the *Module* metaclass in Figure 4.4, it is shown that more than one *inModel* or *outModel* may be attached to a *Module*. This fact is expressed in the template by using the built-in *separator* argument of TCS. In the illustrated TCS example, a comma is used as the delimiter between multiple *OclModels*.
The references to the metaclasses *LibraryRef* and *ModuleElement* are defined inside the square parentheses on lines 6 and 7. Hence, the respective textual representations of the involved metaclasses are inserted at these points of the template.

Listing 4.3: TCS template of the metaclass Module

```
1  template Module context
2    : "module" name ";" <newline>
3     "create" outModels{separator = ","} (isRefining ? "refining" : "from")
4       inModels{separator = ","} ";"
5     [
6      libraries
7      elements
8     ] {nbNL = 2, indentIncr = 0}
9    ;
```

Another example of a TCS template is given in Listing 4.4. The *Binding* is responsible for instantiating a certain attribute (also called property) of a metaclass with a defined value. The *propertyName* is, depending on the boolean value of the *isAssignment* attribute, followed by one of two possible arrow symbols. It must be noted that the second displayed arrow is the standard arrow whereas the first one is hardly used in any available ATL example. The *value* that is used for the instantiation of the property follows after the arrow.

Listing 4.4: TCS template of the metaclass Binding

```
1  template Binding
2   : propertyName{as = identifierOrKeyword}
3    (isAssignment ? "<:=" : "<-") 'value'
4   ;
```

After this concise introduction to the notion of TCS, the most important TCS concepts, and the connection between an ATL metaclass and a TCS template, it is time to define the three templates for the new metamodel elements.

New TCS Templates

The insertion of the three new metaclasses into the abstract syntax of ATL requires for the definition of three new templates in the textual concrete syntax. In other words, the new templates are defined in order to reflect the changes of the abstract syntax also in the concrete syntax.

Template for the ApplyPattern

The *ApplyPattern* template (cf. Listing 4.5) determines that the ATL4pros extension for applying a stereotype to a UML model element has to start with the keyword *apply*. This keyword may subsequently be followed by a comma-separated list of *elements*, whereby these elements refer to the *ApplyPatternElements* of the extended metamodel (cf. Figure 4.3). This assures that the constraints of the metamodel are correctly implemented, and that multiple stereotypes may be applied to one single target model element.

Listing 4.5: TCS template for ApplyPattern

```
1  -- Defining the textual concrete syntax of the new element ApplyPattern
2
3  template ApplyPattern
4   : "apply" [ elements{separator = ","} ] {endNL = false}
5   ;
```

Templates for the ApplyPatternElement and SimpleApplyPatternElement

The two templates for the remaining new metaclasses are presented in Listing 4.6. As the *ApplyPatternElement* is defined as an abstract class, also the corresponding template is declared as abstract.
The template for the *SimpleApplyPatternElement* is, compared to the other new templates, the most extensive one. The *type* (cf. line 7 of Listing 4.6) is used to represent a certain stereotype. Given the fact that the metaclass for the *SimpleApplyPatternElement* has no *type* attribute, there needs to be some other explanation for its occurrence: The inheritance hierarchy for the *SimpleApplyPatternElement* does not end with the *ApplyPatternElement* superclass. As illustrated in Figure 4.3, the *ApplyPatternElement* inherits from the abstract class *PatternElement*. The *PatternElement* in turn inherits from a metaclass called *VariableDeclaration*. This class is located in the OCL-part of the ATL metamodel and contains the *type* attribute. The data type of this attribute is set to *OclType* which allows for a flexible application of all kinds of metaclass types.

The reuse of the existing *Binding* construct is illustrated on lines 9 to 13 of Listing 4.6. Please remember that *Bindings* represent tagged values and the connected *ApplyPatternElement* (or rather the concrete *SimpleApplyPatternElement*) represents a certain stereotype. The tagged values of a specific stereotype are defined between parentheses and separated by a comma.

Listing 4.6: TCS templates for ApplyPatternElement and SimpleApplyPatternElement

```
1  -- Defining the textual concrete syntax of the new elements
2  -- ApplyPatternElement and SimpleApplyPatternElement
3
4  template ApplyPatternElement abstract addToContext;
5
6  template SimpleApplyPatternElement
7   : type
8    -- Reuse of the existing element Binding
9    (isDefined(applyBindings) ?
10    <space> "(" [
11     applyBindings{separator = ","}
12    ] ")"
13   )
14  ;
```

Please note that the existing *Binding* template of Listing 4.4 does not need to be changed due to the fact that the textual representation of a feature assignment does not change. Even though a new relationship was added to the metaclass *Binding* in the abstract syntax, this has no effects on its textual structure. Therefore, the *Binding* template remains unchanged and is only reused in the context of a *SimpleApplyPatternElement*.

Modification of Predefined TCS Templates

The definition of new elements in the abstract syntax of ATL led to the modification of existing metaclasses. Therefore, it is only natural that the extension of the concrete syntax involves a modification of existing TCS templates. As outlined in the previous paragraph, the *Binding* template remains unchanged. The only template that needs to be adapted is the one for the *SimpleOutPatternElement*.
To keep the template of the *SimpleOutPatternElement* simple, all preexisting lines of code were omitted except for the one on line 4 of Listing 4.7. Line 7 illustrates the conducted modification. The template is extended in such a way that an *ApplyPattern* may be attached to a *SimpleOutPatternElement*. Basically, this extension defines the anchor for the *ApplyPattern* within the textual concrete syntax.

Listing 4.7: TCS template for SimpleOutPatternElement

```
1   -- Enhancing the SimpleOutPatternElement to comprise ApplyPatterns
2
3  template SimpleOutPatternElement
4   : varName ":" type
5    ...
6    -- Embedding a new element into an existing element
7    (isDefined(applyPattern) ? applyPattern)
8   ;
```

The modification of the predefined template completes the extension of the textual concrete syntax. All extensions that are present in the abstract syntax of ATL are now provided in the

concrete syntax as well. Thus, the new keyword *apply* is ready for use. Nevertheless, one final detail needs to be addressed.

Syntax Highlighting

One important detail is still missing at the end of this process step. The new elements in the metamodel are defined and the corresponding textual representations are specified. Having accomplished this, it is possible to use the new keyword *apply* within the ATL editor. What has not been considered, though, is the colored syntax highlighting for reserved keywords in the editor. In order to provide syntax highlighting for the new keyword, one specific Java file needs to be modified. The String „apply“ is simply added to a String array and, as a result, the phrase *apply* is printed in bold, purple letters in the ATL editor.

4.4 Step 3: Defining the Operational Semantics

After extending the abstract and the concrete syntax of ATL with new constructs, the final step of the extension process is to define an operational semantics. A transformation defined using the extended ATL syntax cannot be processed by the existing compiler and virtual machine without modifying these components to support the extended syntax also in the runtime. This would require for a heavyweight extension affecting practically all ATL runtime components which would lead to a separate runtime. Please note that this additional runtime has to be maintained separately from the standard runtime. However, following the preprocessor approach discussed in Section 3.2, the desired behavior may be addressed using the standard ATL syntax. The preprocessor allows for the implementation of a more lightweight extension mechanism, leaving the ATL compiler and virtual machine untouched. A model transformation defined in the extended ATL syntax is simply preprocessed in order to produce standard ATL code. Before focusing on the actual definition of an operational semantics, the most important terms and concepts are introduced.

Operational Semantics

The operational semantics determines how the concepts of the extended ATL version are translated to concepts of the standard ATL version.
Example. The keyword *apply* is not provided by the original ATL syntax. In the extended syntax, however, this keyword is used to apply stereotypes of a UML profile to concrete UML model elements. To achieve this application of stereotypes in the standard ATL version, the corresponding code statements have to be generated. Listing 4.8 illustrates the application of the stereotype *Identifier*. To keep the example simple and understandable, all other parts of the transformation rule are omitted. Normally, of course, a target model element is specified in front of the keyword *apply*, cf. Listing 3.2.

Listing 4.8: Using the new keyword apply in ATL4pros

```
1 apply PRO!Identifier
```

This single-line statement including the new keyword is not executable in the standard ATL version. The corresponding statements that are needed for performing the actual application of the *Identifier* stereotype are depicted in Listing 4.9. The imperative *do*-block contains the two statements that need to be executed. As previously discussed in Section 3.1, the desired stereotype is obtained from the UML profile and, for later reuse, stored in the attribute helper named *stereo*. The invocation of the external Java UML2 operation *applyStereotype* finally links the specified target model element (*t*, in this example) to the stored stereotype *Identifier*.

Listing 4.9: ATL statements for applying a stereotype

```
1  do {
2    thisModule.stereo <- profile!Stereotype.allInstances()
3     -> any( e | e.name = 'Identifier');
4    t.applyStereotype(thisModule.stereo);
5  }
```

These two listings exemplify what is meant by and what is needed for the operational semantics. Apparently, such mappings between code statements in the one ATL version and compliant, executable code fragments in the other ATL version need to be specified for the entire ATL4pros extension.
There are different alternatives on how to define such an operational semantics. The two most evident approaches regarding the implemented ATL4pros extension are introduced below.

Defining an Operational Semantics: Two Possible Approaches

The operational semantics that is needed for the preprocessor approach (see Section 3.2) may be implemented in various ways. The two alternatives that are worth to be considered for the ATL4pros extension are the following:

- **Programming language Java**
  The high-level language Java may be used to modify the ATL transformation by adding the required code parts. This is possible since the Eclipse Modeling Framework provides applicable ATL model handlers. With these model handlers it is possible to create, load, and save ATL models, to add, delete, and modify ATL model elements and to read and write element properties. The allocation of the new statements within the existing ATL code may also be accomplished by Java methods. As this Java-based approach is escaping to a technology different than ATL, another possibility is explained below.

- **Higher-order transformation**
  Basically, a higher-order transformation works like any other model transformation: a given source model is transformed to some desired target model. The only difference is, as the term *higher-order* already indicates, that this type of transformation may be used for transforming a *source model transformation* to a *target model transformation*. In the case of ATL, a given ATL transformation may be transformed to a new ATL transformation, containing additional information. By this, new ATL statements may be inserted into a given transformation. The main advantage of this approach is that a higher-order transformation may be defined with ATL. This implies that no further technological knowledge is needed, saving both time and effort.

The decision to implement the operational semantics using the higher-order approach is based on the following reasons:

1. The definition of the operational semantics is done using ATL. As ATL is a familiar technology, it seems more reasonable to stick to this technology.

2. The possibility to fully explore and exploit the whole functionality of ATL seems very interesting. It offers the opportunity to use the refining mode of ATL which is normally not used when writing a standard ATL transformation. Thus, this approach offers the chance to acquire additional and deeper knowledge about ATL and gain further insight into the ATL refining mode.

3. The third reason is that the use of a higher-order transformation written in ATL is not mixing up different platforms. Although the model handlers that are provided by Java seem promising, the combination of different technologies is rarely working without problems.

The concepts and functionalities of a higher-order transformation are explained in the following section.

Higher-Order Transformation (HOT)

A definition of the term Higher-Order Transformation may be found in [49] and reads as follows:
„A Higher-Order Transformation, commonly abbreviated as HOT, is a model-to-model transformation that takes a model as input and generates a model as output. The input and/or output model itself is a transformation model.“
As ATL transformations are themselves models that conform to the ATL metamodel, they may be preprocessed by a HOT. In general, the execution of a HOT is not possible in cases where the transformation language is not based on a specific transformation metamodel.
For a better understanding of the higher-order transformation concept in the context of ATL, Figure 4.5 gives a typical example. The source as well as the target model are both ATL transformations. The functionalities of the three involved components *TCS injector*, *HOT*, and *TCS extractor* are explained below [48]:

- **TCS injector**
  The TCS injector takes an ATL transformation as input and generates the respective ATL transformation model as output. As the name of this component already suggests, the injection task is done using TCS (see Section 4.3). As illustrated in Figure 4.5, the produced ATL model conforms to the ATL metamodel.

- **HOT**
  The ATL transformation model serves as the input for the higher-order transformation, which then generates an ATL transformation model as output. In the example of Figure 4.5, the HOT is defined as an ATL transformation using the refining mode of ATL. By this, the HOT itself conforms to the ATL metamodel.

- **TCS extractor**
  The TCS extractor does the exact opposite of the TCS injector. The transformation model that is generated by the HOT is extracted to an ATL transformation. Thus, this step restores the textual representation of the ATL transformation.

As mentioned in [48], the TCS injector and extractor may not always be necessary for the execution of a HOT. If the source transformation is already in the form of a transformation model, no injection task is needed. This might be the case, for example, if several HOTs are executed in succession. Likewise, the extraction task may also be unnecessary in some transformation scenarios.

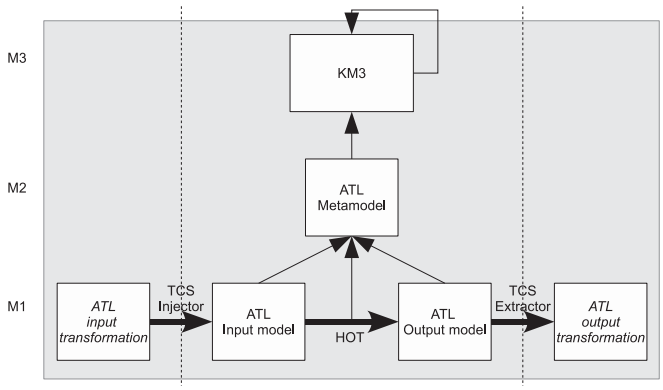

Figure 4.5: Example of a higher-order transformation [48]

HOT Patterns

The authors of [49] identify and analyze four different base *transformation patterns* that are associated with the use of HOTs. The distinction is drawn between (i) transformation synthesis, (ii) transformation analysis, (iii) transformation composition, and (iv) transformation modification. The distinguishing criteria are the source and the target models of the HOT. The HOT that was implemented for translating ATL4pros to standard ATL is considered a *transformation modification*.

Transformation modification. The criteria for this type of transformation pattern are as follows: 1) one model transformation servers as input to the HOT and 2) one model transformation is the output of the HOT. During the execution, the input transformation gets modified by the HOT in order to create the output transformation. The authors further divide the transformation modification pattern into six different subcategories. One of them is named *transformation language extension* and the implemented HOT is exactly of this type.

The survey conducted in [49], explaining the various application possibilities and the identified transformation patterns, shows that there is a certain demand for HOTs in the model transformation community. There are several ATL transformation examples available that make

use of a higher-order transformation, e.g., the ATL2BindingDebugger transformation[4] or the ATL2Tracer transformation[5]. These particular examples are mentioned as both belong to the *transformation modification* category and both of them make use of the ATL refining mode. Therefore, these two examples have been carefully analyzed prior to the implementation of the required HOT.

Implementing a HOT for ATL4pros

The definition of an operational semantics in the form of a HOT needs to be designed and planned carefully. The ATL code statements that are to be generated for the final standard ATL version have to be identified at the very beginning of the implementation work. In the case of the ATL4pros extension, the HOT needs to fulfill the following requirements:

1. The occurrence of the newly introduced *ApplyPattern* element in a given rule triggers the generation of an *ActionBlock* (*do*-block), if there is not already one. Subsequently, the statements for applying the stereotype are generated inside this *ActionBlock* (cf. Listing 4.9).

2. For the rule creating the UML element *Model*, an *ActionBlock* needs to be created, if there is not already one, and the statement for applying the UML profile needs to be added (cf. Listing 4.10).

3. The statements for setting the tagged values of a stereotype need to be generated as well. As these statements are added to an *Endpoint Rule* (please refer to Section 4.5), this special rule is created first and subsequently, the code parts for setting the tagged values are inserted.

4. All occurring *ApplyPattern* elements have to be removed from the transformation in order to conform to the standard ATL grammar. As a consequence, the contained *ApplyPattern-Elements* are deleted as well.

Listing 4.10: ATL statement for applying a profile

```
1  t.applyProfile(profile!Profile.allInstances().asSequence().first());
```

Implementing HOTs with ATL Refining Mode

A transformation written in the extended ATL version differs only slightly from a transformation written in standard ATL. Most of the model representing the extended transformation will remain without changes. However, with a model-to-model transformation operating in the normal execution mode of ATL, this is no advantage. Not only would it be necessary to create rules in order to apply the desired changes, but also to define rules that copy all the unmodified elements. As the metamodel of ATL is rather complex, the task of creating the so-called copy rules is quite

[4] http://www.eclipse.org/m2m/atl/atlTransformations/#ATL2BindingDebugger
[5] http://www.eclipse.org/m2m/atl/atlTransformations/#ATL2Tracer

time consuming and even worse, it is error-prone. Instead, this is the typical scenario where using the ATL refining mode [50] is appealing.
A transformation in refining mode is performed *in-place*. This means that the changes are directly applied to the input model for producing the target model incrementally. Using this refining mode, it is only required to define transformation rules for the elements that are actually changing. All model elements which are not matched by the transformation rules are kept as they are without the necessity of copying them. It is important to note that to avoid rule interaction problems, a transformation in refining mode is internally performed in two steps. The changes promoted by the rules are calculated in a first step without modifying the input model and applied afterwards in a second step.

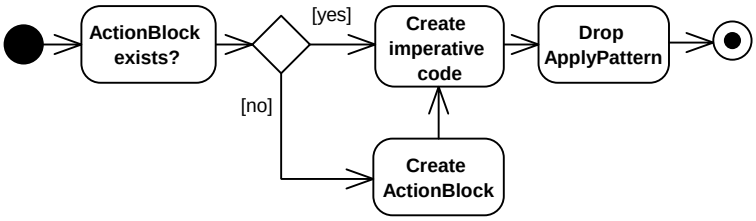

Figure 4.6: Rewriting process for transformation rules with ApplyPatterns

From ATL4pros to Standard ATL

Due to the aforementioned advantages, the preprocessor HOT is implemented in the refining execution mode of ATL. To reduce the complexity of the higher-order transformation, it is split into three successive steps. In the first step, all required *ActionBlocks* are created whereas in the second step, these *ActionBlocks* are matched and filled with the proper statements. Additionally, the endpoint rule is created and filled with the imperative code statements for setting the tagged values. In the third and last step, the *ApplyPattern* elements are eliminated. Figure 4.6 illustrates an overview of the process followed by the preprocessor HOT.

To get a better idea about the complexity of the implemented HOT, Listing 4.11 gives a concrete example of a code statement. This single-line statement is part of the code in Listing 4.9. It is generated and included in the standard ATL version in order to invoke the external Java operation *applyStereotype* that applies a stereotype to a UML model element.

Listing 4.11: ATL statement for applying a stereotype

```
1  t.applyStereotype(thisModule.stereo);
```

The corresponding excerpt from the HOT is illustrated in Listing 4.12. All these statements are needed for the creation of the simple statement in Listing 4.11.
The *CreateStereotypeApplication* rule matches all *SimpleApplyPatternElements* in the source model and subsequently, creates all the required ATL metamodel elements, namely instances of the types *ExpressionStat*, *OperationCallExp*, *VariableExp*, *NavigationOrAttributeCallExp* and *VariableDeclaration*. These elements represent the individual parts of the statement in List-

ing 4.11. For instance, lines 10 to 14 only accomplish the creation of the *applyStereotype* fragment.

Please note that this excerpt of the entire HOT is only responsible for the creation of the single-line statement itself. The accurate allocation within the *ActionBlock* is not carried out yet. For the correct nesting of different statements and the placement in the final transformation, the specification of complex operation helpers is needed. Due to the intricacy and the length of these helpers, they are not presented in this work.

Listing 4.12: Excerpt of the implemented HOT

```
1  rule CreateStereotypeApplication {
2   from
3    s : ATL!SimpleApplyPatternElement
4   to
5    t : ATL!SimpleApplyPatternElement,
6
7    expStat : ATL!ExpressionStat (
8     expression <- applySt
9    ),
10   applySt : ATL!OperationCallExp (
11    operationName <- 'applyStereotype',
12    source <- varExp,
13    arguments <- Sequence { navStereo }
14   ),
15   varExp : ATL!VariableExp (
16    appliedProperty <- applySt,
17    referredVariable <- s.applyPattern.outPatternElement
18   ),
19   navStereo : ATL!NavigationOrAttributeCallExp (
20    name <- 'stereo',
21    parentOperation <- applySt,
22    source <- variableExp
23   ),
24   variableExp : ATL!VariableExp (
25    appliedProperty <- navStereo,
26    referredVariable <- varDecl
27   ),
28   varDecl : ATL!VariableDeclaration (
29    varName <- 'thisModule',
30    variableExp <- variableExp
31   )
32 }
```

Drop Implementation

The last task of the HOT is to delete the *ApplyPattern*, cf. Figure 4.6. Listing 4.13 illustrates the corresponding deletion rule that is part of the HOT. This rule is needed so that the generated model transformation conforms to the standard ATL syntax. The deletion of the *ApplyPattern* element ensures the deletion of all contained *SimpleApplyPatternElements* since they are connected via a composition relationship.
It must be noted that this so-called *drop implementation* was not an official part of ATL at the time of the implementation work. It is a contribution that was proposed by members of the At-

lanMod team[6] and was kindly provided to me before it was even released. However, the drop feature has been integrated into the official ATL implementation recently. It is a useful feature that enables a deep deletion of ATL metamodel elements in the ATL refining mode [50].

Listing 4.13: HOT rule for dropping the ApplyPattern

```
1 rule deleteApply{
2  from
3   s : ATL!ApplyPattern
4  to
5   drop
6 }
```

Concluding Remark on HOTs

To conclude this final process step, it must be said that the specification of a HOT in the ATL refining mode is not a simple task. Quite the opposite is true: Although everything is written in the ATL language and the used concepts are familiar to an ATL engineer, some other severe and unexpected difficulties arise.

Iterations. The development of the higher-order transformation is achieved via several iteration steps. As a large number of ATL elements is needed for the creation of a short statement, a repeated execution of the HOT is helpful to check if all elements are existing and if the connections between those elements are set correctly. Therefore, the HOT may not be implemented in one single step but has to be defined in an iterative way.

Element matching. In the case of ATL4pros, it was difficult to specify which elements are to be created in which rule, i.e., which source model elements are appropriate matching subjects in order to create certain statements. In many cases it was necessary to define filtering criteria for certain source model elements and, depending on the outcome of the filtering, create different elements. To be more precise, the metaclasses *Module*, *MatchedRule*, *ActionBlock* and *SimpleApplyPatternElement* are all matched twice in the HOT, each with a particular filtering criterion.

Nesting of statements. The various statements that are created and inserted into the standard ATL transformation have to be nested. For this task, complex operation helpers are used that facilitate the correct interlinking of statements. The definition of these helpers turned out to be complicated and consumed a lot of implementation time.

Length of HOT. The creation of all required ATL statements led to a high number of lines of code in the HOT. The HOT for the proposed profile-specific extension comprises slightly more than 1,000 lines of code. Bearing in mind that the ATL4pros extension is rather small, the resulting dimension of the HOT is enormous. The entire HOT consists of 13 transformation rules. Each rule contains exactly one source model element (a larger number of source model

[6] http://www.emn.fr/z-info/atlanmod

elements is not supported by the refining mode) and a varying number of target model elements. The number of instantiated target model element differs for each rule, ranging from a minimum of 1 to a maximum of 42 target elements per rule. In addition to the rules, a total number of 11 operation helpers is required for the correct assembly of the created statements.

4.5 Implementation

The final section of this chapter presents two important details of the ATL4pros implementation. The first issue is concerned with the outsourcing of ATL code parts into an ATL library. The second issue is a bit more demanding: During the implementation work of this thesis, a number of problems and technical difficulties arose that were not apparent at the beginning of the extension. As most issues were only of minor complexity, they could be resolved very quickly. One problem, however, affected the entire higher-order transformation and resulted in a large and time-consuming redesign of the HOT. This main challenge was caused due to the existence of *bidirectional references* between stereotype applications. The detailed problem description as well as the successfully realized solution follow after the section discussing the ATL library.

Using an ATL Library

Due to the large dimension of the HOT the following decision was made: Complex operation helpers, needed in the ATL transformation for calculating certain values, are outsourced to an ATL library. This library gets imported into the ATL module and as a result, all helpers defined within this library may be used as if they were part of the actual module. This approach has two important benefits:

- The ATL library may be reused. In the case of the library which was specified for the ATL4pros extension, it may be reused for every transformation which makes use of the extended ATL syntax.

- The size of the HOT is reduced. For importing the library, only the short statement *uses libraryName;* needs to be created by the HOT. This saves a lot of code lines, reduces the complexity of the HOT, and improves maintainability.

Example. Lines 12 and 13 of Listing 4.16 illustrate the *setTaggedValue* operation which represents a call to a helper specified in an ATL library. This helper tests the given parameters and, depending on a sequence of if-then-else decisions, sets the tagged values of a stereotype by invoking the respective method of the underlying Java UML2 API.
Nevertheless, it is important to note that only selected code parts like generic helpers may be outsourced in order to reuse the library for other model transformations. Creating a new library for each new transformation is not conform to the basic intention of a library. Moreover, relocating too much ATL information into a library only leads to transformations that are hard to understand.

Transformation Challenge

A *bidirectional reference* between two stereotype applications *A* and *B* means that one tagged value of stereotype *A* is pointing to an instance of stereotype *B* and that one tagged value of stereotype *B* is in turn referencing an instance of stereotype *A*.
To give the reader a better idea of the problem, Figure 4.7 illustrates an example.

(a) The *DSL* metamodel contains the two metaclasses *Attribute* and *Identifier*. They are connected via a relationship that reads as follows: One *Attribute* may be contained in 0 to multiple *Identifiers* and one *Identifier* contains 0 to multiple *Attributes*.

(b) The corresponding part of the UML profile contains the UML metaclass *Property* that is extended by the two stereotypes *Attribute* and *Identifier*. The stereotype *Attribute* has a tag definition named *containedInIDENT* which is of the data type *Identifier*. The *Identifier* stereotype is analogously defined, holding the tag definition *containsATTR* which points to an instance of the *Attribute* stereotype. These two tag definitions constitute the bidirectional reference that is also present in the *DSL* metamodel.

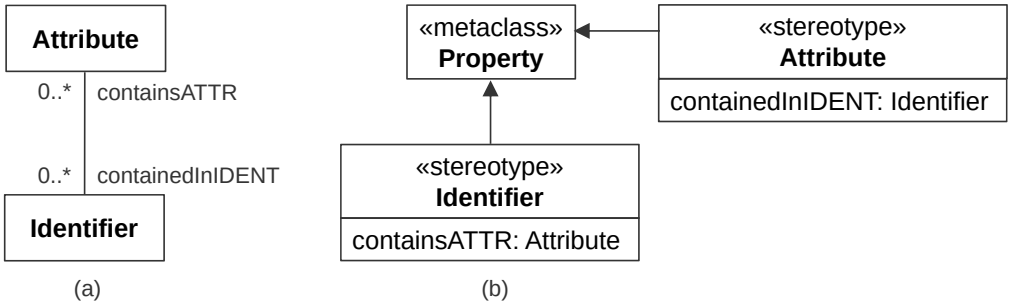

Figure 4.7: Bidirectional reference between the two stereotypes Attribute and Identifier

The depicted UML profile definition by itself is correct and not yet problematic. An ATL model transformation using this specific UML profile may be defined as follows (cf. Listing 4.14):

Att2Prop. One rule matches all *Attribute* elements of the *DSL* metamodel and generates the respective *Property* elements of the *UML* metamodel. In the imperative part of the rule, the *Attribute* stereotype is applied to the *Property* and the tagged value *containedInIDENT* is set (indicated by the corresponding comments on lines 7 and 8 of Listing 4.14).

Ident2Prop. Similarly, the second rule matches all *Identifier* elements of the *DSL* metamodel and creates the *Property* elements. As before, the stereotype *Identifier* is applied to the target model element and the tagged value *containsATTR* is set (cf. lines 18 and 19 of Listing 4.14).
Further, it is assumed that these two transformation rules are executed in the presented order. The Att2Prop rule executes first, followed by the execution of the Ident2Prop rule.

Problem. The problem that arises is due to the existence of the bidirectional reference between the two stereotypes. The instantiation of the tagged value *containedInIDENT* on line 8 requires for an instance of the *Identifier* stereotype. The problem is that these stereotype instances are not yet existing as they are only created in the following rule. Setting the tagged value *containsATTR* on line 19 works without problems as the stereotype instances for the *Attribute* are already initialized at this step of the execution. As a consequence, the resulting target model may only contain references from the *Identifier* to the contained *Attributes* but no links from an *Attribute* to the associated *Identifier*. Obviously, the generated target model is incorrect.

Listing 4.14: Problematic ATL rules due to a bidirectional reference

```
1  rule Att2Prop {
2   from
3    s : DSL!Attribute
4   to
5    t : UML!Property (...)
6   do {
7     -- apply stereotype 'Attribute'
8     -- set tagged value 'containedInIDENT'
9    }
10 }
11
12 rule Ident2Prop {
13  from
14   s : DSL!Identifier
15  to
16   t : UML!Property (...)
17  do {
18    -- apply stereotype 'Identifier'
19    -- set tagged value 'containsATTR'
20   }
21 }
```

Solution to the Challenge

The occurrence of a bidirectional link between two stereotype applications is not unusual, indeed this constellation ought to be quite frequent in large model transformations which make use of UML profiles. The solution to the discussed challenge applies a specific construct of ATL – the endpoint rule. The characteristics of this rule and the solution details are described below.

Endpoint Rule

An *endpoint rule* is a special kind of called rule in the ATL language. The endpoint rule, in contrast to a normal called rule, does not need to be explicitly called as it is automatically executed at the end of a transformation. The idea behind this type of rule is as follows: Feature assignments or other imperative constructs may need to be executed at the very end of a transformation. This may have different reasons, bidirectional references are only one possibility. Listing 4.15 illustrates a general example in reference to the problems depicted in Listing 4.14. This example shows how the instantiation of tagged values may be transferred to the endpoint rule.

Listing 4.15: Solution by means of an endpoint rule

```
1  rule Att2Prop {
2   from
3    s : DSL!Attribute
4   to
5    t : UML!Property (...)
6   do {
7     -- apply stereotype 'Attribute'
8    }
9  }
10
11 rule Ident2Prop {
12  from
13   s : DSL!Identifier
14  to
15   t : UML!Property (...)
16  do {
17    -- apply stereotype 'Identifier'
18   }
19 }
20
21 endpoint rule setTaggedValues {
22  do {
23    -- set tagged value 'containedInIDENT'
24    -- set tagged value 'containsATTR'
25  }
26 }
```

As before, the stereotypes *Attribute* and *Identifier* are applied to the UML *Property* within their associated rules. But, in contrast to Listing 4.14, the two tagged values *containedInIDENT* and *containsATTR* are set in the endpoint rule of the transformation, cf. lines 23 and 24.
With this solution, the occurrence of bidirectional links is no longer a problem. The handling of tagged values is transferred to the endpoint rule. As the endpoint rule is executed at the end of a transformation, all required stereotype applications (that is, the stereotype instances) are certainly existing. In the example above, the target model elements with either of the two stereotypes applied, are created during the execution of the two rules. Thus, establishing a link from a certain tagged value to an existing stereotype instance is applicable.

Endpoint Rule in ATL4pros

Due to the huge benefits of an endpoint rule regarding the occurrence of bidirectional references, this construct is also embedded into the ATL4pros extension. More precisely, the endpoint rule is generated and integrated when transforming the extended ATL4pros version to standard ATL code. This automatic generation of ATL code is the responsibility of the HOT. Listing 4.16 presents an excerpt of the endpoint rule which is created for the final ATL transformation.
The endpoint rule of Listing 4.16 consists of one imperative *do*-block that contains a *for*-loop. This loop iterates over all existing *Identifier* instances of the source model. For each source instance *s*, the corresponding target instance *t* is fetched via the *resolveTemp* operation and, if the target instance really exists, this target model element is stored in the *targetEl* attribute helper. Subsequently, the two tagged values *phase* and *contATTR* of the stereotype *Identifier* are instantiated with the respective values taken from the features of the source model element.

Due to space limitations, the aforementioned *containsATTR* feature is trimmed to the name *contATTR*.

Listing 4.16: Endpoint rule example

```
helper def: stereo : uml!Stereoype = OclUndefined;
helper def: targetEl : UML!Element = OclUndefined;

endpoint rule EndRule() {
 do {
  thisModule.stereo <- thisModule.allStereotypes->any(e | e.name = 'Identifier');

  for(s in DSL!Identifier.allInstances()) {
   if(not thisModule.resolveTemp(s, 't').oclIsUndefined()){
    thisModule.targetEl <- thisModule.resolveTemp(s, 't');

    thisModule.targetEl.setTaggedValue(thisModule.stereo, 'phase', s.phase);
    thisModule.targetEl.setTaggedValue(thisModule.stereo, 'contATTR', s.contATTR);
   }
  }
 }
}
```

By this solution, an additional execution phase is simulated. By default, the *module initialization phase* is followed by the *matching phase* and the *target model elements initialization phase*. With the created endpoint rule, a simulated fourth phase follows at the end of the execution. A drawback of the endpoint rule is, however, that a costly iteration over all source elements and the subsequent trace resolution for fetching the respective target elements is needed. A more elegant solution may be to apply the stereotypes already during the matching phase, but this would require for a reconstruction of the ATL virtual machine and the compiler.

CHAPTER 5

Evaluation

An extension of any (modeling) language should always be accompanied by an evaluation that investigates different quality aspects in order to guarantee a holistic analysis and to disclose the extensions' improvements. The following quality parameters may be considered:

1. The first subject of evaluation should concentrate on the expressivity of the language extension. Basically, an extension should at least provide the same level of expressive power or, even better, increase the level of expressivity. In the case of ATL4pros, an evaluation of the expressivity is not required due to the following rationale: The invocation of different UML2 API operations is necessary for setting profiles, stereotypes, and tagged values. These API calls, which are basically still possible but not necessarily needed in an ATL4pros transformation, are automatically generated by the HOT. Therefore, the generated model transformation has exactly the same expressiveness as a standard ATL transformation and thus, no further evaluation is required.

2. Secondly, the focus may be set on quality attributes which are reflecting a transformation's internal structure. In order to determine the improvements of using ATL4pros instead of using standard ATL, evaluating quality attributes of ATL transformations may be the key for this task. In particular, different metrics are computed for the original transformation (i.e., the transformation of the original case study, see Section 1.5), for a refactored version of this original transformation, for the ATL4pros version, and for the transformation that is generated by the HOT. As the aim of ATL4pros is to have concise instead of verbose transformation code, the calculated metric values should reflect these improvements.

3. In addition to the use of metrics for quality attributes, also the performance in terms of the execution time of the aforementioned transformations may be evaluated and compared. Possible explanations for differing execution times need to be identified and, based on the outcome of these analyses, conclusions are drawn.

The following two sections address the evaluation based on code quality attributes on the one hand and the evaluation of the execution performance on the other hand.

5.1 Evaluation based on Code Quality Attributes

The calculation and comparison of different metrics is a well-established evaluation method for model-to-model transformations [28, 51]. The metrics comparison conducted for the ATL4pros extension is comprised of different metric features. The standard metric lines of code (LoC) and also other measures that are computed by reusing a transformation from van Amstel et al. [52] are taken into account. The following ATL transformations are considered:

T1. T1 is the original ATL transformation of the ModelCVS case study.

T2. T2 is a refactored version of transformation T1. The refactoring includes the following steps: A sophisticated operation helper that assures the correct setting of tagged values by differentiating between simple, complex, single-, and multi-valued attributes is outsourced into an ATL library. Subsequently, this library is inserted into T2 and the defined operation helper is invoked whenever a tagged value has to be set. It is important to note that the defined ATL library (38 LoCs) is, in a marginally extended form (41 LoCs), also reused in transformation T4.

T3. This transformation is defined with the ATL4pros syntax. As this version is not executable, the HOT is used to transform it to transformation T4. Consequently, this transformation is not part of the evaluation concerning the execution time in Section 5.2.

T4. Transformation T4 is the generated model transformation resulting from the HOT execution. Like T2, also this transformation makes use of an external ATL library containing the complex operation helper for setting tagged values and an additional attribute helper for storing all existing stereotypes of the used UML profile. Moreover, this transformation contains the endpoint rule that is responsible for the correct setting of bidirectional references.
Please note that all four transformations are defined in such a way that, given one specific input model, the same target model may be generated. Whether this requirement is met by the different transformations or not is also part of the following discussion.
Table 5.1 illustrates the gathered results of the metrics evaluation. In the following, the different metrics will be discussed in more detail.

Lines of Code. As can be seen in the entries of the LoC measure, the original transformation T1 is the largest one whereas the ATL4pros version T3 is the smallest one. Thus, the aim of having *concise transformation code* is clearly achieved. Due to the existence of the keyword *apply*, and the intuitive way of setting tagged values, the size of the transformation is kept very small. Thereby, the maintainability of the entire transformation is improved significantly.
The generated transformation T4 has, compared to the refactored version T2, more lines of code. However, there is an important reason for this: Both transformations T1 and T2 are not able to

Metric	*T1*	*T2*	*T3*	*T4*
LoC	370	211	186	262
LoC (declarative)	93	95	186	100
LoC (imperative)	277	116	0	162
Ratio declarative/imperative	25 / 75	45 / 55	100 / 0	38 / 62
#Elements	1436	901	383	1210
#Links	3183	2077	892	2632
#Rules	10	10	10	10
#Bindings	13	13	95	13
#Helpers	1	1	0	2

Table 5.1: Results of the metrics evaluation

handle bidirectional references correctly. It is only possible to set one side of the reference instead of both sides. As a result, the generated target models of T1 and T2 are incorrect. A proper instantiation of bidirectional references is only guaranteed by the endpoint rule of transformation T4. The task of this special rule is to fetch the target model elements that were generated by the matched rules and use them for setting the tagged values of a stereotype. Please remember that the endpoint rule is executed at the very end of a transformation and thus, all elements that were instantiated beforehand are not directly available. To make use of these elements, they have to be queried again, resulting in additional code statements. Even though this endpoint rule requires for a few more code statements, the benefits of setting bidirectional references are of more importance.

A comparison between declarative and imperative lines of code illustrates another benefit of the ATL4pros extension. Please remember that declarative code is the recommended ATL programming style. As may be seen in Table 5.1, the ATL4pros version is doing best, consisting of declarative code statements only. With 38 percent of declarative code, the generated transformation T4 is also doing very well. Comparing T4 to T2, the higher ratio of imperative code is again due to the endpoint rule since it contains imperative code only. But again, the huge benefits of the endpoint rule compensate for the slightly higher number of imperative statements.

The LoC metric should always be handled with care, as discussed in [21]. It is not a standardized measure since there is no default specification whether to use the number of physical or the number of logical code lines. The LoC value is only focusing on the concrete syntax level without considering the abstract syntax level. Having a language like ATL, consisting of concrete as well as abstract syntax, the code formatting on the concrete (i.e., the textual) level may have major impacts on the line numbers. Thus, measuring the number of abstract syntax elements may be considered more significant, and is therefore discussed in the next paragraph. Another problem associated with the LoC value is the fact that the number of code lines is highly depending on the power of a language. Comparing one and the same program written in different languages will eventually lead to an unequal and imprecise result.

Number of Elements and Links. The number of elements, that is, the number of instantiated abstract syntax elements, has a significantly small value for transformation T3. A transformation written in the ATL4pros syntax implies that only 383 elements have to be instantiated. In comparison, transformation T2 leads to the instantiation of 901 abstract syntax elements. The reason why T4 requires for a higher number of elements than T2 is again due to the use of the endpoint rule. Retrieving the generated target model elements in order to use these elements, requires for an additional attribute helper and additional model queries. As a result, more abstract syntax elements have to be instantiated.
The same argument is also true for the number of links, i.e., the number of instantiated references between the abstract syntax elements. As the ATL4pros version of T3 is the most concise transformation, also the number of required links is very small. As before, the use of additional elements in T4 and thus, the existence of additional links between those elements, is connected to the endpoint rule.

Number of Rules, Helpers, and Bindings. The number of transformation rules remains unchanged for all four transformations. The number of binding statements is also equal for transformations T1, T2, and T4. The high number of bindings in T3 is based on the fact that all tagged values are now defined like normal bindings. Thus, the total number of 95 bindings arises from 13 normal bindings (as in T1, T2, and T4) and 82 tagged value bindings. The number of involved helpers is also not differing considerably with T4 containing one additional helper and T3 containing no helpers at all.

The presented metrics evaluation clearly demonstrates the good quality of the implemented ATL4pros extension. A transformation conforming to the extended ATL syntax may be written using nothing but declarative code. Not only is declarative programming the recommended style in ATL, but also the size of the transformation in terms of LoC is largely reduced due to the absence of complex imperative code statements. This in turn increases both readability and maintainability of the transformation.
The use of the endpoint rule, which leads to a slightly higher number of instantiated elements and links, has great benefits for the ATL engineer. As outlined in Section 4.5, the existence of bidirectional references is completely handled by the generated endpoint rule and thus, the transformation engineer does not have to take care about the correct invocation of statements.
Another advantage that is not captured by the metrics evaluation is connected to the implementation time of an ATL transformation. The ATL4pros syntax eases the use of UML profiles by the domain-specific keyword *apply* and, as a result, the required time and afford for defining a transformation is reduced to a great extent.
After having discussed the most important code quality attributes, it is time to focus on an evaluation in terms of execution performance.

5.2 Evaluation based on Execution Performance

The performance evaluation is based on the execution time of the three aforementioned model transformations T1, T2, and T4. Additionally, a speedup value is captured which indicates the

ratio between the execution time of the original transformation T1 and the two other transformations, T2 and T4 respectively. Transformation T3 is not evaluated as the extended ATL4pros syntax is not executable on the ATL virtual machine.

Setting. The performance analysis is accomplished by using the same source model for all three transformations and for all execution runs. The resulting execution time is calculated as the average result of ten execution runs. The first run, however, is always omitted due to the fact that it requires considerably more time compared to the successive runs. The source model consists of 170 model elements that have to be transformed. Apart from the generation of the respective target model elements, each target element is specialized by a stereotype. Additionally, tagged values are set for each stereotype, ranging from a minimum of 3 to a maximum of 19 tagged values per stereotype. The execution is done using the EMF-specific Virtual Machine of ATL.
Table 5.2 summarizes the execution times (in seconds) and the respective speedup ratio, comparing the execution time of T1 and T2, and the one of T1 and T4 respectively.

Transformation	***Execution time***	***Speedup***
T1	1.50 s	1.00
T2	1.91 s	0.78
T4	1.45 s	1.03

Table 5.2: Results of the performance evaluation

The results of the performance analysis show that the generated transformation T4 performs best, with an average execution time of 1.45 seconds. The original transformation T1 is close behind, with an average execution time of 1.50 seconds. The execution time of the refactored transformation T2 amounts to 1.91 seconds and thus, it is the slowest one. The reason for this performance loss is as follows: As mentioned before, transformation T2 makes use of a library which contains an operation helper for setting the tagged values. This helper includes a number of if-then-else branches and also type checking invocations. Thus, this operation is very expensive in terms of execution time and is therefore responsible for the poor performance value. Even though T4 is using the same operation helper as T2, the execution performance of T4 is largely increased by the additional attribute helper. The attribute helper stores all existing stereotypes and thus, it is not required to query the profile every time a stereotype is applied, as it is the case in T2. As a result, transformation T4 is working more efficiently than transformation T2.
Based on the outcomes of the execution time, the speedup ratio is calculated. The speedup between T1 and T2 has a value of 0.78, indicating that the second transformation is with a ratio of 0.22 slower than the first one. On the contrary, a comparison between the execution times of T1 and T4 shows that T4 is with a ratio of 0.03 faster than T1. Even though this speedup value is not significant, it is important to remember that transformation T1 is not producing a correct target model due to the absence of bidirectional references. This implies that transformation T4 provides more functionality, and is still performing slightly better than transformation T1.

Based on the outcome of the metrics evaluation and the performance evaluation, it may be concluded that the higher number of instantiated elements and links of transformation T4 is not diminishing the performance of this transformation. Even though a larger number of code statements is required within the endpoint rule, this has no adverse effects on the execution time. The developed preprocessor in form of the higher-order transformation and the defined ATL library are specified in such a way that the generated ATL transformation benefits from several ATL best practices and other tricks. Accordingly, the resulting model transformation is highly optimized and efficient in terms of code quality and also execution performance.
To conclude this section, it must be said that the extension of ATL with a profile-specific keyword in order to facilitate the use of stereotypes and tagged values within an ATL model transformation offers lots of benefits, and that these benefits are also reflected in the values of the metrics evaluation.

5.3 Lessons Learned

During the implementation of the profile-specific ATL4pros extension, technological possibilities as well as limitations were encountered and summarized. This section presents a set of lessons learned, experienced while extending ATL for natively supporting UML profiles. This compendium of lessons learned may be seen as a design guideline for future extension work and as a starting point for a critical discussion about the extensibility of ATL.

Defining extension points in the ATL metamodel. It is important to define appropriate places for the new metamodel elements within the ATL metamodel. As Figure 4.2 partly depicts, the entire metamodel of ATL is rather large, containing 87 metaclasses in total. When investigating the ATL metamodel, it becomes clear that the abstract class *LocatedElement* has a significant role: Every other class directly or indirectly inherits from this superior class. Given this fact, the first natural conclusion is to define all new elements as direct or indirect subclasses of the *LocatedElement* class. The exact allocation of new classes by means of containment references as well as non-containment references to other classes depends on the purpose of the extension and has to be determined by the extension engineer from case to case. Also, the references between the extension classes differ for every extension. There is no written guideline or manual that may be consulted in order to find the right place or arrangement. Another important implication is concerned with the modification of the original ATL metamodel. Basically, the extension of existing metaclasses by adding new metaattributes or relationships should be minimized. Even though an extension needs to be integrated into the standard language, the actual integration should not affect the original constructs in the abstract syntax too much. This is due to the fact that all modifications of the abstract syntax need to be included in the textual concrete syntax as well. Thus, a large number of modifications in the abstract syntax leads to an extensive and time-consuming rework of the textual concrete syntax.

HOTs as in-place transformations. As mentioned in Section 4.4, the implemented HOT is designed as an in-place transformation using the ATL refining mode. Every single ATL code fragment that is needed in the standard ATL language to represent the introduced keywords of

the extended ATL version, has to be created by instantiating the corresponding metamodel elements. As a vast number of such elements is required for only short statements, the definition of the HOT turned out to be a complex and time-consuming task. To give the reader an idea about the dimension of the higher-order transformation, the entire HOT involves approximately 1,000 lines of code for supporting only small language extensions. As a consequence, before building the HOT, the aim should be to out-source as much reusable parts from the standard ATL transformations as possible into an ATL library. This library gets imported into the standard ATL transformation and thereby, all helpers defined in the library may be used. This helps a lot in keeping the HOTs small, which has in turn a major impact on both the development and the maintainability efforts. When it comes to the transformation of model transformations themselves, there are not many alternatives that have the same expressive power as HOTs. The possibilities of escaping to high-level languages like Java or using totally different approaches are currently not satisfactory as additional technological knowledge is needed. Further improvements for developing in-place HOTs may be explored similar to what has been done in [48] for model-to-model HOTs.

Test-driven development. An important characteristic that needs to be investigated refers to the correctness of the implemented extension. Therefore, an appropriate test suite may be required for assuring a test-driven development of ATL extensions. The workflow for ensuring the correctness of the extension is basically as follows. The standard ATL transformations, which are used for abstracting the ATL extension, are executed on sample source models. These transformations consequently create corresponding target models. As a starting point for the correctness test, the transformations expressed in the extended syntax are transformed to standard ATL transformations based on the implemented HOT. Afterwards it is possible to execute the generated ATL transformations on the same sample source models as before, resulting in corresponding target models. Assuming that the extension and the HOT are working correctly, these target models need to be equivalent to the target models generated by the initial transformations. This test-driven development allows for building an extension in an incremental and iterative process, meaning that the additional language features are introduced and tested consecutively. For such a framework, previous work on model transformation testing [33, 35] seems to be an appropriate basis, but has to be extended for testing HOTs. One important building block may form model transformation orchestration languages [43] for modeling the testing process.

ATL extension framework. Unfortunately, the ATL language is not designed to be extended by new concepts. While implementing the presented extension of ATL, it became apparent that an integrated ATL extension framework is needed. This is merely due to the fact that the tooling is quite time-consuming. To be more specific, different versions of ATL plug-ins are required for implementing the extension, and a lot of copy-and-paste tasks have to be performed. It turns out that a wizard-driven extension may be more appropriate to keep the development/test cycles short. An integrated extension framework may provide all required artifacts (that is, metamodel, TCS artifact, artifact for syntax highlighting, etc.) in one place, and automatically copy the generated files into the right folders and set the correct connections. Having such an extension framework at hand, probably more researchers would start to experiment with introducing new

language features in ATL and providing domain-specific preprocessors for different domains. Such experimental implementations of new language features may also provide valuable input for the general evolution of ATL.

5.4 Limitations of ATL4pros

Implementing an extension for the ATLAS Transformation Language was a very interesting and challenging experience. The envisioned extension was realized and finished as planned. Nevertheless, there are a number of limitations to the current ATL4pros extension that are based on technical restrictions or other constraints. These limitations are discussed in the next paragraphs.

Traceability information. An initial hope at the beginning of this master's thesis concerned the ability to store traceability information between elements of the extended syntax and their counterparts in the standard syntax of ATL. This type of information may disclose all connections between new and existing language constructs.
On the one hand, traceability information is useful for reproducing how a specific keyword in ATL4pros was converted by the higher-order transformation to standard ATL statements. This may enhance reusability and may help in the implementation phase during debugging. On the other hand, the efficient discovery of errors may be possible. Consider the following example: An ATL engineer defines a new model transformation in ATL4pros. After completion, the HOT is used to convert the extended transformation to the standard form. If the execution of this standard transformation results in errors, the error messages are only addressing the lines of code in the standard ATL version but not the line numbers of the extended ATL transformation. However, it may be more helpful for the ATL engineer if the error messages were related to the ATL4pros version.
The execution of a model-to-model transformation in the standard execution mode of ATL automatically produces trace links between source model elements and the generated target model elements. Thereby, the relationship between each source and each target element is clear and may be used for further actions. Unfortunately, there is no such traceability information provided for the refining execution mode. The only imaginable possibility would be to set up an independent metamodel that contains particular metaclasses for fetching different internal ATL information. During the execution of the HOT, instances of these metaclasses may be generated and stored in a target model that conforms to the additional metamodel. As this functionality would go far beyond the scope of this thesis, the production and storage of traceability information is not facilitated.

Limitations regarding ATL elements. A number of limitations are directly connected to certain elements within an ATL transformation. These constraints have to be taken into account when defining an ATL transformation in the extended ATL4pros version:

- **Only matched rules are considered**
  The current implementation of ATL4pros is limited to matched rules. This means that the keyword *apply*, the new construct for applying a stereotype to a UML model element,

may only be used within a matched rule. Consequently, called rules and lazy rules are not yet supported and target model elements that are created by these two kinds of rules may not be specialized by a stereotype.
The reason for this restriction is related to the complexity of the HOT. At the beginning of the specification, the entire HOT was designed for the handling of matched rules only. This imposed restriction may come as no surprise since this type of rule is the most important one. Every matched rule needs to have exactly one *InPattern*, which in turn consists of at least one *InPatternElement*, i.e., one source model element. Information related to the source model element was used for putting the final HOT together. Called and lazy rules, however, do not have such a source model element and as a result, the defined HOT may not work on these rules.

- **One source model element per rule**
  Talking about source model elements, there is a second limitation in relation to this part of an ATL transformation. The current version of the ATL4pros extension is only applicable in cases where the matched rule contains a single source model element.
  This constraint is again arising from the intricacy of the HOT. Creating statements in the standard ATL version requires for the direct generation of all parts of the statement. These individual parts need to be connected and, in some cases, the connection may only be established by navigating to the specified source model element. Currently, the HOT is only prepared for the occurrence of one source model element. Thus, a transformation rule with two or more source model elements may cause problems. However, this limitation has only limited impact since usually, the majority of all ATL transformations feature only a single source model element.

- **Rule hierarchy has no impact**
  It may be desirable to apply an arbitrary stereotype to a UML element inside an abstract matched rule and, as a result, the stereotype is also applied to all related model elements inside the concrete subrules. Unfortunately, this scenario is not yet supported by the ATL4pros extension.
  As before, the reason that this feature is not available in the extended ATL version is merely due to the complexity level of the HOT. The difficulty is in finding the appropriate matching subjects for creating all desired code statements. As one specific type of source model element may only be matched once, the options are very limited. In addition, the necessity of this hierarchy-based feature has not been considered at the very beginning of the HOT definition. The integration of this property may not have been possible at a very advanced stage of the implementation work without facing severe problems.

Higher-order transformation. The last issue is more a logical rather than a technical limitation. The higher-order transformation that was created for the ATL4pros extension is only usable for this special scenario. This is due to the fact that all elements which are created by the HOT are tailored to this specific extension purpose. A different domain-specific extension requires for the creation of a different HOT.

A reuse of the HOT, or at least of parts thereof, may be very beneficial since its development has taken a lot of time. Unfortunately, such a reuse is not feasible from the current perspective.

CHAPTER 6

Related Work

The related work is divided into three parts. First, an overview of available model transformation approaches is given. Second, an orthogonal application of using UML profiles in the context of model transformations is discussed. And third, works concerning the automatic generation of UML profiles and transformations are presented.

6.1 Overview of Model Transformation Approaches

The notion of model transformation is an essential part of the model-driven paradigm. Transforming one model to another model may be due to a variety of application purposes, based on distinct metamodels, and also accomplished using different approaches. Besides ATL, *QVT* [38] and *Graph Transformations* [18] are the two most important representatives. The next two subsections give a short introduction to the mentioned technologies in order to understand their functionalities as well as their differences compared to ATL.

Query/View/Transformation

The MOF 2.0 Query/View/Transformation[1] (QVT) approach was specified by the OMG and is currently under version 1.1. Similar to ATL, QVT is a comprehensive, standardized language for defining model transformations. Although these two approaches share the same goal, namely transforming a given source model to a desired target model, the way of how a transformation is defined differs. The most important concepts of QVT are explained below. Basically, QVT is divided into three named language levels:

- **Core**
  The Core language of QVT is a small language that only provides pattern matching over a flat set of variables. Conditions over those variables are evaluated against a set of models [38]. The Core language is equally powerful to the Relations language, but, as it

[1] http://www.omg.org/spec/QVT/1.1/

is simpler, a transformation defined with the Core language is more verbose than one defined with the Relations language. Trace information between transformed model elements need to be explicitly defined.

- **Relations**
  The Relations language (QVT-R) is the most prominent part of the QVT standard. With this language it is possible to define relationships between MOF models and elements of these models. As stated in the QVT specification, this language supports complex object pattern matching. More information and a simple example of the Relations language is given below.

- **Operational Mappings**
  The Operational Mappings language provides imperative implementations, meaning that a transformation may be defined in a completely imperative style, or that imperative parts may be used within the Relations language. Imperative parts are used whenever complex statements may not be expressed in a declarative way. Operational mappings are comparable to imperative ActionBlocks in the ATL language.

As in the case of ATL, also QVT is a hybrid language, meaning that it supports declarative and imperative code parts. The Relations and the Core language constitute the declarative components whereas the Operational Mappings and the Black Box are considered as imperative parts. Figure 6.1 depicts the relationship between the involved language levels.
The Black Box implementations make it possible to plug-in additional, external mechanisms. Thus, it is possible to define complex algorithms with any programming language that has a MOF binding. By this, the calculation of model property values may be accomplished by using external domain-specific libraries. As some calculations may be too difficult to be expressed with OCL, an escape to a high-level programming language like Java is a helpful possibility.

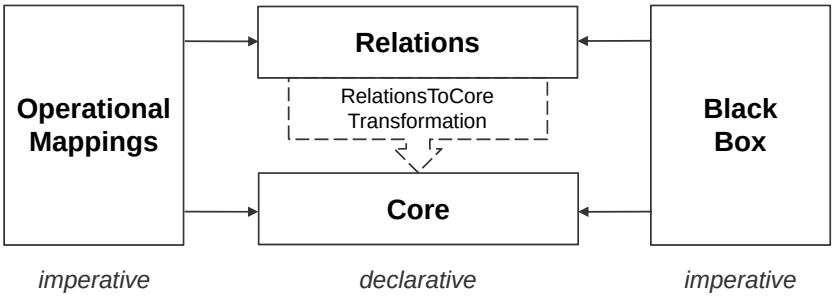

Figure 6.1: Relationships between QVT language levels [38]

QVT-Relations (QVT-R)

The Relations language is used for a declarative specification of the relationships between so-called candidate models, i.e., source and target models. A transformation defined in QVT-R

usually consists of multiple *relations* which contain the details of the transformation. The relations are used to specify certain constraints which have to be met by elements of the candidate models. If the relations hold, the transformation may be executed successfully. The *domains* within a relation (cf. Listing 6.1) address specific element types of the candidate models and describe the correlations (*patterns*) that must hold between those elements. With the relations and the domains of the QVT-R language, it is possible to check two models for consistency, to update a (target) model or to transform a model.
The Relations language has two different notations, namely a graphical and a textual syntax. Another important fact is that, in contrast to the Core language, the Relations language implicitly creates trace information to capture the occurrences during a transformation execution. As the Relations language may not be executed directly, it has to be transformed to the Core language, see Figure 6.1. For details about this transformation please refer to the QVT specification [38].

Example. The ATL transformation example of Listing 2.3 may also be defined in the QVT-R language. The corresponding transformation code is outlined in Listing 6.1.

Listing 6.1: QVT-R example

```
1 transformation datamodelUml (datamodel : DSL, uml : UML) {
2  relation DataModelToModel {
3   domain datamodel d:DataModel { name=dn }
4   domain uml m:Model { name=dn }
5  }
6 }
```

In this example, the transformation is named *datamodelUml* and it is comprised of two candidate models, namely *datamodel* and *uml*. The candidate models are typed with a specific metamodel package, *DSL* and *UML* in this example. The relation *DataModelToModel* contains two domains that match *DataModel* instances of *DSL* models and *Model* instances of *UML* models respectively. The defined domain patterns are simple and require for the values of the elements' *name*-property to be equal.
In comparison with ATL, the *transformation* is equivalent to a *module* and the *relation* correlates to a *rule*. Accordingly, the *domains* specify the source and the target model elements used for the execution of the transformation. For a complete definition of all QVT functionalities please refer to the QVT specification [38].

QVT is a comprehensive model transformation toolkit that facilitates the definition and execution of model transformations. With its declarative and imperative constructs, it offers a great number of possibilities to transformation engineers. Nevertheless, it suffers from a severe shortcoming in relation to UML profiles – they are simply not supported in the language. Hence, applying profile-specific information to annotate target model elements is not possible with QVT. Compared to ATL, this total lack of profile support is a huge disadvantage and may be one explanation for the greater popularity of ATL.

Graph Transformation

Another relevant model transformation approach is known as *Graph Transformation* (GT). The connection between model transformations and graph transformations is as follows: *„Models can easily be seen as graphs“* [18].
Thus, the notion of graph transformations may be applied for different types of model manipulations, including model transformations. To make the correlation between the different concepts more explicit, the following comparisons may be drawn:

- **Type graph**
  A *type graph* is comparable to a *metamodel*, representing the conceptual level with possible generalizations (abstractions) of elements.

- **Graph**
  A *graph* is conform to a type graph and thus, it represents a *model* which is conform to a metamodel.

The basic terms graph, directed graph, typed attributed graph and graph transformation rule are explained in the following paragraphs.

Typed Graph

A *graph* consists of two disjoint sets: a set of vertices V and a set of edges E. In a *directed graph*, the vertices are connected via directed edges, thus, each edge has a source and a target vertex. For being able to represent model information, *typed attributed* graphs are required. This means that each vertex and each edge has (1) a type and (2) may contain a number of attributes. In order to illustrate typed attributed graphs in a model-like manner, it is common practice to represent a graph as an object diagram and the type graph as the corresponding class diagram.

Graph Transformation Rule

A graph transformation rule p is a structure-preserving mapping between two graphs [27] and is defined as follows: $p\colon L \rightarrow R$
L defines the left-hand side (LHS) of the rule, R the right-hand side (RHS). Both L and R represent two graphs which participate in the graph transformation. The LHS of a graph transformation rule describes the pre-condition whereas the RHS describes the post-condition. In other words, the LHS describes which conditions (vertices and edges) must hold before the transformation is executed, and the RHS illustrates what has to be available after a successful transformation. The execution of a graph transformation rule p results in a graph transformation t.

Example. The following example in Figure 6.2 is taken from [11] and shows a very simplified form of a graph transformation.
The example illustrates the loading of a container onto a truck. As it may be seen in the figure, most parts of the graph remain unchanged as only two edges are affected. Basically, vertices

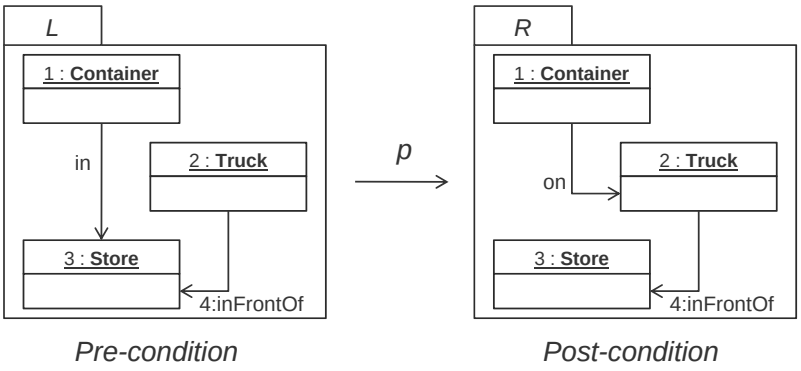

Figure 6.2: Graph transformation rule for loading a container onto a truck [11]

and edges may be deleted from and others may be added to the graph. In this example, the edge *in* gets deleted whereas the edge *on* is added to the graph. All other parts of the graph remain unaltered.
Apart from deleting and adding vertices or edges, graph transformations may also contain more sophisticated functionalities. Having directed, typed and attributed graphs, it is possible to calculate certain values based on the given attributes. Thus, constants and variables may change or OCL expressions may be evaluated in the course of a transformation.

To summarize the presented approach, it needs to be said that model transformations and graph transformations share the same idea but offer different functionalities and have diverse characteristics. Both approaches have their supporters and are important ingredients of model-driven engineering. In general, a model-to-model transformation may be expressed as a graph transformation, given that a model represents a graph and a metamodel represents the associated type graph. Graph transformations are able to handle complex transformation scenarios by providing mature techniques and a wide range of functions. However, graph transformations do not support UML profiles natively. Profile-specific information has to be integrated directly into the definition of a graph transformation and thus, the separation between a model and a UML profile is lost.

6.2 UML Profiles for Supporting Model Transformations

UML profiles are not limited to be directly used within a model transformation for applying profile-specific information to UML model elements. In [44], a completely different application scenario of UML profiles in conjunction with model transformations is introduced. The starting point is the idea of modeling model transformations with UML and therefore, UML profiles are needed. More precisely, the aim is to translate a given UML profile instance to a model transformation. Hence, the profile is not part of the model transformation itself but rather the basis for the transformation.
The SDM (Story Driven Modeling) metamodel, which may be used for defining a model trans-

formation, requires for the use of a specific tool (Fujaba). Subsequently, the model transformations based on the SDM metamodel are only executable in the Fujaba development environment. To overcome this drawback, the authors developed a template-based code generator that accepts a UML profile for SDM as input and generates a model transformation as output. The two components are now briefly discussed.

UML profile for SDM. As the SDM metamodel and the corresponding profile need to share structural commonalities, each SDM component is mapped to a UML counterpart. Moreover, specific stereotypes are used to represent all types of component variants found in the SDM metamodel. This mapping assures that all modeling elements of the SDM metamodel remain available. The UML profile may be designed with any CASE tool, meaning that the transformation specification is no longer bound to the Fujaba editor. Additionally, any standard-based MOF repository may be used for storing the stereotyped UML model, since the UML is an instance of the MOF meta-metamodel.

Code generator. Having a model transformation based on SDM, the code generation is fully dependent on the Fujaba environment. In contrast, the transformation models based on UML and the UML profile are translated by using a standard API for model access: the Java Metadata Interface[2] (JMI) standard. The code generator used by the authors first of all analyzes and, subsequently, transforms the given transformation model to the desired transformation code by invoking JMI calls. By this, the code generator is no longer limited to one specific tool.

This approach clearly demonstrates that UML profiles may be leveraged for a wide variety of application areas. Due to its standardization and its huge success in combination with UML, the profile mechanism has become an essential integral part in the modeling, the metamodeling, and also the model transformation context.
In this special case presented above, the UML profile helps in solving two drawbacks of model transformations based on the SDM metamodel:

1. A transformation based on SDM may only be created within the Fujaba editor. However, the UML profile makes it possible to define a transformation model within any CASE tool, i.e., within any modeling tool that is based on the UML metamodel. Thus, the profile assures a standardized and tool-independent specification of the model transformation.

2. The deployment of the profile-based transformation model is done by a code generator which is based on standardized API calls. Therefore, the dependence on a specific development environment, like in this case Fujaba, is no longer given.

Based on the findings and possibilities of the presented approach, and on the traditional way of applying profile-specific information within an ATL model transformation, it is fair to say that model transformations and UML profiles form an attractive combination. Not only is it possible to set profiles, stereotypes and tagged values within a transformation to annotate

[2] http://java.sun.com/products/jmi/

target model elements, but also to use UML profiles as the basis for the subsequent creation of transformations.

6.3 Automatic generation of UML Profiles based on Mapping Models

The applications of UML profiles discussed so far assume that the profile is already available for being used in a model transformation. Transformation rules for generating the target model elements are defined and the desired stereotypes and tagged values from the manually produced UML profile are applied inside this rules. Apart from this direct assignment of UML profiles, there is another valuable approach that follows a different direction.

In Wimmer et al. [54] (a related approach is given in [17]), the integration of DSL and UML models is reported and a semi-automatic approach based on a mapping model is followed. This approach is not relying on some existing UML profile, like it is for example reported in Abouzahra et al. [1], but is instead focusing on an automatic creation of the profile and the required transformation out of a mapping model.
As outlined in the paper, the so-called ad-hoc approach for bridging DSLs to UML has several drawbacks. Ad-hoc in this context means that the connections between the elements of the DSL and the ones of the UML metamodel have to be defined and that the corresponding UML profile is manually created afterwards. The assignment of profile information is then directly added inside the model transformation. This methodology leads to a high coupling between the transformation and the profile and, further, the correspondences between DSL and UML elements are not reusable in a different scope. To overcome these drawbacks, the authors propose a new approach consisting of two fundamental parts:

1. The core of this approach is a *mapping model* which is manually specified. This user-defined mapping model contains all correspondences between elements of the DSL and appropriate elements of the UML metamodel. The mapping model is expressed using a metamodel bridging language.

2. The second important part is a *Bridge Generator* which takes the mapping model as input and generates both the final model transformation and the required UML profile. Thus, the UML profile is not available from the very beginning but is automatically created.

Mapping model. For differentiating between the various metamodel concepts, different mappings are available: *Cl2Cl* mappings are provided for mapping classes of the involved metamodels, *Att2Att* mappings connect related attributes and *Ref2Ref* mappings are used for mapping corresponding references. Apart from these three, also other mappings are available to cover the remaining metamodel parts. For an example of *Cl2Cl* mappings please see Figure 6.3. Metaclasses of a DSL metamodel on the left hand side and metaclasses of the UML metamodel on the right hand side are related to each other by entries in the mapping model.

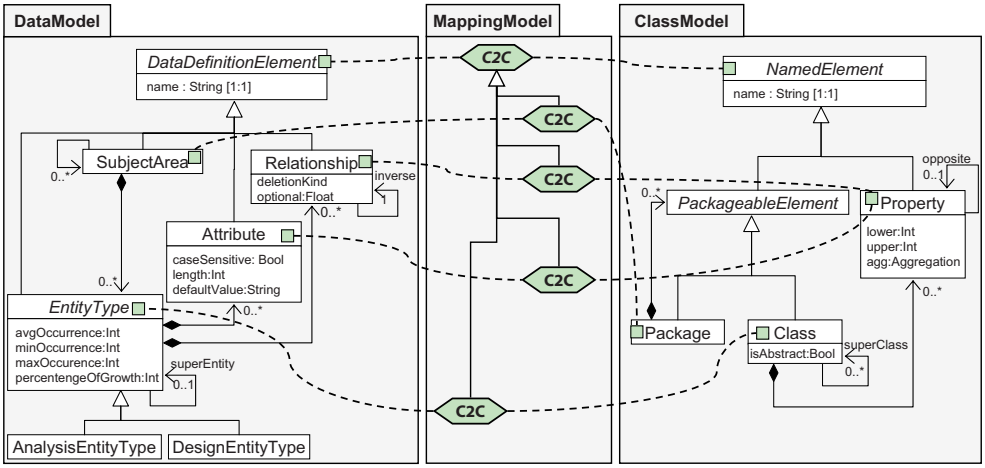

Figure 6.3: Example of Cl2Cl mappings [54]

Bridge generator. The proposed bridge generator takes the mapping model as input and subsequently generates two artifacts: (1) the UML profile(s) used within the model transformation and (2) the model transformation itself. The different mappings that are defined in the mapping model instruct the bridge generator for the generation of the required profile elements as well as the transformation rules. Concerning the profile generation, the bridge generator is able to distinguish between model features which are supported in both metamodels and those features that are only provided by the DSL and the UML metamodel respectively.

As outlined in the paper, this approach is not fully mature as different challenges are not yet considered. Apart from small unresolved issues, the idea behind this approach is very appealing since it puts the extension problem into a new perspective. The automatic generation of stereotypes and tagged values as well as the creation of code statements for handling these constructs in a transformation rule simplifies the work of an ATL engineer and enhances maintainability. The complex statements that are needed for applying profiles and stereotypes or setting tagged values may be created by the bridge generator. Thereby, the problems and difficulties that arise when using UML profiles within an ATL model transformation are tackled from a different point of view. The ATL language is not enriched with new, intuitive language concepts, as it is done with ATL4pros, but instead the definition of a model transformation is accomplished by a generator.
Due to the fact that this approach is totally different from the concept of ATL4pros, it is not possible to compare the two and tell which one is better or more intuitive. It shows, however, that different perspectives may be used when solving domain-specific problems of model transformation languages. Even though it is hard to compare the two approaches, they may be combined. Transforming an arbitrary domain-specific model to a UML model using UML profiles is a specific scenario, but, with the help of ATL4pros, allows for an automated generation of ATL code. The bridge generator, in turn, may be reconfigured to produce ATL4pros code which is subse-

quently transformed by the HOT. Thus, the two approaches are combined in order to cover a larger application spectrum.

The discussed related work shows that the area of model-driven software engineering involves a wide variety of different approaches for solving similar challenges. The specification of a model-to-model transformation is not restricted to a standardized transformation language like QVT or ATL but may also be accomplished with the technique of graph transformations. Likewise, UML profiles may not only be used during the execution of a transformation, but may also be used for generating a model transformation. Nevertheless, there are still many tasks to be resolved concerning the model-driven paradigm in general and model transformations in particular. The available approaches are, however, constantly improving to overcome these pending issues.

CHAPTER 7

Conclusion and Future Work

7.1 Conclusion

In this master's thesis I have presented an approach for extending ATL for natively supporting UML profiles. The necessity of a profile-specific extension of the ATL language may be seen as the result of the continuing success of the model-driven software engineering paradigm. The rise of OMG's Model-Driven Architecture (MDA), the popularity of the Unified Modeling Language (UML) in combination with UML profiles, and the need for model transformations all play their part. The ATLAS transformation language (ATL) is the de-facto standard in the area of model transformation and is a comprehensive tool when models are based on metamodels only. In contrast, applying UML profiles within a model transformation is more complicated. To overcome this serious drawback, the ATL4pros extension has been developed.

ATL4pros gives ATL engineers the possibility to apply stereotypes in a completely uncomplicated manner. The new keyword *apply* eases the application of stereotype information and, furthermore, the setting of tagged values is simplified as the existing binding construct of ATL is reused. In summary, this extension has significantly improved the use of UML profiles in the ATL language. However, it must be admitted that the main limitation of the discussed approach is the fact that it is not able to enhance the expressivity of standard ATL as such. As the extended syntax is converted to the standard syntax in order to be executable, the original core of the ATL language remains unchanged. Nevertheless, the concept of ATL4pros is certainly a good basis for further investigations concerning the future orientation of ATL.

The successful implementation of ATL4pros has been accomplished in three successive steps. First, the extension of the abstract syntax, i.e., the metamodel of ATL, has been completed. This step included the design and definition of new language elements as well as the integration into the predefined structure. The second step assured that all modifications in the abstract syntax are also present in the textual concrete syntax of ATL, requiring for a modification of the respective artifact. In the final implementation step, an operational semantics has

been defined. The operational semantics is needed to specify how new model elements of the extended syntax are translated to elements of the standard syntax. The procedure followed in this last step corresponds to the definition of a preprocessor. This preprocessor has been implemented by means of a higher-order transformation (HOT), defined in the refining execution mode of ATL. The benefits of this approach are the complete reuse of the standard ATL editor as well as the ATL runtime.

The last step of the extension process has not been carried out without problems. In general, higher-order transformations defined in the refining mode of ATL are a powerful technique. They offer the possibility to modify ATL transformations by adding extra information. Transformation elements that are not affected by a modification are simply copied. But, as every technology has certain strengths, it also has a number of weaknesses. One such shortcoming in relation to HOTs in refining mode was discovered during the implementation. An enormous number of metamodel elements have to be instantiated for building very small ATL code fragments. Change requests at a late stage of the implementation phase are a huge problem since the entire HOT is affected and has to be changed. Consequently, a mature plan at the beginning of the higher-order definition saves a lot of time and possible redesigns are considerably smaller.
The operational semantics defined for the ATL4pros extension required for the definition of various operational helpers. These helpers tend to become large and bulky. Therefore, a good documentation about the functionalities and the expected results is inevitable. Furthermore, the transformation rules require for a precise description for reproducing which rules are responsible for the creation of which code statements. In summary, a well-documented HOT prevents the ATL engineer from laborious maintenance tasks.

The need for a profile-specific extension proves that transformation languages like ATL have to be able to align to new challenges as new technological requirements emerge. Especially in the area of information technology a quick adaption is of most importance. In this context, it is necessary to take a closer look at the term extensibility in connection with ATL and examine the current status quo.
Basically, ATL is not designed for being enhanced with new language constructs by means of extension definitions. Even though it is possible for independent developers to commit contributions for improving ATL, this requires for expert knowledge about the ATL language on the one hand and the underlying architecture on the other hand. Such contributions are normally made by members of the ATL development team and not by normal users. However, a proper interface to plug in domain-specific extensions is missing. As outlined in Section 2.4 and also explained in [8], the ATL framework itself has a modular structure. But this modularity is limited to the ATL architecture itself. Comparable approaches for supporting modular language units are not available to this day. However, it may be desirable for ATL users to have the possibility of defining their own small language extensions for addressing domain-based requirements. Apparently, the integration of such user-built extensions is not feasible with the current framework and would require for a complete reconstruction of the ATL toolkit.

The topics *higher-order transformation* and *modularity of ATL* have been part of a lively

discussion during the *3rd International Workshop on Model transformation with ATL*[1]. All participants of the workshop had either personal experience with the specification of a HOT or at least knew the concepts behind this technique. Those who were using the ATL refining mode for the definition of a HOT all agreed that (i) the specification is a highly time-consuming task and that (ii) currently, no better alternatives are available. Therefore, developers who want to extend ATL with new and innovative features or useful add-ons have to accept the fact that the specification of HOTs is not unproblematic.
The second discussed topic concerning the modularity of ATL has been more difficult. All workshop members agreed that the need for extensions is given. Some argued that the structure of ATL has to be changed completely if such extension points were to be provided. But, as ATL is a very mature language with a comprehensive toolkit, it is doubtful if such a tremendous change in structure is really achievable.
Others suggested that it would be better to focus on the further enhancement of higher-order transformations and a rework of the ATL refining mode in order to integrate new extensions more smoothly. This solution seems to be more promising since the further development of ATL and its components is still in progress and a complete readjustment of the underlying structure is complicated.
When talking about a potential improvement of the ATL refining mode, the latest contribution of the drop implementation (cf. Section 4.4) sets a good example. This new feature is certainly improving the refining mode by facilitating the deletion of model elements. Prior to this contribution, a removal of elements has not been possible. The creation of more of this type of features is hopefully the first step into the right direction and may lead to a better support of domain-specific extensions.

The focus of this master's thesis was set on the ATLAS transformation language and an extension based on UML profiles. However, the ability to tailor ATL and also other transformation approaches to new, domain-specific needs and requirements may certainly become an important quality criterion in the future. Presumably, the demand for such extensions will further increase and modeling frameworks will have to adapt to the needs of the community.

7.2 Future Work

The current limitations of the ATL4pros extension discussed in Section 5.4 provide a good starting point for future work. In particular, all present restrictions regarding the ATL elements may be tackled in one step since all these limitations are only solved by a comprehensive redesign and rework of the higher-order transformation. To be more specific, the following three issues may be resolved as future work:

- **Consideration of all rule types**
  Not only matched rules but also called and lazy rule may be considered in the extension. As a result, the application of stereotypes is feasible for all three types of ATL rules.

[1] http://www.emn.fr/z-info/atlanmod/index.php/MtATL2011

- **Multiple source model elements**
  Mostly, only one source model element per transformation rule is defined. Regardless of this assumption, an ATL engineer may have the chance to define multiple source model elements.

- **Direct support of rule hierarchy**
  Defining a stereotype application in an abstract rule may result in an automatic application of the respective stereotypes in the subrules. As abstract rules ease the creation of model transformations, this additional implementation detail may be useful.

However, what seems even more important as a goal for future work is related to the traceability information between the extended and the standard ATL version. More precisely, enhanced debugging capabilities in form of advanced ATL editor features may be desirable. Standard ATL already provides a set of such facilities, e.g., including step-by-step transformation execution, running a transformation to the next breakpoint, and introspection of variables. Apart from debugging, also the disclosure of compile-time and runtime errors in the ATL editor is an important feature.
Debugging and error messages are supported for the final preprocessed transformation, but not for the transformation expressed in ATL4pros, which is the specification the transformation engineer would prefer to debug. However, the ATL refining mode builds an internal change computation model during its transformation process. Presumably, this model is used to store all changes promoted by the matching rules, like created elements, deleted elements, and modifications of elements. Therefore, this computation model is somehow relating the two different versions of the model transformation. Thus, the goal for future work is to explore the possibility of using this change computation model to serialize information as an additional output model by applying techniques as presented in [22, 56] for model-to-model transformations. It needs to be studied if it is possible to enable debugging for transformations written in the extended ATL syntax. In particular, debugging messages, state information, and further specific messages like error messages may be propagated from the standard ATL transformation specification to the ATL4pros transformation specification.

List of Figures

Listings

List of Tables

Bibliography

[1] Anas Abouzahra, Jean Bézivin, Marcos Didonet Del Fabro, and Frédéric Jouault. A Practical Approach to Bridging Domain Specific Languages with UML profiles. In *Proceedings of the OOPSLA Workshop on Best Practices for Model Driven Software Development @ OOPSLA'05*, 2005.

[2] Colin Atkinson and Thomas Kühne. The Role of Metamodeling in MDA. In *Proceedings of the International Workshop in Software Model Engineering @ UML'02*, 2002.

[3] Colin Atkinson and Thomas Kühne. A Tour of Language Customization Concepts. *Advances in Computers*, 70:105–161, 2007.

[4] Colin Atkinson, Thomas Kühne, and Brian Henderson-Sellers. Systematic stereotype usage. *Software and Systems Modeling*, 2:153–163, 2003.

[5] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, 4:171–188, 2005.

[6] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, 2001.

[7] Jean Bézivin and Frédéric Jouault. Using ATL for Checking Models. *Electronic Notes in Theoretical Computer Science*, 152:69–81, 2006.

[8] Hugo Brunelière, Jordi Cabot, Frédéric Jouault, Massimo Tisi, and Jean Bézivin. Industrialization of Research Tools: the ATL Case. In *Proceedings of the 3rd International Workshop on Academic Software Development Tools and Techniques (WASDeTT-3) @ ASE'10*, 2010.

[9] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Proceedings of the OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture @ OOPSLA'03*, 2003.

[10] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45:621–645, 2006.

[11] Hartmut Ehrig, Gregor Engels, and Hans J. Kreowski. *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*. World Scientific Publishing Company, 1997.

[12] Gregor Engels, Reiko Heckel, and Stefan Sauer. UML – A Universal Modeling Language? In *Proceedings of the 21st International Conference on Application and Theory of Petri Nets (ICATPN'00)*, 2000.

[13] Ryan Stansifer. EBNF Grammar for Mini-Java. `http://cs.fit.edu/~ryan/cse4251/mini_java_grammar.html` Accessed: 2011-07-14.

[14] Lidia Fuentes-Fernández and Antonio Vallecillo-Moreno. An Introduction to UML Profiles. *UPGRADE, European Journal for the Informatics Professional*, 5:5–13, 2004.

[15] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. A Domain Specific Language for Expressing Model Matching. In *Proceedings of the 5ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM'09)*, 2009.

[16] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The Missing Link of MDA. In *Proceedings of the 1st International Conference on Graph Transformation (ICGT'02)*, 2002.

[17] Giovanni Giachetti, Beatriz Marín, and Oscar Pastor. Using UML Profiles to Interchange DSML and UML Models. In *Proceedings of the 3rd IEEE International Conference on Research Challenges in Information Science (RCIS'09)*, 2009.

[18] Reiko Heckel. Graph Transformation in a Nutshell. *Electronic Notes in Theoretical Computer Science*, 148:187–198, 2006.

[19] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *Management Information Systems Quarterly*, 28:75–105, 2004.

[20] Martin Hitz, Gerti Kappel, Elisabeth Kapsammer, and Werner Retschitzegger. *UML @ Work, Objektorientierte Modellierung mit UML 2*. Dpunkt Verlag, 2005.

[21] Capers Jones. Software metrics: good, bad, and missing. *Computer*, 27:98–100, 1994.

[22] Frédéric Jouault. Loosely Coupled Traceability for ATL. In *Proceedings of the ECMDA Traceability Workshop (ECMDA-TW'05) @ ECMDA'05*, 2005.

[23] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72:31–39, 2008.

[24] Frédéric Jouault and Jean Bézivin. KM3: a DSL for Metamodel Specification. In *Proceedings of the 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, 2006.

[25] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06)*, 2006.

[26] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Satellite Events @ MoDELS 2005 Conference (MoDELS'05)*, 2005.

[27] Stefan Jurack, Leen Lambers, Katharina Mehner, Gabriele Taentzer, and Gerd Wierse. Object Flow Definition for Refined Activity Diagrams. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE'09)*, 2009.

[28] Lucia Kapová, Thomas Goldschmidt, Steffen Becker, and Jörg Henss. Evaluating Maintainability with Code Metrics for Model-to-Model Transformations. In *Proceedings of the 6th International Conference on the Quality of Software Architectures (QoSA'10)*, 2010.

[29] Milan Karow, Andreas Gehlert, Jörg Becker, and Werner Esswein. On the Transition from Computation Independent to Platform Independent Models. In *Proceedings of 12th Americas Conference on Information Systems (AMCIS'06)*, 2006.

[30] Stuart Kent. Model Driven Engineering. In *Proceedings of the 3rd International Conference on Integrated Formal Methods (IFM'02)*, 2002.

[31] Anneke Kleppe. A Language Description is More than a Metamodel. In *Proceedings of the 4th International Workshop on Software Language Engineering (SLE'07)*, 2007.

[32] Anneke Kleppe. The Field of Software Language Engineering. In *Revised Selected Papers of the 1st International Conference on Software Language Engineering (SLE'08)*, 2008.

[33] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model Comparison: A Foundation for Model Composition and Model Transformation Testing. In *Proceedings of the International Workshop on Global Integrated Model Management (GaMMa'06)*, 2006.

[34] Thomas Kühne. Matters of (Meta-)Modeling. *Software and Systems Modeling*, 5:369–385, 2006.

[35] Yuehua Lin, Jing Zhang, and Jeff Gray. Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In *Proceedings of the OOPSLA Workshop on Best Practices for Model-Driven Software Development @ OOPSLA'04*, 2004.

[36] Olaf Muliawan. Extending a Model Transformation Language Using Higher Order Transformations. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE'08)*, 2008.

[37] Object Management Group. MOF Specification: Meta Object Facility (MOF) Core Specification Version 2.0. `http://www.omg.org/spec/MOF/2.0/PDF/` Accessed: 2011-05-20.

[38] Object Management Group. QVT Specification: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.1. http://www.omg.org/spec/QVT/1.1/PDF/ Accessed: 2011-05-20.

[39] Object Management Group. UML Specification: OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3. http://www.omg.org/spec/UML/2.3/Superstructure/PDF/ Accessed: 2011-05-20.

[40] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[41] Dan Pilone and Neil Pitman. *UML 2.0 in a Nutshell*. O'Reilly Media, Inc., 2005.

[42] Andrea Randak, Salvador Martínez, and Manuel Wimmer. Extending ATL for Native UML Profile Support: An Experience Report 49-62. In *Proceedings of the 3rd International Workshop on Model Transformation with ATL (MtATL'11)*, 2011.

[43] José E. Rivera, Daniel Ruiz-Gonzalez, Fernando Lopez-Romero, José Bautista, and Antonio Vallecillo. Orchestrating ATL Model Transformations. In *Proceedings of the 1st International Workshop on Model Transformation with ATL (MtATL'09)*, 2009.

[44] Hans Schippers, Van Gorp Pieter, and Dirk Janssens. Leveraging UML Profiles to Generate Plugins From Visual Model Transformations. *Electronic Notes in Theoretical Computer Science*, 127:5–16, 2005.

[45] Ed Seidewitz. What Models Mean. *IEEE Software*, 20:26–32, 2003.

[46] Marten Sijtema. Introducing Variability Rules in ATL for Managing Variability in MDE-based Product Lines. In *Proceedings of the 2nd International Workshop on Model Transformation with ATL (MtATL'10)*, 2010.

[47] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley, 2009.

[48] Massimo Tisi, Jordi Cabot, and Frédéric Jouault. Improving Higher-Order Transformations Support in ATL. In *Proceedings of the 3rd International Conference on Theory and Practice of Model Transformations (ICMT'10)*, 2010.

[49] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In *Proceedings of the 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA'09)*, 2009.

[50] Massimo Tisi, Salvador Martínez, Frédéric Jouaoult, and Jordi Cabot. Refining Models with Rule-based Model Transformations. Technical report, AtlanMod, INRIA & École des Mines de Nantes, 2011.

[51] Marcel van Amstel, Christian F. J. Lange, and Mark van den Brand. Using Metrics for Assessing the Quality of ASF+SDF Model Transformations. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT'09)*, 2009.

[52] Marcel van Amstel and Mark van den Brand. Using Metrics for Assessing the Quality of ATL Model Transformations. In *Proceedings of the 3rd International Workshop on Model Transformation with ATL (MtATL'11)*, 2011.

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

[54] Manuel Wimmer, Andrea Schauerhuber, Michael Strommer, Wieland Schwinger, and Gerti Kappel. A Semi-automatic Approach for Bridging DSLs with UML. In *Proceedings of the OOPSLA Workshop on Domain-Specific Modeling @ OOPSLA'07*, 2007.

[55] Manuel Wimmer and Martina Seidl. On Using UML Profiles in ATL Transformations. In *Proceedings of the 1st International Workshop on Model Transformation with ATL (MtATL'09)*, 2009.

[56] Andres Yie and Dennis Wagelaar. Advanced Traceability for ATL. In *Proceedings of the 1st International Workshop on Model Transformation with ATL (MtATL'09)*, 2009.

[57] Wei Zhang, Hong Mei, Haiyan Zhao, and Jie Yang. Transformation from CIM to PIM: A Feature-Oriented Component-Based Approach. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05)*, 2005.