Steffen Kruse

Co-Evolution of Metamodels and Model Transformations

An operator-based, stepwise approach for the impact resolution of metamodel evolution on model transformations.

Abstract

With the continual increase in size and complexity of modern software systems, the evolution of software remains one of the big challenges of software engineering. Model Driven Engineering (MDE) is one approach to meet this challenge – by breaking down the description of software systems under development into manageable abstractions, each embodied by a suitable kind of artefact. Kinds of artefacts are models and metamodels, along with model transformations, (modelling) languages, generators and others. Yet the benefits expected from MDE can only be fully realised when the complexity of the MDE artefacts and their relationships remain manageable themselves.

The challenge addressed in this context by this thesis is the co-evolution of metamodels and model transformations; expressed as transformation descriptions in common, dedicated transformation languages. Transformations are used to produce output models conforming to a metamodel based on input models conforming to another (or the same) metamodel and are expressed in terms of the metamodels used. This enforces a tight coupling between metamodels and transformation descriptions. In consequence, any change made to one or more metamodel potentially invalidates existing transformations so that every change requires the validation and adaptation of all dependent transformations. This can lead to an exceeding amount of effort necessary to keep metamodels and transformations consistent as the number of transformations and metamodels increase.

This work presents an operator-based, stepwise approach to support software architects in the co-evolution of metamodels and transformations. To this end we propose a set of operators which, when applied to metamodels, perform an evolution step on the metamodel. The impacts of such a change in the form of an operator applied to a metamodel can be predicted for transformations that depend on the metamodel. The approach further allows the resolution of the impacts to restore consistency, either automatically or with minimal human input – depending on the type of change and the kind and complexity of the transformation. In the worst case, the use of operators at least indicates potentially invalid transformation parts that need further validation to fulfil their original intended purpose. Overall the approach reduces the effort needed for co-evolution.

The approach is implemented and integrated into MDE tooling commonly used for modelling and transformation creation to demonstrate its feasibility. The operators are formalised on the basis of the Essential MOF (EMOF) and the impact resolution is provided for the ATLAS Transformation Language (ATL).

Contents

CHAPTER 1

Introduction

Complexity is an essential property of software systems that increases in a non-linear fashion with the size of the software system [16]. Model Driven Engineering (MDE) is a software engineering approach that aims to alleviate this complexity in software development and maintenance by utilising models and modelling activities to raise the level of abstraction and to automate the production of artefacts. One specialised approach with this purpose is the *model transformation*, which allows the automated creation and modification of output models based on input models. Model transformations can be expressed in specialised transformation languages to be executed by a transformation engine. As models and model transformations are used in a productive capacity in software engineering, they underlie the same evolutionary pressure that conventionally build software systems do. Here the tight coupling between model transformations and metamodels becomes problematic, as changing the one often results in the need to check and adapt the other accordingly.

This thesis presents an approach to lessen this co-evolution problem by providing a systematic method to software architects of describing changes done to a metamodel and determining and resolving the impact on related model transformations.

The remainder of this chapter motivates this approach. Section 1.1 describes the co-evolution problem for metamodels and model transformations in more detail and Section 1.2 provides a simple illustration as an application scenario. Section 1.3 lists the proposed scientific contributions of this work. An outline of this thesis is provided in Section 1.4.

1.1. Motivation

During the MoDELS'08 conference a workshop was held to identify *'Challenges in Model-Driven Software Engineering'* and to propose the ways in which to best address these challenges in the next 10 or more years. Two of the challenges brought forward by the participants were the lack of proper *process support* for model driven engineering activities and specifically the lack of support for *model evolution and inconsistency management* in large, multi-user and distributed development settings. [105]

The participants argued that models can become inconsistent where dependencies between them exist, especially if the models are developed in a distributed fashion. As a first step to a solution, the need to identify the change types possible and the possible ways to resolve their impact on dependencies were identified. Furthermore, rapid prototyping for domain-specific modelling languages would be needed, as rapid prototyping provides the advantage of 'continuous, incremental feedback' for development in MDE. Yet rapid prototyping can only be achieved if handling the dependencies between modelling artefacts can be done as rapidly as evolving the prototype. The focus here was mainly on the relationship between models and metamodels, but the argument is also valid for model transformations that depend on the metamodels they are build on. [105]

In 2011, Hans Vangheluwe identified the evolution of development artefacts used in MDE, specifically models and modelling languages as a major challenge for the community, as the evolution of models has repercussions on all related artefacts, amongst them model transformations. He suggest for MDE and the related Domain Specific Modelling (DSM):

> 'If MDE and DSM are to be usable at an industrial scale, modeling language evolution has to be dealt with, ideally by (semi-) automatically co-evolving artifacts.' [106]

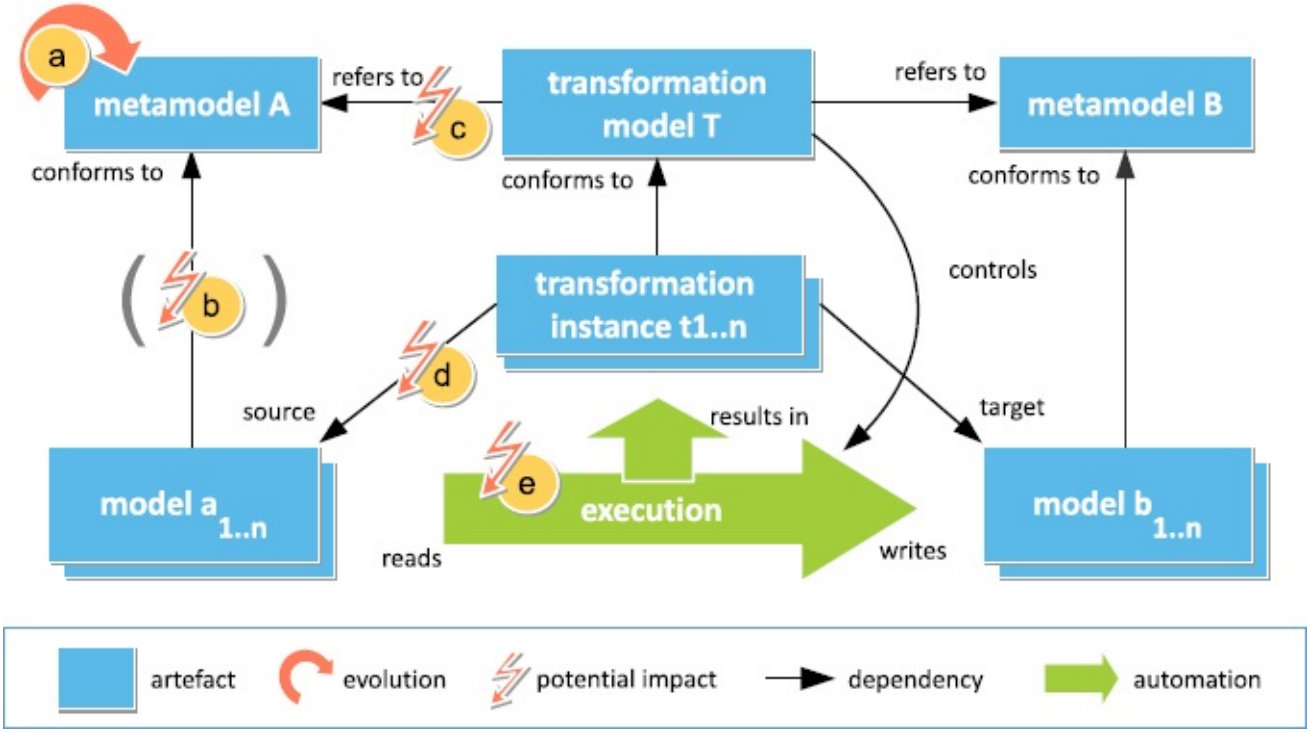


Figure 1.1.: The impact on metamodel evolution on dependent artefacts

Co-evolution is the model engineering task of changing dependent artefacts together to maintain consistency [22]. What constitutes *consistency* depends on the artefacts involved, for metamodels and models it is given in part by the conformance of the models to their metamodels. For metamodels and transformations, it is for one the suitability of the metamodel to represent the input and output models of the model transformation. But the expected function of the transformation also needs to be taken into account; an evolutionary change can introduce inconsistencies that cause a transformation to produce the wrong output, although it is still executable.

Figure 1.1 illustrates the problem of metamodel evolution and the impact it has on the artefacts involved in a transformation. It contains a transformation T, which is specified by a transformation model which can be expressed in a transformation language like ATL. The transformation model refers to two metamodels *A* and B, where *A* serves as the source or left-hand-side (LHS) and *B* as the target, or right-hand-side (RHS) in this case. If the transformation model describes an executable transformation and it is executed on some input model *a*, the execution results in a concrete transformation instance and produces a model *b* as output. If the transformation is correct, the model *b* conforms to its metamodel B. The transformation instance can be produced as an actual model in the form of a trace model[1] by the execution or it can simply be the concrete pair of models *a* and *b* that are linked by the transformation.

Evolution occurs in the form of some change done to metamodel *A*, represented by the curved arrow at ⓐ. An important part of handling the evolution of dependent artefacts is determining or recording what kind of change has taken place in a metamodel. A variety of different approaches dedicated to this problem exist for different purposes and technologies. These are discussed in chapter 8. Chapter 6 introduces an operator-based approach to describe possible changes made to a metamodel that is suited to resolving co-evolution issues with dependent transformations written in ATL.

After the metamodel is changed, the relationships to the other artefacts involved in the transformation may no longer hold and need to be evaluated and updated accordingly for the transformation to be consistent. This is represented in the figure by the flash symbol. The first relation is that of all the models *a* and their metamodel *A* at ⓑ. This relationship is one of conformance, indicating whether or not the model fulfils all constraints dictated by its metamodel, or in linguistic terms, whether the model is a valid statement in the language that the metamodel represents. This co-evolution problem has been investigated by numerous researchers and a number of different approaches to address it exist (see Section 8.1 for further details). This problem is not covered further in this work and we assume that it can be handled adequately.

The next relationship that may have been impacted is that between the metamodel *A* and any transformation model that makes use of the metamodel. This relationship is named 'refers to' and the impact is illustrated at ⓒ Whether or not the relationship is impacted at all by an evolution step and whether it can be resolved automatically depends on a number of factors. For example, the transformation may not even refer to the element that was changed in the metamodel and continue to function as before without needing any adaptation. Furthermore, if an impact occurs, what needs to be done to check and resolve it depends greatly on how it is expressed in the given transformation language or modelling technology. We investigate this problem for ATL as a central part of this work and suggest an approach for the impact resolution in chapter 7.

Without looking at the impact of the metamodel change and updating the transformation model accordingly, the updated models *a* may no longer fulfil the role of source models in their transformation instance ⓓ. In practical terms this means that an executable transformation can no longer read the input models without error or even worse, it executes without raising an error but delivers false results. Both these problems can be reduced to the problem of co-evolution of the metamodel and the transformation model, as when the transformation model is updated to reflect the change in the metamodel, re-executing the model transformation leads to a new and correct set of models *b* and accordingly to a set of new transformation instances.

Figure 1.1 illustrates a metamodel evolution on the left-hand-side of a transformation. The problem can also occur on the right-hand-side and affect target models. Should metamodel *B* be changed, the target models *b* may become invalid, i.e. the transformation no longer produces the correct result. Here the relationship between the target metamodel and the transformation model needs to be updated to reflect the evolution. Source and target elements are referred to differently in many transformation languages so that the side of the change plays an important role when resolving its impact. For this reason many of the impact resolutions introduced in chapter 7 make this distinction.

The next section introduces an application scenario containing two metamodels and a transformation between them in ATL. It further illustrates what impact changes done to one of the metamodels have on the transformation and how resolving inconsistencies can be addressed.

1.2. Application Scenario

This section contains a small example consisting of a model transformation between two metamodels and a scenario of metamodel evolution that affects the transformation. It illustrates how the changes made to a metamodel can impact the validity of a transformation and how this impact can be resolved. The example is also used in the remainder of this work to illustrate different aspects of operators or resolutions.

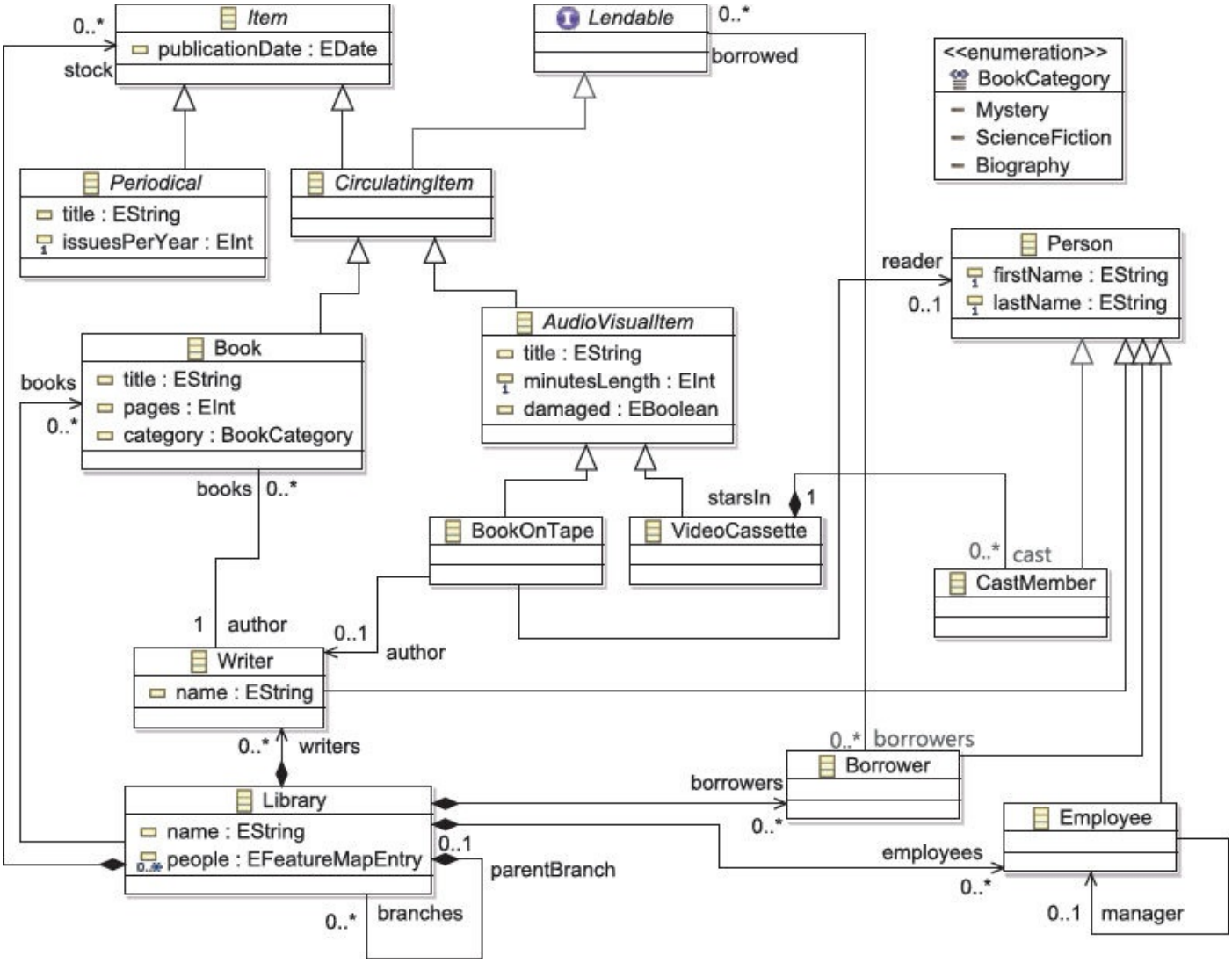


Figure 1.2.: The extended EMF-Library Metamodel example

Both metamodels are slightly modified versions of those we used in [19]. The first metamodel as seen in Figure 1.2 is an extended version of the common EMF-Library metamodel example [101]. The EMF-Library is a simple domain model of a public library, with entities like ‘Book’, ‘Writer’, ‘Borrower’ etc. In this fictional scenario, we assume the EMF-Library example metamodel serves as the basis for the development activities for software applications to manage a public library.

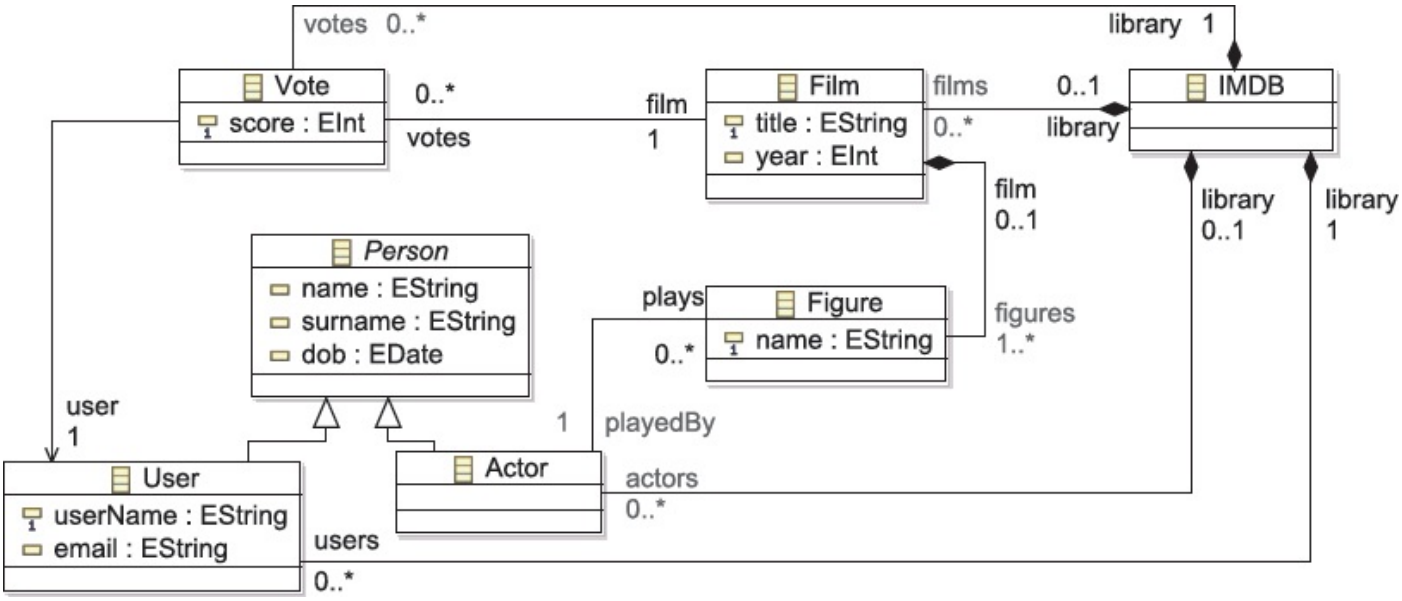


Figure 1.3.: The IMDB Metamodel example

The second metamodel represents a small and artificial model of the domain of the Internet Movie Database (IMDB) which is an internet database of information related to films and television programs[2] currently owned by Amazon. The metamodel contains entities like ‘Film’ with attributes like ‘title’ and ‘year’ of release and the ‘Actor’s that star in the film. The two domains of the two metamodels are related, as a library may also contains films that can be borrowed on video cassette.

The transformation example in listing 1.1 converts instances of the IMDB metamodel to those of the extended EMF-Library metamodel. It is written in ATL and consists of two rules. An entity of type *VideoCassette* is created for each IMDB entity *Film.* Furthermore, the information on any actor that stars as a *Figure* in the film is created as a *Cast* element and associated with the *Film.* This transformation could implement or model the task of adding new films into the stock of the library.

```
1  module transformation;
2  create OUT: emflib from IN: imdb;
3
4  rule Film2VideoCassette {
5   from film:imdb!Film
6
7   to cassette:emflib!VideoCassette (
8     title <- film.title,
9     damaged <- false,
10    cast <- film.figures
11  )
12 }
13
14
15 rule Figure2Cast {
16  from figure:imdb!Figure
17
18  to cast:emflib!CastMember (
19   firstName <- figure.playedBy.name,
20   lastName <- figure.playedBy.surname
21  )
22 }
```

Listing 1.1: A simple transformation between the IMDB Metamodel and the extended EMF-Library Metamodel

As an example of metamodel evolution, we propose the removal of a shortcoming of the extended EMF-Library metamodel: in its current state, each *VideoCassette* contains the general information of the film on the cassette and a list of *CastMembers* that star in the film. This is fine as long as there are few duplicate cassettes of the same film in a library. Otherwise, the same information is duplicated for each copy, causing redundant information to be stored and managed. Furthermore, if we wanted to use the model to create a list of all the films an actor starred in, this is not easily done, as the *CastMember* is duplicated for each new cassette.

Should the library decide to start buying numerous copies of especially popular films, the model can be improved to reflect this change in the domain. One approach would be to introduce an entity 'Film' to represent the film itself and to associate the actors with the film. The entity *VideoCassette* would continue to hold the information on the actual copies available, like who borrowed a cassette and whether it is damaged or not and would be linked to the film that is on the cassette.

Three steps are necessary to accommodate this change:

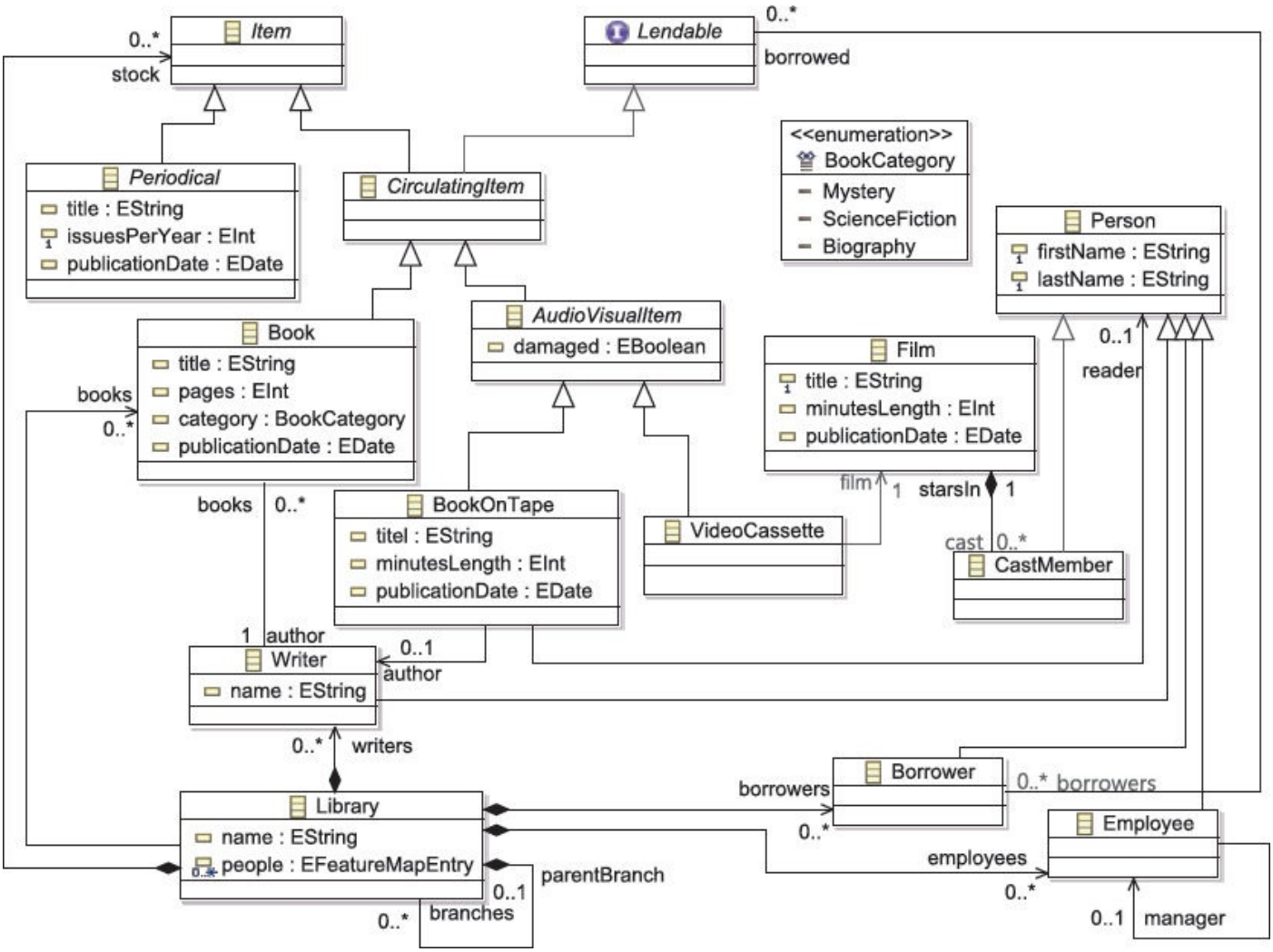


Figure 1.4.: The modified extended EMF-Library Metamodel example

1. Some of the properties needed for the new entity *Film* currently belong to parent entities of *VideoCassette* and are available to *Video-Cassette* by inheritance. These are moved down into *VideoCassette* in preparation so that they can later be moved to *Film.* The relevant properties are *publicationDate* in *Item* and *title* and *minutesLength*

in *AudioVisualItem*. This means that they also need to be moved down into other subclasses which are siblings of *VideoCassette* so that they remain available there. These are *Periodical*, *Book* and *BookOnTape* for the property *publicationDate* and *BookOnTape* for both the properties *title* and *minutesLength*.

2. In the next step we can create a new entity *Film* and link the two entities by an association.

3. Now we can move the relevant properties and the association to *Cast-Member* from *VideoCassette* to *Film*.

The resulting metamodel is shown in Figure 1.4. Please note that other entities were also updated as a result so that *Book* now also owns an attribute *publicationDate* directly.

```
1  module transformation;
2  create OUT: emflib from IN: imdb;
3
4  rule Film2VideoCassette {
5   from film:imdb!Film
6
7   to cassette:emflib!VideoCassette (
8     damaged <- false,
9     film <- emfFilm
10   ),
11   emfFilm:emflib!Film (
12    title <- film.title,
13    cast <- film.figures
14   )
15 }
16
17
18 rule Figure2Cast {
19  from figure:imdb!Figure
20
21  to cast:emflib!CastMember (
22   firstName <- figure.playedBy.name,
23   lastName <- figure.playedBy.surname
24  )
25 }
```

Listing 1.2: The updated transformation between the IMDB Metamodel and the evolved extended EMF-Library Metamodel

To cater for the evolution of the metamodel, we also need to update the transformation in listing 1.1:

1. The first change of moving the properties along the inheritance hierarchy down to the *VideoCassette* entity does not affect the transformation at all, as the attributes were previously available by inheritance and are now available directly.

2. The second evolution step requires an update of the first transformation rule, as for every new *VideoCassette* that is created we now also need to create a new *Film* entity (or link to an existing one).

3. In the third step, properties are moved from the *VideoCassette* to the *Film* entity. Here

the first rule needs to be updated again so that the correct values are set on *Film* instead of *VideoCassette.*

The updated transformation is given in listing 1.2. When the transformation is updated, it is valid again and performs the same task as before. Further improvements for the metamodel are possible, like performing the same kind of differentiation between the novel and the actual physical copies (books) in stock. These would require further updates to the accompaning transformations, depending on the nature and impact of the change.

1.3. Contribution

The central contribution of this work is made in addressing the co-evolution problem for metamodels and model transformations. It consists of the following parts:

An approach for the co-evolution of metamodels and model transformations. The approach is aimed to support software architects in evolving dependent artefacts of the Model Driven System (MDS) in stepwise fashion by applying predefined operators of common evolution steps on metamodels and detecting and resolving the possibly occurring impacts on model transformations.

A Set of Co-evolution Operators for the co-evolution of metamodels and model transformations. The operators are based on the Unified Modeling Language (UML) EMOF metamodel and are formalised as relations using the Object Management Group (OMG) standard QVT Relations (QVT-R) graphical notation.

A Set of Impact Resolutions that address the impact on ATL transformations that depend on evolved metamodels. The impact resolutions are formalized as QVT-R relations for the set of operators. The impact resolutions capture the type of impact possible for the different operators and provide semi- or fully automated resolution of the impact to restore consistency between metamodel and model transformation.

A Prototypical Implementation of the approach, the operators and the impact detection and resolution. The implementation is integrated into common MDE tooling for the creation and editing of metamodels and model transformations. The implementation provides support to the software architect when applying an operator to a metamodel, in determining the resulting impact on dependent model transformations and by providing and performing available resolutions.

This work provides support for co-evolution in MDE. It addresses the problem of complexity and the tight-coupling of dependent artefacts in the context of software evolution and reduces the effort needed to prevent the occurrence of inconsistencies due to software evolution tasks for metamodels and model transformations.

1.4. Outline

This thesis consists of three main parts. In part I the foundations for this work are laid out. It is structured as follows:

Chapter 2 introduces the foundations concerning MDE: Section 2.1 introduces the concept of the ‘model’ and Section 2.2 that of the ‘metamodel’. Section 2.3 covers the Meta-Object Facility (MOF), being the standardised metamodelling framework of the

UML and Section 2.4 covers Ecore, an implementation of MOF for the Eclipse IDE. In Section 2.5 the concept of model transformation is discussed.

Chapter 3 provides the concepts and languages available for creating model transformation used in this work. Section 3.1 introduces the general features of model transformation languages. Section 3.2 covers the transformation language ATL and Section 3.3 the Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT). As the Object Constraint Language (OCL) plays an important part in both languages, it is introduced in Section 3.4.

Chapter 4 covers software evolution in the context of MDE. To this end, Section 4.1 relates the co-evolution problem to the MDS and Section 4.2 introduces the related concepts in the area of model refactoring.

Part II covers the central contributions of this thesis. It contains:

Chapter 5 covers the approach presented in this thesis. It provides a problem description in Section 5.1 and details the goal pursued by our approach in Section 5.2. The requirements for the approach are detailed in Section 5.3 after which an overview of the approach is presented in Section 5.4. Sections 5.5 through 5.7 discuss the details of the three phases that make up the approach, after which a summary is provided in Section 5.8.

Chapter 6 provides the formalisation of the operators of our approach for EMOF as QVT-R relations and provides a discussion of the use and effect of each operator.

Chapter 7 covers the impacts of the operators from the previous chapter on ATL model transformations. Each operator is discussed and the proposed impact resolutions for ATL are formalised as QVT-R relations.

Chapter 8 discusses work related to our approach. Section 8.1 covers approaches dealing with the co-evolution of metamodels and models, Section 8.2 those that handle co-evolution of models and linked OCL constraints and Section 8.3 approaches that compete in the co-evolution of metamodels and model transformations.

Finally, part III discusses the evaluation of the approach with the implementation and concludes this thesis. It is structured as follows:

Chapter 9 covers the evaluation of our approach. Section 9.1 discusses aspects of completeness of the operators both on the metamodel and the transformation level. In Section 9.2 the use of operators in a coevaluation scenario containing a stepwise metamodel evolution and the resolution of impacts on a dependent transformation is presented. Section 9.3 summarises the chapter.

Chapter 10 contains the prototypical implementation of our approach. Section 10.1 provides an overview of the prototype and Section 10.2 covers the considerations made in the implementation. In Section 10.3 the details of the implementation for ATL are covered. The implementation of the operators and the integration into a graphical metamodel editor are discussed in sections 10.5 and 10.4 and the implementation of the impact resolutions in Section 10.6.

Chapter 11 concludes this thesis. Section 11.1 provides an overall summary. In Section 11.2, the benefits seen in this approach are recapitulated. Finally, Section 11.3 concludes with an outlook on future work.

---

[1] *Trace models* specify which input model element was transformed into which output model element and allow further aspects of model transformations like traceability and incremental transformations which are not relevant to this work.

[2] http://www.imdb.com

PART I

Foundations

CHAPTER 2

Model Driven Engineering

Model Driven Engineering (MDE) is a software engineering approach that aims to alleviate the complexity of software development and maintenance. It supports software engineers with suitable methods and tooling to handle the complexity of modern software systems. The complexity of a software system is broken down into manageable abstractions, each embodied by a suitable type of artefact. The most prominent artefact type is the *model*, along with transformations, (modelling) languages, code generators and many others. Together the artefacts constitute a Model Driven System (MDS), which serves as a representation of the software system of interest - from a software engineering perspective. In MDE, models and model engineering are utilized to raise the level of abstraction of primary artefacts and generative technologies to automate the production of secondary artefacts.

The concept of ‘model’ plays a central role in MDE. In this vein, Bézivin coined the principle ‘Everything is a model’ as the common principle behind the many different techniques and technologies used in MDE [12]. It is meant to play the same role that the principle ‘Everything is an object’ does in object technology and is an attempt to unify the different technologies and understandings of MDE and thus better define what is and what is not a part of MDE. Bézivin sees a step beyond object technology in MDE in which models take up the significance of objects and model transformation that of object composition:

> ‘Very differently [to object technology], what seem to be important now is that a particular view (or aspect) of a system can be captured by a model and that each model is written in the language of its metamodel. [12]

Although Bézivin does not attribute a single unique goal to MDE, he sees MDE as a result of the necessity for

> ‘a complete new rise in abstraction’ in dealing with ‘increasingly complex and rapidly evolving systems we are building today.’ [12]

Atkinson and Kühne follow this sentiment in that MDE is a further step in the need to handle growing and more complex systems, as

> ‘Model driven development (MDD) can be regarded as the natural continuation of [the trend to] … raise the level of abstraction at which the activity [to program computers] takes place.’ [4]

Models are used by programmers to specify what functionality is required of a system and what architecture is to be used. This holds the potential to reduce the complexity of problems and achieve a higher degree of automation in software development. The underlying goal of MDE is thus to improve productivity or maintain productivity when systems grow and become more complex. Atkinson and Kühne name two areas of improvement [4]:

- improving *short-term productivity*, by increasing the amount of functionality that can

be derived from a software artefact. The main mechanism to achieve this goal is the automated generation of source code from visual models.

- improving *long-term productivity* can be achieved by increasing the life-span of the value of artefacts.

Kühne further highlights the role of formal modelling for MDE:

> ‘It is one of the primary goals of model driven engineering to shift the emphasis from informal, non-binding models to rigorous, binding models.’ [56]

To achieve these goals for MDE in practice, a variety of artefacts and artefact types are used to capture the different aspects of the development process and a variety of tools to mange the different artefacts, both for the initial development and during the maintenance of the system [51]. These artefacts are tightly coupled by the different dependencies that exist between them and need to be managed along with the software system itself. Therefore, the artefacts used during development together with the dependencies between them form a complex system themselves [9]. We use the term Model Driven System (MDS) to refer to this system in this work:

Definition 1 (Model Driven System): *A* Model Driven System (MDS) *is the collection of artefacts and tools used for the development of a software system in MDE and the dependencies between them.*

Artefacts that are usually part of the model driven system are models and metamodels, but also transformations, transformation languages and codegenerators. In the next sections we look at models and metamodels in more detail (2.1, 2.2) and discuss the technologies provided by the UML and the Eclipse Modeling Framework (EMF) project to facilitate metamodelling activities (2.3, 2.4) and discuss model transformations in Section 2.5. Chapter 3 is dedicated to the different transformation languages relevant to this work.

2.1. Model

Stachowiak provides a general definition for the concept *model* for model theory. He associates three features with the term ‘model’: a mapping feature *(Abbildungsmerkmal)*, as a model is always a representation of something, a reduction feature (*Verkürzungsmerkmal*), as the model omits attributes that are deemed irrelevant by the model’s creator or user and a pragmatic feature *(Pragmatisches Merkmal*), as model and original may only be matchable under certain conditions. [91, pp. 131-133]

While many different uses of the term ‘model’ exist in different contexts [30], models in software engineering are commonly made of hard- or softwarebased, real or logical systems; as representations with the purpose of reasoning about or to perform analysis on some properties of the systems.

Kühne provides a definition of ‘model’ along these lines:

> ‘A model is an abstraction of a (real or language-based) system allowing predictions or inferences to be made.’ [56]

This narrows down Stachowiak’s definition to the notion of a model in the context of MDE: The original that a model represents is considered to be a real or abstract *system*,

the reduction feature is an *abstraction* and the purpose for reduction and the pragmatic feature is to reason about the system (instead of the system).

This directly corresponds to Bézivins and Gerbés definition of ‘model’:

> ‘A model is a simplification of a system built with an intended goal in mind. […] The model should be able to answer questions in place of the actual system.’ [13]

When using the term ‘model’ in this work we further refer to the models commonly used in software engineering, which are abstract and languagebased (as opposed to for example mathematical or scale models) and are abstract systems themselves. As such, the concept of a model relates to that of a system. The ISO/IEC/IEEE 12207:2008 standard defines the term *system* as follows:

Definition 2 (system): *A* system *is the ‘combination of interacting elements organized to achieve one or more stated purposes.’[47]*

It further limits its scope to *man-made* systems, that ‘may be configured with one or more of the following: hardware, software, data, humans, processes (e.g., processes for providing service to users), procedures (e.g., operator instructions), facilities, materials and naturally occurring entities.’ [47]

Although many other types of systems exist and broader definitions of the term also cover e.g. biological systems, we deem this definition sufficient for the remainder of this work and take no further position as to what constitutes a system. We assume that models represent systems in general and that the nature of the system is left to the user or creator of the model (unless otherwise noted).

We can therefore summarize the definition of model as follows:

Definition 3 (model): *An abstract system is a* model *of another system, if it is a simplification of the other system, created for a given purpose. The relationship between the model and the system it represents is named* representationOf.

According to Kühne [56], two classes of models are relevant to software engineering, *token models* and *type models.* Token models ‘capture singular (as opposed to universal) aspects of the original’s elements, i.e., they model individual properties of the elements in the system.’ Type models instead ‘capture the *universal* aspects of a system’s elements by means of *classification.’*

Whether or not the relation between system and model holds; i.e. the question whether a model correctly represents a system depends on the purpose for which the relation is observed. This provides a context under which this relation between two systems is that of one being a representation of the other; i.e. one system *has the role of a model* in the given context (Stachowiak’s pragmatic feature). This allows for the treatment of models as systems and the other way around depending on the context. Nevertheless, we presume it is safe to use the term *model* as short hand for ‘a system in the role of a model in context X’ if the context is deducible.

2.2. Metamodel

Formal and rigorous modelling requires methods to specify and to reason about the validity of models. This is achieved in part by metamodelling concepts in MDE, where modelling techniques are applied to models as systems. Bézivin defines the term metamodel as follows:

> ‘A metamodel is a formal specification of an abstraction, usually consensual and normative. From a given system we can extract a particular model with the help of a specific metamodel. A metamodel acts as a precisely defined filter expressed in a given formalism.’ [12]

This definition contains two central aspects of metamodelling: a constructive aspect, as the metamodel guides the creation process of a model from a system and a representative aspect, in that the metamodel acts as a filter on models.

Generally speaking, a metamodel is the model of a modelling language as it defines the abstract syntax of the modelling language. Kühne states: linguistic metamodelling is ‘the explicit specification of a language’s wellformedness constraints’ [57]. The advantages of such an explicit specification are: 1) the specification is not hidden in the code of a tool; 2) the specification can be altered by users of the tool; and 3) one can reason about the specification and the models it describes. [57]

The metamodel makes it possible to determine which models are valid according to the represented modelling language. Language-based models are abstract systems and it is possible to combine models into sets and treat these sets as abstract systems. Therefore the metamodel determines which models are members of the set of valid models. Seidewitz defines metamodels in reference to systems (system under study (SUS)) as follows:

> ‘A metamodel is a specification model for a class of SUS where each SUS in the class is itself a valid model expressed in a certain modeling language. That is, a metamodel makes statements about what can be expressed in the valid models of a certain modeling language.’ [87]

We adapt this definition slightly to incorporate the notion of the abstract system:

Definition 4 (metamodel): *A* metamodel *is a representation of a set of models, which can be called a modelling language. If a model is a member of this set, it* conforms *to its metamodel and this relationship is named* conformsTo.

The next section introduces the metamodelling framework of the UML and Section 2.4 its implementation for the Java programming language.

2.3. The Meta Object Facility (MOF)

The Meta-Object Facility (MOF) is a modelling language for metamodels, specifically for the metamodels of the family of languages maintained by the OMG, of which the UML is a prominent member. The MOF is also a part of the UML, as it reuses parts of the UML’s class modelling syntax and semantics. It contains two language versions, the first being a minimal set of elements named EMOF. The EMOF is designed for simplicity and reuse:

> ‘A primary goal of EMOF is to allow simple metamodels to be defined using simple concepts while supporting extensions.’ [75, p. 31]

The full MOF, which extends EMOF by more sophisticated language capabilities is called Complete MOF (CMOF). [75]

According to Bézivin and Gerbé, the MOF ‘emerged progressively’ in the work of different communities like the CASE data interchange format (CDIF) committee and the American National Standards Institute (ANSI) Information Resource Dictionary System (IRDS) standards. Different metamodels were competing with the UML and evolving separately, which brought about the risk of incompatibility. Thus the MOF was build as a ‘global integration framework for all the meta-models in the software development industry.’ [13]

In the next section, we introduce the UML and the aspects that are relevant to the MOF. We then cover the EMOF and its implementation for Java in more detail, as these form the basis for the co-evolution operators defined in this work.

2.3.1. The Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a modelling language for software systems. It is based on object oriented principles and is best known for its visual diagram types, of which the class diagram is most commonly known. The first version of the UML was created by Grady Booch, James Rumbaugh and Ivar Jacobson to unify their different object oriented methods and submitted to the OMG for standardisation in 1997. In November 1997 the revised UML version 1.1 was adopted by the OMG. The maintenance versions up to version 1.5 were produced by a revision task force until the major revision UML 2.0 was adopted in 2005. [15]

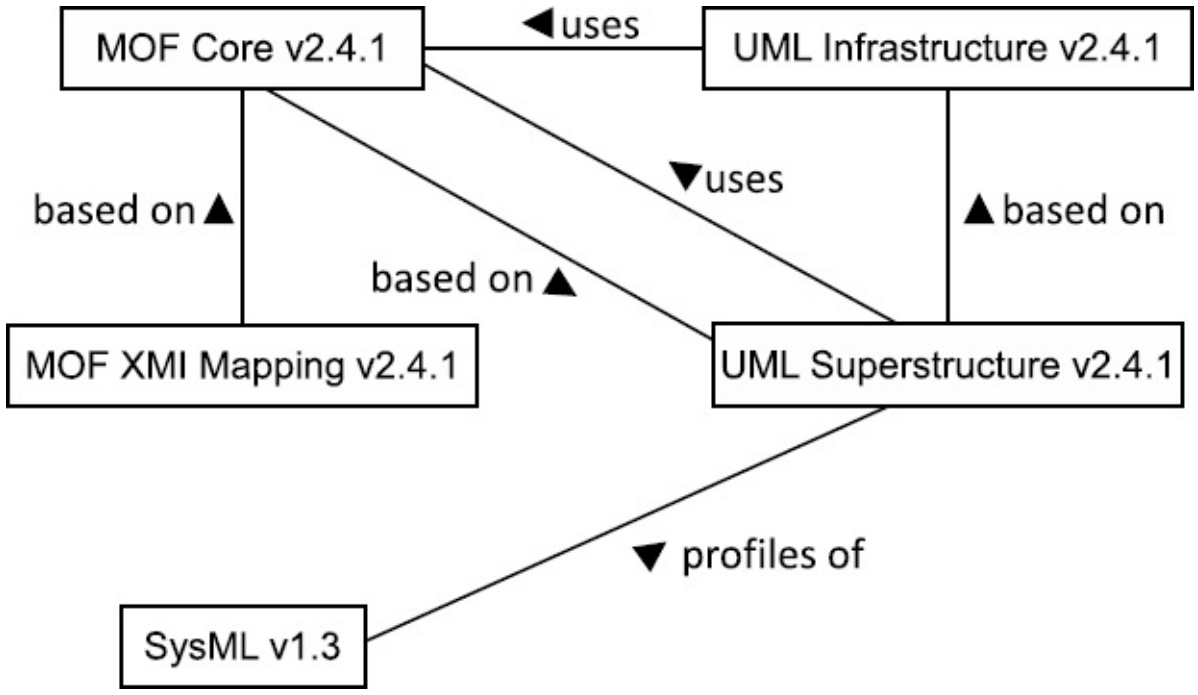


Figure 2.1.: Structure of the UML Specification as of Version 2.4.1 [86]

The current official release of the UML is version 2.4.1 from August 2011 while version 2.5 is being prepared and currently available for evaluation [78]. Unless otherwise stated, this work is based on and refers to the current version 2.4.1 of the UML standard.

The UML Specification Architecture

The organisational structure of the UML specification is relevant to this work and changed between versions 2.3 and 2.4.1 [86]. Figure 2.1 provides a short overview of the current configuration.

The UML in version 2.4.1 consists of two related specifications: the *OMG Unified Modeling Language Infrastructure* (Infrastructure) [76] and the *OMG Unified Modeling Language Superstructure* (Superstructure) [77]. Both consist of a natural language document and a set of models in the XML Metadata Interchange (XMI) format.

The Infrastructure specification contains the language foundation for the UML. It contains the core packages Basic, Constructs, Profiles and Abstractions and defines language features like namespaces, typing, classes, associations and profiling etc.

The Superstructure specification defines the language elements of the UML commonly used for software system modelling like the UML diagram types, such as the class and activity diagram. For this purpose the Superstructure reuses and extends the elements defined by the Infrastructure specification.

The MOF specification also reuses and adapts both the Superstructure and the Infrastructure packages (see Figure 2.1) so that all three specifications need to be consulted for the definitive specification of the language elements belonging to the MOF. Furthermore, the UML provides different levels of compliance that modelling tools can adhere to.

How elements can be adapted and extended for the purpose of reuse in the UML and MOF specifications is defined by the *package merge* mechanism, which is described in the next section.

Package Merge Mechanism

The UML is divided into four *compliance levels* (L0–L3), where L0 is defined in the UML Infrastructure specification and the levels grow in the available features by building on the previous level. The key mechanism used for this structure is the *package merge,* which defines how elements from different packages can be integrated to form new language specifications [77, pp. 113]. A merge is conceptually like a ‘generalization’, as the result of the merge combines the characteristics of the two merged packages. Merging can be applied to elements that have the same name and that represent the same concept. Merging represents both the process of transformation and the element resulting from the transformation, which is a stand-alone element in its own right. How elements are merged and what constraints apply to a successful merge is defined in the UML Superstructure specification, while the specifics depend on the type of element. [77]

> ‘Different metatypes have different semantics, but the general principle is always the same: a resulting element will not be any less capable than it was prior to the merge. This means, for instance, that the resulting navigability, multiplicity, visibility, etc. of a receiving model element will not be reduced as a result of a package merge.’ [77, p. 116]

Therefore, as a general rule of thumb, merging can not reduce capabilities: for the availability of properties, multiplicity or visibility the merge always results in the most general combination of the merged elements.

The MOF Specification Architecture

The MOF specification defines two language levels, the EMOF and the CMOF. The EMOF merges the `Classes::Kernel` package from the UML Superstructure with own language capabilities. `Classes::Kernel` merges the `Core::Basic` package from the UML Infrastructure in turn [75, p. 32]. The CMOF merges the EMOF and extends it with more sophisticated extension and reflection features [75, p. 47].

The two MOF levels are designed to be self contained, i.e. after merging the language elements from the different sources, the EMOF metamodel is reflective in that it can be specified by itself. Thereby, the metamodel of the EMOF is the EMOF itself. The CMOF further serves as the metamodel for the UML Infrastructure, Superstructure and compliance levels, as implemented by the MOF XMI models of the ‘Normative Machine Consumable Files’ as part of the UML standard. [77]

Semantics of the UML

Unfortunately the UML does not provide precise formal semantics in its current version. The abstract syntax and static semantics are provided by the normative MOF XMI models files and in the Superstructure and Infrastructure specification documents along with the concrete (graphical) syntax. The dynamic semantics are provided in ‘precise natural language’ [76, p. 22].

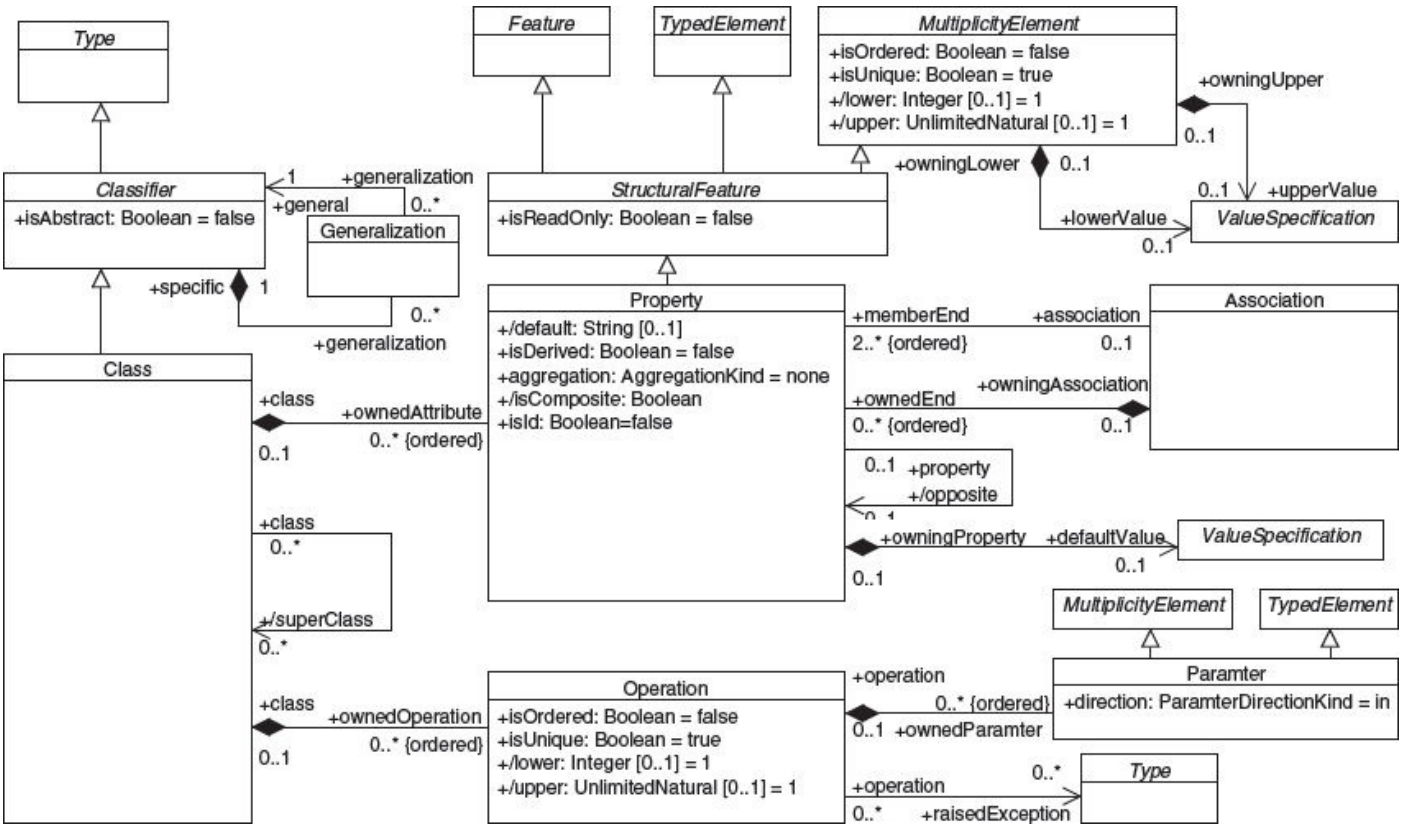


Figure 2.2.: The *EMOF Classes* package

This lack of formal semantics has been criticised early on in the history of the UML [29] and has also been discussed for the *UML2* revision of the UML standard (see for example the work by the UML 2 Semantics Symposium [17]). The difficulties of providing overall formal semantics stem from the complexity of the UML and the tailoring and variability mechanisms it contains [83]. For some of the sophisticated language features, individual formalisation approaches have been proposed, like semantics for the package merge mechanism [110] or for the association concept [2]. Yet complete formal semantics for the UML remain an open research problem.

2.3.2. The Essential MOF (EMOF) Model

This section covers the EMOF language architecture and the most important EMOF elements in some detail, as the co-evolution operators defined in chapter 6 are based on EMOF.

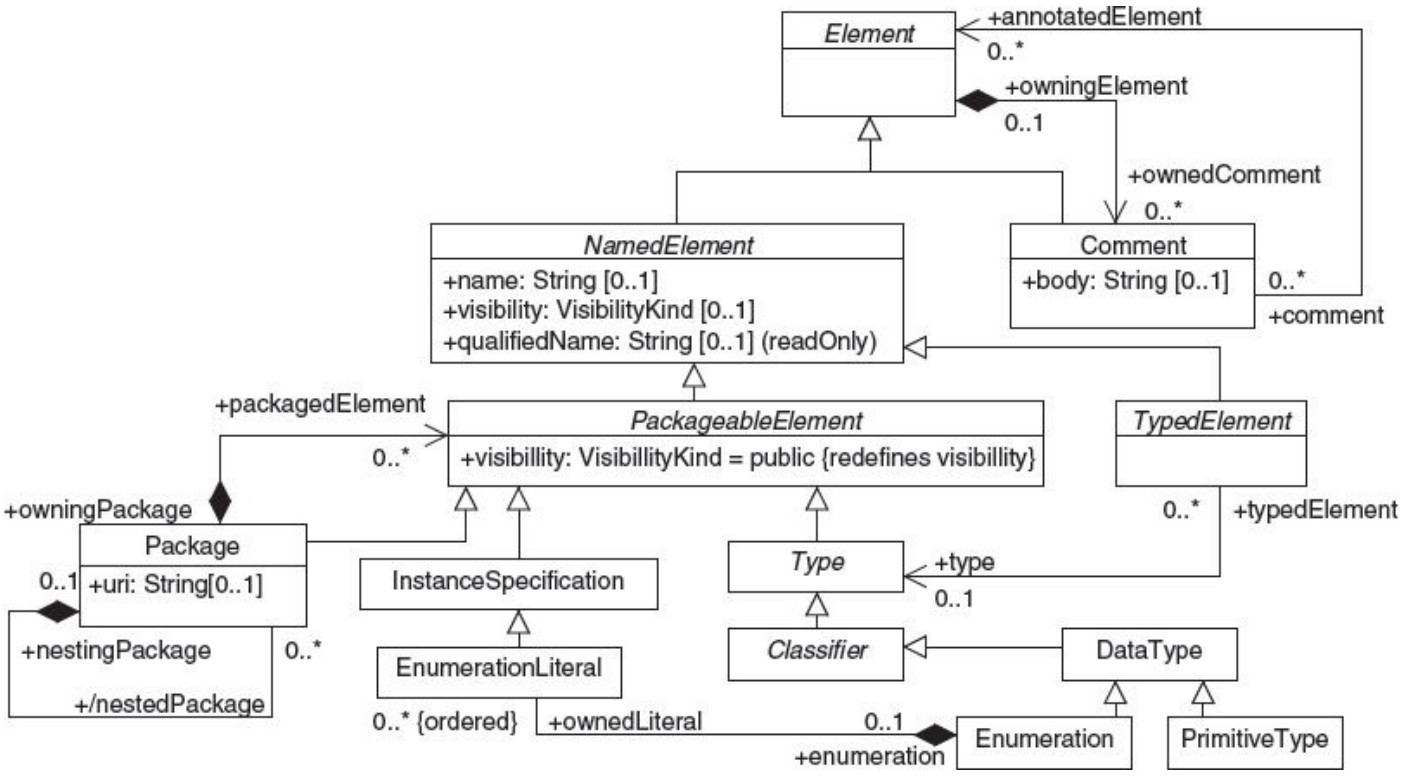


Figure 2.3.: The *EMOF Types and Data Types* package

EMOF is defined in clause 12 of the *OMG Meta Object Facility (MOF) Core Specification* [75, p. 31]. It does not fully repeat the elements which are imported from the UML Superstructure but rather provides the omissions and redefinitions. The accompanying XMI models redefine elements from the UML Superstructure using the package merge mechanism described in the previous section. EMOF merges the packages `Identifiers`, `PrimitiveTypes`, `Extensions` and `Reflection` from MOF, of which the last merges the `UML::Classes::Kernel` from the UML Superstructure specification in turn. In this manner most EMOF language features are defined by the Superstructure specification, while the EMOF specification only makes minor additions and excludes unnecessary classes and attributes from the `UML::Classes::Kernel` via constraints. Figure 2.2 shows the EMOF class element with the related elements and Figure 2.3 the type and data type elements after package merge of the Superstructure and MOF *(compare* [75, pp. 33-34] *with* [77, pp. 22-34]).

The main concrete EMOF language elements are:

Class: Classes are the central elements of a model. Classes are classifiers and provide attributes and operations as features. Classes inherit the generalization mechanism from classifier, serve as types, are named and packageable.

Package: Packages provide a structuring mechanism for all elements. They are uniquely identifiable by a Uniform Resource Identifier (URI) and provide uniqueness to their content. As opposed to CMOF and UML, in EMOF packageable elements are redefined to always be publicly visible.

Property: Properties declare named relationships of classifiers to values of simple or complex type. Properties are owned by classifiers or associations and can be part of an association, making them an associationend. Properties defined on classifiers (directly owned or inherited) are also called attributes. If a property is part of a bidirectional association, the property on the other end is given by the /opposite feature.

Association: Associations define valid links between typed instances. In EMOF, an association has a property at each end, which defines the type of the association-end. The properties may be owned by the classifier at the association-end or (one for each association) by the association itself. If the property is owned by a classifier, the association is defined to be navigable from that classifier. If both properties are owned by a classifier, the association is bidirectional. The exact semantics of *navigability* depend on the implementation as:

> ‘Navigability means instances participating in links at runtime (instances of an association) can be accessed efficiently […]. The precise mechanism by which such access is achieved is implementation specific.’ [77, p. 38]

The derived feature /endType [1..*] provides the type(s) of the properties serving as association-ends.

Operation and Parameter: Operations allow the modelling of behaviour and define what parameters need to be supplied to initiate the behaviour of an instance and what return type is expected. The implementation is platform dependent. For example, the MOF makes use of the OCL to further specify the behaviour of operations.

Generalization: Generalizations provide the inheritance mechanism common to object oriented languages. Generalizations allow the redefinition of classifiers and a parent – child relationship between classifiers. Classes provide the class / superClass association for convenient access to the classes related by a generalization.

Enumeration, EnumerationLiteral: Enumerations are data types that have enumerable values, the enumeration literals.

Comment: Comments allow the annotation of elements with human readable text.

InstanceValue: Instance values are value specifications that identify an instance, either in textual or graphical notation.

LiteralSpecification: The types `LiteralBoolean`, `LiteralInteger`, `LiteralNull`, `LiteralReal`, `LiteralString` and `LiteralUnlimitedNatural` encode constant values of the corresponding simple types.

DataType and PrimitiveType: Data types represent instances that are only identified by their value. Primitive types have no further substructure in the MOF, examples are `Boolean`, `Integer` and `String`.

2.4. The Eclipse Modeling Framework (EMF) / Ecore

The Eclipse Modeling Framework (EMF) is a modelling and code generation framework for the Java programming language and the Eclipse platform [94]. It contains a metamodel called *Ecore*, which is closely related to EMOF. Ecore started off as a MOF implementation and was extended and adapted to provide a 'highly efficient Java implementation of a core subset of the MOF API' [101]. With the creation of the EMOF as a similar subset of MOF in version 2.0, close alignment between Ecore and EMOF was created. Although some differences exist, Ecore can read and write serializations of EMOF. [101]

EMF provides model driven features on top of Ecore for the Java language, such as Java code generation from Ecore models, XMI serialisation, the creation of models from Java code using annotations, notification and adaptation functionality of the generated Java classes and a reflective Java API to access Ecore models. [94]

One example of a difference between Ecore and EMOF in version 2.4.1 is the relationship of classes, properties and associations *(compare* [94, p. 107] *with* [77, p. 29]). In EMOF, attributes with complex types are build as associations with properties serving as association-ends on both sides. Attributes of simple type are properties without associations (just with a simple type). In Ecore, the property element is called 'attribute' *(EAttribute)* which only serves for simple types. Association elements connect attributes of complex type to their type-class instead. While directionality is expressed in EMOF by the ownership of the remote property (if it is owned by the association, it is uni-directional; if it is owned by the remote class it is bi-directional), in Ecore associations are bi-directional if two associations between the classes exist that are explicitly set as each others opposite.

2.5. Model Transformations

Model transformations are the mechanism that allows the automatic manipulation of models and play a central role in MDE [11, 27, 33, 57, 68]. They have been called 'the heart and soul of Model-driven Software Development' [88] and according to Bézivin, take up the same importance in MDE that object composition does in object technology:

> 'The central idea of object composition is progressively being replaced by the notion of model transformation.' [12]

Kühne defines the term 'model transformation' as the

> 'information on a mapping from one model to another, created by a transformation engineer, for the transformation engine, in order to automate a translation process.' [56]

Kleppe et al. in turn define 'transformation' as the process itself and rather use the term 'transformation description' to refer to the mapping information:

> ‘A transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.’ [52]

This difference highlights the dual usage common for the term ‘transformation’ [30]: in the first it refers to the mapping information between two models and in the second to the process of applying this information in creating one model from another.

Mens and Van Gorp propose the extension of the definition by Kleppe et al. so

> ‘that a model transformation should also be applicable to multiple source models and/or multiple target models.’ [68]

Although a set of models can also be considered to be a model, this addition explicitly allows for transformations to be defined between sets of models. In summary, we derive the following working definition:

Definition 5 (model transformation): *A model transformation is the information on a mapping between two sets of models, to be executed by a transformation engine. It consists of a set of rules that describe how elements of the target language can be created from elements of the source language automatically.*

Whether or not something constitutes a transformation depends on the term ‘model’ (see definition 3). For example, it is valid to assume that programs are models, as they represent an abstract system, conform to their programming language and exist for a specific purpose [12]. A common scenario along this definition is a code generator, which creates executable code from some model on a higher level of abstraction. Kurtev et al. suggest the concept of the *technological space* to differentiate transformation (and modelling) concepts in different technological contexts:

> ‘A technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. It is often associated to a given user community with shared know-how, educational support, common literature and even workshop and conference regular meetings.’ [58]

Examples of technological spaces with transformations are the XML / XSL Transformations space [24] or triple graph grammars [85]. We limit the scope of this work to transformation languages for MOF-based models, which corresponds to a choice of the UML/MOF technological space.

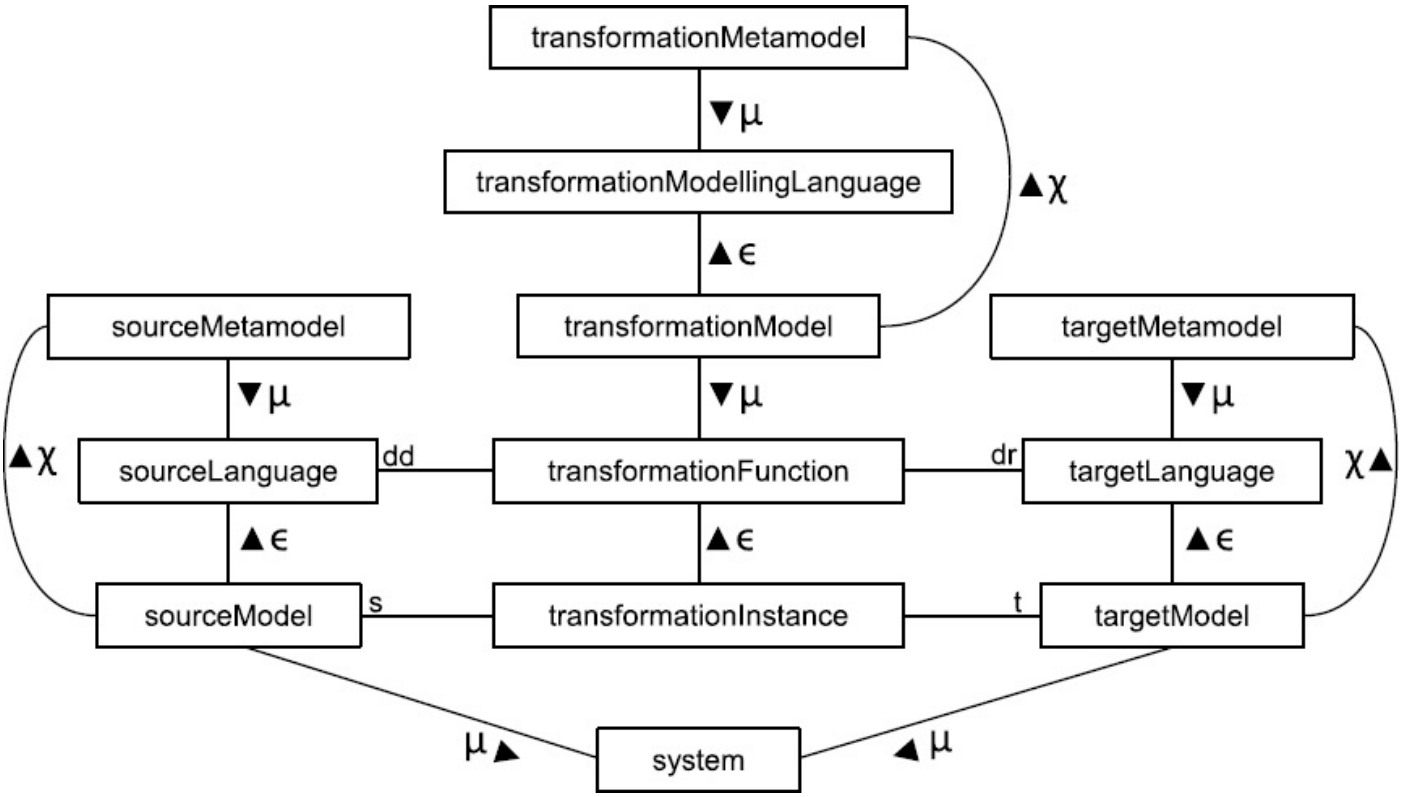


Figure 2.4.: Favre's *Transformation Pattern* [32]

The next sections cover some of the important aspects of model transformations.

2.5.1. Transformations as Functions

The abstract system represented by a model transformation can be seen to be a function in set theory. This *transformation function* is the set of transformation instances which are pairs of related source and target models. In this sense, the set of all valid input models is the *domain* and the set of all output models the *range*. As metamodels define sets of models and allow the reasoning of whether or not a model is valid i.e. a member of the set, metamodels lend themselves to defining domain and range of the transformation function. [32]

2.5.2. Transformations as Models

Model transformations can be seen as models (as long as one understands ' 'transformation' as 'description of a transformation function' ' [56]): The transformation conforms to its transformation language (metamodel), it represents the transformation function between one or more sets of models and is created for the purpose of execution by a transformation engine.

Figure 2.4 contains a pattern of the notion of model transformation according to Favre [32]. It illustrates the roles the elements of a model to model transformation play in terms of the relations *RepresentationOf* (μ), *ElementOf* (∈)and *ConformsTo* (χ). The transformation itself is given as a transformation function, defined between a source and target modelling language. When executed for a specific pair of models, this one transformation instance is an element of the transformation function, as source and target models are elements of the set of all valid models, i.e. the source and target modelling

languages. The languages are represented by their metamodels, to which all valid models conform. The transformation function is represented by a transformation model (an executable description of the transformation) which is an element of all valid transformations, i.e. a member of the transformation language of choice. The transformation language is represented by a metamodel, to which the transformation model conforms.

When transformation functions are represented by models, these models can themselves serve as input or output of transformations. In this manner transformations can be handled by the mechanisms and tools common to MDE and be treated as ‘first-class citizens’ of MDE [57], with a number of promising application cases [103]. These transformations are called ‘higherorder’ [9]:

Definition 6 (higher-order transformation): *Higher-order transformations represent transformation functions with transformations as input or output models.*

2.5.3. Transformations as Programs

To achieve the goal of MDE of improving productivity while raising the level of abstraction of software artefacts, model transformations are needed that can be evaluated and executed automatically. As mentioned above, a transformation model can represent a transformation function and can be interpreted and executed by a transformation engine. When applied to a set of input models, the transformation engine produces a set of output models.

Here the executable model of the transformation function takes the form of a set of instructions and is thus a program. Models that make up the source or input of the execution are also referred to reside on the LHS and target or output models on the RHS of the transformation.

The transformation model can also be seen as the representation of a set of mappings between models. An engine provided with models for both (or all) sides of the mapping can then validate whether the models are related by the transformation, i.e. whether the mapping holds in the given case. This makes two models a pair in the ordered pair of the transformation function.

While all programming languages are generally suited to program transformations, dedicated languages exist to specify transformations in different technological spaces with properties related to their technological space [27]. In the next section we look at two transformation languages prominent in the UML / MOF technological space, their architecture and the supportive tooling.

CHAPTER 3

Model Transformation Languages

This chapter covers the ATLAS Transformation Language (ATL) and the languages contained in the Meta Object Facility (MOF) 2.0 Query/View/ Transformation (QVT) and their supportive tooling in more detail. Both approaches are closely related to UML / MOF and have tool support for the Eclipse IDE and the Eclipse Modeling Framework (EMF). ATL is introduced in Section 3.2 and QVT in Section 3.3. Both also reuse the Object Constraint Language (OCL) for model navigation and expressions, so that OCL is covered in Section 3.4. The next section provides a short comparison of the two approaches in terms of their general features.

3.1. Model Transformation Language Features

Czarnecki and Helsen developed a feature based approach for the classification of the variety of model transformation approaches and technologies available [26, 27]. The classification provides means of comparison of the large number of responses to the OMG's Request for Proposal for MOF/QVT (see Section 3.3) [26] and a means to make the design choices behind model transformation approaches explicit [27].

Their feature model provides the following top-level and mandatory features of model transformations:

- **Transformation rules** Model transformations consist of transformation rules that describe how the transformation occurs. Besides being mandatory features of transformations in Czarnecki and Helsen's feature model, transformation rules are also part of the definition for model transformations (see *Definition 5* on page 32). The transformation rules map elements of the source model to the elements of the target models to make up the transformation. The domain of a transformation rule is defined by the rule body, which must specify the use of variables, and can contain sets of patterns and rule logic. Patterns are defined as being a set of zero or more variables used during the transformation process (metavariables), a syntax and a structure, being one of strings, terms or graphs.

- **Rule application control** Transformation approaches define how the locations where in a model the rules apply are determined and in which order the rules are executed.

- **Rule organization** Rules need structuring mechanisms to provide features like modularisation or reuse to developers.

- **Source-target relationship** This feature describes how transformation approaches relate models, like if the source and target are the same or separate models.

- **Incrementality** Incrementality defines whether transformation approaches can update target models based on changes made to source models.

- **Directionality** This describes if transformations can be executed in one or more directions (only from LHS to RHS or in reverse or in the direction of any of an arbitrary number of domains.)

- **Tracing** Tracing defines if and how the transformation approach records the transformation execution process for later analysis or to provide incrementality.

In respect to this feature model, the declarative language provides by the ATL and the declarative and imperative languages of the QVT are rule- and pattern-based, and use a combination of implicit and explicit rule application control. Both also provide mechanisms for rule organisation and re-use.

While the primary use-case of ATL is the transformation between two different models [49], QVT supports the use of any number of models per transformation. With the exception of the QVT Core (QVT-C) language, all languages provide tracing mechanisms and support for incremental transformations by using the tracing information from previous executions. The QVT languages and ATL differ in the directionality feature, ATL transformations are defined in the direction of a specific domain, while for QVT transformations, the target model can be chosen at runtime.

3.2. The ATLAS Transformation Language (ATL)

The ATLAS Transformation Language (ATL) is a textual model to model transformation language. It is maintained in the Eclipse ATL Project which is part of the Eclipse Modeling Project [95] and is closely related to the ATLAS Model Management Architecture (AMMA) platform, which bundles a set of projects concerned in general with MDE and contributes in particular to the tool support of ATL [49].

ATL was developed as a response to the OMG *MOF/QVT Request for Proposal* for a transformation language from which QVT originated and is thus similar to the final adopted QVT specification. Like QVT, ATL transformations are made up of rules which map elements belonging to a source model to one or more elements of a target model. It consists of both declarative and imperative language structures although the main focus is on the declarative parts. ATL also uses OCL (see Section 3.4) for the navigation of model elements but uses its own partial implementation of the OMG OCL standard. Unlike QVT, ATL transformations are unidirectional [5].

3.2.1. Language Structure

The ATL transformation language syntax is made up of a hybrid of declarative and imperative elements. The promoted programming style is declarative, while imperative language structures are provided for complex mappings. An ATL program is composed of a set of rules. Rules can be either *matched rules* that match source model elements and produce target model elements or *called rules* which only produce output elements and are called from matched rules as a mechanism for re-use and to structure complex cases. Rules use patterns to describe the model elements they match or create. These *in-* or *out-patterns* consist of *pattern elements* that bind to matching instances in the source and target models. [6]

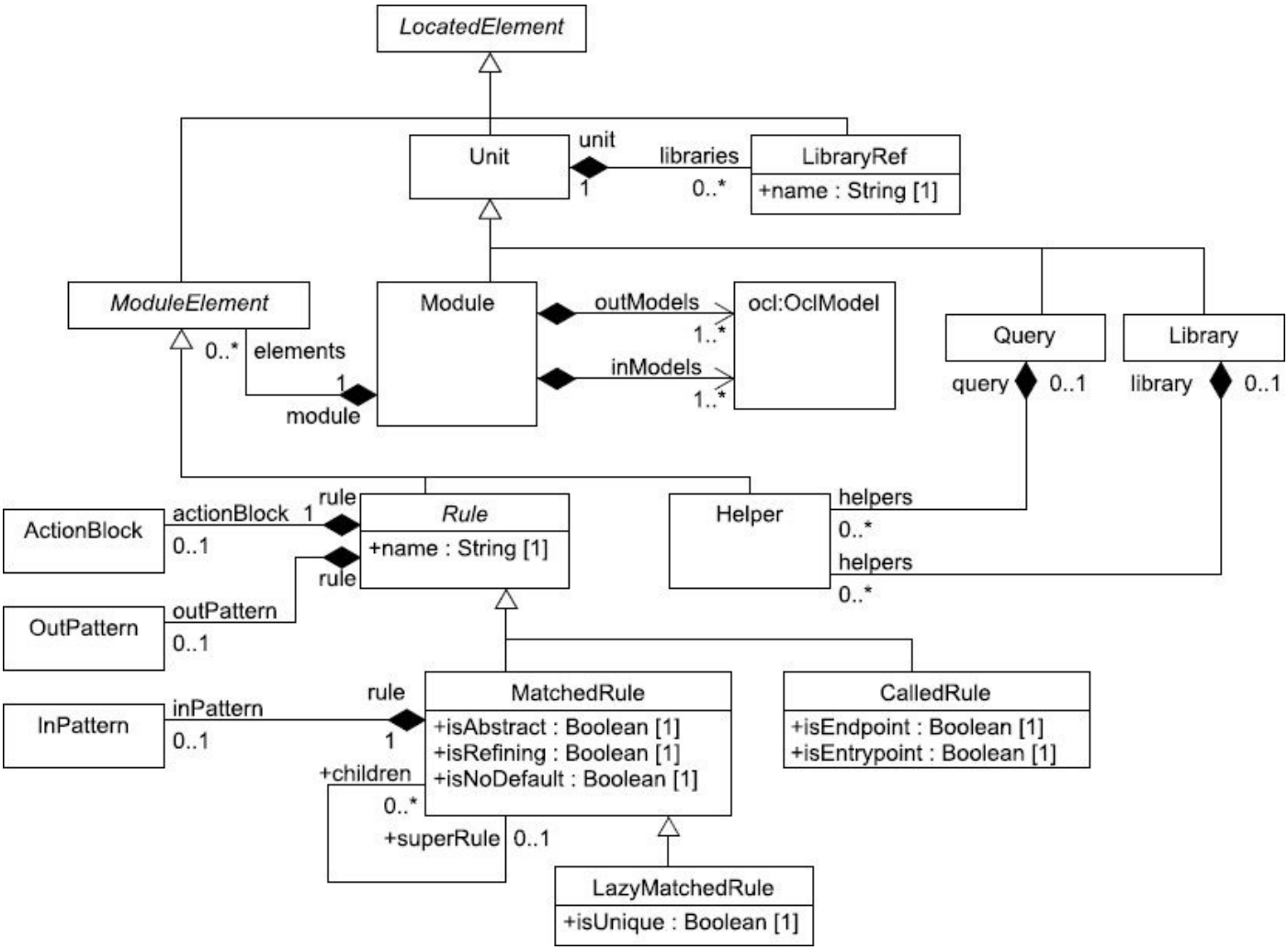


Figure 3.1.: The ATL abstract syntax: *Modules* and *Rules*

Figure 3.1 contains an excerpt from the ATL abstract syntax metamodel[3]. One transformation is made up of one `Module`-element that references a set of in and out `OclModels` which define the Ecore source and target metamodels of the transformation. Each transformation is made up of `ModuleElements` which can bei either `Rules` or `Helpers`. `Helpers` provide a mechanism to reuse OCL expressions and are called from within transformation rules.

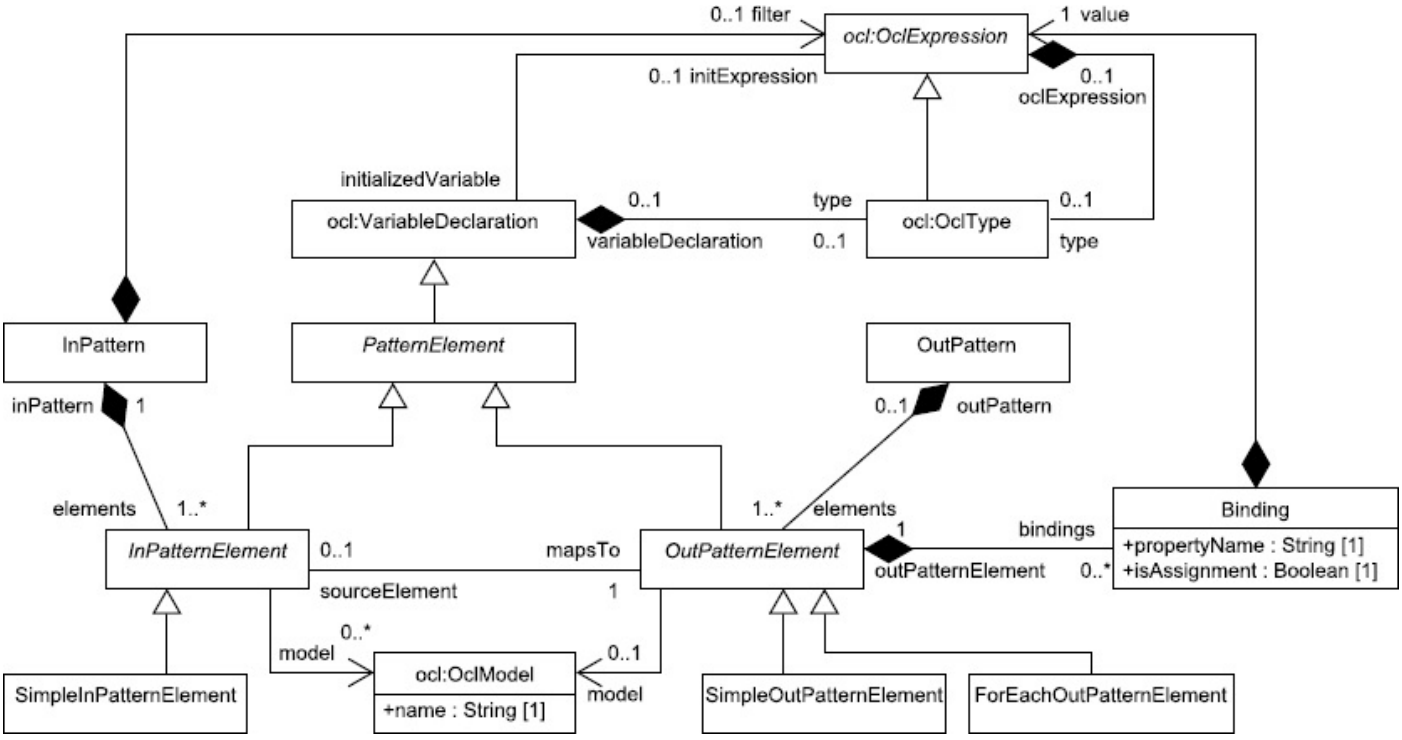


Figure 3.2.: The ATL abstract syntax: *In- and OutPatterns*

Rules consist of patterns and an optional `ActionBlock` that can contain imperative statements that are executed after the target model elements are initialised using the rule patterns. Rules can be either `MatchedRules` or `CalledRules`. Matched rules execute for each `InPattern` that is matched in the source model. Called rules can be called from matched rules and provide a mechanism to structure complex rules and can be reused from different matched rules. Both rule types have an `OutPattern` that describes the changes made to the target model. The detailed syntax for patterns is shown in Figure 3.2.

Patterns are used in ATL to describe and locate the parts of source and target models that are used in a transformation rule. Patterns are made up of one or more `PatternElements` which are OCL variable declarations and can be used as such in expressions. The matching process binds concrete model element instances to the declared pattern elements (variables) during execution. Pattern elements are further divided into in- and out-pattern elements for the LHS and RHS side of the transformation rule.

```
1  module transformation;
2  create OUT: emflib from IN: imdb;
3
4  rule Film2VideoCassette {
5   from film:imdb!Film
6
7   to cassette:emflib!VideoCassette (
8     title <- film.title,
9     damaged <- false,
10    cast <- film.figures
11  )
12 }
13
14
15 rule Figure2Cast {
16  from figure:imdb!Figure
17
18  to cast:emflib!CastMember (
19   firstName <- figure.playedBy.name,
20   lastName <- figure.playedBy.surname
21  )
22 }
```

Listing 3.1: A simple example of an ATL transformation

Each matched rule has one `InPattern` to describe the source model part that must be matched for the rule to execute. The in-pattern consists of exactly one[4] `SimpleInPatternElement` which matches a model element `(OCLModel)` and is mapped to exactly one out-pattern element. The mapping reveals which target element was created from which source element and can be used for tracing purposes. The `OutPattern` of a rule consists of one or more `OutPatternElements` which can be either a `SimpleOutPatternElement`, which denotes a single model element that is created in the target model or a `ForEachOutPatternElement`, which allows the creation of a number of target model elements of the same type for each match of the pattern element. Out-pattern elements are initialised using a set of `Bindings`.A binding describes how the value for an attribute of a target model element is calculated. The value calculation is given by an OCL expression which can make use of any valid OCL variable in the current (rule) scope, i.e. including all pattern elements of the rule.

Listing 3.1 contains a simple example of an ATL transformation. It transforms elements of the IMDB metamodel to elements of the extended EMFLibrary metamodel (see the application scenario introduced in Section 1.2). The transformation contains two matched rules: the first one matches instances of type `Film` and creates instances of type `VideoCassette` in its out-pattern, setting a value for the title and the boolean attribute `damaged` to **`false`** (assuming all new video cassettes in the library are not damaged). The `cast` association is linked to the `figures` association of the source `film` element, which means that the values are to be taken from the rule that matched the type of the given source association, which is the second rule of the transformation as it matches the `Figure` type.

The second rule creates a `CastMember` for each `Figure`. The values of the properties `firstName` and `lastName` are calculated using OCL expressions which navigate the model

along the `playedBy` association to the associated instance of `Actor`, from which the values of `name` and `surname` are read.

When the transformation is executed it creates a `VideoCassette` for each `Film` in the source model with an associated `CastMember` for each `Figure` defined for the `Film`. The values of the `CastMember` attributes are taken from the `Actor`-instance associated with the `Figure` in the source.

The ATL project [95] provides an editor for the Eclipse framework and a virtual machine to execute ATL transformations, which is described in the next section.

3.2.2. The ATL Virtual Machine

ATL makes use of a virtual machine that executes ATL-specific byte-code to perform a model to model transformation. The specification of the ATL Virtual Machine (ATL VM) describes an abstract computing machine for which two implementations exist; a model infrastructure independent reference implementation and one based directly on the EMF Ecore implementation which is optimised for the transformation of Ecore-based models. [6]

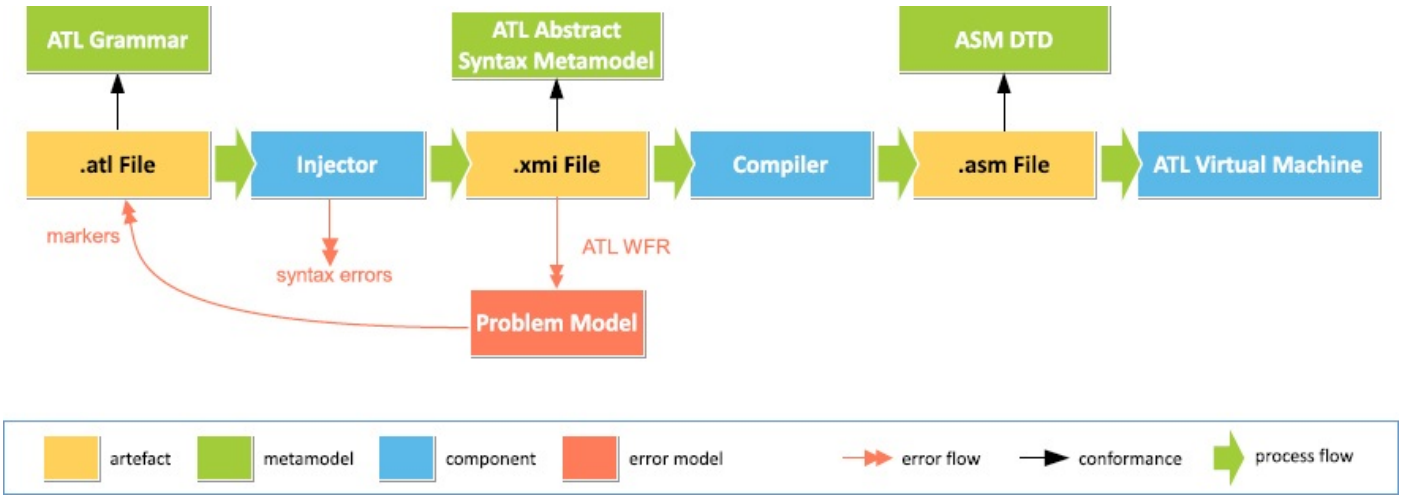


Figure 3.3.: The ATL compilation process [96]

The compilation process is as follows (see Figure 3.3): Transformations are written in the ATL textual syntax (.atl file). The textual rules are processed by an injector into an Ecore-based model that conforms to the ATL abstract syntax. The injector reports any syntax errors. The model is validated and occurring problems are collected in a problem model that links to markers in the textual representation to indicate the error sources. After the compilation to the VM-specific byte code, the resulting .asm file can be interpreted by the ATL VM. The intermediate compilation process allows for alternative languages to be used to specify transformations on top of the ATL VM. [96]

3.3. MOF Query/View/Transformation (QVT)

The Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) is a model transformation framework specified by the OMG as part of the MOF collection of modelling standards [73]. In general terms, it provides model transformation functionality for MOF-based languages like the UML. It caters both to the use of transformations as executable programs, where output models are created or modified based on input models

and to the use as relations, to define and verify rules that must hold between models. Transformations are made up of rules between models expressed in terms of their MOF-metamodels and additional OCL expressions. They can be defined between an arbitrary number of models, where a transformation between the same model as input and output is an *in-place* transformation, but a transformation can also create a model based on a number of different input models. As the direction in which a transformation is executed can be chosen at runtime, the definition of bi- or multi-directional transformations is possible.

QVT is made up of three distinct but related languages: the declarative QVT Relations (QVT-R) and QVT Core (QVT-C) languages and the imperative QVT Operational Mappings (QVT-O) language. The QVT-C language is designed to be the smallest and to be executed by a virtual machine in analogy to the Java virtual machine and to serve as the semantic basis for the other languages. The other two languages are made to be more verbose, reside on a higher level of abstraction and be better understandable by humans. They can be transformed into QVT-C for execution, but the standard also foresees tool vendors to implement the direct execution of the higher-level languages as long as the results are correct. [73, p. 10]

3.3.1. Semantics

The OMG specification provides no formal semantics of the QVT languages. The QVT-R semantics are provided by a mapping to QVT-C, which in turn is only semi-formally defined [41]. Other scientific work provides some approaches to a formalisation of parts of the QVT, like supporting QVT-R with Coloured Petri Nets by de Lara and Guerra [59], their algebraic semantics for check-only execution of QVT-R [41] or the alignment of the hybrid relationship between QVT Operational Mappings (QVT-O) and QVT-R with problem theory by Giandini et al. [39]. Further semantic issues stem from the lack of formalisation of OCL, which is used as part of QVT and the lack of formal semantics of MOF (see sections 3.4 and 2.3).

3.3.2. The QVT Core Language

The QVT Core (QVT-C) language is a declarative language to define transformations on the basis of pattern matching over sets of variables against models. Although suited for transformation execution, it is used mainly as a foundation for the QVT set of languages as it serves to illustrate the semantics of QVT-R in the QVT standard. Most aspects of the QVT-R language are defined by a transformation from the QVT-R syntax to the QVT-C syntax. [73, p. 163] In this vein QVT-C is designed to be as expressive as QVT-R (and QVT-O) but to be simpler, which makes the transformation descriptions in QVT-C more verbose than in the other languages. [73, p. 145]

The QVT-C language is build for the definition of transformations using a number of *Mappings* that relate sets of *Patterns* of candidate models to each other. Patterns define variables, constraints and value assignments and can be matched against model instances in execution to define relevant parts of candidate models to a transformation (a *Binding).* The number of metamodels *(Domains)* usable in a transformation is not fixed; mappings

can be defined for instances ranging from one metamodel (in-place transformation) to potentially any number of metamodels.

The execution of QVT-C transformations is defined for two modes. In *checking* mode, bindings are calculated for candidate models and errors are reported for inconsistent mappings. In *enforcement* mode, one domain has to be chosen and inconsistent mappings are fixed by creating or deleting elements to fulfil the transformation. QVT-C specifies a tracing mechanism, but provides no metamodel for tracing models nor a mechanism to derive such. How the tracing information is derived for an execution run is transformation specific and explicitly specified in each mapping definition. The tracing metamodel is as such one domain of a multi-directional transformation. In consequence, the common use case of transforming models of one metamodel into models of another metamodel would result in a transformation of three domains (source, target and tracing) which is executed in enforcement mode in the target direction.

3.3.3. The QVT Relations Language

The QVT Relations (QVT-R) language features the highest level of abstraction of the QVT languages. It has a declarative language style and provides the potential for multi-directional interpretation of relations between multiple metamodels, while the common cases found are unidirectional transformations with a defined source and target or bidirectional transformations, if a relationship between two metamodels is established. The QVT standard contains a textual concrete syntax and the suggestion for a graphical variant of the concrete syntax for QVT-R [73].

The central concept of all QVT languages is the *pattern* and the accompanying pattern matching semantics for models [73, pp. 21]. A pattern defines a part of a model that features in a transformation, in terms of its metamodel. During the execution, candidate (input) models are checked for occurrences of the pattern, where each valid match is a binding instance. This basic concept is defined in the *QVTBase* package and extended by the different languages (QVT-C, QVT-R, QVT-O) for their individual interpretations. In the QVT-R language, a transformation consists of one or more *relations*, which establish a set of constraints between candidate models. Transformations can be executed in two modes. In the *checking* mode, the relations (constraints) between all candidate models are checked and violations are reported. If the constraints hold, the transformation is said to be valid. In *enforcing* mode, the constraints are enforced by modifying one selected model as the target of the transformation execution until all relations hold (or the transformation fails). The model is modified by creating or deleting elements and setting values for simple properties.

Relations between models are expressed using one or more *domains*, which are typed variables with an associated pattern. Relations can define further variables and contain two sets of predicates, the *when* and *where* clauses, that provide further detail to the relation. For the transformation to be valid, the relation must hold only when all constraints in the when-clause hold (pre-condition). If the relation holds, all constraints of the whereclause must also hold (post-condition). Other relations can be referred to in the clauses, providing a structuring mechanism for relations and subrelations.

Domain variables and associated patterns are also known as *object template expressions* as they specify how elements for target models are created in enforcing mode. The variables of a domain are bound to values during the matching process. For the remaining unbound (or *free)* variables, the object template expression specifies how values are to be created. QVT-R makes use of OCL for expressions to calculate values or navigate over model elements. The QVT-R abstract syntax extends elements of the EMOF and OCL metamodels to tightly integrate with the OMG family of languages.

```
1  transformation Film2Cassette(source: imdb, target: library)
2  {
3   top relation CreateVideoCassette {
4    checkonly domain source film : imdb::Film
5    {
6    };
7
8    enforce domain target cassette : library::VideoCassette
9    {
10    title = film.title,
11    damaged = false
12   };
13  }
14
15  top relation CreateCastMember {
16   checkonly domain source figure : imdb::Figure
17   {
18    film = film : imdb::Film {}
19   };
20
21   enforce domain target cast : library::CastMember
22   {
23    firstName = figure.playedBy.name,
24    lastName = figure.playedBy.surname,
25    starsIn = cassette : library::VideoCassette {}
26   };
27
28   when {
29    CreateVideoCassette(film, cassette);
30   }
31  }
32 }
```

Listing 3.2: A simple example of a QVT transformation

Listing 3.2 contains a simple transformation in the QVT-R textual syntax. It transforms elements of the IMDB metamodel to elements of the extended EMF-Library metamodel (see Section 1.2). The transformation is made up of two relations: the first one creates instances of type `VideoCassette` from instances of type `Film`, setting a value for the title and the boolean attribute `damaged` to **`false`** (assuming all new video cassettes in the library are not damaged). The second relation creates a `CastMember` for each `Figure` and is a bit more complex: it defines a simple in-pattern to match all instances of `Figure` and binds them to the variable `figure`. This variable is used in the out-pattern / object template expressions to set the values for the `CastMember` instance that is created or updated. The actual values are calculated using OCL expressions: for the attribute `firstName` the expression `figure. playedBy.name` navigates the model along the `playedBy` association to the associated instance of `Actor`, from which the `name` attribute is read. For the attribute

`starsIn`, the other relation is referenced using the whenclause: it contains the pre-condition that the relation *CreateCastMember* only needs to hold when the relation *CreateVideoCassette* held previously for the two variables `film` and `cassette`, i.e. when a `VideoCassette` was created for the `Film` the current `Figure` belongs to. When the `starsIn` reference is set, the when-clause also ensures that a viable value has been bound to the `cassette`-variable.

When the transformation is executed in enforcement mode (not check-only), it is executed in the direction of the extended EMF-Library metamodel (already named `target` in the transformation) and an empty model is selected as the target, this model will contain a `VideoCassette` for each `Film` in the source model with an associated `CastMember` for each `Figure` defined for the `Film` after execution. The values of the `CastMember` are taken from the `Actor`-instance associated with the `Figure`.

3.3.4. QVT Relations Graphical Notation

Because of the general significance of the diagrammatic syntax of the UML, the QVT standard contains a suggestion for a complementary graphical notation to the textual syntax of QVT-R [73, p. 38]. Unfortunately, the syntax is not well-defined and the standard only provides a small set of examples and explanations in natural language, which causes ambiguity and provides room for interpretation. This has caused others to create and use different variants of the syntax (compare for example Marković and Baar's use in OCL refactoring [65] and Wachsmuth's use in co-adapting models [107]). Nevertheless, their usage shows that the graphical notation is very compact and arguably intuitive, so that it is used in this work as described below.

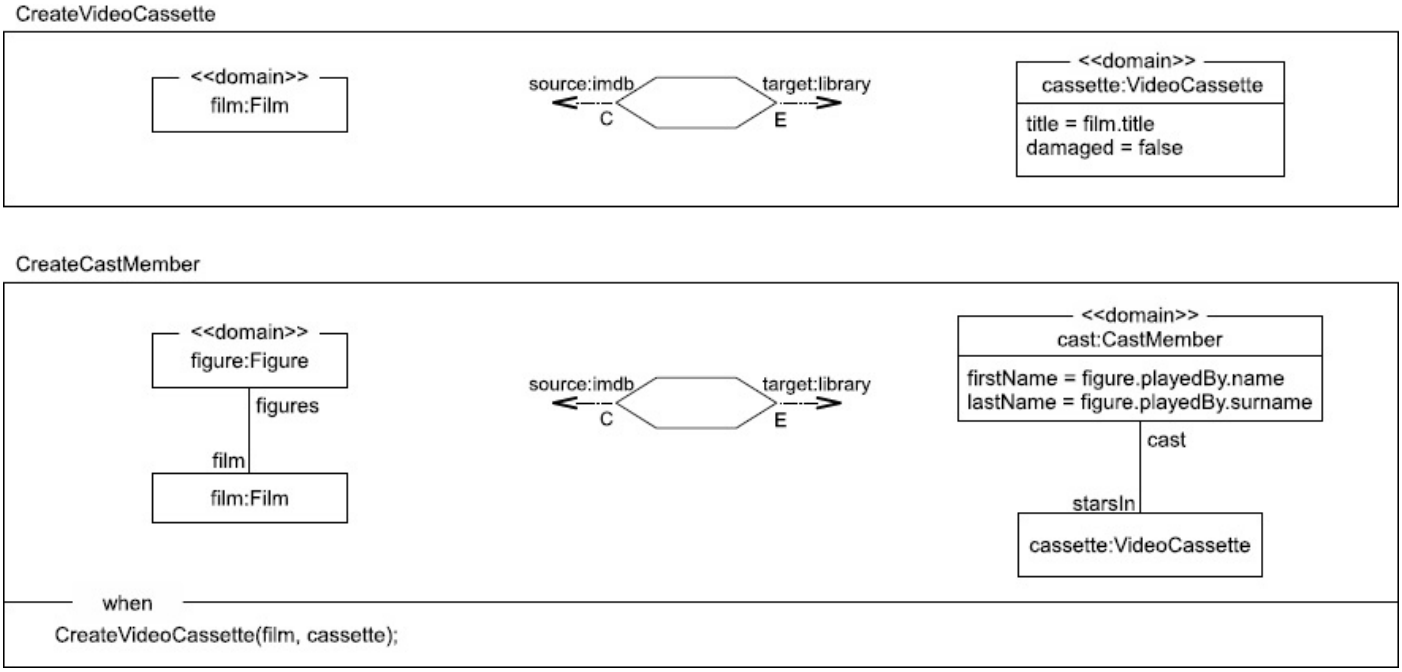


Figure 3.4.: The simple example of a QVT transformation in graphical notation

For each relation, the graphical syntax prescribes a rectangle that contains the patterns of the domains of the relation, a title with the name of the relation and optional regions below for the when- and where-clauses of the relation. The contents of the clauses is given in OCL textual notation. (See Figure 3.4 for the transformation example of listing 3.2 in QVT-R graphical syntax.)

Domain patterns are expressed using a notation similar to the UML object diagram notation. Table 3.1 details the syntax used and the assumptions and extensions made for this work. We annotate associations with their association-end names where necessary to distinguish between different possible links between elements. In places where the choice of association is unambiguous (because only one exists between two types), the names can be omitted.

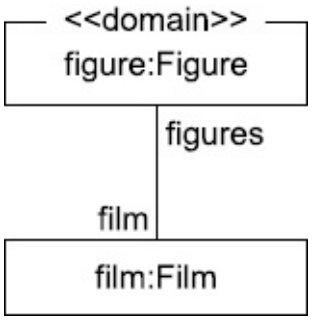
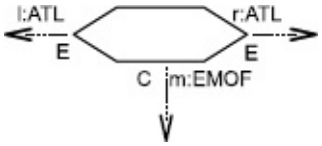
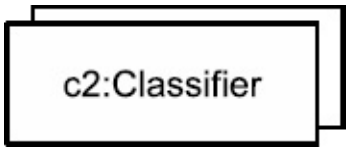
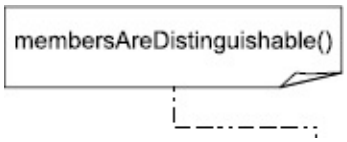
	Object template patterns are expressed similar to UML object diagrams. The stereotype-like marker <<domain>> signals the domain of the pattern. While not explicitly stated by the QVT standard, we annotate the association-ends to differentiate the links involved.
	The 'relation symbol' configures the relation, where each arrow defines one domain. It shows the model and metamodel used (l, r, m and ATL, EMOF) and whether a direction is checkable and/or enforceable (C/E). Here we assume the standard to imply that a relation with more than two domains has an arrows per domains.
	This notation represents a set of elements. Unfortunately it does not indicate which collection type was used, which makes it ambiguous. We suggest using annotations in the form of comments where necessary.
	The 'not template' matches when no element fulfils the template in a pattern. This is to be equivalent to a constraint that forces the association that links the element to be empty such as `->isEmpty()` or `->size()=0`.
	Constraints can be attached to pattern elements using this notation. It is assumed to be exchangeable with using the constraint in the when-clause.

Table 3.1.: Elements of the QVT-R graphical syntax as adapted from the QVT standard in version 1.1 [73].

3.4. The Object Constraint Language (OCL)

This section provides a short introduction to OCL and its usage in transformation languages. Another aspect of OCL is important to this work, namely the co-evolution problem of UML models and associated OCL constraints. Research in that area provides some foundation and is covered in detail in Section 8.2.

3.4.1. The OCL and Model Transformation Languages

The Object Constraint Language (OCL) is a formal textual language to describe invariant conditions or queries for model elements of the OMG family of languages (most commonly known to specify constraints on UML classes). OCL is usable with MOF and any MOF-based languages and is used in transformation languages like QVT and ATL. This work uses OCL according to the OMG Object Constraint Language specification in version 2.3.1 from January 2012 [74] or in the form of the ATL OCL dialect where noted.

OCL itself is free of side effects, meaning that OCL expressions can not change the model they are assigned to. But the specification allows for the use of OCL to describe operations that manipulate models when executed. This makes OCL usable in transformation languages to describe the changes the application of a rule performs when executed. OCL can also be used to formulate constraints and for its type system. Specifically, OCL fulfils three purposes in the transformation languages QVT and ATL:

Filter Expressions: Filter expressions are used to further restrict the matching of model elements in rules and thereby the applicability of rules. The initial selection of model element candidates for rule matching is done by type. This set of elements can then be reduced further by boolean OCL expressions that are applied to each element.

Model Navigation: The selection of sets of elements using some element as starting point and the manipulation of sets of elements is achieved with OCL navigation expressions both in ATL and QVT. Navigation expressions begin with some variable that refers to a model element and then describe the navigation across associations to other linked sets of elements. The sets can be reduced using OCL filters and manipulated using set operations. In this manner OCL provides the mechanisms to define patterns of model elements for transformation languages.

Value Calculation: The values of simple types are calculated using the OCL type system and expressions for value binding in ATL and QVT. The available operations provide String and Integer manipulation for example.

3.4.2. The OCL Context

As both QVT and OCL are maintained by the OMG, the QVT syntax specification and abstract syntax models import the EssentialOCL specification and model directly. Variables used in QVT inherit from `EssentialOCL::Variable` and expressions are of type `EssentialOCL::OclExpression` [73, p. 159].

ATL on the other hand uses its own dialect and implementation of OCL instead. The implementation is not complete, for example, the type-casting OCL expression `.oclAsType()` is not available in ATL, which prevents some resolution strategies for ATL transformations (see Section 7.7 for example).

Semantics of the OCL Context

OCL expressions are written in relation to a syntactical context; in general this context is given by an object model on which the expression is to be interpreted. The OCL standard defines the context for OCL expressions for invariants and pre- and postconditions when

used to constrain UML models. The most general context is the UML model itself and the data signature Σμ of an OCL expression contains the types and operations on attributes available for the expression. [74, p. 212]

For invariants and pre- and postconditions the context can further be declared with the notion of *free variables*, which are defined using a `context` clause. Any number of variables can be explicitly defined and typed or the provided variable `self` is typed and then usable in a subject expression.

Essential OCL

In alignment with the separation of MOF into CMOF and the smaller EMOF (see Section 2.3) and the Ecore implementation of EMOF for Eclipse, a minimal set of OCL is available for EMOF that references EMOF types explicitly [74, p. 187]. It is called *EssentialOCL* and is implemented for Ecore and the Eclipse framework as part of the Model Development Tools project[5].

---

[3] The graphical representation of figures 3.1 and 3.2 was created from the `ATL.ecore` abstract syntax definition available in the ATL eclipse plugin *org.eclipse.m2m.atl.common* in version 3.3.1 available from the Eclipse Modeling Project [95].

[4] The ATL abstract syntax is imprecise here as the multiplicity for in-pattern elements (1..*) would cater for more than one element per in-pattern, yet the editor and concrete syntax only allow one. [5, p. 37]

[5] http://www.eclipse.org/modeling/mdt/

CHAPTER 4

Software Evolution and MDE

The evolution of software is one of the big challenges in software engineering [69]. The complexity of software systems is constantly increasing. At the same time, the business needs which drive the development of software continue to change with growing pace and software systems are expected to adapt accordingly. Depending on the sources, maintaining and adapting software systems after the initial development process are estimated to make up to 90% of the overall cost [89, p. 489].

The challenge of software evolution and dealing adequately with the growing complexity of software systems over time has been known in software engineering for a long time. In the 1980s, Lehman proposed eight laws of software evolution based on case studies and analysis of software processes over a 20 year period. According to these laws in summary, any *real-world software system* (in the sense that it provides a solution to an acceptable approximation of a real world problem or that it is embedded in the real world as a universe of discourse) is bound to undergo continuous change or become less (cost-) effective over time instead. The need for evolution is *intrinsic* to large software systems and evolution increases software complexity unless actively countered. [60, 61]

Because of the high cost and slow realisation associated with software evolution, in 2000, Bennett and Rajlich draw up a research roadmap for software maintenance and evolution for the next ten years to identify key problems and find promising solutions [10]. In this roadmap, the three research topics directly associated with the software evolution stage in the software life-cycle are:

1. Architectures which will allow considerable unanticipated change in the software without compromising system integrity and invariants.
2. Architectures which themselves can evolve in controlled ways.
3. Managing the knowledge and expertise of the software team.' [10]

Furthermore,

> 'a highly promising area of research is therefore to find ways to raise the level of abstraction in which evolution is expressed, reasoned about and implemented.' [10]

Evolution is also recognised as an architectural concern; the ISO/IEC/IEEE 42010 standard names the 'principles governing the evolution of the system over its life cycle' as a possible 'essential or fundamental characterization' of a system and thereby as possible constituents of architectures and architecture descriptions of interesting systems [48, p. 4].

Therefore, the question of how to formalize and apply evolution and to assists software engineers in performing evolution and analysing the impact of software evolution on software systems are important and ongoing research topics for software engineering in general.

The next section covers the specifics of MDE and how the problem of software evolution relates to the artefacts of the MDS. Refactoring as an approach to software maintenance

has been adapted for MDE and model transformations and is related to MDS evolution. It is addressed in Section 4.2.

4.1. MDE and the Evolution of MDS

MDE promises to support software engineers with suitable methods and tooling to handle the complexity of modern software systems - during development and maintenance. In essence, MDE strives to break down the complexity of a software system into manageable abstractions, each embodied by a suitable type of artefact. The benefits of MDE have been widely discussed both in industry and academia. Among these is the superior expressiveness of models over source code, the suitability of Domain Specific Languages (DSLs) to capture specialized domains, the ability of better partitioning systems according to (technical) concerns and describing their relationship using model transformations, the increased reusability of artefacts in the development process, mechanisms for concealing (implementation) detail where appropriate and more. [92]

Yet, these expected benefits can only be fully realised when the complexity of the MDS itself remains manageable. Due to the high number of artefacts and the complexity and diversity of artefact types, the MDS itself becomes susceptible to maintenance issues. Therefore the challenges posed to MDE are the same as those of regular development methods. These range from applying suitable development processes, handling distributed development to the versioning of models and model driven artefacts. If the effort necessary to maintain the MDS (being ‘only’ a representation of the software system of concern) becomes overwhelming and circumventing changes to the software seem beneficial, its representative nature and overall use are threatened. For this reason these is a need for methods and tooling for the evolution of the MDS [71].

Atkinson an Kühne relate the problem of evolution in MDE directly to the software artefacts common in MDE and the goal of improving short-term and long-term productivity (see section 2). A large amount of long-term productivity depends on

> ‘the longevity of a primary software artifact. The longer an artifact remains of value, the greater the return derived from the effort that went into the creation of that artifact. Thus, a second and strategically even more important aspect of MDD is reducing the sensitivity of primary artifacts to change.’ [4]

Therefore, one key aspect of the value of MDE lies in the means of handling the evolution of the artefacts involved. Atkinson an Kühne further identify four fundamental forms of change with particular importance to MDE artefacts and tooling [4]:

- **Personnel** Knowledge and skills are always bound to the developers working in projects; as the project staff fluctuates, so increases the risk of a loss of information. This can be alleviated primarily by using a ‘concise and tailorable presentation’ of primary artefacts.

- **Requirements** Changing requirements are a common problem in software engineering. Atkinson an Kühne especially stress the need to reduce the impact of rapid requirements changes both on maintenance and in terms of disruption of online services. They see the need to support ‘dynamic addition of new types at runtime’ in

terms of primary artefacts.

- **Development Platforms** The constant evolution of development platforms may endanger the longevity of primary artefacts should these be dependent on a particular tool. A ‘high-level of interoperability’ between MDE tools is needed.

- **Deployment Platforms** The rapid evolution of platform technology is a growing problem. Shielding primary artefacts from platform change can increase their longevity – for example by automating the process of generating platform specific artefacts from platform independent artefacts by applying ‘user definable mappings’.

The need to provide explicit support for the handling of evolution of MDS can be derived from the first two areas of change. For one, with the fluctuating nature of project personnel, providing a ‘concise and tailorable presentation’ of the evolution of the MDS along with the primary artefacts should counter the expected information loss. Furthermore, the thread to the longevity of artefacts from changing requirements can be countered by providing adequate tooling to describe, perform and predict the impact of evolution.

To this end, Engels et al. capture the evolution of models as atomic, elementary steps. In their method, model transformations are used to capture single evolution steps. These can be composed to express the complex evolution of a model. For each step, the consistency preserving properties are defined. [28]

One advantage of representing evolution as elementary steps and combining these to form more complex evolution tasks is the exploitation of the *locality* principle from component-based software development approach for MDE. This means that the effects of the changes made are kept local. In general, the consistency of the overall system may be endangered by an evolution. But if the consistency preserving properties of a set of evolution steps can be shown, any evolution that is made up of consistency ensuring steps can be performed without harm. Engels et al. show the consistency preserving attribute of creation and deletion rules (evolution steps) for specific domain elements and in relation to their chosen domain (not in general):

> ‘In order to manipulate a given model we have to determine the kind of change we want to achieve as well as its location in the model. This corresponds to the selection of a rule and its application area. Then, the rule may be applied provided that all those application conditions are satisfied that are required for the consistency properties to be preserved. If there is no consistency-preserving transformation for the change we intend to do, we have to resort to ad-hoc evolution with a complete consistency check of the new model.’ [28]

Wachsmuth follows this approach in providing a method for the *co-evolution* of metamodels and models [107]. Co-evolution is defined by Favre as the problem of requiring system architecture and implementation to evolve in a ‘synchronized way’, as they are linked, because otherwise ‘architectural drift’ or ‘architectural erosion’ can occur [31], which are maintenance issues of the MDS. According to Favre, the co-evolution problem also occurs for other linked artefact types, like languages and programs, model and metamodels, and more. We therefore define co-evolution for this work as follows:

Definition 7 (co-evolution): *Co-evolution is the synchronized evolution of linked artefacts to maintain the consistency of the MDS. The* co-evolution problem *refers to inconsistencies that may occur when artefacts are changed individually.*

Wachsmuth's approach provides for the stepwise co-evolution of linked models and metamodels by providing a set of 16 adaptations that can be applied for common and recurring types of changes [107]. The adaptations prescribe how both models and metamodels need to be adapted to remain consistent. The approach is based in parts on concepts of object-oriented (OO) refactoring which has been adapted to MDE. The next section covers model refactoring in more detail. In chapter 8 a number of different approaches for the co-evolution of related artefacts of the MDS are compared.

4.2. Model Refactoring

Refactoring plays an important role in the maintenance of software systems in OO software engineering. According to Fowler,

> 'Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.' [34, p. xvi]

Opdyke initially provided a catalogue of refactorings ('restructuring operations') in his PhD-thesis for programs written in C++ [80]. The list of refactorings has been extended by Fowler [34] and others. Opdyke defines the behaviour preserving properties of refactorings according to seven criteria, of which six relate to syntactic correctness and the seventh prescribes 'semantic equivalence' as follows:

> 'let the external interface to the program be via the function main. If the function main is called twice (once before and once after a refactoring) with the same set of inputs, the resulting set of output values must be the same.' [80, p. xvi].

Therefore a refactoring is correct if it provides a syntactically correct output program from any applicable syntactically correct input program and the refactored program delivers the same output for the same input as before the refactoring. It has been shown that the behaviour preserving property of refactorings is difficult to prove in general, so that more fine grained criteria such as 'access preservation', 'update preservation' and 'call preservation' have been proposed to be used instead [7, 70].

Fowler states that refactoring serves a number of purposes, from making source-code easier to read and grasp for humans to preparing a software system to be extend with new features. Refactoring is an ongoing process, starting with the first steps in implementation and continuing over the entire software life-span. It is a key activity in software evolution, as it maintains evolvability (in improving structure) or in preparing new features. [34]

Many approaches have been proposed that carry over the idea of refactoring from OO-technology to MDE [64], resulting in the field of *model refactoring*. While refactoring in the object-space is well understood, refactoring in the model-space is ongoing research, mainly because what constitutes the internal behaviour of a model is an open question [104]. Approaches include catalogues of refactoring patterns for UML class diagrams [3,

14] or Marković and Baar's formalisation of the most important refactoring rules for UML class diagrams and dependent OCL constraints [65].

In model refactoring, the central artefacts of interest are models. This provides a set of challenges open to research [67], two of which are:

- **Model quality** As the central goal of refactoring is the improvement of the quality attributes of the model in question, a precise definition for quality in terms of models is needed. Models are the representation of a system for some purpose, so that the quality of the model may hinge on how well the model fulfils this purpose. Mens et al. further suggest attributes like usability, readability and performance as possible quality aspects for models, as well as the use of design patterns [35, 43] in models of OO-systems [67].

- **Behaviour preservation** The external behaviour of the artefact that is refactored should not be changed by the refactoring. This requires the definition of what properties make up the external 'behaviour' of a model and how they can be effectively measured. One hindrance for the precise definition of such properties for UML models is the lack of formal semantics of the UML [67] (see Section 2.3.1).

Model refactoring and evolution approaches in MDE are similar in their use of elementary steps to represent change that can be applied to models [7, 107]. In both cases, the simple steps can be combined to represent more complex scenarios. Also, in many approaches the basic sets of steps available are very similar and describe the same or similar kinds of model change. The main difference can be seen in the focus: refactoring approaches are mainly concerned with improving some defined properties of quality of the model in question while external behavioural or semantic properties remain unchanged. The focus of evolutionary approaches is generally broader; these also attempt to maintain the quality aspects of the MDS but cater to all kinds of change, explicitly including change of the external or semantic properties of the model [31]. Furthermore, co-evolution approaches take the dependencies between models and other artefacts into account and attempt to maintain the quality of related artefacts along with the changed artefacts.

PART II

Operator-based Co-Evolution of Metamodels and Model Transformations

CHAPTER 5

Supporting the Evolution of Model Driven Systems: A Stepwise Approach

In this chapter, we introduce an approach to support software architects in performing common co-evolution tasks for metamodels and ATL model transformations. The expected benefit of the approach is the reduced effort when performing commonly occurring changes to metamodels and the reduced effort in adapting dependent model transformations accordingly. Depending on the change performed and the resulting impact, a portion of the adaptations can be performed automatically while the software architect can be supported when performing manual adaptation for the rest. Furthermore, the approach allows the automatic detection of the impacts on dependent transformation.

The next section recapitulates the co-evolution problem for metamodels and model transformations first introduced in Section 1.1. Section 5.2 details the goal pursued in addressing the problem and Section 5.3 covers the requirements that need to be fulfilled by a solution. Section 5.4 introduces our proposed approach which is detailed in the subsequent sections. The last section covers the combined steps of our proposed approach.

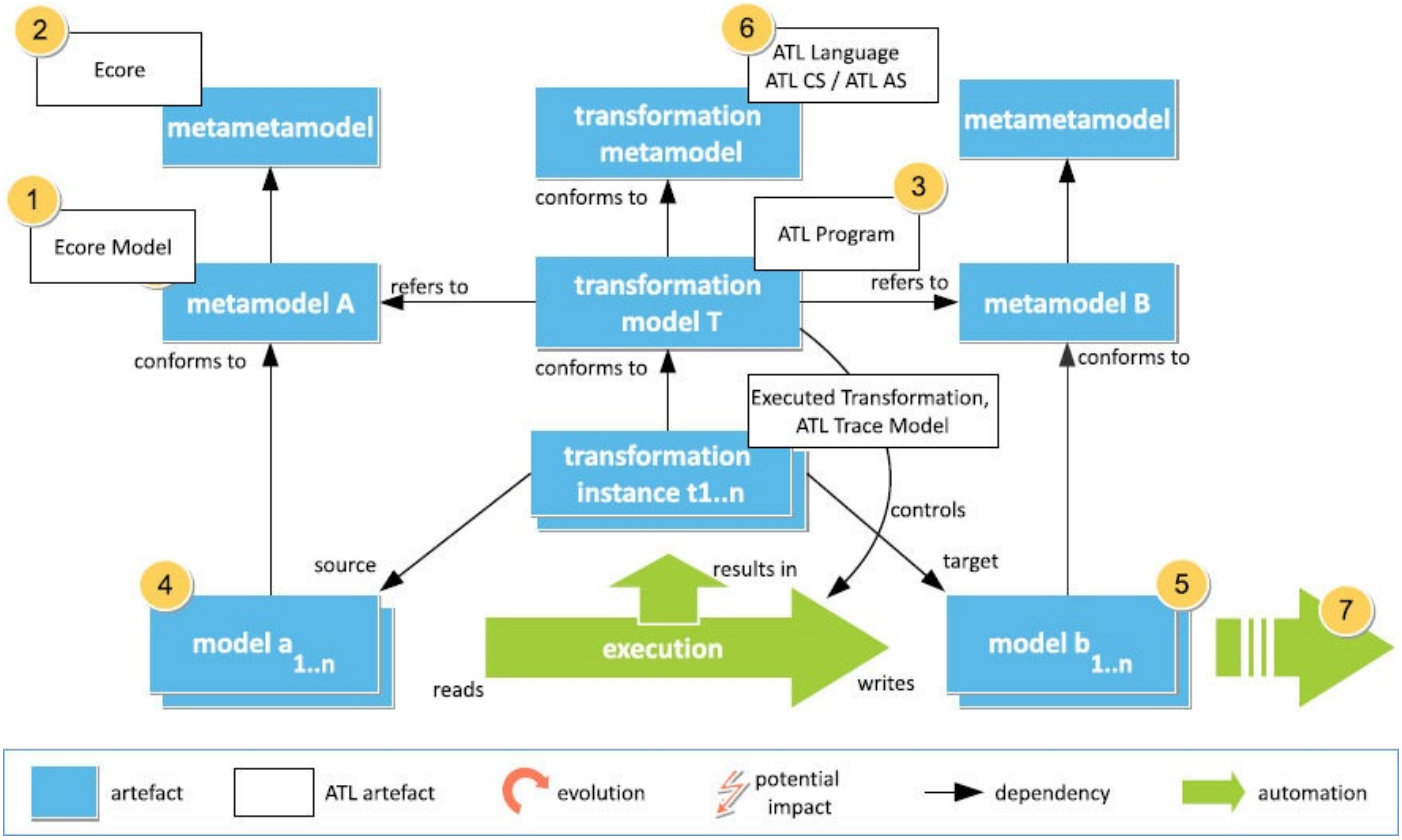


Figure 5.1.: The extended problem illustration

5.1. Problem Description

When using MDE methods to develop and manage complex software systems, the software systems are represented by a possibly high number of varying modelling artefacts. The artefacts themselves, their composition and the dependencies between them

may also be expected to be complex [9]. We refer to these artefacts collectively as Model Driven System (MDS) (see Definition 1 on page 19).

Figure 5.1 contains the schematic illustration of a typical use case of model transformations as part of the MDS, annotated with the artefact types of the ATL framework. It consists of: two or more different metamodels ① which specify the type of models for which the transformation is defined. The metamodels conform to the common Ecore metametamodel ②. The transformation is implemented as a set of ATL transformation programs ③ which transform a set of input models ④ on the LHS into output models ⑤ on the RHS. If valid, the ATL transformations conform to the ATL language ⑥, in that they conform both to the ATL concrete and abstract syntax.

The output models produced by the transformation may further serve as input for any number of further automated processes, for instance for the generation of source code ⑦. This usage adds additional artefacts and dependencies via the code generation framework to the MDS, like configuration files, generation code templates etc. The different artefact types involved are created and maintained by dedicated and specialized tooling, like the ATL editor which provides functions like code completion, graphical model editors for Ecore models that manage additional layout information and further typical development infrastructure for version control, testing etc. to support the process as a whole.

MDE promises to support software engineers to handle software system complexity by means of the superior expressiveness of models over source code, the suitability of Domain Specific Languages (DSLs) to capture specialized domains, the ability of better partitioning systems according to (technical) concerns and describing their relationship using model transformations, the increased reusability of artefacts in the development process and mechanisms for concealing (implementation) detail where appropriate and more. [92] Yet, the expected benefits can only be fully realised when the complexity of the MDS itself remains manageable. Due to the high number of artefacts and the complexity and diversity of artefact types, the MDS itself becomes susceptible to maintenance issues.

The artefacts involved in a model transformation can be said to be *tightly coupled* [93], as model transformation written in ATL navigate over and make use of model elements in many different parts of the transformation description; such as rule selection, rule application constraints, value calculation, element creation and more. There is no difference between public or private sections of models or transformations and no concept of an application programming interface. Changes to either the transformation or any of the involved metamodels can potentially impact any of the other artefacts.

Furthermore, transformations depend on a pattern of input and output model elements, which are more complex than method signatures in the object space. Declarative transformation languages allow the definitions of rules, so that one transformation can be made up of an arbitrary number of rules. These rules are defined on the basis of one or more metamodels, which makes the metamodels the types of the transformation [93]. Yet, typing transformations by the involved metamodels is very coarse-grained as any change to the metamodel would mean the creation of a new type, even if the transformation is not affected by the change. As a result, the interface provided by a transformation is not easily delimited and the internal and external behaviour defined. This makes the maintainability, reusability and evolution of transformations more difficult. [50, 108]

OO-technology explicitly provides mechanisms for the decoupling of artefacts and encapsulation of functionality – like class contracts or method access modifiers – which help to differentiate source code regions that are expected to be accessible by clients from those that can be changed safely without causing external impact. This means for instance that in Java, the impact of a change can be detected by the development environment in many cases; especially if the change is static and the impact can be determined by the compiler. In Java, any sophisticated development infrastructure has information at hand to determine many types of impact. When changing a method signature, all uses of the method in the current project can be determined and adapted for example.

Similar concepts either don't exist for model technology or are not as readily available: for instance, encapsulation by visibility of package content is only available in CMOF and not in EMOF [75, p. 35]. Another example is the navigability of associations; whether or not an association is navigable may vary with the use case:

> 'If an end [of an association] is not navigable, access from the other ends may or may not be possible, and if it is, it might not be efficient. Note that tools operating on UML models are not prevented from navigating associations from non-navigable ends.' [77, p. 38]

This means that non-navigable association-ends may be made available for navigation by a tool or platform implementation regardless.

The complexity caused by the number and variety of elements in an MDS and the tight coupling of elements motivates the need for suitable approaches to support software architects in common development and maintenance tasks. The programming environments used to create metamodels and model transformations play a key role in this goal. Model transformation languages are programming languages and share the common properties and functionality of other programming languages [1]. The rules that represent transformations are expressed as programs consisting of declarative or imperative statements in a textual concrete syntax. The statements undergo lexical and syntactical analysis phases that provide an abstract syntax tree (AST) of the transformation program. In the cases of QVT and ATL, the AST is then translated into a simpler but more verbose code that can be executed by a virtual machine to execute the transformation [73, 95].

The features of model transformation approaches described above are implemented in textual editors that provide mechanics to aid architects, like code completion and syntax highlighting. For this purpose, the editors maintain an AST of the current transformation in memory. Changes made to the transformation program in source code are parsed and when recognized, inserted into the AST immediately. As transformations are closely linked to the metamodels they are defined on, some environments also maintain a representation of the metamodels involved in memory and provide code completion for expressions that use model elements. One particularity of tooling for transformation languages as opposed to other programming languages is that models are traditionally created in different environments than the textual programming editors. For example, Ecore models can be created in graphical notations or a table-like view on an XMI document or in pure Extensible Markup Language (XML). This means that the maintenance of the relation between the AST of a transformation and the metamodel representation in memory needs to be updated across different editors and languages to

offer the same support as purely textual languages. This adds to the complexity of MDE tool support [109].

In summary, the complexity caused by the number and variety of elements in a MDS and the tight coupling of elements motivates the need for suitable approaches to support software architects in common development and maintenance tasks. This constitutes the goal of our approach discussed next.

5.2. Goal: Providing Support for Co-Evolution

The goal of this work is the development of a method to support software architects in performing common co-evolution tasks for metamodels and model transformations. *Co-evolution* tasks entail changing dependent artefacts together to maintain overall consistency [22].

Consistency, in the case of metamodels and model transformations, is for one the suitability of the metamodel to represent the input and output models of the model transformation. But the expected function of the transformation also needs to be taken into account; an evolutionary change can introduce inconsistencies that cause a transformation to produce the wrong output, although it is still executable.

The benefit of such a method is the reduced effort of performing changes to the MDS due to the partial automation of managing the dependencies of artefacts and providing a solution to combined evolution. Additionally, the use of operators as opposed to ad-hoc changes supports the prediction of the impact an evolution step has and the effort needed to resolve it.

5.3. Requirements

To achieve the goal described in the previous section, a number of requirements have to met. These cover the functional aspects of evolution and impact resolution and ensuring correctness. Furthermore, the method should integrate with state-of-the-art MDE technology and should provide tooling that integrates with common MDE tools. The topical sections below cover the requirements in detail.

Evolution Support The central benefit of a support method is the reduced effort of performing changes to metamodels and managing the dependent model transformations. The method should support software architects with the most common types of changes:

Requirement 1 (Support common evolutionary changes): *The method should support a set of evolutionary changes that are commonly encountered in metamodel and transformation co-evolution.*

Impact Detection When a change is applied to a metamodel, any dependent model transformations may or may not be affected by the change, depending on the type of change and the type of relation between metamodel and transformation. To support the software architect in predicting the impact of the change and finding a suitable strategy to resolve the effect, the method should provide a mechanism to detect which parts of dependent transformation descriptions are impacted:

Requirement 2 (Detect impact): *For the provided change types, the method should identify the parts of a dependent transformation description that may be impacted by the change.*

Impact Resolution When the impact of a change on a dependent transformation description is known, and an automatic resolution is possible, the method should update the transformation accordingly:

Requirement 3 (Resolve impacts automatically): *The impact of a change on a model transformation should be resolved automatically, if possible.*

If an automatic resolution is not possible, the method should provide suitable suggestions for a manual resolution:

Requirement 4 (Support manual resolution): *If the automatic resolution of the impact of a change is not possible, the method should provide adequate user support for the manual resolution of the impact.*

Application Support The method and the supportive tooling should ensure that changes are performed correctly. This means checking that all conditions necessary for a change type are met by the metamodel and the targeted elements before the change can be applied and that the metamodel is in a valid state afterwards. The conditions to be met can be expressed as pre- and post-conditions:

Requirement 5 (Ensure applicability of change): *The method should supply pre- and post-conditions for each change type to ensure that the change is applicable and that the metamodel is in a valid state state before and after the operator is applied.*

MDE Technology and Principles Riebisch defines six fundamental principles of software engineering to manage the complexity of software systems. These are: *Abstraction* (‘Abstraktion’), *Modularisation* (‘Modularisierung’), *Encapsulation* (‘Kapselung’), *Hierarchical Decomposition* (‘Hierarchische Dekomposition’), *Separation of Concerns*, and *Consistency* (‘Einheitlichkeit’) [82, p. 72-77]. The method and the supportive tooling should generally adhere to these design principles to be useful. Specifically, the use of MDE methods and languages are seen to support the principles of *Abstraction* and *Consistency.* Furthermore, formal methods and languages should be used to describe changes and impact resolutions to improve correctness and to avoid ambiguity for both the method and supportive tooling:

Requirement 6 (Use state-of-the-art modelling): *The evolutionary changes and the impact resolutions should be described using stateof-the-art MDE modelling languages and methods.*

Tool Support Tool support for the method should be integrated with other tooling commonly used in developing metamodels and transformations and follow interaction patterns close to those provided by tooling for similar tasks. For instance, refactorings are commonly applied by selecting a set of target elements or portion of source code and choosing the refactoring to be applied from a context menu. The application is performed through a dialog called a ‘wizard’ which provides a preview of the effect of the refactoring. If the user is satisfied with the predicted result, the refactoring is executed immediately and all effects take place. (See the refactoring support provided

by the Eclipse framework for the Java language [99] or that of the Elipse EMF Refactor plug-in for EMF models [97] for examples).

The application of co-evolution changes should be performed in a similar vein, if possible be integrated into the tooling commonly used for development and embedded in the general development process. This is expressed as Requirement 7:

Requirement 7 (Integrate with common tooling): *The tool support for the method should integrate with common MDE tooling for metamodel and transformation development and follow common usage patterns.*

Figure 5.2.: The three general phases of co-evolution

5.4. Approach

This section introduces our operator-based approach to describe changes commonly performed on metamodels and deriving impact calculation and resolution strategies for dependent model transformations from these. The details below address the requirements above directly or support the development of tooling to fulfil the requirements.

The approach is structured according to three general phases, as shown in Figure 5.2:

- **Phase 1: Metamodel Adaptation** During this phase, the software architect selects the change operator to be performed and the metamodel elements to perform the change on. The metamodel is adapted accordingly.

- **Phase 2: Operator Impact Detection** When the operator is applied, the software architect can select a number of transformations that are dependent on the metamodel. The impact of the change is detected for each transformation and classified according to the resolution that is possible.

- **Phase 3: Impact Resolution** For each detected impact a resolution is performed to restore consistency in this phase. The resolution for an impact is either executed automatically or the software architect is supported in performing a semi-automatic resolution.

After the phases are complete, consistency of the MDS is restored and the software architect can continue with other development activities or apply another operator.

In the next sections, the aspects relevant to each phase and the relevant design decisions are discussed. Based on these, each phase is broken down into the individual steps necessary to complete the phase. Furthermore, the contributions each phase makes to fulfil the requirements described above are discussed. Finally, in Section 5.8 the overall process is presented.

5.5. Phase 1: Metamodel Adaptation

The first phase of the co-evolution approach is concerned with how changes are performed. Our approach uses metamodels for the origin of change operations and treats transformations as dependent artefacts that need to be updated accordingly. This design decision is discussed in the next section. Section 5.5.2 covers the expected advantages for our use-case of an operatorbased approach – as opposed to state recording approaches. In Section 5.5.3 the details of how operators are used are given. Finally, Section 5.5.4 contains the individual steps that make up the first phase.

5.5.1. Metamodels as the Primary Artefacts in Co-Evolution

To provide a solution that supports the software engineer in handling coevolution, one must decide which of the dependent artefacts are primary and which are secondary, i.e. on which artefact types the change is performed and which artefact types are updated accordingly. For metamodel and transformation co-evolution, either the metamodel or the transformation could be chosen as the primary artefact type. Choosing the transformation as primary artefact entails that the software architect performs changes to the transformation and the dependent metamodels are updated accordingly to co-evolve. In this work, we choose the metamodel as primary artefact instead, for two main reasons:

- **Metamodels are central to the MDS.** The metamodel in which a change occurs may be involved in numerous transformations. It may also be used on different sides of the same or different transformations, for example when the output models (RHS) of one transformation be used as input for another (LHS). Would one start by changing one transformation and then adapting the metamodel afterwards, this change would need to be carried over to tertiary dependent transformations. At this point the metamodel becomes the origin of the change and the primary artefact to the co-evolution problem of that transformation. Therefore, metamodels play a central role in an MDS so that they present a suitable choice as primary artefact.

- **Co-evolving models from metamodels is possible.** The related work on the co-evolution of metamodels and models (see Section 8.1) shows that it is possible to co-evolve models as secondary artefacts to metamodels using operators. If the change operations provided for both metamodel and model and metamodel and transformation co-evolution are the same or similar, performing change may only be necessary once to perform the co-evolution of both transformations and models. This may reduce the overall effort of evolving the MDS.

For these reasons we choose metamodels as primary artefacts and to adapt the model transformations as secondary artefacts accordingly in our method. The reverse approach of changing transformations and adapting dependent metamodels instead may hold merit non-the-less. It is considered an interesting but out of scope research topic in this work.

5.5.2. Change Tracking for Co-Evolution

Handling the evolution of a software system requires the tracking of changes that occur and representing them adequately. In the case of MDE, with a focus of models as primary artefacts, a lot of change occurs in models and the artefacts that are dependent on models.

As the traditional, textbased methods of tracking and representing change do not convert well to graph-based models [72], new methods for the appropriate tracking of model changes are needed and have been proposed by different researchers. These approaches can be categorised in two general classes: *state* and change-based approaches, where operation-based approach are a sub-class of change-based approaches [54].

- **State recording approaches** store the state of the model after some amount of change occurred. The exact actions performed for the change may or may not be reconstructible, depending on the chosen time span between recorded states and the atomicity and unambiguity of the actions performed.

- **Change-based approaches** record change while it occurs so that no comparison of versions is needed as the versions can be constructed by applying, or (if possible) reversing the recorded change. Operationbased approaches are a subclass of these, as they provide pre-defined operators to express the change that can be applied to a model to derive a new model version: operation-based approaches can provide primitive and complex functions that take one model state as input and provide the changed state as output [63].

  Change-based and operator-based approaches have been shown to have advantages over state-based approaches in general [54] and especially for metamodel and model co-evolution [44, 107]. These advantages can be expected to also apply for metamodel and transformation co-evolution, for the following reasons:

- **Calculating model differences is hard.** As Kolovos et al. state, determining model differences is based on model matching, which can be reduced to the graph isomorphism problem, which is NP-hard [55]. Potentially, every model element of one version needs to be compared with every other model element of the other to determine the difference. On the other hand, recording the change while it occurs only requires the artefacts involved in the change.

- **Operators are unambiguous in effect.** Calculating model differences is subject to heuristics, so that the quality of the solution depends on the intended application [55]. For instance, changing a large portion of the attributes of a class can be indistinguishable from deleting the class and creating a new, independent one. The accuracy of the matching depends on the heuristic used and can not be ensured for all cases. When using operators instead, the impact of one unit of change is easier to predict and the exact parts of transformation descriptions that are impacted can be calculated automatically.

- **Operators are semantically rich.** When using operators to predetermine the possible impact, additional information besides the pure effect on the metamodel is available for co-evolution tasks. For instance, an operator for pushing a property down along the inheritance hierarchy of classes (see Section 6.5 for a description) has the same effect as deleting and adding the property and is redundant when only considering the effect on the metamodel. But the additional knowledge that the property was pushed allows for a better resolution of the impact on dependent

transformations. (We use this additional knowledge in our proposed resolution of a push property operator as described in Section 7.7.)

Furthermore, operators provide added benefit for other development tasks, like reviews, traceability and documentation: by recording evolution steps undertaken using operators, the nature, impact and source of changes to transformations are made explicit. In this manner, more knowledge about the evolutionary history of the system can be made available for instance to new members of the software team [54].

- **The disadvantage of specific tooling is lessened.** One disadvantage of recording change instead of calculating it is the specific tooling needed to perform the recording (or the application of operators) [40]. This means that the choice of modelling tools is limited and all participants of the development process need to agree on specific tooling. Yet, when considering both metamodel and transformation development, specialized tooling is already widely used, as models are commonly created using graphical editors and textual transformation editors benefit from the functionality of understanding models to provide code-completion and error checking capabilities. Therefore, the use of an integrated development infrastructure for MDE is common [94].

The change-based representation and especially the use of operators to describe and apply change when performing a metamodel evolution can therefore be expected to hold advantages over state-recording approaches and to represent a viable option to address the metamodel and transformation coevolution problem.

5.5.3. Using Operators

We make use operators to describe a change that can be performed on a metamodel. Operators are defined as follows:

Definition 8 (Operator): *An* operator *is a mapping from a set of metamodel elements to another set of metamodel elements. For the purpose of co-evolution, the operator describes how metamodel elements are adapted to perform a predefined type of change.*

Operators encapsulate the changes performed and provide an abstraction from the manipulation of individual metamodel elements. To fulfil Requirement 1, *Support common evolutionary changes* we propose a set of operators commonly used for metamodel evolution. These are detailed in chapter 6.

We chose to define the operators using QVT-R to come as close as possible to fulfilling Requirement 6, *state-of-the art modelling*. While the semantics of QVT-R are not fully formalised, it is well integrated with the UML and as long as overall formal semantics for the UML are still lacking (see Section 2.3.1), complete formalisation of operators working on UML-based metamodels is not possible. Yet the graphical, declarative syntax provided by QVT-R can be seen as advantageous for both the design principles *Consistency* and *Abstraction:* the application patterns of the operators can be modelled in the same level of abstraction as the metamodels they are defined for and the graphical syntax is consistent with graphical MOF models.

Furthermore, we chose to define the operators in chapter 6 for metamodels conforming to EMOF. EMOF also provides a strong formal background and can be considered to be well established in MDE as part of the UML. Furthermore, the EMF Ecore implementation of EMOF for the Eclipse framework provides a good foundation for the prototypical implementation and integration with other MDE tooling, to address Requirement 7 *(Integrate with common tooling)*.

5.5.4. The Metamodel Adaptation Phase

Figure 5.3 shows the details of the steps performed in the first phase for selecting and applying an operator to perform a metamodel adaptation. These are:

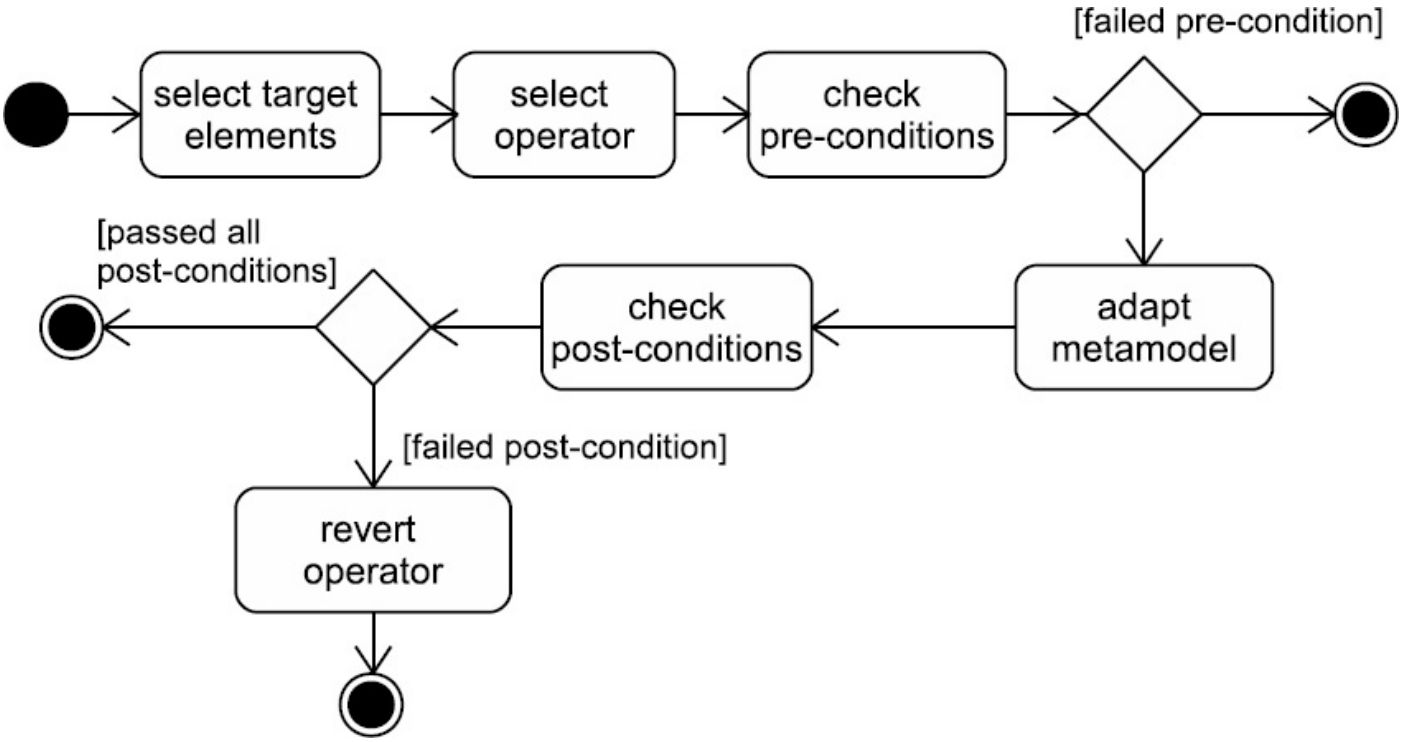


Figure 5.3.: The individual steps of the *metamodel adaptation phase*

1. **select target elements:** One or more elements of the metamodel are chosen by the user to perform a change on.

2. **select operator:** The user chooses the operator to apply to the elements chosen in the first step from a set of predefined operators.

3. **check pre-conditions:** In this step, the applicability of the operator is checked. One criterion is whether or not the chosen elements match the parameters required by the operator. Furthermore, operators may supply further pre-conditions that must be met to ensure that the operator can be applied. Should any pre-condition fail, the overall process is terminated. This phase contributes to Requirement 5 *(Ensure applicability of change)*.

4. **adapt metamodel:** If all preconditions are met, the operator is applied and the selected target elements are adapted.

5. **check post-conditions:** After the metamodel is changed, the postconditions of the

operator are checked. If all post-conditions pass, the first phase completes. This phase also contributes to Requirement 5.

6. **revert-operator:** Should any post-condition fail, the operator is reverted and the metamodel returned to its original stage. At this point the overall process is terminated.

5.6. Phase 2: Operator Impact Detection

The second phase of the process is concerned with the detection of the impact of the change that was performed in phase one by applying the operator. For this purpose, all transformations that are to be co-evolved are selected by the software architect. For these, the impact the operator has on the transformation rules is detected, to allow the resolution of the impact in the last phase. Which parts of a transformation rule are affected by an operator depends on whether or not the metamodel elements that are modified by the operator are relevant to the transformation rule part. Finding a use of these elements in a transformation rule equates detecting an impact. How metamodel elements are referred to in ATL is discussed in the next section. Section 5.6.3 summarizes the two steps that constitute the second phase.

5.6.1. Detecting Operator Impact on ATL Rules

A model transformation program written in ATL (see Section 3.2) is made up of a set of rules, where each rule produces one or more target model elements from a source model element. The application of a rule is determined by a source pattern, defined in terms of a source metamodel. The created elements are conformant to a target metamodel. ATL relies on an adapted version of the OCL to perform model navigation and to calculate values for target properties.

To analyse the impact of operators on transformation rules in ATL, two factors need to be taken into account:

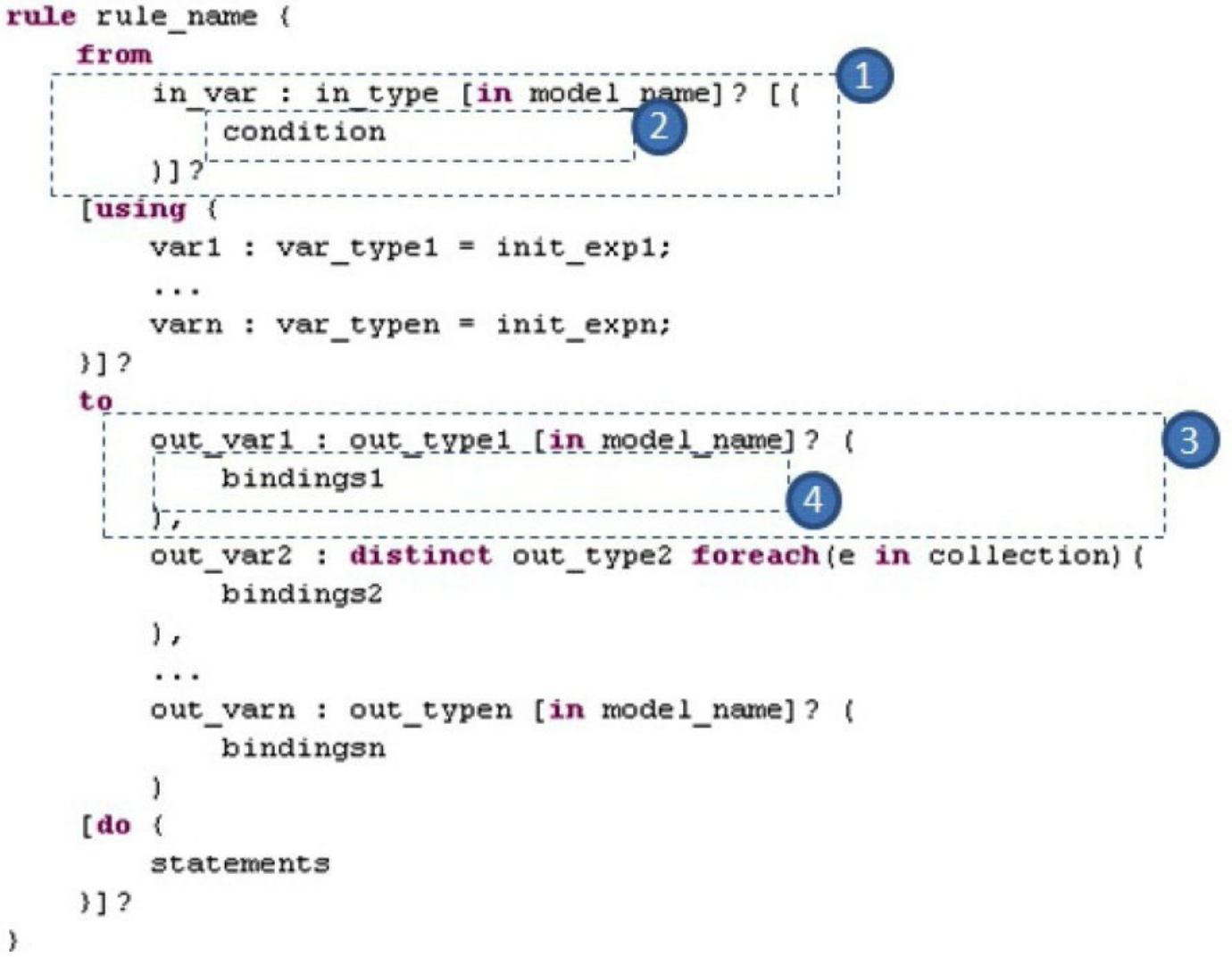


Figure 5.4.: The ATL syntax for a matched rule

1. **Transformation Direction** The impact depends on whether the operator is used on a metamodel that is a source (LHS) or a target (RHS) of a transformation. For example, adding an obligatory property on the LHS of the transformation has no impact on validity, while the same change on the RHS will make the transformation invalid, as it is impossible to deduce a correct value for the given property automatically. Therefore, the possible impacts for each metamodel change are different for LHS and RHS metamodels and require different resolutions.

2. **Locality** The impact of an operator further depends on the part of a transformation rule that the elements affected by the operator are used in. We identified the following relevant rule parts:

   - Changes can occur in the **domain definition** in-pattern of a rule (LHS metamodel),
   - the **filter condition** constraints of the in-pattern (LHS metamodel), and
   - the **binding assignment** in out-patterns for both LHS and RHS metamodels.

Figure 5.4 shows an informal excerpt of the syntax of an ATL Matched Rule, taken from the ATL Language Guide [90]. Metamodel changes due to an operator can impact the

following parts of an ATL rule on the LHS:

① The in-pattern ②, filter condition constraints imposed on the in-pattern element (expressed in OCL), and ④ the value calculation for properties in the binding assignment, also expressed in OCL. Changes to the RHS are potentially reflected in: ③ the target pattern (each rule can have numerous target patterns) and the ④ properties to which values are assigned in the value binding.

OCL plays an important role in ATL rules. It is used for two purposes: to express constraints in filter conditions on the applicability of rules and to calculate values in the property binding assignment. In both cases, OCL statements may navigate over models and can potentially involve any part of the LHS metamodel. This can lead to very complex source patterns for complex rules. To detect the impact of an operator on an ATL transformation requires the incorporation of OCL expressions. Also, adapting the OCL expressions that are part of a transformation may provide better impact resolution. The role of OCL is discussed in more detail in the next section.

5.6.2. The OCL and Metamodel Evolution

OCL plays a central role in transformation languages in the UML / MOF technological space, including ATL. This section covers the impact of metamodel evolution on OCL expressions in more detail. We provide our findings on which impacts can be detected for which OCL language features using static analysis and which cannot. We define how detected impacts can be resolved and provide a set of impacts that cannot be resolved automatically.

Both ATL and QVT use OCL for model navigation, the selection of model elements and the calculation of values. ATL uses its own variation of OCL, which does not conform to the standard and is not complete. The semantics of the OCL expressions of ATL are only loosely defined. [5, p. 15] QVT on the other hand uses the essential OCL according to the OMG standard for OCL 2.0 [73, p. 11].

The semantics of the OMG OCL version 2.0 are not fully formalised [18]. It has to therefore be noted that the correctness of a resolution strategy for an impact may not be proven completely and depends ultimately on the interpretation of the underlying transformation engine.

Refactoring OCL Expressions

The coupling between OCL expressions and UML models and the impact of refactoring UML models on linked OCL expressions has been investigated by other researchers. Cabot and Teniente for example propose techniques for the generation of semantically equivalent syntactic alternatives of OCL constraints [20] and Marković and Baar published a set of operators for the co-evolution of models and OCL expressions [65] (also see Section 8.2).

Yet, when comparing the co-evolution of OCL expressions and models with that of OCL expressions and transformations, three general differences can be noted, which make transferring the results difficult:

- **Evolution (and refactoring) has to cover whole transformations.** In other words, the transformation structure, the rule inheritance and the rule selection mechanisms have to be considered along with OCL expressions in the transformation to properly address evolution impact.

- **Transformation languages only use a subset of OCL**. As OCL expressions are used only for model navigation, element selection and calculation of primitive values in transformations, only a subset of the full OCL is used by transformation language implementations. Therefore, OCL language features may not be available or resolution strategies not necessary that are needed when handling models and expressions.

- **OCL used for constraints on models in itself is a different application domain.** OCL constraints defined on (UML) models relate to the models themselves. OCL as it is used for navigation in transformation languages relates instead mainly to the metamodels and sometimes to the primitive values of attributes of model instances. The application cases are different so that different resolution strategies are needed.

It can be noted in summary that while the work on OCL and model coevolution or refactoring provides good approaches for some cases, the problem is both broader and different so that taking only resolution strategies for OCL into account can be expected to fall short.

Impact on Embedded OCL Expressions

To assess the impact of metamodel changes on the OCL expressions used in transformations, the types of possible links between metamodels and OCL are important. In this section, we first look at how model element references are resolved by the OMG OCL standard definition. In preparation of the impact resolution for the ATL language, we compare this to the OCL implementation found in ATL and highlight the differences.

The OMG OCL standard defines which parts of a UML model can be referred to in an OCL expression in general:

> ‘OCL expressions can refer to Classifiers, e.g., types, classes, interfaces, associations (acting as types), and datatypes. Also all attributes, association-ends, methods, and operations without side effects that are defined on these types, etc. can be used.’ [74, p. 16]

OCL is a strongly typed language. All OCL expressions have a result type and all variables in all places of an expression are typed. As OCL expressions are always written in a context, available types are either the basic and collection types defined by OCL or stem from the classifiers of the associated UML model. [74, p. 11]

The OCL guide suggests the following approach to resolve bound properties[74, p. 50]: Properties are always associated with an object at any place in an expression. The *self* variable and iterator-variables in collections can be left out and are introduced implicitly. This makes the binding process dependent on the context in which a part of an expression is evaluated. If a property is used without an explicit variable, it is interpreted on an

implicit variable. If more than one variable is available in the current expression scope and the used property is defined for more than one object of the variables available, the variable of the most inner scope is chosen. This means that variable binding and property evaluation can be ambiguous and the resolution process has to be undertaken in the context of the entire expression.

The value of a property of an object in the context of a classifier is specified with the infix navigation operator ('.') in an OCL expression. For example:

```
1  context VideoCassette inv:
2  self.minutesLength > 60
```

Operations may be provided with parameters, given in brackets. Associationends are navigated using the dot notation, providing access to related objects and their property values:

```
1  context Book inv:
2  self.author.lastName.size() > 3
```

In the case of ATL, the context for OCL expressions in transformation rules is provided by the `PatternElements` which inherit from `ocl:VariableDeclaration` or by direct variable declarations that span the scope of the rule (see Figure 3.2). The `InPatternElements` provide variables for LHS model elements, while `OutPatternElements` are variables that contain elements of the RHS model. OCL expressions are featured in the form of the `Binding`, which refers to an OCL expression to calculate the value of a target model property. In this case the *self* context of the expression is the property the value is assigned to. The properties that are valid in this scope are all `PatternElements` and the declared rule variables. The other case of expression usage are filter expressions of `InPatterns` which are interpreted in the context of the `InPatternElement` that they are related to.

The navigation of models is expressed in the OCL variant of ATL by a `NavigationOrAttributeCallExp` which uses the infix navigation operator mentioned above to call properties or to navigate across associationends.

OCL Reflection

One problem in determining the impact of metamodel changes are the OCL language mechanisms for reflection. The `allInstances()` operator allows the collection of all available instances in an expression of a type which is not the context. This makes it possible to establish 'relationships' between classes in a rule and at runtime that were not intended in the original metamodel. As the OCL reflective operators are only evaluated at runtime, the static analysis for change impact is difficult and can be impossible, depending on the case. For this reason, the usage should be discouraged, as many explicit mechanisms of transformation languages can be circumvented in this manner.

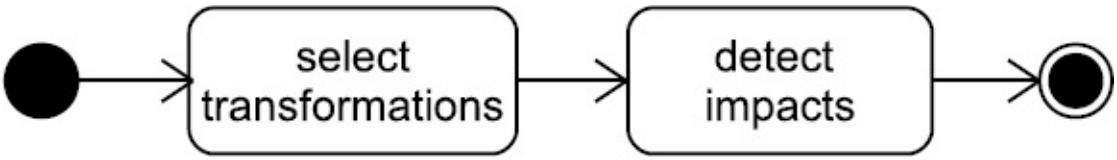


Figure 5.5.: The individual steps of the *impact detection phase*

5.6.3. The Impact Detection Phase

Phase two performs the detection of the impact of the change that was performed in phase one by applying the operator. For this purpose, all transformations are selected by the software architect that are to be coevolved. This is necessary, as metamodels can be used in any number of transformations and no explicit link between metamodels and transformation programs that use them exist in ATL; the metamodel to work with is only selected at runtime, prior to execution of the transformation. Then the impact detection for the selected transformation is performed. The detected impacts are then resolved in the subsequent third phase. Therefore, the two steps that make up the second phase, as shown in Figure 5.5, are:

1. **select transformations:** The transformations to be co-evolved with the metamodel are selected by the software architect.

2. **detect impacts:** For the selected transformations, the impact of the changes is detected.

The next section covers the third phase, in which the detected impacts are resolved.

5.7. Phase 3: Impact Resolution

One or more impacts of applying an operator on one or more ATL transformation rules was detected in the previous phase. In this third phase of our approach, these impacts are attempted to be resolved. The resolution may or may not be automated, depending on the operator and the usage of the elements modified by the operator in the dependent transformation rules.

The next section discusses possible approaches to classifying operators according to their impact.

5.7.1. Operator Classification

The changes performed on a metamodel in co-evolution can be classified according to the impact they have on dependent artefacts. This classification is useful from a practical standpoint, because it allows the prediction of the effort needed to resolve the impact of the change, before the change is performed. It may be ‘safe’ to apply an operator because it has no impact at all on dependent artefacts or the impact may be automatically resolvable or the user may need to perform a manual resolution of the impact.

Gruschko et al. identified three classes of metamodel change according to their effect on conformant models [40]:

- **Not Breaking** Changes in this class do not break the conformance of models to the corresponding metamodel, they have no impact on the validity of the dependent models.

- **Breaking and Resolvable** These changes break the conformance of models but the models can be adapted automatically to restore their validity.

- **Breaking and Unresolvable** Changes in this class break the conformance of models so that models can not be automatically coevolved. They require further user activity to re-establish model conformance.

Building on these classifications, Cicchetti et al. identify a set of operations on metamodels and categorise them according their change class, again with respect to the conformance of models [23], see Table 5.1. For instance, the operation *flatten hierarchy* refers to the change of eliminating a superclass and moving all its properties to the remaining subclasses. This is a breaking change, as old models no longer conform to the new metamodel, but it can be resolved, as instances of the missing class can be removed from the model and all properties and values copied to the corresponding subclass instances.

Change class	**Operation**
Non-breaking changes	Generalize metaproperty Add (non-obligatory)metaclass Add (non-obligatory) metaproperty
Breaking and resolvable changes	Extract (abstract) superclass Eliminate metaclass Eliminate metaproperty Push metaproperty Flatten hierarchy Rename metaelement Move metaproperty Extract/inline metaclass
Breaking and unresolvable changes	Add obligatory metaclass Add obligatory metaproperty Pull metaproperty Restrict metaproperty Extract (non-abstract) superclass

Table 5.1.: Metamodel change classification according to the impact on conforming models according to Cicchetti et al. [23]

But Cicchetti et al. argue further that metamodel changes commonly occur neither individually nor independently as modifications of metamodels tend to occur with arbitrary multiplicity and complexity [22]. This makes a clear distinction of change types difficult and less useful in practice.

Unfortunately, the classification according to the breaking nature of change can not be carried over easily from metamodel and model co-evolution to metamodel and transformation co-evolution. While the impact of an additive change of the metamodel is easily determined for conformant models, as conformance refers to the model as a whole, it is different for model transformations, as application patterns in transformation rules usually only refer to some part of their metamodel. For example, if a new property is added to a class, it can be set to be either obligatory or not. For dependent models, if it is not mandatory, the models remain syntactically conformant and the user has to decide if the attribute is semantically required for some instances. If the attribute is mandatory, the change is always *breaking* and most likely *unresolvable* (bar default values). For transformations, even an added mandatory property does not imply a breaking change, as the transformation will remain syntactically valid and it depends on the intended function of the transformation if the given change requires adaptation.

Furthermore, whether or not a transformation can be adapted automatically in response to the use of an operator depends on how the elements that are modified by the operator are used in the transformation, as discussed in Section 5.6.1. The same operator may lead to a resolvable impact in one transformation rule and a non-resolvable one in another. In conclusion, operators can not be classified in isolation when looking at the impact on transformations. We propose the classification of the different impact resolutions instead. This approach is discussed in Section 5.7.4.

5.7.2. Impact Resolution: Syntax and Semantic Preservation

Marković and Baar distinguish between syntax and semantics preserving refactoring rules for the co-evolution of UML 1.5 class diagrams and dependent OCL constraints [65] and illustrate the semantics preserving property of the *MoveAttribute* rule [7]. A rule is *syntax preserving*,if

> ‘for every syntactically correct source model the obtained target model is syntactically correct as well. Syntactically correct models are exactly the valid instances of the metamodel, what boils down to the following three criteria: (1) all model elements are well-typed, (2) all multiplicity constraints are met, and (3) all well-formedness rules are obeyed.’ [65]

Whether or not a rule is syntax preserving has to be shown for every valid metamodel that serves as input.

The semantic preservation property of a refactoring rule ensures that the semantics of the model coincide before and after the application of the rule. Opdyke defines this as *behavioural preservation* so that

> '…if the function *main* [the program] is called twice (once before and once after a refactoring) with the same set of inputs, the resulting set of output values will be the same' [80, p. 40].

The lack of formal semantics of the common modelling languages poses a hindrance to defining criteria for semantic preservation which explains the lack of proof techniques for semantic preservation [7]. The same can be expected for the co-evolution of metamodels and transformations, for two main reasons:

- Both ATL and QVT are based on MOF and make use of OCL internally, so that the lack of formal semantics is as critical. Furthermore, the semantics of QVT and ATL are also not fully formalised (see sections 3.3.1 and 3.2.2) so that even if the semantics for the metamodel evolution can be proved, the resolution strategy may still not be provable for the transformation languages involved.

- The application of an evolutionary operator is not meant to result in a semantically equivalent model but rather in a semantically correct change of the model in accordance with the semantics of the operator. Therefore, the semantics of the operator need to be taken into account.

In conclusion this means that the syntactical preservation of evolutionary operators for ATL can be shown by showing that all three criteria for model conformance are given for all input metamodels. Semantic correctness instead is very hard to prove, without fully formalising all involved languages. Whether or not an operator performs as intended is left to the user, who needs to check the result and adapt it as needed.

5.7.3. Impact and Resolution Definition

Chapter 7 introduces a number of impact resolutions for the operators defined in chapter 6. For each operator, we discuss the possible impacts on ATL transformation rules and provide resolutions to restore the consistency of the transformation involved. To remain in line with the operator definitions, we also use QVT-R to define the impact resolutions available for each operator. The impact and impact resolution for ATL are defined on the basis of the ATL AST model. If an automatic resolution of the operator is possible, the AST is adapted according to the resolution. In a tool implementation, the software architect should be provided with an overview of the suggested impact resolutions and accept or reject these individually. The nature of the different impacts is classified in the next section.

5.7.4. Classifying Impact

Instead of classifying the overall impact of operators according to their breaking or resolvable nature – as possible for metamodel and model coevolution but not for metamodel and transformation co-evolution – we need to take the many different kinds of impact an operator can have depending on how the changed metamodel elements are used in subject transformations into account. In other words, the same operator can be automatically resolvable or not, depending on how a dependent transformation program is build. Therefore, we define the different impacts possible for each operator on ATL

transformations in chapter 7 and classify them according to the resolution possible. The classes are:

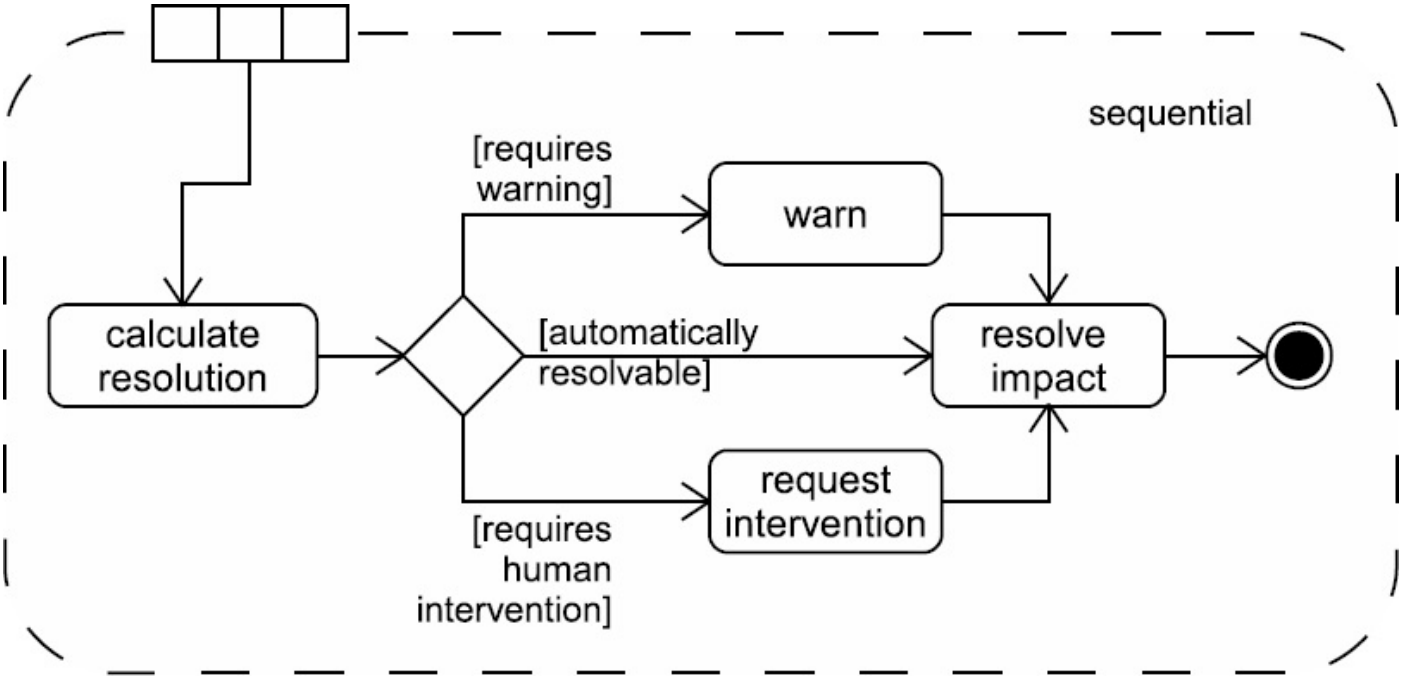


Figure 5.6.: The Individual Steps of the *Impact Resolution Phase*

- **none:** The model elements that are modified by the operator are used in a transformation rule, but no change to the rule is needed as it continues to function as before.
- **automatic:** A resolution strategy for the impact is available that can be executed automatically.
- **warn:** An automatic resolution strategy for the impact is available, but it may not have the effect intended by the software architect. After the resolution, the model transformation is syntactically correct, however, the software architect may need to perform further manual adaptation.
- **human intervention:** No automatic resolution strategy is available and the software architect needs to perform a manual adaptation of the model transformation rule to resolve the impact.

Table 7.1 on page 136 provides an overview of the impact definitions and the resolution strategies possible for the operators defined in chapter 6.

5.7.5. The Impact Resolution Process Phase

The third phase of the approach is structured as shown in Figure 5.6. It is performed for each detected impact. The individual steps are:

1. **calculate resolution:** For each impact, a resolution is calculated.
2. **warn:** If the resolution may have unintended consequences, warn the user.

3. **request intervention:** If the resolution can not be performed automatically, the user needs to provide further information or resolve the impact manually.

4. **resolve impact:** If an automatic resolution is possible or the user has provided enough information for the resolution, the resolution is performed and the transformation is updated accordingly.

5.8. The Overall Co-Evolution Process

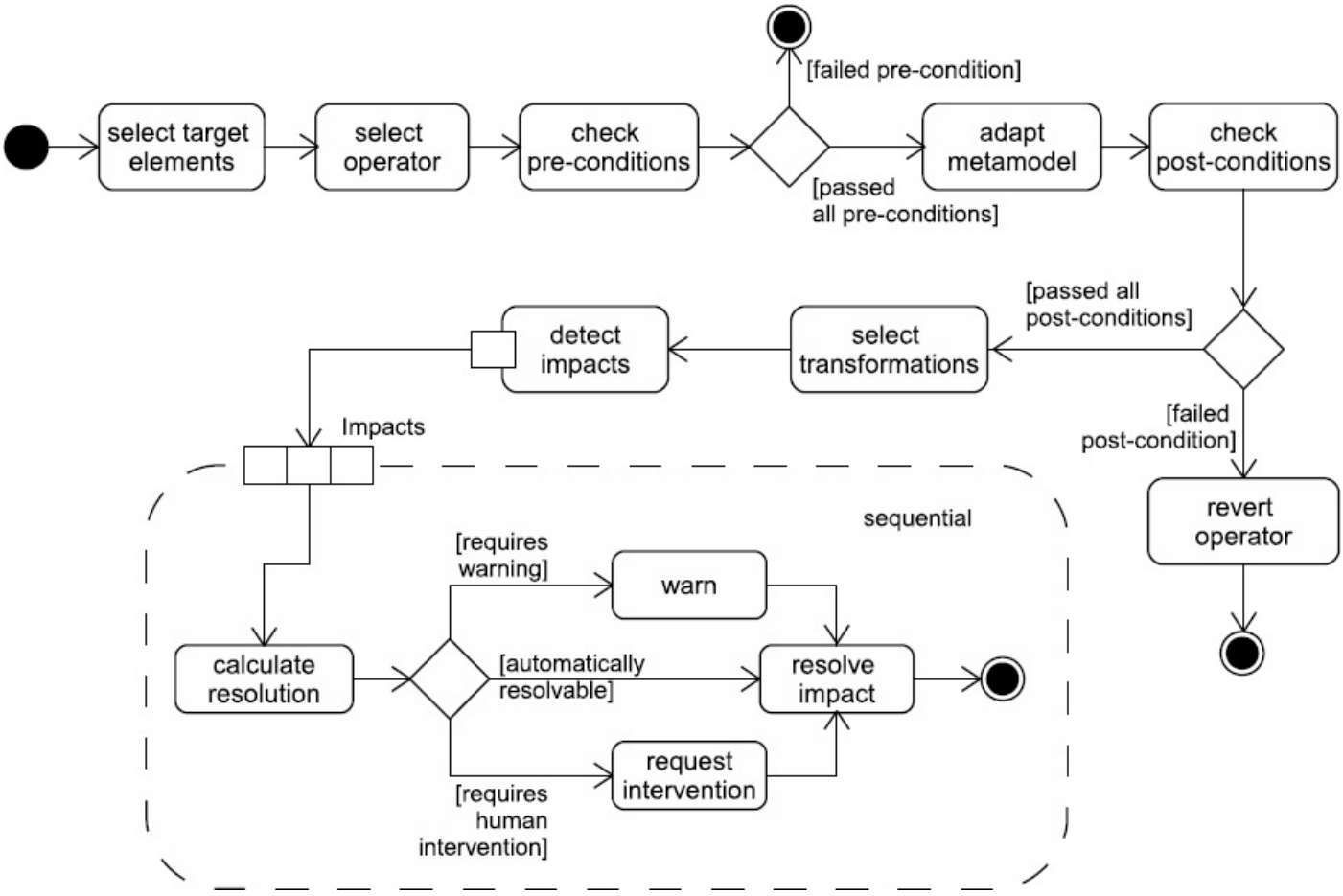


Figure 5.7.: Approach

The overall steps of all three phases of our approach are shown in Figure 5.7. The approach is stepwise and operator-based and supports software architects in the co-evolution of metamodels and ATL model transformations. The approach uses operators to define units of change that are commonly made to metamodels. The software architect selects an operator and chooses one or more metamodel elements to apply the operator to, to perform one evolution step. The operators may contain one or more pre-conditions that guide the application to ensure that the selected metamodel elements are in a state for which the operator is intended and ensure that the result is valid. The pre-conditions are checked and, should the validation pass, the input metamodel is manipulated automatically according to the operator and an evolved metamodel is produced as result.

To assist in the evolution of model transformations, the operator and the changes made to the metamodel are used to calculate the impact on an arbitrary number of dependent model transformations. The model transformations to be checked are selected by the software architect. The impact detection stage performs a static analysis of the model transformations to detect ATL rules or rule parts that may be impacted by the change made

to the metamodel. For each impact, a resolution is calculated. The resolutions are categorised according to how they can be performed, either automatically, with a warning or with human intervention. The resolutions are performed and the transformations are updated accordingly.

The next chapter introduces a set of operators for common evolution tasks to support our approach. In chapter 7, the impact and resolution of each operator for different parts of an ATL transformation are discussed. Chapter 9 provides an evaluation scenario and chapter 10 covers the prototypical implementation of tool support possible for our approach.

CHAPTER 6

Operators for EMOF Metamodel Evolution

This chapter introduces a set of operators for the evolution of EMF-based metamodels. The operators stem from different sources and are formalised using the graphical notation for QVT-R. Most operators stem from related work on metamodel and model co-evolution (see Section 8.1) or coadaptation of models and OCL constraints (Section 8.2) or general model or OO refactoring work (Section 4.2). Some of the operators are available in a QVT-R-like syntax for MOF or UML 1.5 and 2.0 in the sources. As both UML MOF are closely related to EMOF, we build upon the existing formalisations to adapt them to EMOF, or (in some cases) to make them better suited for transformation co-evolution.

The operators are grouped in topical section. For each operator, we provide a formalisation using the graphical notation for QVT-R. In some cases, operators have an opposite with an opposite effect (such as addition and removal of an element) and can be formalised by the same relation; by interpreting the relation in the opposite direction. This is noted in the description where appropriate. We further list the OCL pre- and postconditions relevant for each execution direction. These ensure that the operator can be applied as intended and that the metamodel resulting from executing the operator remains valid (we assume that the input metamodel is valid).

6.1. On the Usage of the QVT-R Graphical Notation for Operator Definition

The operators presented in this chapter are expressed using the QVT-R graphical notation as described in Section 3.3.4. One assumption concerning bi-directionality was made, as the QVT standard is open to interpretation in this matter. We also defined some black-box operations to make the relations less verbose. These are described next.

We base the metamodels on EMOF and also make occasional use of EMOF-functions, like membersAreDistinguishable() or isEmpty(). These are defined by the EMOF standard as described in Section 2.3.2.

6.1.1. Black-Box Operations create() and createX()

We abstract from the creation of elements from simple values by means of a black-box operation in the QVT relations to make the relations more concise and hide verbose value parsing. Otherwise, the relations which create new elements would need simple domains for each parameter of an element. As the creation relations for elements are used in other relations, these parameters would have to be passed on, forcing these relations to again have simple domains for all parameters of all elements they create, distracting from the purpose of the operator and making relations unnecessarily verbose. To circumvent this, we propose the introduction of a create() black-box operation, which handles the provisioning of values for the simple attributes of new elements (like the name).

It is assumed that the create() operation returns a complete and configured element. In an implementation of the operation, this could be achieved by providing a dialog to the user when a new element is created for all required simple values. After the dialog completes,

the new element exists but is not yet owned by an element of the model (it is not ‘inserted’ into the model). Usage of the create() operation on its own would leave the resulting model in an invalid state, as the created element has no owner and other constraints may also be violated. Yet this is permissible, as it is always used in conjunction with a corresponding *addElement* relation, in which the new element is added to an owning model element and all pre-and postconditions needed for a valid addition are checked and, if met, the model is in a valid state.

To further improve the readability of the relations, we assume that a `create()` operation is available for each type in the form `createX()` where X is the type name. The create operation for classes is called `createClass()` for instance.

6.1.2. Black-Box Operations: clone() and cloneDiscName()

Similar to the `create()` operation described above, a `clone()` operation is assumed to provide the functionality to create a copy of an element without adding it to the model. The cloned element can then be added to an owning element by the relation that calls the `clone()` operation. In checking mode, the `clone()` operation can be seen to verify that an element is the clone of another in all features but the owner, evaluating to either true or false. The `cloneDiscName()` operation (‘clone discounting name’) clones the element in all regards but the owner and the name, i.e. elements with a different owner and a different name match in a query and a clone is created with a name-prompt for the user to supply a new name during execution.

6.1.3. Bidirectional Execution and Model Candidates

As QVT relations are potentially bi- or multi-directional (during execution, one candidate model is chosen to be modified in dependence of the other model(s)), the execution direction needs to be defined for the correct semantics of the relation in question. In general, the relations used here are defined to be executed ‘from left to right’, i.e. from model ‘l’ to model ‘r’. The reverse is only meant to be useful if explicitly stated. This is the case for all Add/Remove relations, where the direction ‘l’ to ’r’ adds elements and ‘r’ to ‘l’ has the opposite effect and removes elements. When relations call sub-relations, the sub-relations are executed according to the membership of the parameters passed: if the variable belongs to the ‘l’-model in the parent relation, it is expected to also belong to the ‘l’-model in the subrelation. The execution direction is then inherited from the parent; ‘l’-to-‘r’ or ‘r’-to-‘l’ for all relations.

If the same model is chosen for both candidate models, the relations are *in-place* transformations, with the expected effect of providing an evolution step on a single (meta-)model. Another possible application is the comparison of two versions of the same model to find the evolution that may have occurred. Here the two model versions are chosen as the left and right candidates and the relations are executed in checking mode, to verify that the two model versions are related by the given relation.

6.2. Operator Overview

The set of 30 operators and their sources are summarised in Table 6.1. The sources of the operators are the work by Wachsmuth [107], who proposed a set of 16 operator-like adaptations for the co-evolution of metamodels and models, Marković and Barr [65], with 15 rules for the co-evolution of class diagrams and OCL constraints and Hassam et al. [42], with 7 operations for metamodels and OCL constraints. We cover the operators and their formalisation here, how the approaches presented in the related work compare to our approach is discussed in chapter 8.

Some of our operators are not covered by any of the sources (see Table 6.1). In the case of *Add / Remove Element* operators, they are added to provide operators for the addition and removal of all elements of the EMOF metamodel, as detailed in Section 6.3. In the category *Property Manipulation, we* added push and pull operators for complex properties which were not found elsewhere. Finally, we added the operator *Introduce Composite Pattern* as an example of a complex refactoring pattern from Fowler [34].

Type	Operator Name	W.	M&B	H.	Section	Page
Add / Remove Element	Add Class	yes	no	no	6.3.2	102
	Remove Class	yes	no	no	6.3.3	103
	Add Package	no	no	no	6.3.4	103
	Remove Package	no	no	no	6.3.5	104
	Add DataType, PrimitiveType, Enumeration	no	no	no	6.3.6	105
	Remove DataType, PrimitiveType, Enumeration	no	no	no	6.3.7	105
	Add EnumerationLiteral	no	no	no	6.3.8	106
	Remove EnumerationLiteral	no	no	no	6.3.10	107
	Remove Property	yes	no	no	6.3.9	107
	Add Property	yes	no	no	6.3.11	108
	Add Association	no	no	no	6.3.12	109
	Remove Association	no	no	no	6.3.13	109
	Add Operation (Class / DataType)	no	no	no	6.3.14	110
	Remove Operation (Class / DataType)	no	no	no	6.3.15	110
	Introduce Generalization	no	no	no	6.3.16	112
	Remove Generalization	no	no	no	6.3.17	112
Property Manipulation	Move Property	yes	yes	no	6.4	113
	Push Simple Property	yes	*	yes	6.5	114
	Push Complex Property	no	no	no	6.6	116
	Pull Simple Property	yes	yes	yes	6.7	119
	Pull Complex Property	no	no	no	6.8	120
	Restrict (Unidirectional) Property	yes	no	no	6.9	122
	Generalise (Unidirectional) Property	yes	no	yes	6.10	123
Hierarchy Manipulation	Extract Class	yes	yes	no	6.11	123
	Inline Class	yes	no	no	6.12	125
	Extract Superclass	yes	yes	no	6.13	126
	Flatten Hierarchy	yes	no	no	6.14	127
Refactoring Patterns	Association to Class	yes	no	yes	6.15	129
	Generalization to Composition	no	no	yes	6.16	131
	Introduce Composite Pattern	no	no	no	6.17	133

Table 6.1.: Overview of the operators for EMOF metamodel evolution

The set of operators defined in the next Section 6.3 covers the addition or removal of elements to or from models for all EMOF types. Please note that the element creation is not covered by the given relations, as the primitive values of attributes are not included as primitive domains. The relations only summarize the pre- and post-conditions that need to hold for the addition or removal of elements of each type and outline the situation before addition or after removal. Relations for the creation and addition of elements (‘introduction’) would be very verbose and not really helpful, as they only map primitive domains (‘relation parameters’) to the primitive attributes of the types in question. Instead,

we make use of the `create()` operations described above where necessary and cover the addition and removal of elements by the operators in the next section.

6.3. Add / Remove Element

An element is added to or removed from the model. This can be any instance of subclasses of `Element`, such as comments, types, classes but also attributes and associations in EMOF. Besides behaving in a way expected by the user and making no further changes to an existing model, the operator must adhere to the structural requirements and all constraints placed on the added elements by EMOF. For example, EMOF defines a constraint for `Element` that no element may own itself - neither directly nor transitively: `not self.allOwnedElements()->includes(self)`[76, p. 75]. This constraint

is relevant when adding a new `Package` and is covered by the more general constraint that a package added by the *Add Package* operator is completely empty.

EMOF defines the following sub-classes of `Element` (see also Section 2.3.2) that are non-abstract and available to be used in models. We define general constraints for adding or removing elements in Section 6.3.1 and the specifics of each element as follows:

- `Class` and `Package` are handled by their respective *Add* and *Remove* operators (see pp. 102ff.)
- `DataType`, `PrimitiveType` and `Enumeration` are handled by the *Add DataType* and *Remove DataType* operators (see sections 6.3.6 and 6.3.7).
- `EnumerationLiteral` is handled by the *Add Enumeration Literal* (Section 6.3.8) and *Remove Enumeration Literal* (Section 6.3.9) operators.
- `Property`,and `Association` are handled by their respective *Add* and *Remove* operators (see pp. 107ff.)
- `Operation` an their `Parameters` are added to and removed from classes or data types using the *Add Operation* / *Remove Operation* operators (sections 6.3.14 and 6.3.15).
- `Generalisations` are introduced between two classes or removed from them. This is done using the *Introduce Generalisation* / *Remove Generalisation* operators (sections 6.3.16 and 6.3.17).
- The subclasses of the abstract class `ValueSpecification` define constant values for simple types (like Integer or String) or a chosen `EnumerationLiteral` value. These belong to their respective owners (like properties with constant or default values) and are added and removed with them, eliminating the need for separate operators. These are `InstanceValue` for enumeration literals and `LiteralBoolean`, `LiteralInteger`, `LiteralNull`, `LiteralReal`, `LiteralString` and `LiteralUnlimitedNatural` for simple types.
- `Comments` are used in EMOF for human-readable annotations only and have no enforced semantics. Therefore no operator is provided for the addition and removal

of comments. To maintain the structural validity of models however, a general constraint is given that no element can be removed while it still owns a comment (see the next section).

These operators cover the creation and removal of all elements that are available for use in metamodels in EMOF.

6.3.1. General Constraints

Two conditions need to be checked for the correct removal of any element. These are defined by the following constraints.

All elements in EMOF can be annotated by a human readable comment. This comment is stored in the `Comment` element, which can be owned by any sub-class of `Element` and point to the annotated sub-class of `Element` via `Comment.annotatedElement`. For the removal of elements, we define constraints to prevent any comment pointing to a removed element or a removed element owning a comment:

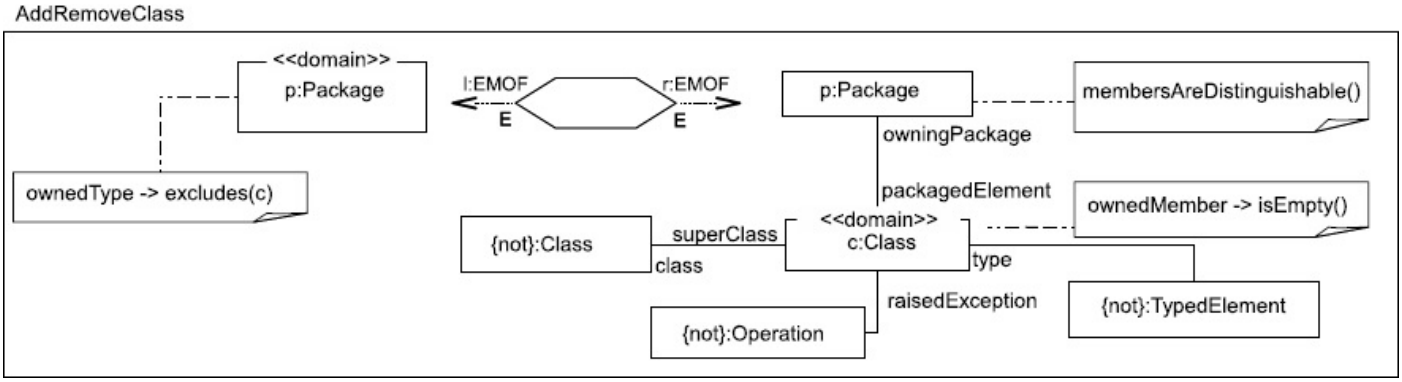


Figure 6.1.: The *Add / Remove Class* relation

- Preconditions for the *Remove*- operators of all elements:
  - `e.ownedComment -> isEmpty()`: the `Element e` to be removed has no attached comments. e removed has no attached comments e removed has no attached comments
  - **`context`** `Comment` **`pre:`** `self.annotatedElement -> excludes(e)`: no `Comment c` annotates the `Element e` to be removed.

The sub-classes of `Classifier` can redefine other classifiers. When a classifier is removed from a model, any redefining classifier could become invalid. This is prevented by the constraint:

- Preconditions for the *Remove*- operators of any classifier:
  - **`context`** `Classifier` **`pre:`** `self.redefinedClassifier -> excludes(c)`: no classifier `c2` is a re-definition of the classifier `c` to be removed.

6.3.2. Add Class

An empty class is added to an existing package. The relation *Add Class* is given in Figure 6.1 in the direction left-to-right.

- Preconditions:
  - `p. ownedType -> excludes(c)`: no Class `c` exists in Package `p`.
- Postconditions:
  - `p.membersAreDistinguishable()`: all members of the given Package p (with the added Class c) are unique in their namespace.

6.3.3. Remove Class

A class is removed from a package. The class is expected to be empty, not to be part of a generalisation (inheritance) relationship or otherwise be used in the model. A class can be put in this state by using other operators first, for example by removing all properties from the class with the *Remove Property* operator. The relation *Remove Class* is given in Figure 6.1 in the direction right-to-left.

- Preconditions:
  - `c.ownedMember -> isEmpty()`: the class to be removed has no other attached elements like attributes or operations.
  - `{not}:Class`: the class to be removed is not superclass of any other class.
  - `{not}:TypedElement`: the class to be removed is not the type of any remaining typed element, which are parameters, properties and instance values. Furthermore, no operation can be typed by the class to be removed, as the type of an operation is derived directly by the type of the return parameter, which is covered by this constraint.
  - `{not}: Operation`: the class to be removed is not an exception raised by any operation.
- Postconditions:
  - `p.ownedType -> excludes(c)`: the removed class `c` no longer exists in package p.

6.3.4. Add Package

An empty package is added, possibly to an owning parent package. The value of the parent package variable parent may have a value of `null` to denote the addition as a root package. The relation *Add Package* is given in Figure 6.2 in the direction left-to-right.

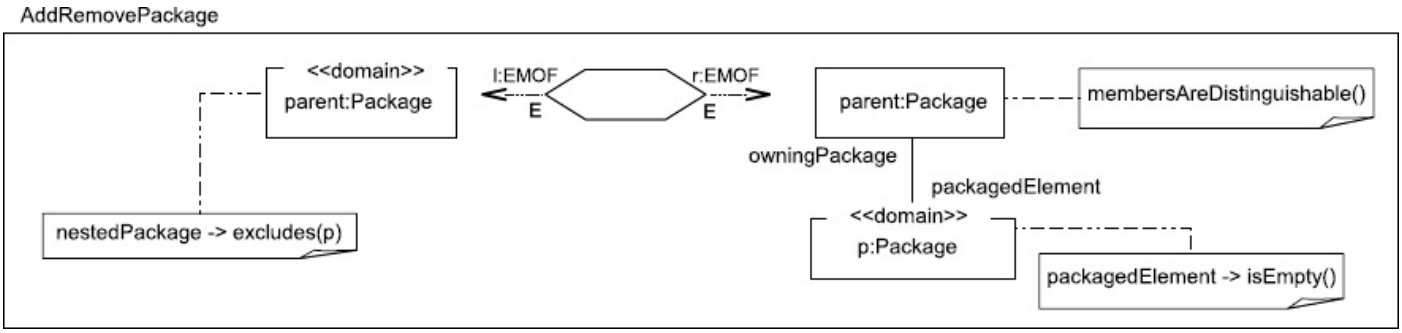


Figure 6.2.: The *Add / Remove Package* relation

- Preconditions:
  - `parent.nestedPackage -> excludes(p)`: the parent package does not own a package p.
- Postconditions:
  - `parent.membersAreDistinguishable()`: all members of the package parent are unique in their namespace after the addition of package p.

6.3.5. Remove Package

An empty package is removed, possibly from an owning parent package. In cases where packages with content are to be deleted, the deletion of the content elements should be undertaken first and then the empty package can be deleted. The relation *Remove Package* is given in Figure 6.2 in the direction right-to-left.

- Preconditions:
  - `p.packagedElement -> isEmpty()`: the package p to be removed has no content.
- Postconditions:
  - `parent.nestedPackage -> excludes(p)`: the removed package p no longer exists in the owning parent package.

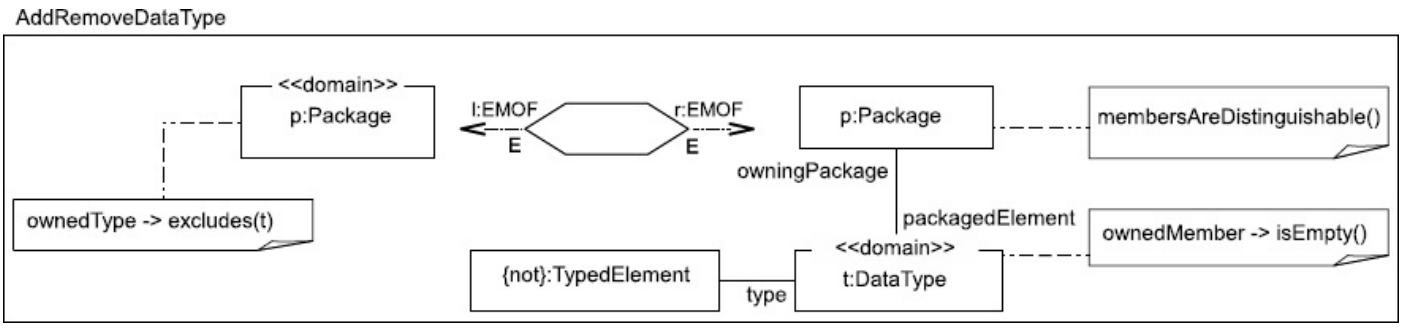


Figure 6.3.: The *Add / Remove Data Type* relation

6.3.6. Add DataType, PrimitiveType, Enumeration

A data type, primitive type or enumeration is added to an owning package `p`. Primitive types in EMOF are represented by the `PrimitiveType` class, which is a sub-class of

`DataType` and which introduces no additional features, and can therefore be handled by the same relation as `DataType`. The `Enumeration` class extends `DataType` by the `ownedLiteral` association, which holds an ordered set of literals. It sub-sets the `ownedMember` association and is covered by constraints on `ownedMember`. The relation *Add Data-Type* is given in Figure 6.3 in the direction left-to-right.

- Preconditions:
  - `p.ownedType -> excludes(t)`: no data type `t` exists in the parent package `p`.
- Postconditions:
  - `p.membersAreDistinguishable()`: all members of the given parent package `p` (with the added data type) are unique in their namespace.

6.3.7. Remove DataType, PrimitiveType, Enumeration

A data type, primitive type or enumeration is removed from an owning package `p`. The relation *Remove DataType* is given in Figure 6.3 in the direction right-to-left.

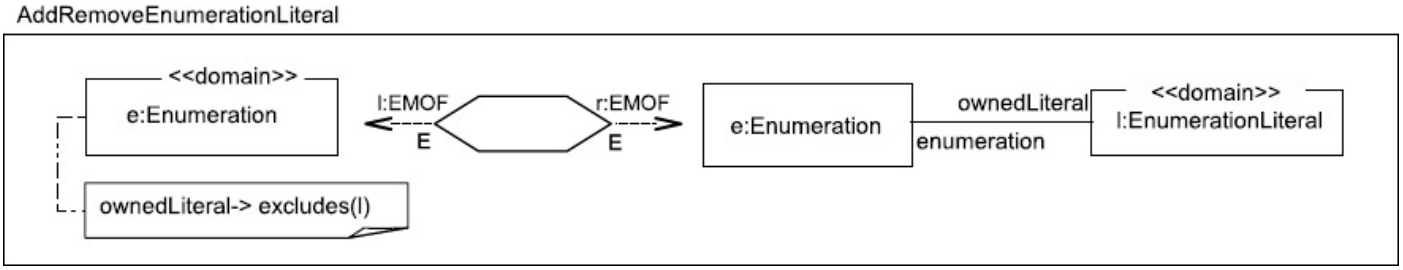


Figure 6.4.: The *Add / Remove EnumerationLiteral* relation

- Preconditions:
  - `t.ownedMember -> isEmpty()`: the data type to be removed has no attached elements like attributes, operations or enumeration literals.
  - `{not}:TypedElement`: the data type to be removed is not the type of any typed element, like parameter or property. Furthermore, this constraint prevents that any remaining operation is typed by the data type to be removed, as the type attribute of an operation is derived from the type of the return parameter of the operation.
- Postconditions:
  - `p.ownedType -> excludes(t)`: the removed data type `t` no longer exists in package `p`.

6.3.8. Add EnumerationLiteral

An enumeration literal is added to an owning enumeration `e`. The relation *Add EnumerationLiteral* is given in Figure 6.4 in the direction left-toright.

- Preconditions:

- `e.ownedLiteral -> excludes(l)`: no enumeration literal `l` exists in the owning enumeration `e`.

- Postconditions: *none*

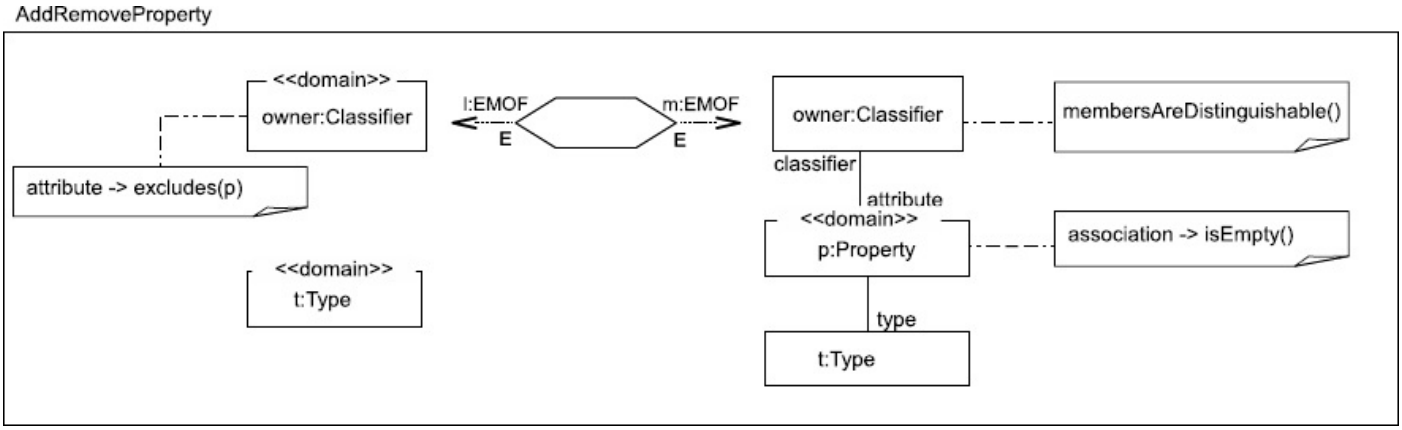


Figure 6.5.: The *Add / Remove Property* relation

6.3.9. Remove EnumerationLiteral

An enumeration literal is removed from an owning enumeration `e`.The relation *Remove EnumerationLiteral* is given in Figure 6.4 in the direction right-to-left.

- Preconditions: *none*

- Postconditions:

  - `e.ownedLiteral -> excludes(l)`: the enumeration literal `l` no longer exists in the owning enumeration `e`.

6.3.10. Add Property

A property of an existing type is added to an existing classifier. Concrete classifiers can be classes, data types or associations. In the case of an association, the property becomes the `ownedEnd` of the association. The relation *Add Property* is given in Figure 6.5 in the direction left-toright.

- Preconditions:

  - `c.attribute -> excludes(p)`: no property `p` exists as an attribute of classifier `c`.

- Postconditions:

  - `p.membersAreDistinguishable()`: all members of the given classifier `c` (including the added property p) are unique in their namespace.

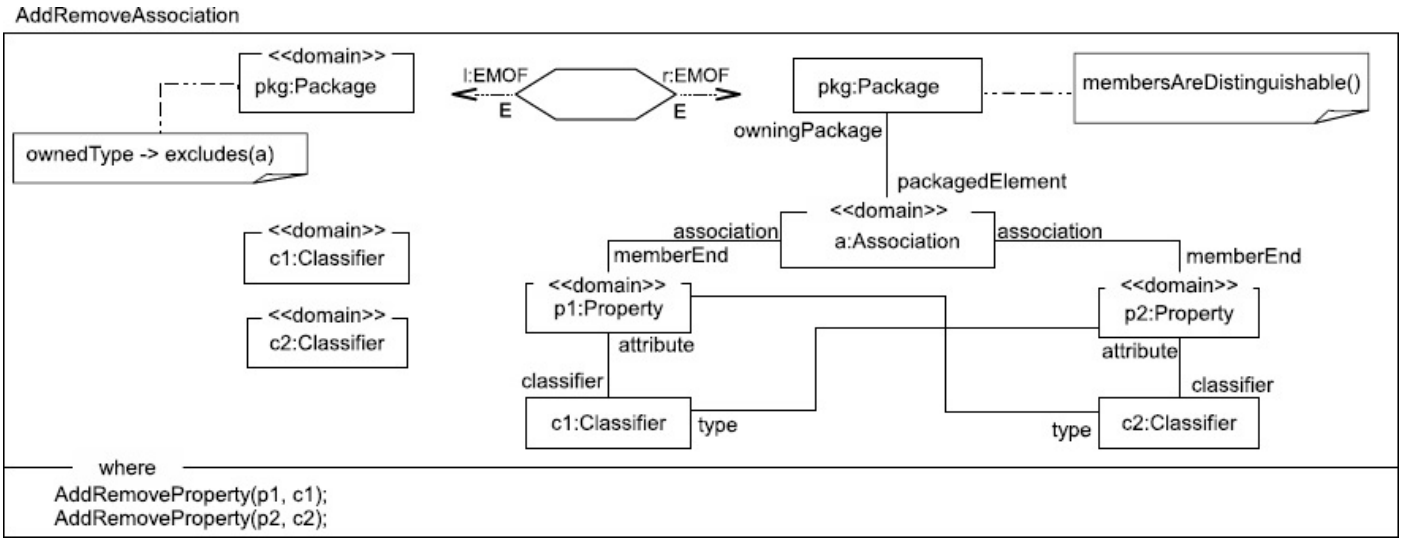


Figure 6.6.: The *Add / Remove Association* relation

6.3.11. Remove Property

A (simple) property of some type is removed from a classifier. The relation *Remove Property* is given in Figure 6.5 in the direction right-toleft.

- Preconditions:
  - `p.association -> isEmpty()`: no association are attached to the property p. Associations can be removed using the *Remove Association* operator in Section 6.3.13. This constraint also ensures that no opposite is set for the property, as this is derived from the association attached.
- Postconditions:
  - `c.attribute -> excludes(p)`: the property p has been removed from classifier `c`.

6.3.12. Add Association

An association between two existing classifiers is added to a package with the specified member-end properties. The member-end properties are added to the classifiers by diverting to the *Add Property* relation in the where-clause. The relation *Add Association* is given in Figure 6.6 in the direction left-toright. If one of the classifiers is the association itself, the corresponding property is owned by the association (ownedEnd).

- Preconditions:
  - `pkg.ownedType -> excludes(a)`: no association a exists in package pkg.
- Postconditions:
  - `pkg.membersAreDistinguishable()`: all members of the given package (including the newly added association a) are unique in their namespace.

6.3.13. Remove Association

An association between two classifiers is removed from an existing package. The member-end properties of the association are also removed. The relation *Remove Association* is given in Figure 6.6 in the direction right-to-left. It calls the *Remove Property* relation in the where-clause to remove the member-ends from the owning classifiers. If one of the properties is owned by the association itself, the association must be given as one of the owning classifiers.

- Preconditions: *none*
- Postconditions:
  - `pkg.ownedType -> excludes(a)`: the association a has been removed from package pkg.

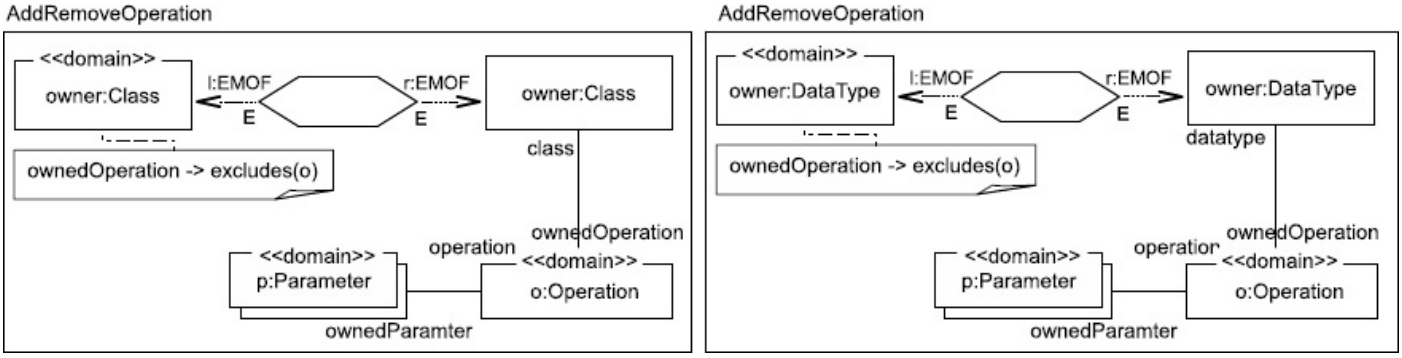


Figure 6.7.: The *Add / Remove Operation* relation

6.3.14. Add Operation (Class / DataType)

An operation with a set of parameters is added to a class or data type. A distinct relation is needed for each case, as the ownership associations of `Class` and `DataType` are not related or available in a common superclass. Apart from that the relations are identical and given in Figure 6.7 in the direction left-to-right.

- Preconditions:
  - `owner.ownedOperation -> excludes(o)`: no operation `o` exists for the given owner.
- Postconditions: *none*

6.3.15. Remove Operation (Class / DataType)

An operation with a set of parameters is removed from a class or data type. A distinct relation is needed for each case, as the ownership associations of `Class` and `DataType` are not related or available in a common superclass. Apart from that, the relations are identical and given in Figure 6.7 in the direction right-to-left.

- Preconditions: *none*

- Postconditions:
  - `owner.ownedOperation -> excludes(o)`: the Operation `o` has been removed.

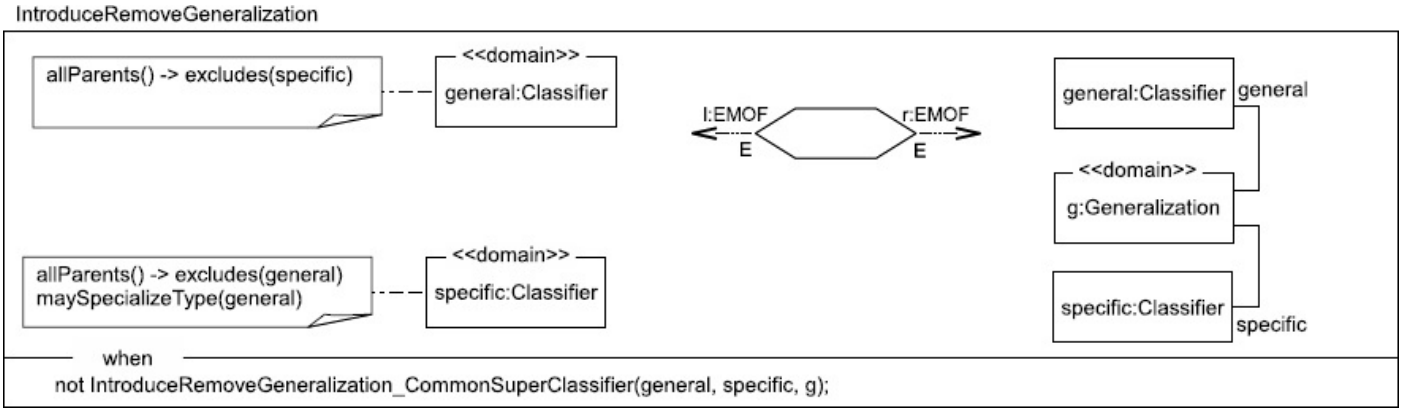


Figure 6.8.: The *Introduce / Remove Generalization* relation

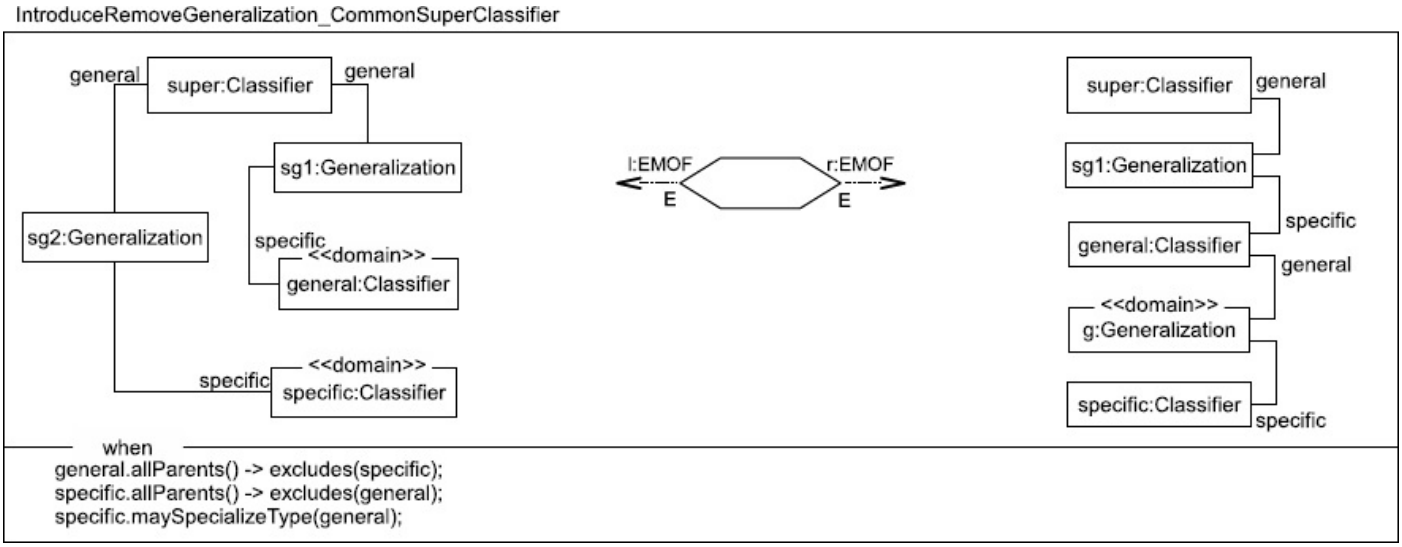


Figure 6.9.: The *Introduce / Remove Generalization* relation with common super classifier

6.3.16. Introduce Generalization

A generalization is introduced between two classifiers. The relation for the common case is given in Figure 6.8 in the direction left-to-right. Should instead the special case be applicable, where the two classifiers share the same super type, the generalization for the specific classifier is redundant and can be removed. This case is covered by the relation in Figure 6.9 .

- Preconditions:
  - `not IntroduceRemoveGeneralization_CommonSuperClassifier(general, specific, g)`: for the common case, ensure that the special case is not applicable instead.
  - `specific.allParents()-> excludes(general)`: The classifier general is not already part of the generalization hierarchy of the classifier specific.
  - `specific.maySpecializeType(general)`: The classifier specific may sub-class the classifier general. The operation maySpecializeType() returns true by default if the type of general is the same or more general than that of specific.

Subclasses of type Classifier may introduce further specialization constraints by overwriting this behaviour. [77, p. 53].

- `general.allParents()-> excludes(specific)`: The classifier specific is not part of the generalization hierarchy of the classifier general (to prevent cyclic relationships in the generalization hierarchy).

- Postconditions: *none*

6.3.17. Remove Generalization

A generalization between two classifiers is removed. The relation is given in Figure 6.8 in the direction right-to-left. Should the general classifier inherit from further, more general classifiers, generalizations to these need to be introduced for the specific classifier, as otherwise properties previously available from classifiers further up the inheritance hierarchy would no longer be available in the specific classifier. This case is covered by the relation in Figure 6.9, also in the direction right-to-left.

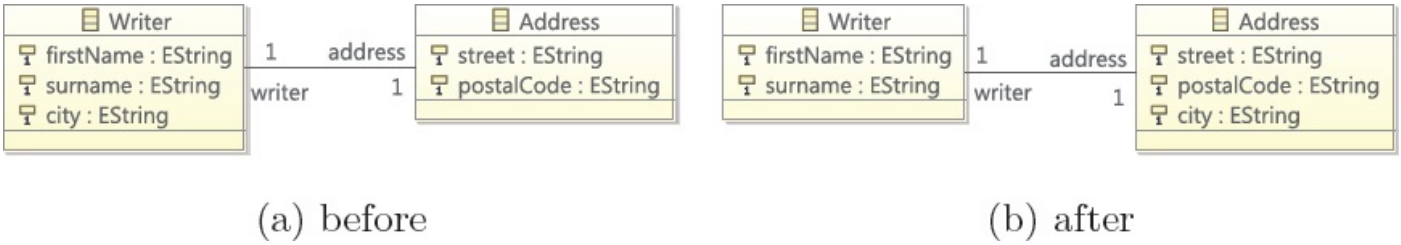


Figure 6.10.: *Move Property* operator example

- Preconditions:
  - `not IntroduceRemoveGeneralization_CommonSuperClassifier(general, specific, g)`: for the common case, ensure that the special case is not applicable instead.

- Postconditions:
  - `specific.allParents()-> excludes(general)`: The generalization relationship has been removed.

6.4. Move Property

A property is moved between classes that are linked by an association with a multiplicity of one-to-one. The restriction on the association is necessary so that values of the property of an instance after the move can unambiguously be associated with the original instance.

In the example in Figure 6.10, the property *city* is moved from *Writer* to *Address* along the one-to-one association between the two classes.

The relation for the *Move Property* operator is given in Figure 6.11, as read from left-to-right. The relation is its own inverse, switching source and target classes reverts the impact of the operator, moving the property back.

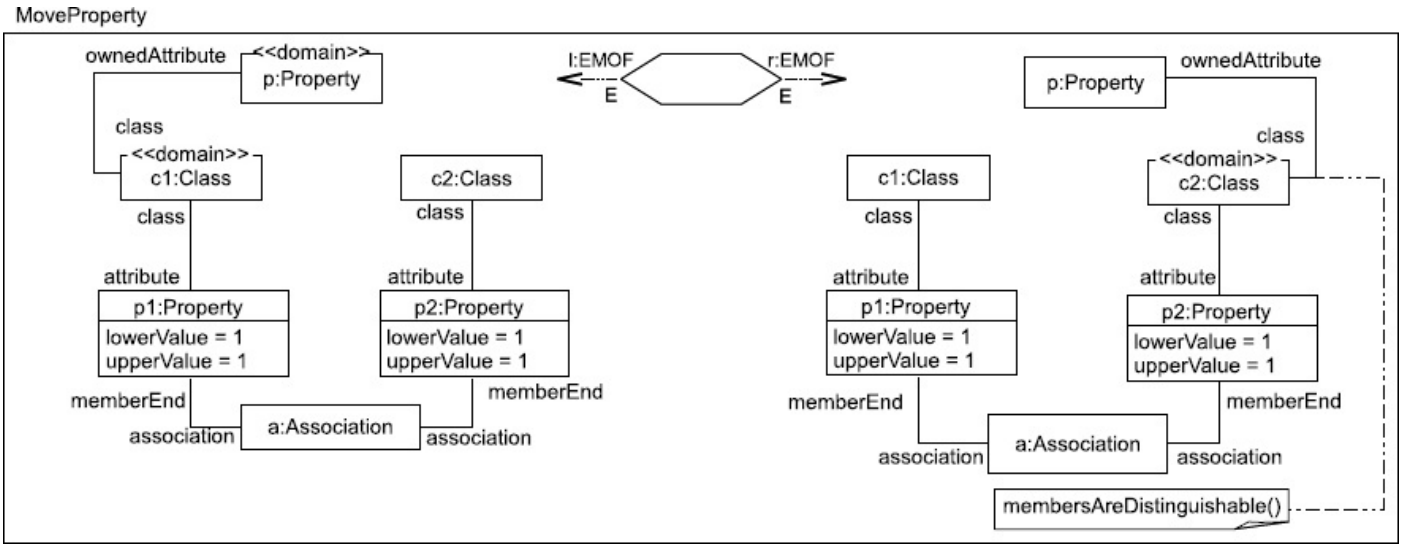


Figure 6.11.: The *Move Property* relation

- Preconditions: *none*
- Postconditions:
  - `c2.membersAreDistinguishable()`: the target class does not already directly or indirectly own a member of the same name as the property to be moved.

6.5. Push Simple Property

The *Push Simple Property* operator is an adaptation for EMOF of Fowlers *Push Down Field* refactoring pattern [34]. A property with a simple type is pushed from a superclass into all subclasses that are immediate children of the superclass. Subsequent application of *Push Simple Property* on the property on the child classes can push properties to further removed classes along the generalization hierarchy. Pushing properties to all the children of a class makes little sense on its own, so the property can be removed from those children that don’t need it using the *Delete Property* operator. This fulfils the original description of Fowler’s pattern. Alternatively, *Push Simple Property* can be used to remove all properties from a superclass and then delete it using *Delete Class.*

We discern between pushing simple and complex properties as the impact is different and extra steps are needed to adjust associations linked to properties for complex types.

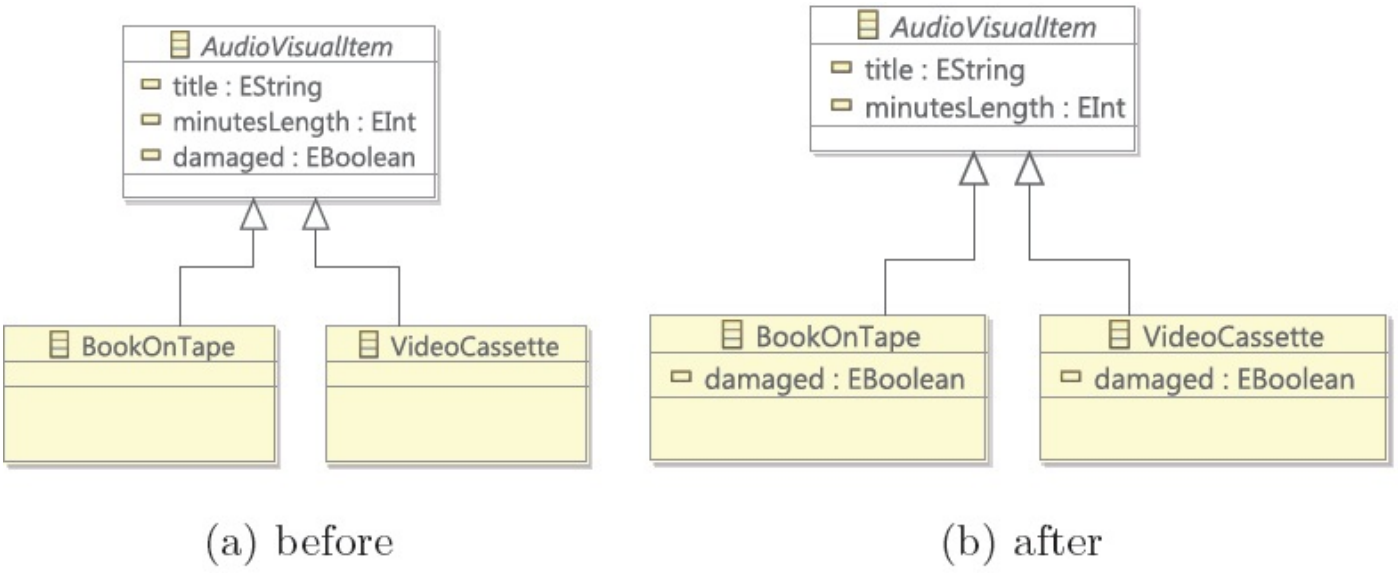


Figure 6.12.: *Push Simple Property* operator example

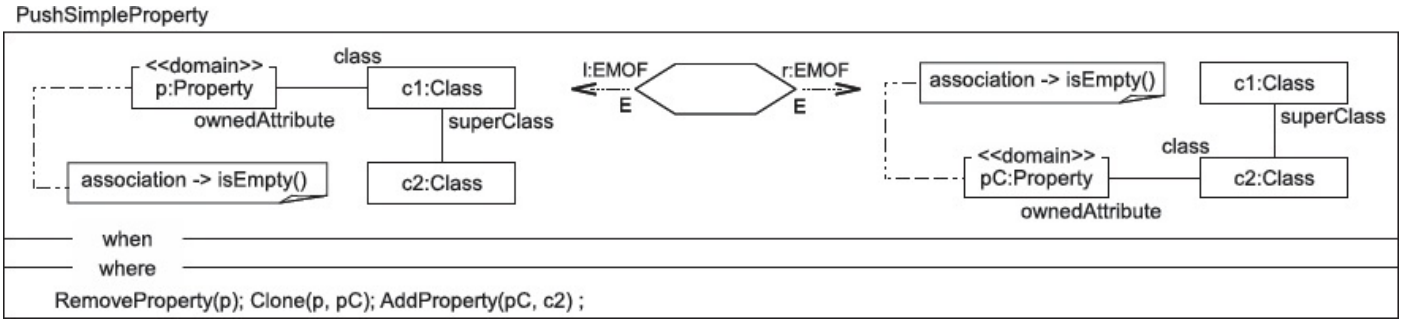


Figure 6.13.: The *Push Simple Property* relation

Wachsmuth proposes a similar version of *Push Simple Property* for CMOF as 'pushProperty' [107], but does not discern between simple an complex properties so that it does not cater for associations.

Figure 6.12 shows a small example of applying the *Push Simple Property* operator: the property damaged is pushed from the parent class AudioVisualItem to both its children.

The *Push Simple Property* operator is defined by the relation in Figure 6.13 . In the application, the simple property p is cloned and set on the class c2 matching the pattern. Please note that LHS pattern produces a match for every child class c2 that inherits from the superclass c1 so that a clone of the property is created for every child class.

- Preconditions:
  - `p.association -> isEmpty()`: ensure that the property p is a simple property.
- Postconditions:
  - `RemoveProperty(p)`: ensures the correct removal of the original property.
  - `Clone(p, pC)`: create a proper clone for each child class c2.
  - `AddProperty(pC, c2)`: add the clone to the child class c2.

6.6. Push Complex Property

Like the *Push Simple Property* operator, the *Push Complex Property* operator is used to push a property from a superclass to all immediate subclasses and then remove the property from the superclass. The *Push Complex Property* operator performs this operation on properties of a complex type, where the link to the value instance and its type is established by an association. The association has to be cloned along with the property for each of the subclasses that receive the pushed property, which makes the differentiation from the *Push Simple Property* operator necessary.

Associations establish links between two instances over the properties of two classifiers (complex types) which serve as association-ends (see Section 2.3.2). On one side, the property is always owned by a class. The link established by the association is always navigable in this direction. On the other side, the property can be owned by the association itself, if the association is directed and the link only navigable in one direction. If the association is bidirectional instead, the property is owned by a class on the other side and not the association.

The need to also adjust the association and the remote property along with the property being pushed makes a separate operator for complex properties necessary.

In the case of a bidirectional association, cloning the property on the far side of the push can lead to name clashes, as a clone of the property has to be created for each receiving child class. As the cloned properties are owned by the same classifier, duplicating the name would violate the uniqueness constraint for names. Here the clone operator needs to assign unique names or allow the user to assign an original name (user intervention). This is enforced by the `membersAreDistinguishable()` constraint in the relation.

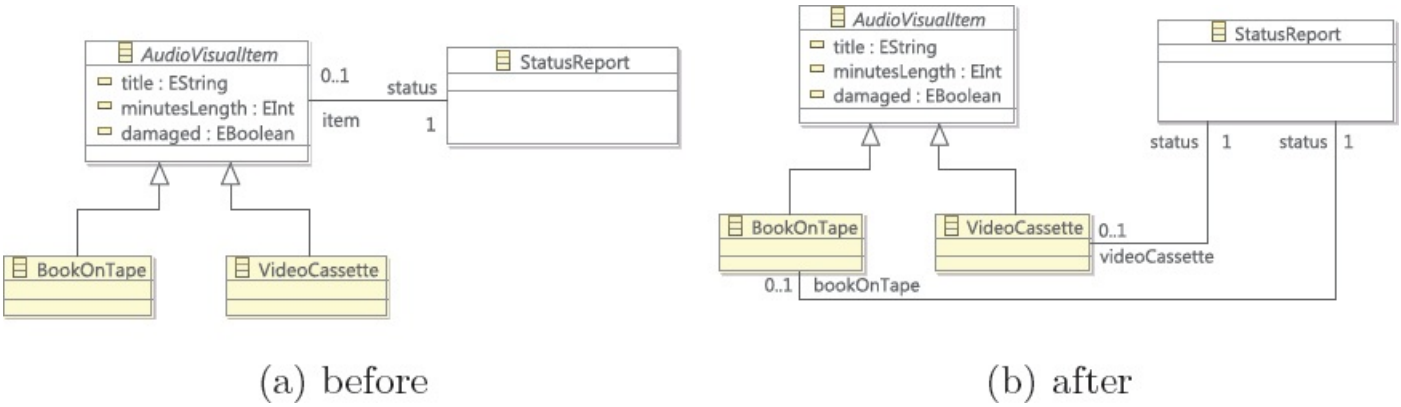


Figure 6.14.: *Push Complex Property* operator example

In the example in Figure 6.14, the property `status` is pushed from the class `AudioVisualItem` to its subclasses `BookOnTape` and `VideoCassette`.As the property is linked to a bidirectional association and the property `item` on the far side, the association and the remote properties are cloned and the remote properties renamed to `bookOnTape` and `videoCassette` respectively – to maintain uniqueness.

The *Push Complex Property* operator is defined by the relation in Figure 6.15 . For each occurrence of the target complex property on a superclass-subclass pattern, a clone of the property and a clone of the association is created and the cloned property is assigned to the subclass. The property on the remote end of the association can be owned either by another class (bidirectional) or the association itself (unidirectional), where the owner is designated by the classifier `cX` in bother patterns. The `type` of the remote property is

adjusted to the subclass `c2:Class`. In cases with more than one subclass and a bidirectional association, the remote property is cloned numerous times; once for each subclass. Here the user needs to assign new and unique names for the cloned properties to ensure distinguishability and to properly reflect the new semantics of each property, as the set of navigable instances is divided up between the cloned properties. For this reason the sub-relation `CloneDiscName()` (‘clone discounting name’) is used, which prompts the user for a new name (see Section 6.1.2).

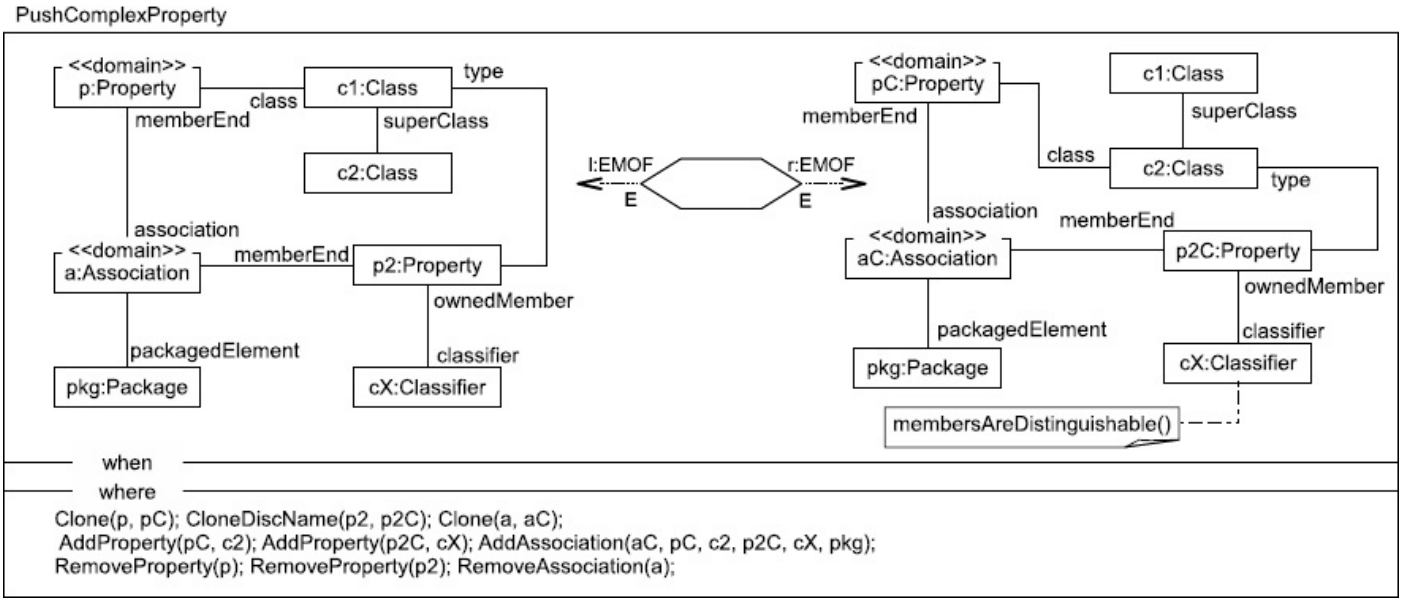


Figure 6.15.: The *Push Complex Property* relation

- Preconditions: *none*
- Postconditions:
  - `Clone(p, pC); CloneDiscName(p2, p2C); Clone(a, aC)`: create clones for each subclass of both properties and the association. For the remote property, a clone with a new, unique name is created.
  - `AddProperty(pC, c2); AddProperty(p2C, cX); AddAssociation(aC, pC, c2, p2C, cX, pkg)`: add the cloned elements to the model.
  - `RemoveProperty(p); RemoveProperty(p2); RemoveAssociation(a)`: remove the old properties and the association.
  - `membersAreDistinguishable()`: ensure that the newly created properties have unique names.

A possible alternative approach would be to first split up any bidirectional associations to two unidirectional ones and then perform the push only on a unidirectional association. This would leave the reverse association pointing at the superclass and maintain the same set of instances as before. Yet, the reversibility of the association would then not be enforced by the model any longer and would have to be maintained by other means and more importantly, the effect of the push property operator would not be reversible, as whether or not two properties are opposites and can take part in a *Pull Property* operator can not be decided based on the model alone. For these reasons we chose to maintain the bidirectional nature over maintaining the whole reverse association.

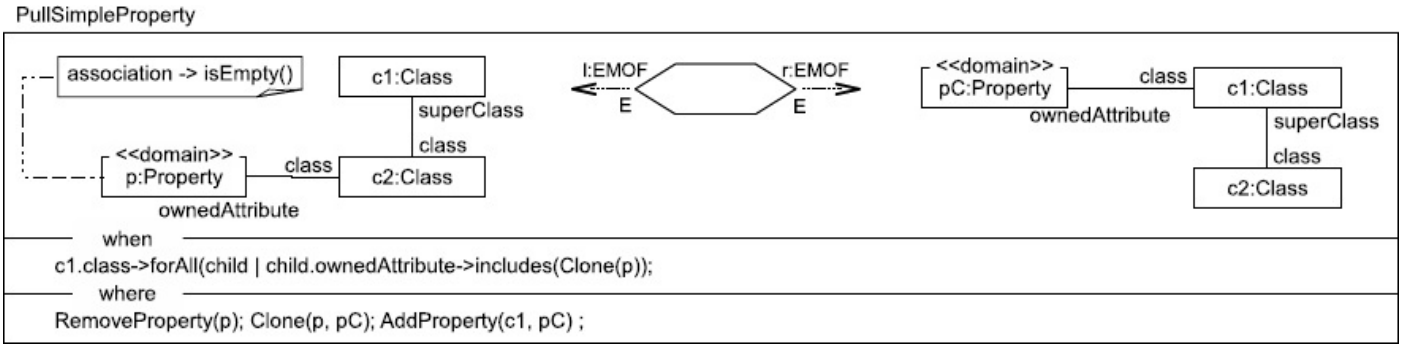


Figure 6.16.: The *Pull Simple Property* relation

6.7. Pull Simple Property

The *Pull Simple Property* operator 'pulls' equivalent simple properties from all subclasses into a common, direct superclass and removes them from the subclasses. This is the reverse effect of the *Push Simple Property* operator (6.5). Should some subclass of the superclass not have the given property, it can be added first by applying *Create Element* for that property. Further applications of the operator can pull the property further up the inheritance hierarchy.

The example in Figure 6.12 serves to illustrate the effect of pulling a property, as it reverses pushing the property, by swapping 'before' and 'after': the property `damaged` is pulled from the children to the parent `AudioVisualItem`.

The relation in Figure 6.16 defines the behaviour of the *Pull Simple Property* operator. It is almost the reverse of the *Push Simple Property* relation (Figure 6.13) with one exception: a precondition ensures that if *Push Simple Property* is applied, it must be applied for all child classes, in other words, all subclasses of the target class must own a clone of the property to allow it to be pulled. The relation is executed for each occurrence of the property on a subclass `c2` and a single clone `pC` remains on the superclass `c1`.

- Preconditions:
  - `p.association -> isEmpty()` - ensure that the property `p` is a simple property.
  - `c1.class->forAll(child | child.ownedAttribute.includes(Clone(p));` - ensure that all subclasses own a clone of the property `p`.
- Postconditions:
  - `RemoveProperty(p);` - ensures the correct removal of the original property.
  - `Clone(p, pC);` - ensure the existence of a clone `pC` for `p`.
  - `AddProperty(pC, c2);` - add the clone to the superclass `c1`.

6.8. Pull Complex Property

The *Pull Complex Property* operator pulls complex properties that are equivalent from all subclasses into a common superclass. It has the inverse effect of the *Push Complex Property* operator of Section 6.6. Like the *Pull Simple Property* operator, it only applies when an equivalent property is owned by all subclasses of the receiving superclass.

Should one be missing from a subclass, it can be added prior to the operator usage using the *Add Property* operator (6.3.10). An example for the effect of the operator can be seen in Figure 6.14, when read from right-to-left: the properties `status` of the classes `BookOnTape` and `VideoCassette` are pushed into the class `AudioVisualItem` and joined.

- Preconditions:
  - `c1.class->forAll(child | let pc: Clone(p)in child.ownedAttribute->includes(pc)and pc.association = Clone(a)and pc.association.memberEnd->includes(Clone(p2));` - ensures that all subclasses `(child)` own a suitable property - association - property construct, of which each element fulfils the role of a clone.

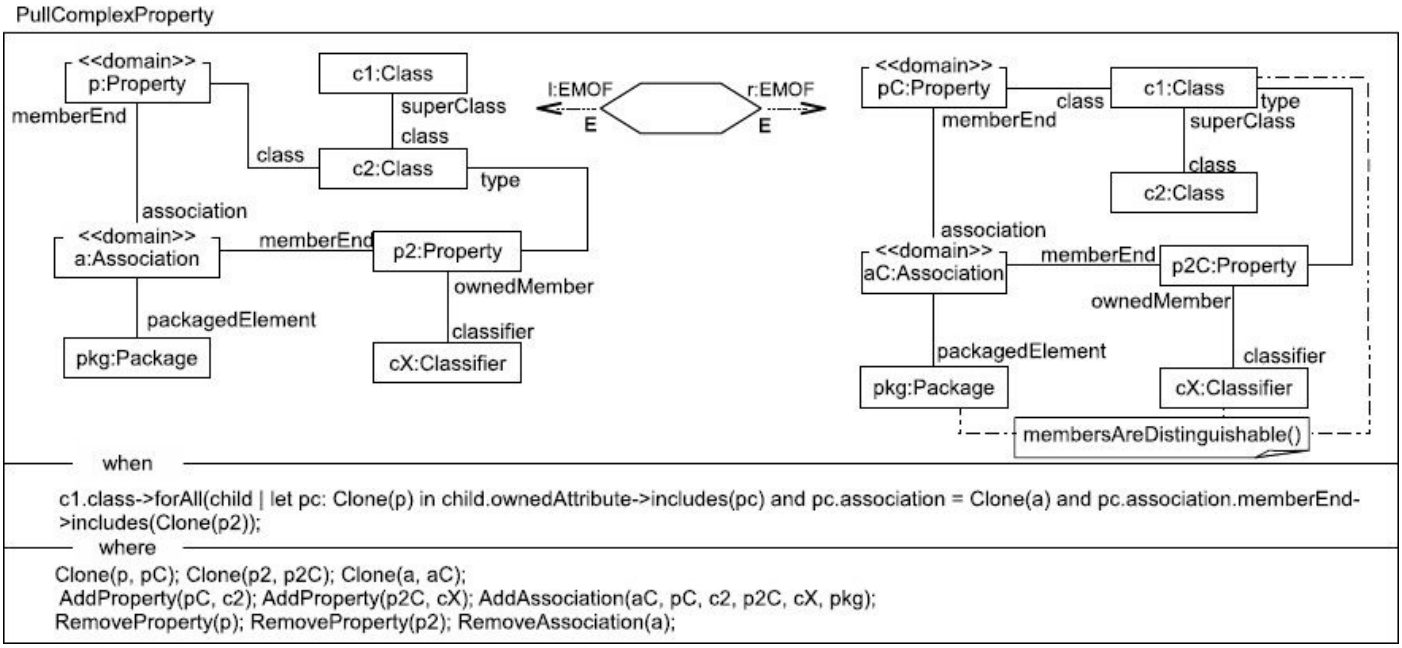


Figure 6.17.: The *Pull Complex Property* relation

- Postconditions:
  - `Clone(p, pC); Clone(p2, p2C); Clone(a, aC)`: clones for the elements connected to the subclass need to exist on the superclass.
  - `AddProperty(pC, c2); AddProperty(p2C, cX); AddAssociation(aC, pC, c2, p2C, cX, pkg)`: add the cloned elements to the model.
  - `RemoveProperty(p); RemoveProperty(p2); RemoveAssociation(a)`: remove the old properties and the association.
  - `membersAreDistinguishable()`: ensure that the newly created elements are unique.

The relation in Figure 6.17 defines the behaviour of the operator. The property p is pulled up from the subclass c2 to the parent class c1. This is achieved by cloning both the property, the attached association a and the property p2 on the remote side of the association. The relation matches and is executed for all subclasses of c1, so that the property is removed from all subclasses. The cloning and addition relations in the where-clause enforce the existence of the elements for the superclass c1 after each execution case.

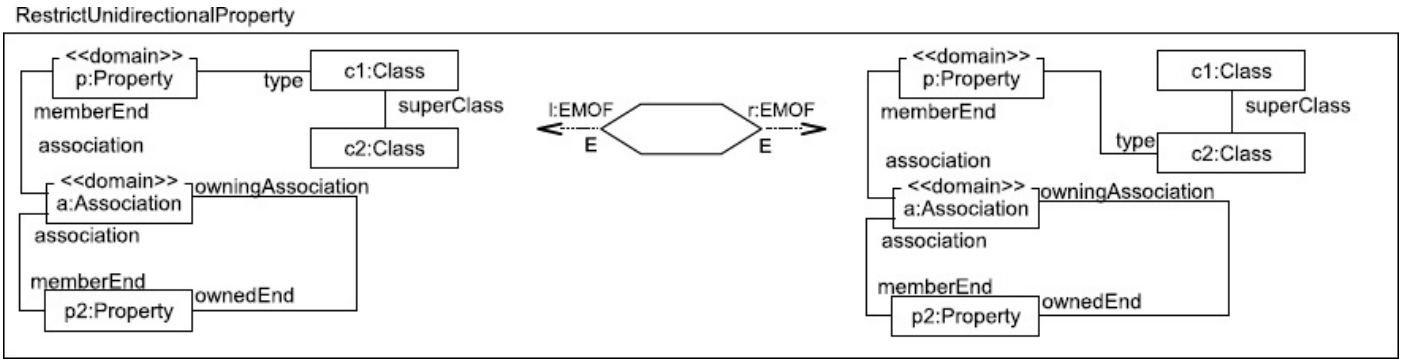


Figure 6.18.: The *Restrict Unidirectional Property* relation

6.9. Restrict (Unidirectional) Property

The *Restrict Property* operator reduces the range of allowed value instances of a complex property by specialising the type of the property (and its values). As complex properties are modelled using associations in EMOF, the nature of the attached association has to be taken into account: If the association is bidirectional, restricting the type of a property on one side of the association equates ‘pushing down’ the property on the other side, as it is owned by the class serving as type. Here we suggest using the *Push Complex Property* operator on the remote property and removing all association clones that have the intended restricted type (see section6.6).

If the association is unidirectional and thus the remote property is owned by the association instead of the opposite class, the *Restrict Property* relation in Figure 6.18 can be applied.

Figure 6.18 defines the *Restrict Property* operator. The application changes the type of the complex property p from the superclass c1 to the subclass c2. The property p2 at the remote end of the association attached to p has to be owned by the association for the relation to apply, which effectively constrains the relation to unidirectional associations.

- Preconditions: *none*
- Postconditions: *none*

6.10. Generalise (Unidirectional) Property

The *Generalise Property* operator increases the range of allowed value instances of a complex property by generalising the type of the property (and its values). It has the opposite effect of the *Restrict Property* operator of Section 6.9. Like its opposite, it is only applied to unidirectional complex properties, as bidirectional associations can be handled by pulling the opposite property up with the *Pull Complex Property* operator (see Section 6.8).

As the association is unidirectional and thus the remote property is owned by the association instead of the opposite class, the inverse of the *Restrict Unidirectional Property* relation in Figure 6.18 can be applied, in the direction right-to-left.

6.11. Extract Class

A class is extracted from an existing class in the same package. An association is created between the two classes with a multiplicity of one-to-one. The restriction on the association is necessary to ensure that one instance of the former class can be mapped to a set of instances of the old and the new class after the operator is applied. The newly created class is empty after the operator is applied. In a subsequent adaptation step, properties can be moved from the original class to the new class with *Move Property* (see Section 6.4) or new features can be added to the new class using *Add Property* (see Section 6.3.10).

In the example in Figure 6.19, a new class `StatusReport` is extracted from the class `AudioVisualItem` and linked by a one-to-one association to the class. The *Move Property* operator could be used next to move the `damaged` property from `AudioVisualItem` to `StatusReport`. `StatusReport` could further be extended to hold more information on the condition an item of the library in the example is in.

Figure 6.20 shows the relation for the `Extract Class` operator. The class `c1` is the parameter of the operator on the LHS to which the operator is applied. The newly created and added elements are set to belong to the same package as `c1`. Sub-relations for the creation and addition of the new elements are called in the where-clause to ensure all features of the new elements are set correctly and all pre- and post-conditions for their addition are met.

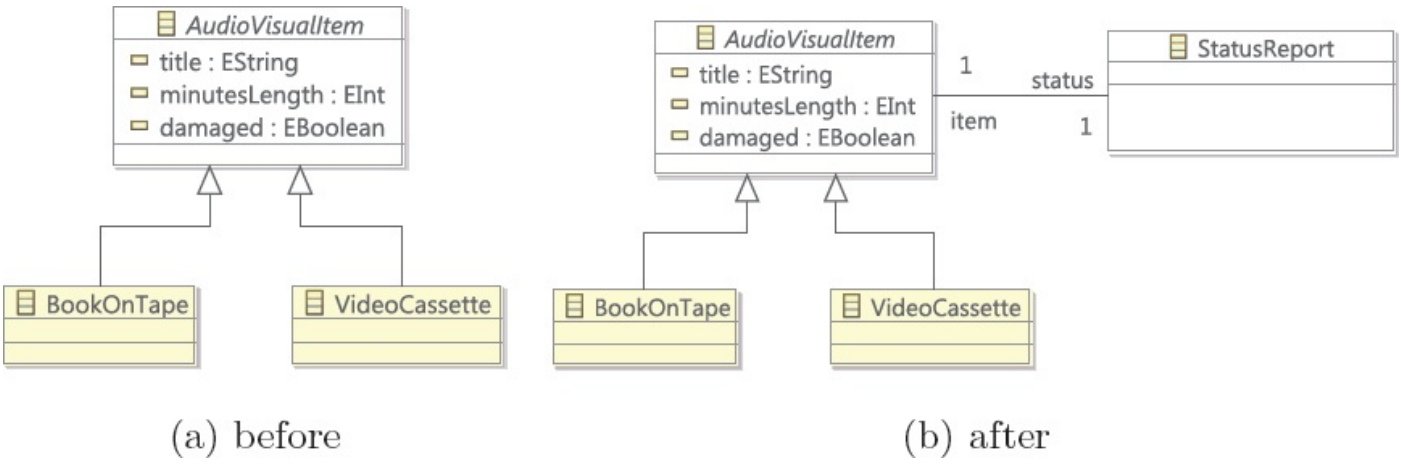


Figure 6.19.: *Extract Class Operator* example

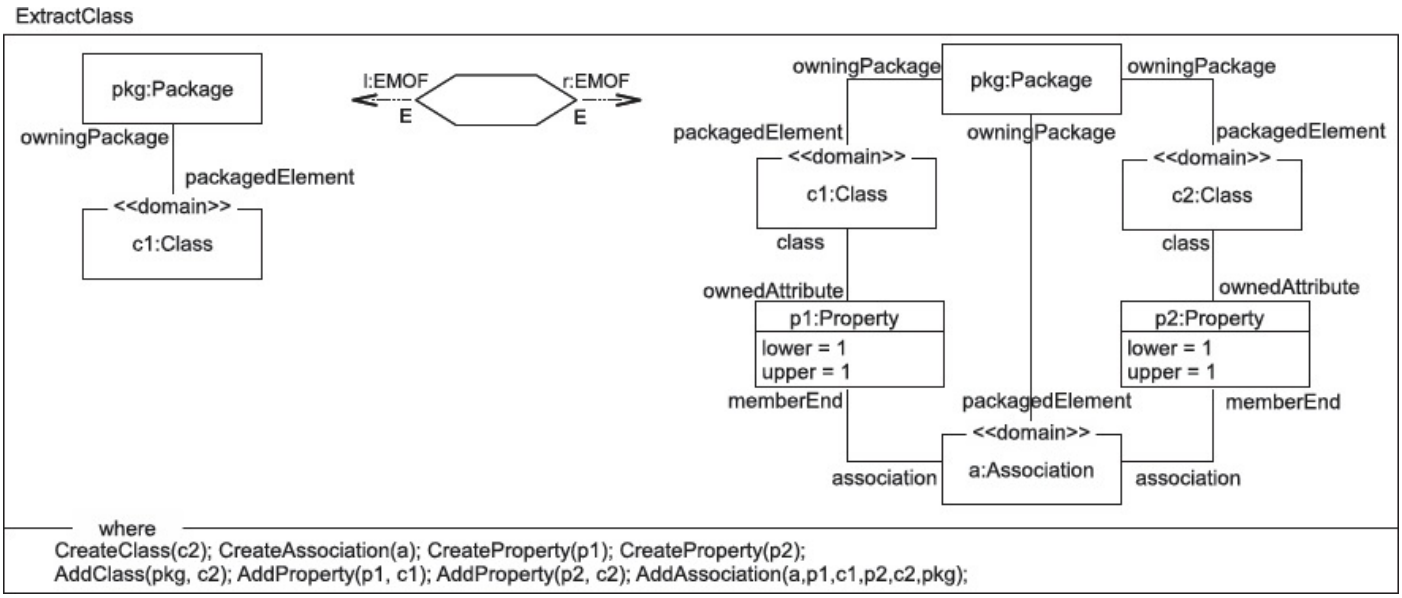


Figure 6.20.: The *Extract Class* relation

- Preconditions: *none*
- Postconditions:
  - `CreateClass(c2); CreateAssociation(a); CreateProperty(p1); CreateProperty(p2)`: The new elements are created.
  - `AddClass(pkg, c2); AddProperty(p1, c1); AddProperty(p2, c2); AddAssociation(a,p1,c1,p2,c2,pkg)`: The newly created elements are added to their respective owners.

6.12. Inline Class

The *Inline Class* operator has the opposite effect of the *Extract Class* operator: a class is merged into another class to which it was connected to by an association with a multiplicity of one-to-one. The class and the association are removed from the model. We refer to the class that remains as the ‘target’ and the class that is merged into it as the ‘merged’ class in the following. When looking at the effect of the operator on the metamodel level only, it is the same as simply removing the merged class and the association. Yet when handling the impact on transformations, the distinction between a merge and a deletion becomes important.

The merged class is purposefully constrained to be empty for the operator to be used, as properties or associations of the merged class can be treated in a number of different ways before using *Inline Class* by using other operators like *Move Property* (see Section 6.4) to move them to the target class first or simply by removing them.

As the effect of this operator is the reverse of *Extract Class*, the example in Figure 6.19 can be used to illustrate its effect: If the operator is applied to the class StatusReport on the right-hand side, it is merged into the class AudioVisualItem (which is indistinguishable from removing it, when considering the effect on the metamodel level only).

The operator is shown in Figure 6.21 . The operator is applied to the merged class c2, which is merged into the target class c1. The association between the two and the

properties connecting the association are removed with the merge. The operator requires no dedicated preconditions as the operators used to remove the elements in the *where*-clause ensure a clean removal, for example that the class c2 is empty for the operator to applicable.

- Preconditions: *none*
- Postconditions:
  - `RemoveClass(c2); RemoveAssociation(a); RemoveProperty(p1); RemoveProperty(p2)`: ensure the proper removal of the given elements.

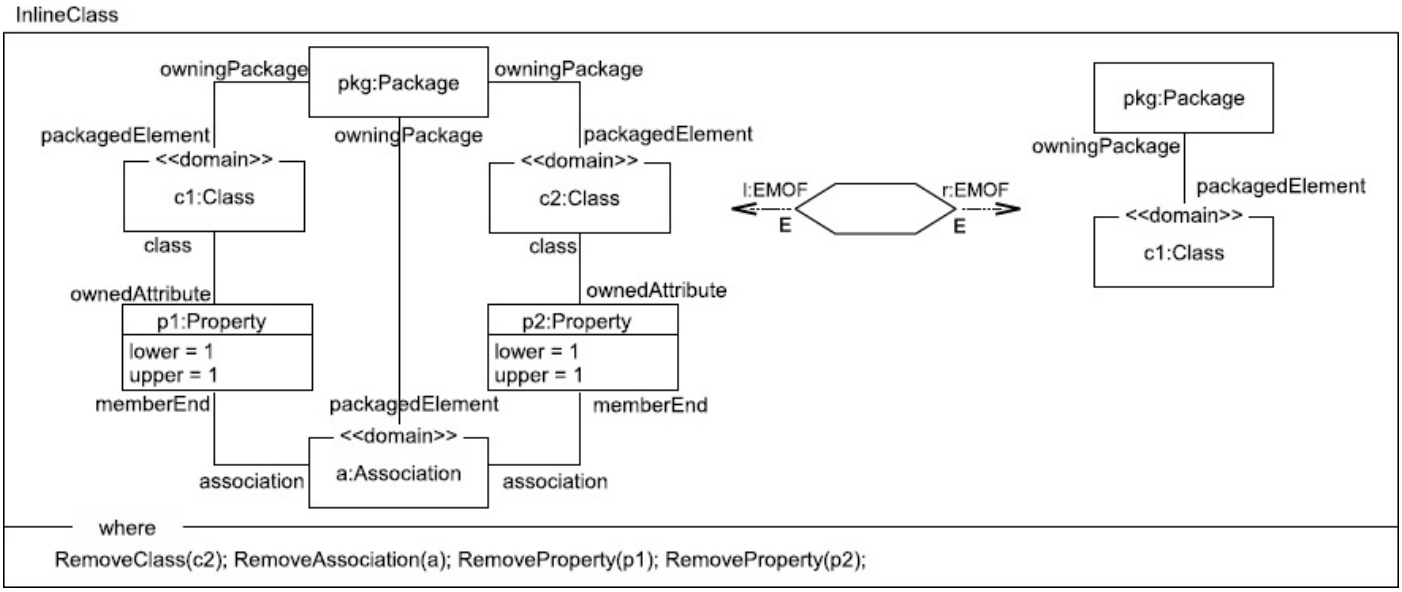


Figure 6.21.: The *Inline Class* relation

6.13. Extract Superclass

A superclass is extracted from one or more child classes. A selection of common features of the children can be pulled up into the new superclass. This operator has the inverse effect to the *Flatten Hierarchy* operator. The new superclass is owned by any given package.

- Preconditions:
  - `props->forAll(p | c1->forAll(c | let clone: Clone(p,clone)in c.ownedAttribute.includes(clone))`: A clone of all properties to be pulled must be present in each of the child classes, i.e. before a property can be pulled up to the new parent class, it has to be present with the same name and type in every child class.
- Postconditions: *none*

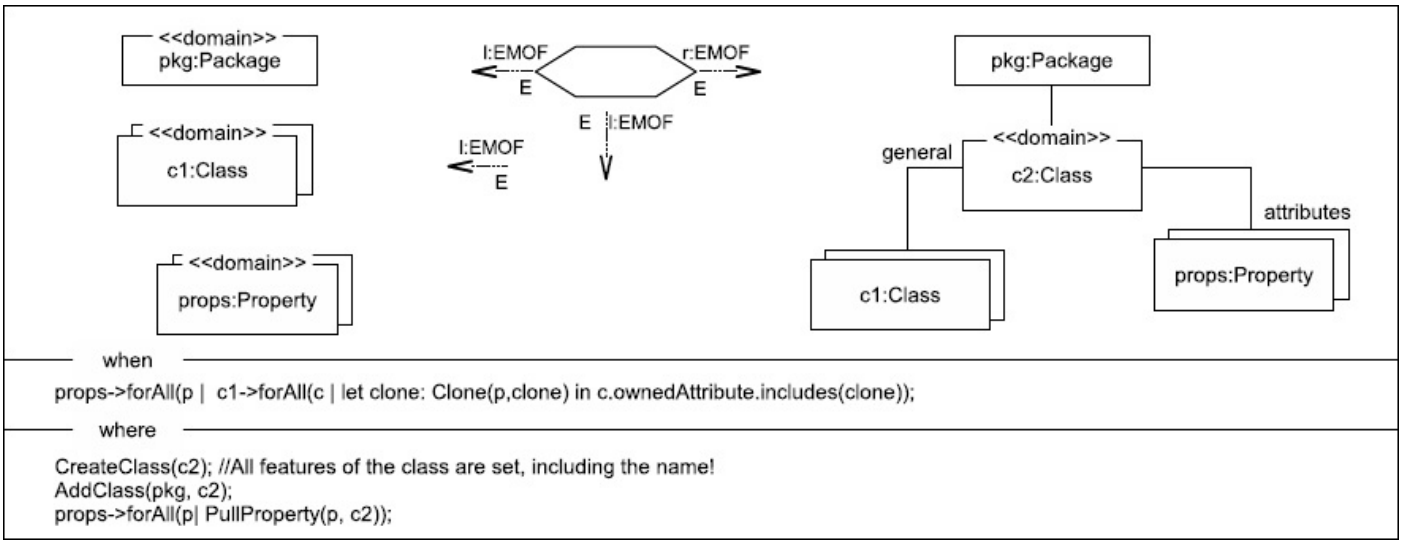


Figure 6.22.: The *Extract Superclass* relation

6.14. Flatten Hierarchy

The *Flatten Hierarchy* operator is used to reduce the inheritance hierarchy of classes: the features of a target class are moved to all child classes and the class is deleted. This operator is implemented by applying *Push Simple Property* (6.5) or *Push Complex Property* (6.6) for all properties of the parent class until it is empty. The generalizations between the parent and the child classes are then removed using the *Remove Generalization* operator (6.3.17). Should the parent class inherit from another class even further up the inheritance hierarchy, the operator ensures that the child classes are set to inherit from that class instead, so that inherited features remain available. The parent class is then removed using *Remove Class* (6.3.3).

Although the task of flattening the hierarchy is achieved by delegating to other operators, the existence of a distinct operator for this task is merited, as the impact resolution for transformations can be improved with the knowledge of its usage (see 7.16).

In the example in Figure 6.23, the level of hierarchy is reduced, by ‘pushing’ the property `title` of the abstract class `CirculatingItem` down into the subclasses and removing the class. The `title` property was previously available to instances of the subclasses by inheritance and is now made explicitly available by the operator.

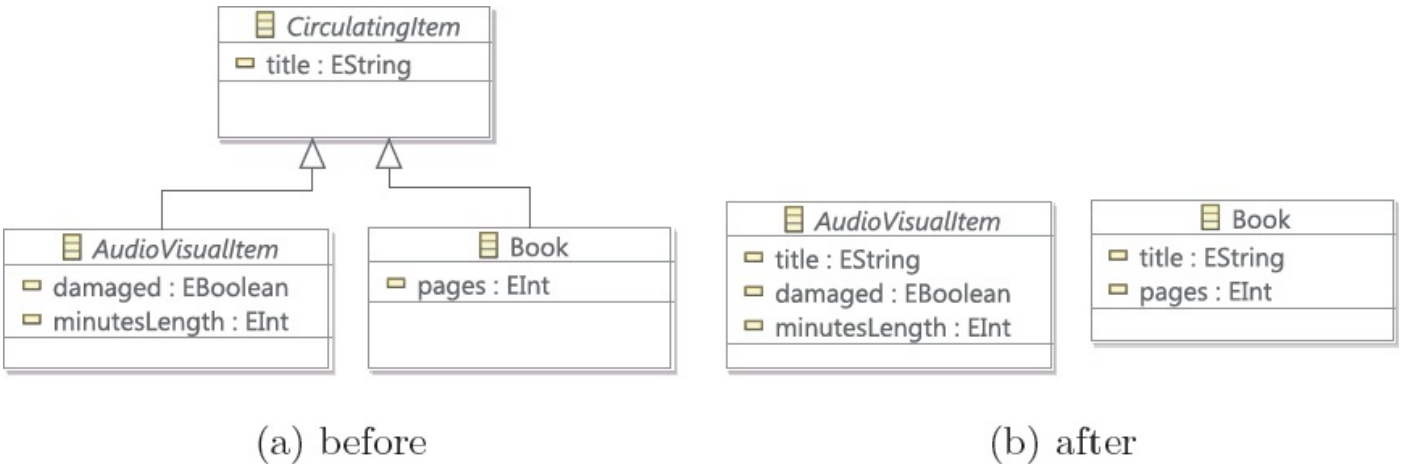


Figure 6.23.: *Flatten Hierarchy* operator example

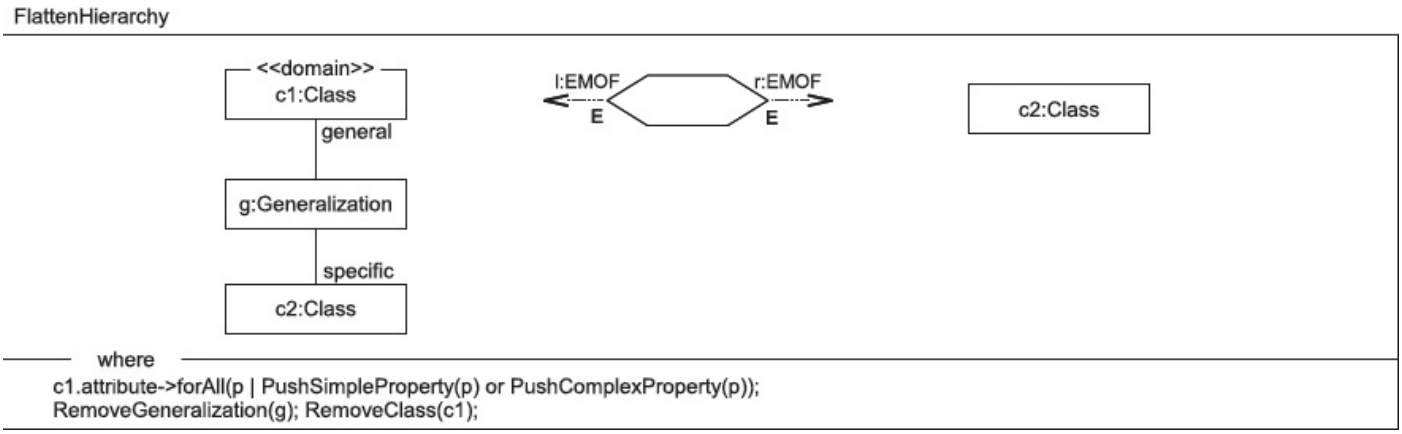


Figure 6.24.: The *Flatten Hierarchy* relation

The *Flatten Hierarchy* operator defined by the relation in Figure 6.24 matches every occurrence of a subclass c2 to the selected superclass c1 so that it is executed for each subclass. A clone of every property of the class c1 is created in every subclass and removed from the superclass with a call to the *Push Property* operator. Subsequently, the empty superclass and all existing generalizations are removed.

- Preconditions: *none*
- Postconditions:
  - `c1.attribute->forAll(p | PushSimpleProperty(p)or PushComplexProperty (p))`: all properties are pushed into the subclasses.
  - `RemoveGeneralization(g); RemoveClass(c1)`: both the generalization and the superclass are removed.

6.15. Association to Class

The *Association to Class* operator replaces an association between two classes by a class connecting the two classes instead. This is useful for example if a relationship between two entities needs to be extended to hold information beyond the fact that the two entities are related. Introducing a new class like this fulfils the same purpose as the *AssociationClass* element available in UML2, which is an association which owns features [77, p. 44].

In the example in Figure 6.25, the directed association between the classes Borrower and Lendable is replaced by the class Loan. This class can be extended afterwards, for example by adding properties which hold further information about the loan, like the due date or the number of loan extensions.

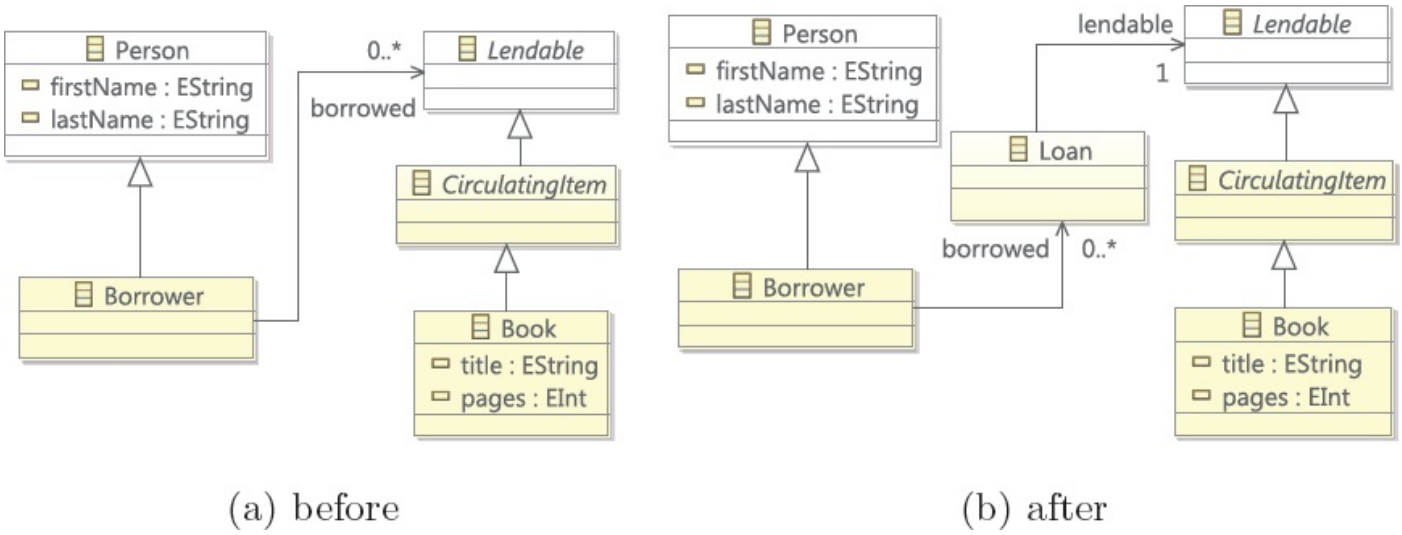


Figure 6.25.: *Association to Class* operator example

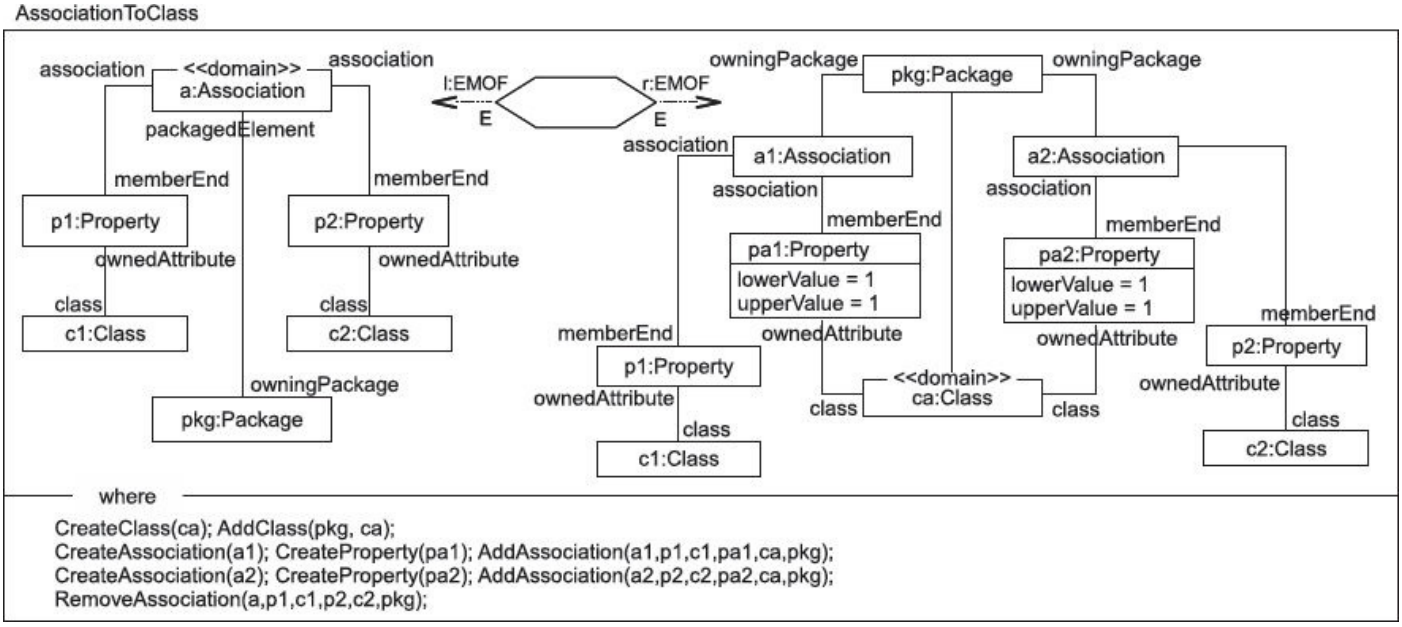


Figure 6.26.: The *Association to Class* relation

The *Association to Class* operator is defined by the relation in Figure 6.26 . The association a is replaced by the new class ca. To establish the link to the new class, the two new associations a1 and a2 are created. The member-end properties of the old association that are owned by the classes are repurposed as member-ends of the new associations on the side of the classes. On the other side, two new properties are created for the new class `ca` and given an upper and lower bound of `1`. The bounds and names of the repurposed properties can be maintained to remain as close to the original configuration as possible. In sum, the allowed number of instances on both side remain the same, as the bounds are kept and are navigable by the same properties (bar the extra navigation step from the new class to the remote end, which can not be avoided). The user can provide names for the new elements, being the new class and the owned properties. This is performed in the `Create..()` operation in the `where`-clause.

- Preconditions: *none*
- Postconditions:

- `CreateClass(ca); AddClass(pkg, ca)`: Create the new class and add it to the package owning the association.
- `CreateAssociation(a1); CreateProperty(pa1); AddAssociation(a1,p1, c1,pa1,ca,pkg)`: Create an association and a property to link to the new class on one side…
- `CreateAssociation(a2); CreateProperty(pa2); AddAssociation(a2,p2, c2,pa2,ca,pkg)`:… and on the other.
- `RemoveAssociation(a,p1,c1,p2,c2,pkg)`: Remove the association that is replaced.

6.16. Generalization to Composition

A generalization between two classes is converted to an association. The association is made essential on the side of the child class, with an upper bound of 1. On the side of the former parent, the upper bound is 1 and if it was abstract, the lower bound is set to 1, otherwise it is 0. If the parent class itself inherited from a third class, both classes are set to inherit from that third class after the resolution. Hassam and al. call a similar operator 'InheritanceToComposition'. [42]

- Preconditions: *none*
- Postconditions: *none*

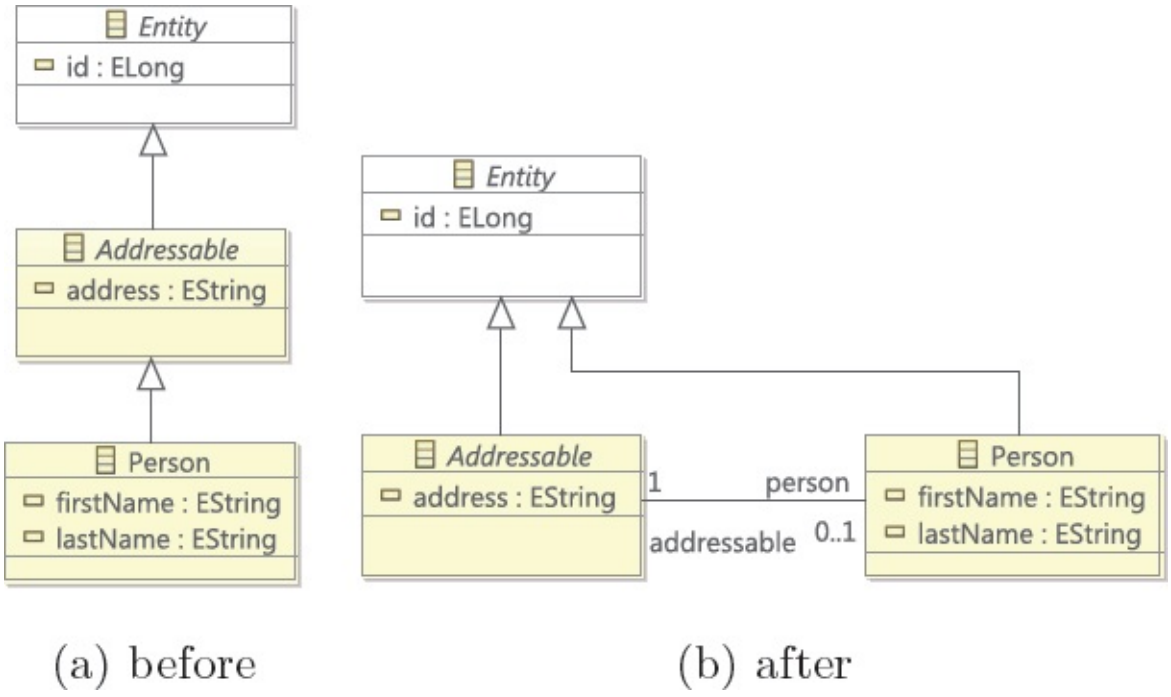


Figure 6.27.: *Generalization to Composition* operator example

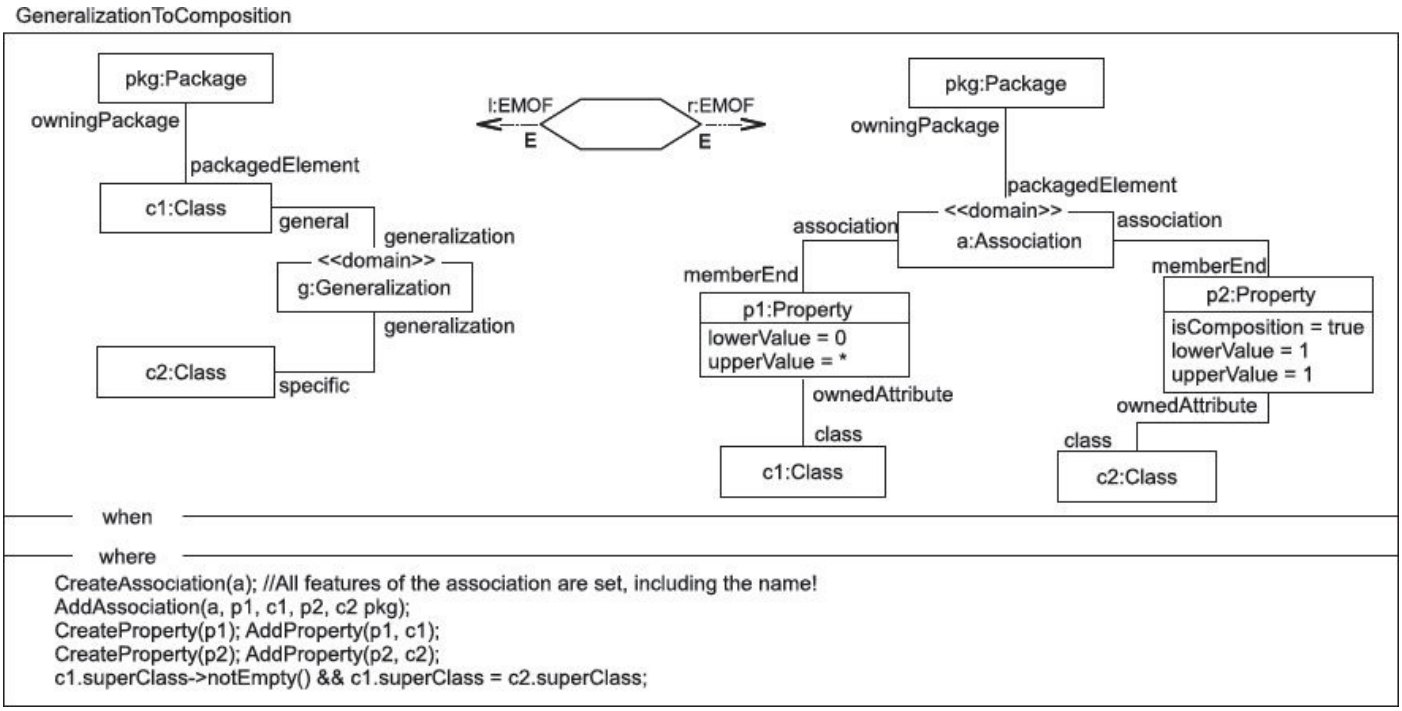


Figure 6.28.: The *Generalization to Composition* relation

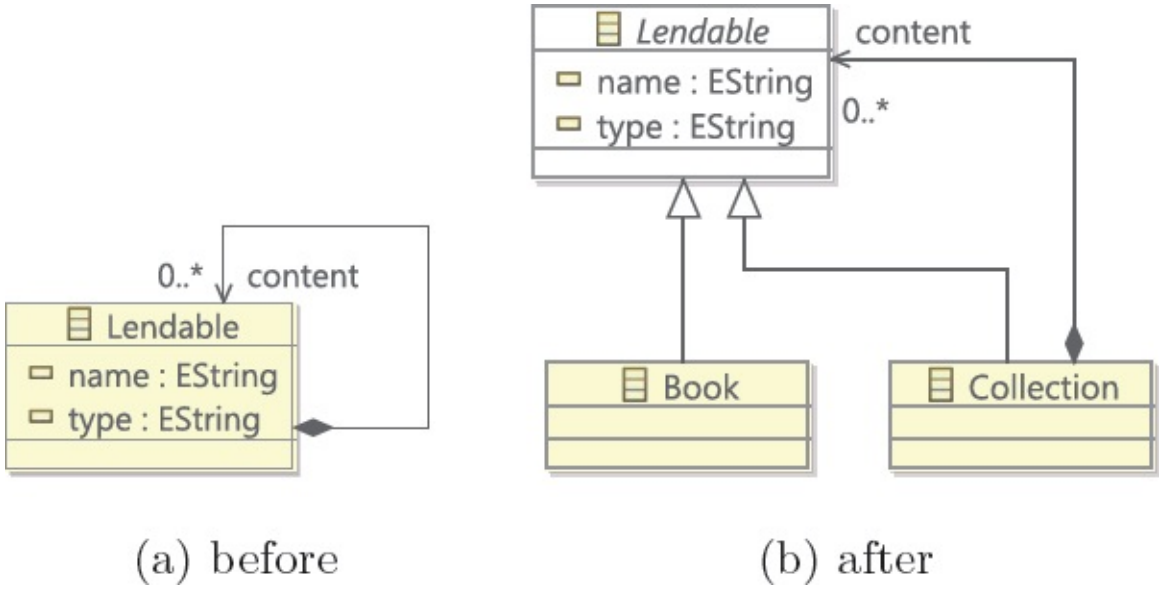


Figure 6.29.: *Introduce Composite Pattern* operator example

6.17. Introduce Composite Pattern

This operator serves the introduction of the *Composite* design pattern as defined by Gamma et al. [35] to a model. The operator is applied to an existing parent-child containment association of a class and divides the class into a class in the role of a *leaf* and one in the role of a *composite* as prescribed by the pattern. The classes both inherit from the original class, which is made abstract and which keeps all previously existing properties and relations left unchanged by the operator. These can be moved to either the leaf or the composite in a subsequent adaptation, if applicable. The parent-child association is redirected as a containment association between the composite and the abstract original class.

In example 6.29, the *Composite* pattern is introduced for the class Lendable and the containment association content. Lendable is split into a Book, which has no further

children and assumes the role of the leaf in the pattern. The Collection class can contain children and serves as the composite. Collections can now be made up of books or further collections. The type and name properties remain with the class Lendable and can be adapted afterwards. The class Lendable becomes abstract.

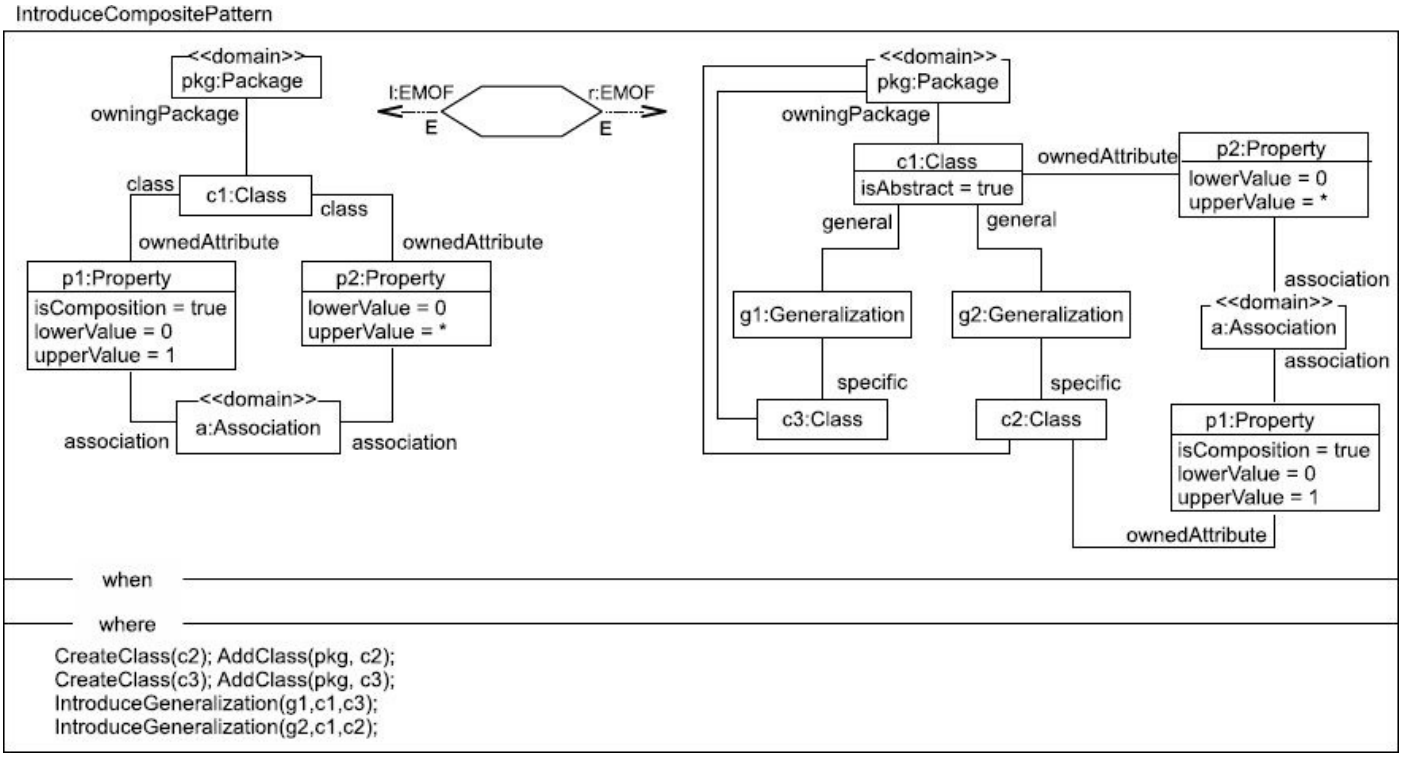


Figure 6.30.: The *Introduce Composite Pattern* relation

CHAPTER 7

Impact Resolution for ATL

This chapter introduces the impact resolution of the operators discussed in chapter 6 on ATL model transformations and in accordance with the third phase of our approach described in Section 5.7. The impact and resolutions suggested for each operator are discussed in detail in separate sections below. The next section provides an overview of the resolutions for all operators. The impact resolution for each operator is formalised as a QVT-R relation defined on the ATL AST and given in graphical notation. Section 7.2 discusses the omissions made for ATL and support functions defined for the relations to make them less verbose.

7.1. Operator Impact Overview

Table 7.1 summarizes the influence of the operators on ATL transformation rules and the resolution suggested for them. The occurring impact resolutions are:

- **none** The operator has no impact. It can be applied without endangering the validity of dependent model transformations.
- **automatic** The operator has an impact that needs to be resolved, but the resolution can be achieved automatically and without the attention of the software architect.

Operator	Section	Page	Impact and Resolution
Add Element	7.3	139	Impact and Resolution
Remove Element	7.4	140	none / HI for add prop.
Remove Class	7.4.1	140	none
Remove Property	7.4.2	141	automatic
Remove Association	7.4.3	146	automatic, warn
Remove Generalization	7.4.4	147	automatic, warn
Rename Element	7.5	147	automatic, warn
Move Property	7.6	147	automatic
Push Simple Property	7.7	152	automatic
Push Complex Property	7.8	154	none, warn, HI
Pull Simple Property	7.9	158	none, warn, HI
Pull Complex Property	7.10	158	none
Restrict (Unidir.) Property	7.11	159	none, HI
Generalise (Unidir.) Property	7.12	159	none, HI
Extract Class	7.13	160	none
Inline Class	7.14	161	automatic
Extract Super Class	7.15	163	automatic, HI
Flatten Hierarchy	7.16	163	none, automatic
Association to Class	7.17	166	automatic, warn, HI
Generalization to Composition	7.18	167	automatic
Introduce Composite Pattern	7.19	168	none, automatic
			automatic, HI

Table 7.1.: Overview of impact resolutions for ATL transformations

- **HI** The impact of the operator can not be resolved and human intervention is required. In an implementation, the user should be alerted to the exact part of the transformation that requires attention.

- **warn** An automatic resolution is possible and the syntactic correctness of the transformation is maintained by the resolution, but the software architect is warned because the resolution without any further change is not deemed to be practical or useful. The software architect is alerted to such cases.

For most operators, a combination of impacts is possible depending on the structure of the transformation and where in a transformation rule the impact occurred or whether the operator was used on the LHS or RHS metamodel of the transformation.

The details of the impact resolution for each operator are discussed individually starting from Section 7.3.

7.2. Support Functions and Omissions

For the impact resolutions defined in this chapter two omissions of the ATL language were made which is deemed permissible when showing the overall applicability of the approach, yet should be addressed when aiming for a complete tool support in the future. These are:

Helpers ATL provides the *helper* mechanisms to structure and reuse OCL expressions. Helpers can act as either global variables or functions defined on model elements and are an optional structuring mechanism [6, p. 14]. They use OCL expressions to calculate a value based on a set of parameters and are called from rules or other helpers. Because they are optional, they are omitted from the impact resolutions defined here, yet, whenever an impact relates to an OCL expression, a helper could also be impacted. Complete tool support of ATL would need to cover helpers in practice. As a workaround for the time being, one can eliminate helpers first by replacing all uses of a helper with the contained expression and removing the helper. This way, the approach is still usable without explicit helper support.

Imperative rule part ATL rules can have an optional imperative part. It is a convenience feature that contains one or more imperative statements that are called after target model elements have been initialised to provide values for uninitialized attributes or to modify values [5, p. 42]. The effect of imperative parts is difficult to predict and omitted from the impact detection. If imperative parts are used, users are advised to check such rules manually after an impact resolution is performed.

To simplify the impact definitions, two custom black-box operations are used in the relations. These provide basic support functions to be called from the relations where needed to make them less verbose. For an implementation of the impact resolutions in QVT-R, the black-box operations could be provided by the executing engine as defined by the QVT standard [73]. They are as follows:

Matches The `Matches()` operation provides the bridge between EMOF and ATL types. As ATL uses OCL expressions to identify model elements, the executing engine and the co-evolution approach need to identify the correct model element represented by an OCL expression. The `matches()` operation is defined to return `true` when the given `OCLModelElement` corresponds to the given EMOF metamodel `Element`.

Clone The `Clone()` operation creates a clone of an OCL expression. AS OCL expressions can be of arbitrary length, the cloning process can be extensive and its definition as a set of QVT-R relation would be verbose, while only creating a copy of each type of OCL expression possible. Therefore, `Clone()` is defined to return a clone of a given `OCLExpression` element, or evaluate to `true` if the two given expressions are valid clones.

The impact resolutions for *Flatten Hierarchy* in Section 7.16 and *Push Complex Property* in Section 7.8 make use of further custom black-box operations. These are defined together with the impact resolution as they are not reused elsewhere.

7.3. Add Element

Adding an element to a metamodel requires no impact resolution for most types of elements, as the existing metamodel elements remain unaltered and the existing ATL transformation rules that use them remain valid. For example, when adding a new class to a package, any existing transformation will neither read nor write instances of the new class until a new rule for this purpose is created.

The only exception is the creation of properties: here the metamodel can dictate that a certain amount of values is required for the property, by setting the lower bound of the property larger than `0`. Models without the proper amount of values for the property are invalid. Should such a requirement be defined for a new, RHS property and an existing transformation rule creates instances of the owning class, the required values will not be set and the instance model resulting from the transformation is no longer valid. Therefore, the following two cases have to be checked for EMOF-based models and require user intervention to provide meaningful values:

- `MultiplicityElement.lower > 0 and MultiplicityElement.isMultivalued()`: The lower bound of a multi-valued property is greater than 0, therefore a value must be supplied. This can not be a default value, as multivalued properties can not have a default value in EMOF, so that any transformation rule with this property on the RHS has to be adapted manually to provide a value (`isMultivalued()` is `true` when the multiplicity has an upper bound greater than one).

- `MultiplicityElement.lower > 0 and not MultiplicityElement.isMultivalued() and Property.defaultValue->isEmpty()`: The new property is not multivalued but no default value is specified so that any transformation rule using the owning class on the RHS must be adapted manually to provide a value.

For an implementation in EMF Ecore, the attribute `ETypedElement. required` fulfils the same function as setting a lower bound greater than 0. It must be checked along with the multiplicity.

Besides these restrictions for the *Add Property* operator, the addition of elements to models has no impact on ATL transformations.

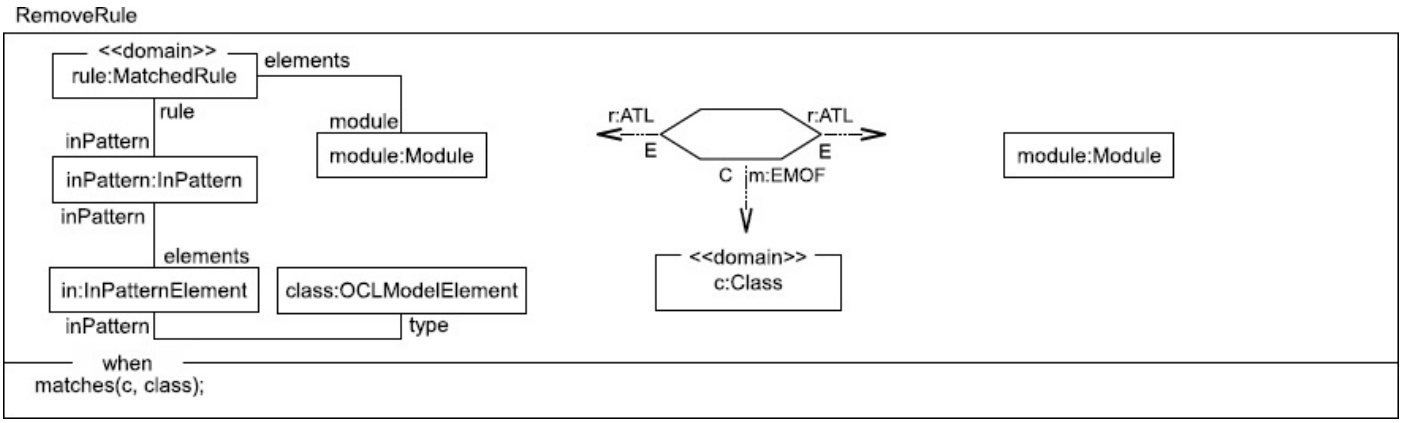


Figure 7.1.: The *Remove (LHS) Matched Rule* relation

7.4. Remove Element

An element is removed from the model. As opposed to the impact of the *Add Element* operator, we have to discern between the types of elements that are removed to determine and resolve the impact. Deleting operations, parameters, data types or packages has no influence on existing transformation rules. The deletion of classes, properties and associations is handled individually, as there are a number of possible impacts on transformation rules. These are detailed below.

7.4.1. Remove Class

The *Remove Class* operator can remove a class from a metamodel both on the LHS and RHS side of an ATL transformation. On the LHS, the class can only occur in in-patterns of rules that match the class. All the other possible occurrences of classes in transformation rules are not possible when the *Remove Class* operator is applied as it constrains the class to be empty and to not be used by any other part of the model (as the type of a property, for example). We assume that the removal of the class means that the rule that matches it can also be removed. Therefore, the impact resolution is straight-forward and automatic: all rules matching the class are removed. This is defined by the relation in Figure 7.1 .

For the RHS, all out-pattern elements that create instances of the removed class can be removed. This is achieved by the relation in Figure 7.2 .

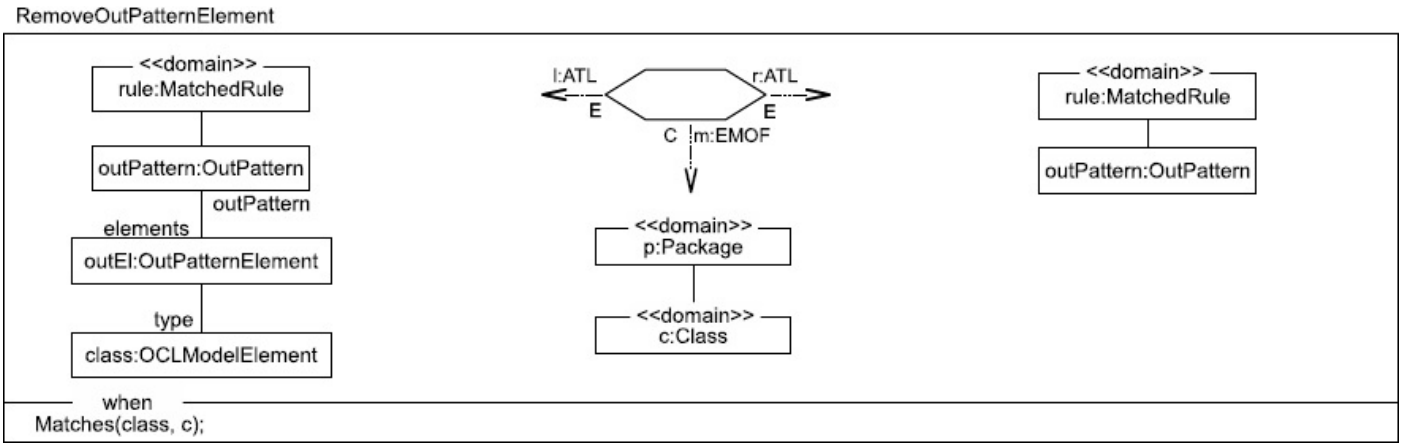


Figure 7.2.: The *Remove Out-Pattern Element* relation

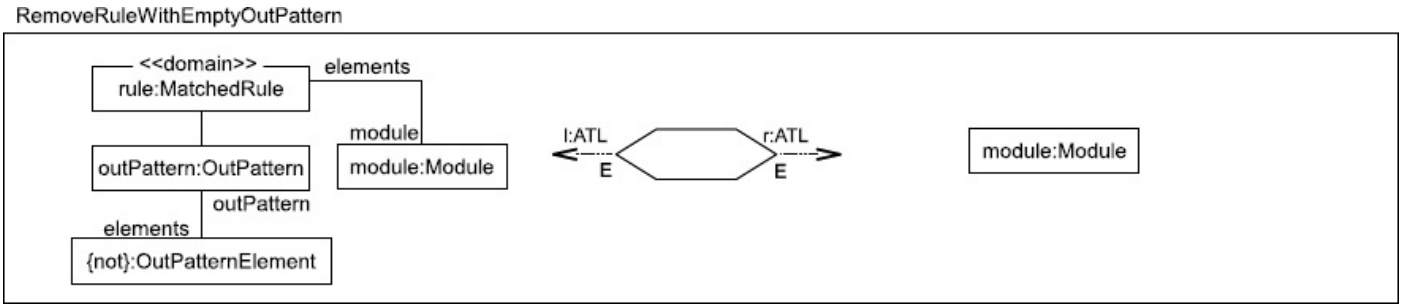


Figure 7.3.: The *Remove Rule with Empty Out-Pattern* relation

Rules can have more than one out-pattern element, so only when this results in a rule with an empty out-pattern, the whole rule can be removed, which is defined by the *RemoveRuleWithEmptyOutPattern* relation in Figure 7.3 .

7.4.2. Remove Property

A simple property is removed from a class. Complex properties are always used as part of associations which are removed using the *Remove Association* operator instead (see Section 7.4.3 for the impact resolution of the *Remove Association* operator.

In simple cases, rules that need to be adapted in response to the *Remove Property* operator either refer to the class from which the property is removed or to one of its subclasses in in- or out-patterns, or contain OCL expressions that refer to the removed property.

```
1  rule ClassAToClassZ {
2   from classA: left!ClassA
3   (
4    classA.isMatched = true
5   )
6
7   to classZ: right!ClassZ
8    (
9     name <- classA.name
10   )
11 }
```

Listing 7.1: ATL rule with constraining OCL expression

In more complex cases, the property can be used in rules that match unrelated classes and then use OCL expressions to navigate to the property to be removed. The suggested impact resolution is the same for both cases but the impact detection is more complex, as every OCL expression in every rule has to be analysed for possible OCL navigation expressions and usage of the property.

The following parts of a rule are possible candidates for the removed property to occur:

Filter Conditions (LHS)

References to the removed property can occur in filter conditions restricting the application of a rule. Listing 7.1 contains an example of an ATL rule matching an arbitrary class `ClassA` which is restricted by an OCL expression that references the arbitrary attribute `isMatched` of type `boolean` (line 4).

When the attribute `isMatched` is removed from the class `ClassA`, the expression becomes invalid. To resolve this, we propose the removal of the conflicting condition, making the rule potentially match more often. The user is warned of the possibly unintended behaviour in this case. Should the property really have been necessary for conditions such as this, the removal appears to be premature.

The relation in Figure 7.4 defines the resolution for matched rules with filter conditions containing a property affected by the *Remove Property* operator.

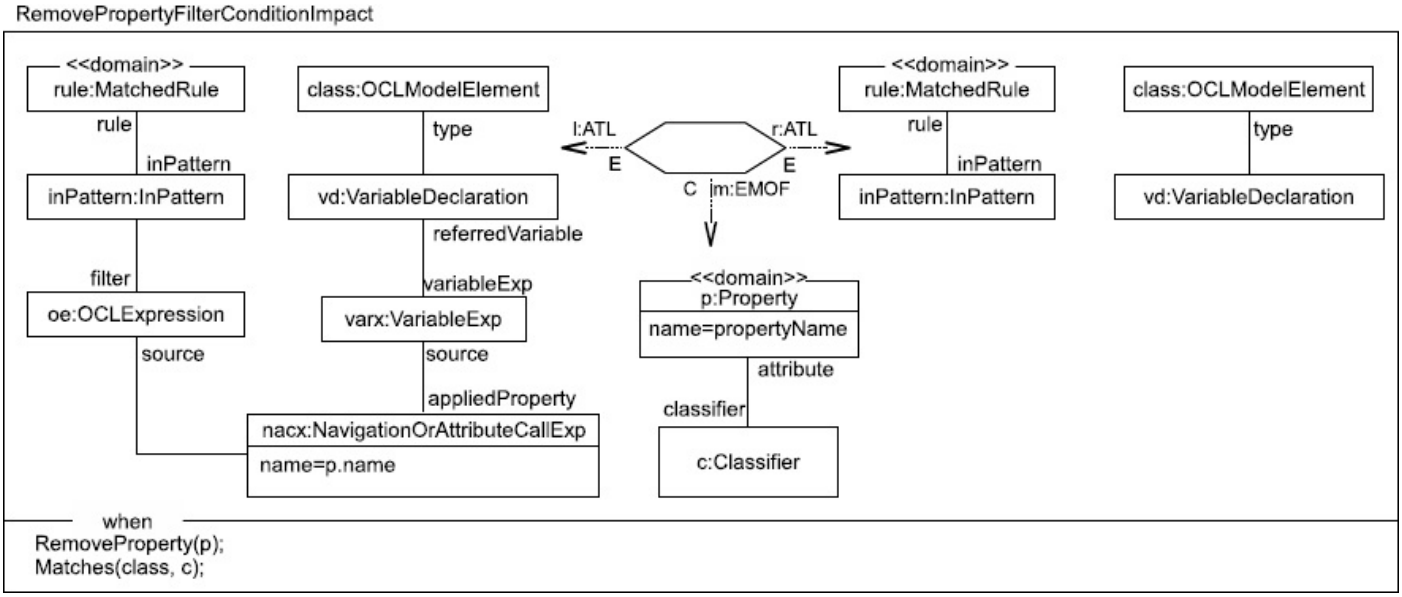


Figure 7.4.: *Remove Property* impact resolution for ATL filter conditions

The domain of the relation consists of the impacted rule and the removed property. The OCL expression restricting the application of the rule in this case is given by the `filter` reference set on the rule's `InPattern`. The filter condition can be any OCL expression evaluating to a `boolean`. When the AST of the expression contains a `NavigationOrAttributeCallExp` which navigates to the removed property, the whole filter expression is removed in the impact resolution, as given by the left side of the relation in Figure 7.4 .

The given relation only covers cases where the removed property is used directly in a `NavigationOrAttributeCallExp` for the sake of brevity. OCL expressions used as filters can be made up of any number of `OCLExpression` elements to form more complex expressions (like `if..then..else..endif`). These are referred to here in sum by the element `oe:OCLExpression`. For an implementation of the resolution strategy, the `oe:OCLExpression` AST used as a filter has to be tested recursively for any occurrence of the `Navigation OrAttributeCallExp` element referring to the removed property in the filter expression. The resolution strategy remains the same no matter how complex the filter expression: the entire filter is removed as soon as any occurrence of the removed property is found.

```
1  rule VideoToMovie {
2   from
3    video: emflib!VideoCassette
4
5   to
6    movie: imdb!Movie
7    (
8     title <- video.title
9    )
10 }
```

Listing 7.2: ATL rule with binding assignment

Binding Assignment (LHS)

For a LHS metamodel, the removed property can be part of a binding assignment (to set the value of a RHS property). For the impact resolution, we propose the removal of the whole assignment, if the RHS property is non-obligatory or if it has a default value. Syntactic correctness would be preserved. The user should be warned in this case, as a RHS property which was previously assigned a value now receives its default value instead. Should the value of the property be obligatory instead, human intervention is needed to create a new assignment statement to provide a value.

In the example in Listing 7.2, the title property of the RHS `Movie` class is bound to the title of the LHS `VideoCassette` class (in line 8). When we remove the property `title` of the class `VideoCassette`, the binding assignment has to be altered. In this example, the `title` attribute is mandatory, so the user has to intervene to provide a new value. Since the rule in its current state becomes obsolete, the best course of action for the user is to remove the entire rule.

The relation in Figure 7.5 defines the resolution for rules with a LHS binding assignment reference to a property that is removed. The domains given are the rule to be adapted and the property. The binding assignment references the property via a `NavigationOrAttributeCallExp` with the name of the property and a source of a `VariableExp` referring in turn to a variable declaration of the type of the owning class. For the resolution, the entire binding assignment is removed.

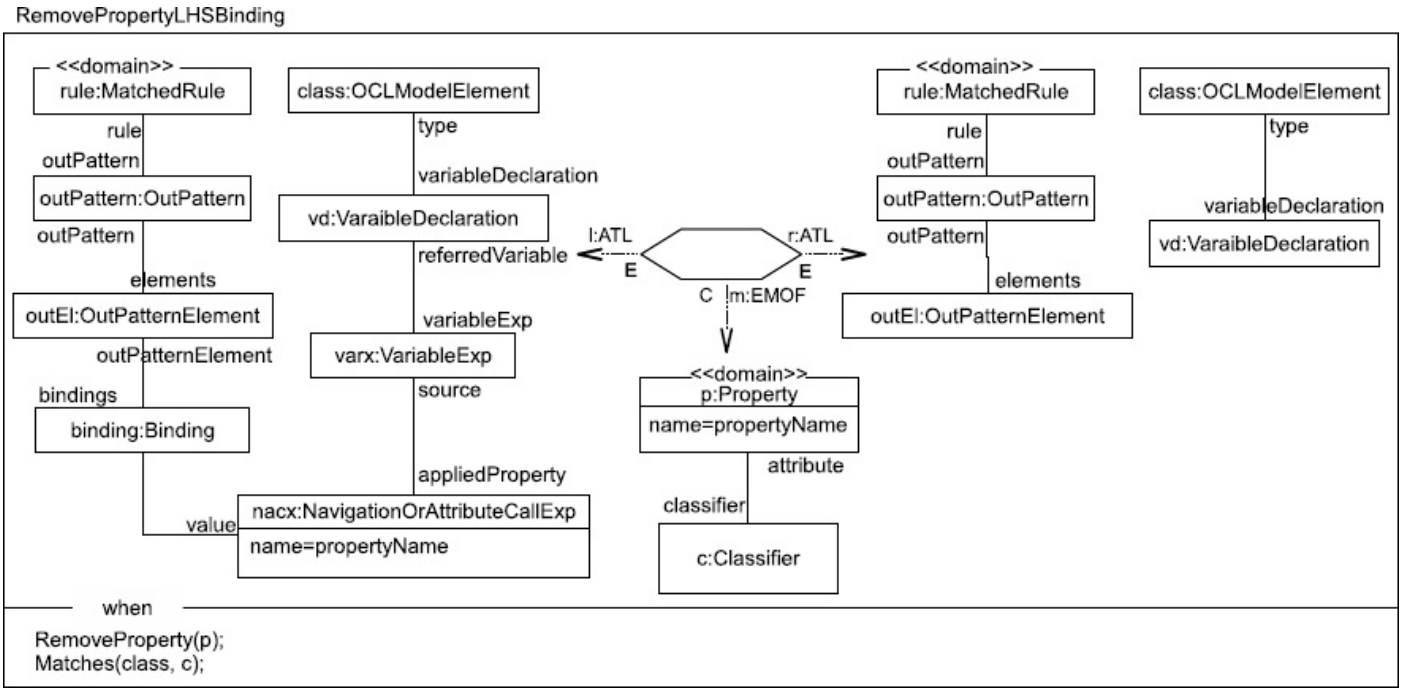


Figure 7.5.: *Remove Property* impact on LHS binding

Same as with the resolution of removed properties used in filter expressions described above, the given relation only resolves cases where the property is used directly in a `NavigationOrAttributeCallExp`. The property usage in more complex expressions is referred to in sum by the binding element `binding:Binding` for the sake of brevity. For an implementation of the resolution strategy, the binding expression AST has to be tested recursively for any occurrence of the `NavigationOrAttributeCallExp` referring to the removed property. The resolution strategy again remains the same: for any occurrence, the whole binding is removed.

Binding Assignment (RHS)

The property can have a value binding for the target pattern of the class it was removed from. The binding assignment is removed in the resolution.

For example, should the property `title` of the class `Movie` used in the rule in Listing 7.2 line 8 be removed by the *Remove Property* operator, the entire binding statement `title <- video.title` is removed in the resolution.

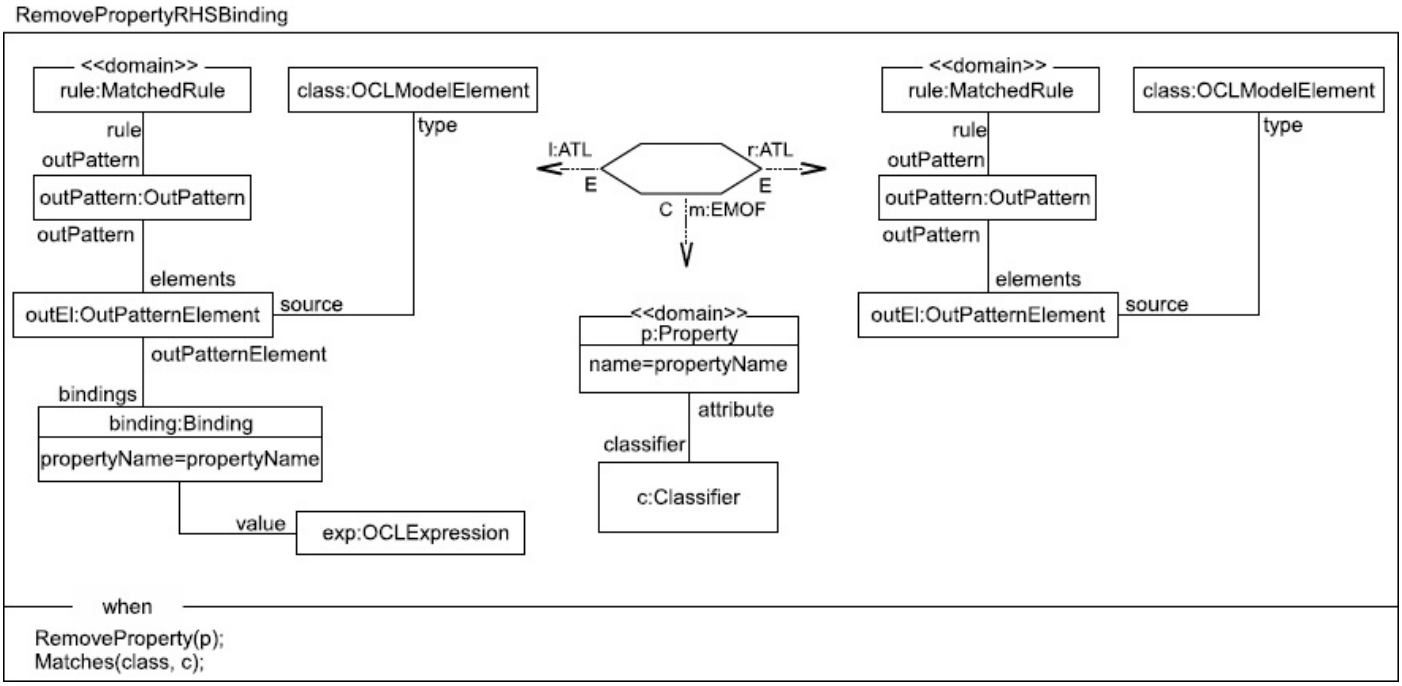


Figure 7.6.: *Remove Property* impact on RHS binding resolution

Figure 7.6 defines the resolution for rules with a RHS binding assignment reference to a property that is removed. The domains are the matched rule to be adapted and the property. The binding assignment references the property via its name and the superclass is the type of the corresponding `OutPatternElement`. For the resolution, the binding assignment and the expression that was bound are removed.

7.4.3. Remove Association

An association and the properties denoting the ends of the association (member-end properties) are removed from a package. The impact resolution can be reduced to resolving the removal of the two member-end properties, as ATL does not distinguish between the properties and the association itself. The *Remove Association* operator already delegates to *Remove Property*, the removal of the properties is handled and no further action is needed.

7.4.4. Remove Generalization

A generalization between a super- and a subclass is removed. The impact can be equated to the removal of all features of the superclass from the subclass, as these are no longer available through inheritance afterwards. (Features of classes even further up the inheritance hierarchy remain available, because the *Remove Generalization* operator introduces a generalization between the subclass and those superclasses in this case.) Therefore, the impact resolution can be delegated to that of *Remove Property* and *Remove Association* for these features (see sections 7.4.2 and 7.4.3).

7.5. Rename NamedElement

All elements with a name can potentially be renamed. In the case of EMOF, this can be any descendant of `NamedElement` (see Section 2.3.2 on page 27). In general, as names

serve as identifiers for many elements, the uniqueness of the name must be checked prior to renaming. For example, the name of an EMOF Class must be unique in the scope of its package. Hassam et al. define the operation *.allConflictingNames()* on elements to exclude name collisions in co-evolution rules for OCL constraints and metamodels [42].

The parsing and binding mechanisms of transformation languages take care of identifying model elements used in transformation descriptions and provide references to the involved metamodel elements. Name identifiers in the transformation descriptions can be changed from the old identifier to the new identifier (bar a use in comments).

7.6. Move Property

A property is moved from one class to another, with the two classes being linked by an association with a multiplicity of one-to-one. This ensures that every instance of the target class is linked to exactly one instance of the source class and the values of the property can be associated correctly with the previously owning instance. In terms of the use in transformations, references to the property are impacted and need to be redirected to the new class along the association link. This resolution can be handled automatically for the places where properties can feature in transformation rules, being binding assignments (LHS), filter conditions (LHS) and binding assignments (RHS).

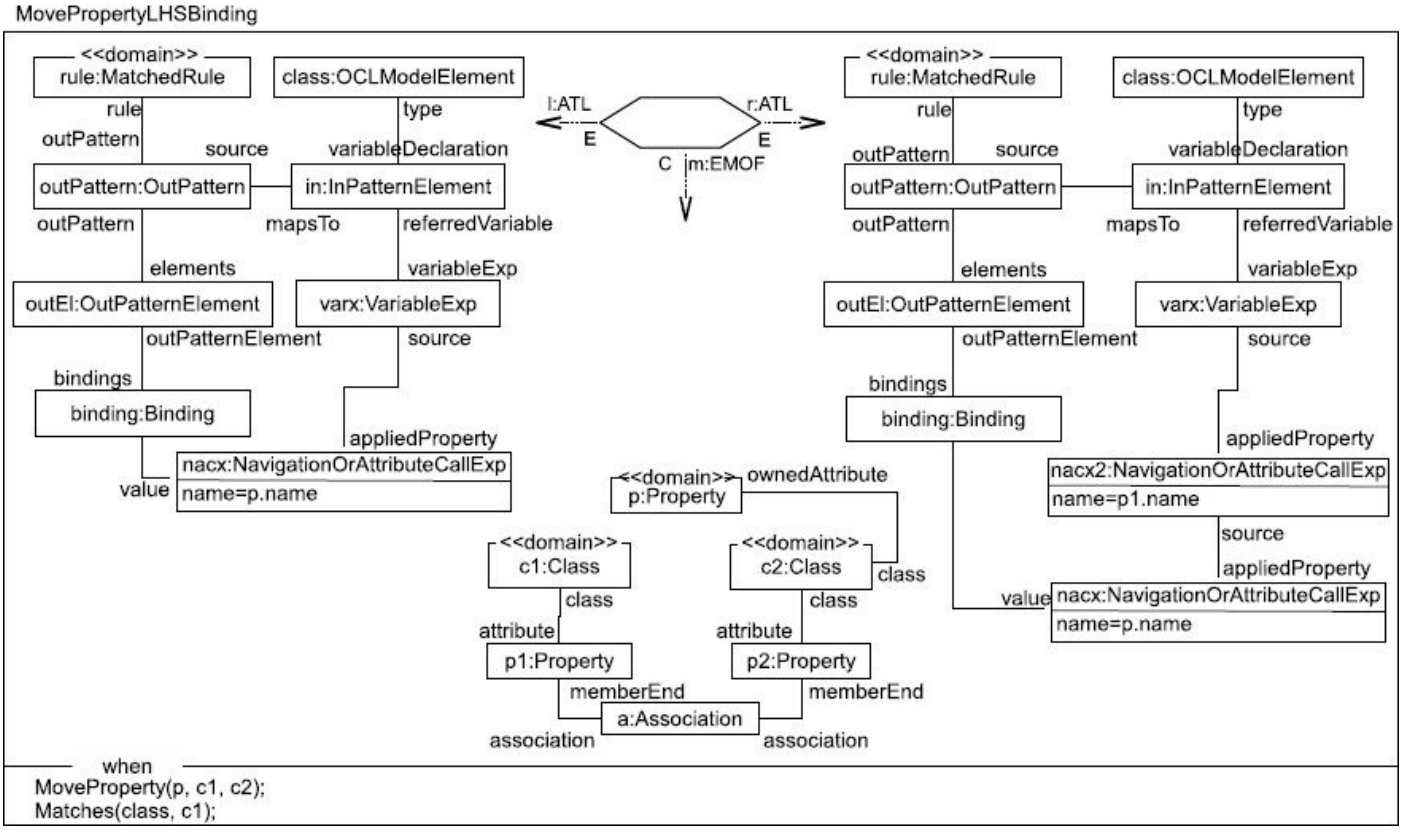


Figure 7.7.: *Move Property* LHS binding assignment resolution

7.6.1. Binding Assignment (LHS)

The relation in Figure 7.7 handles the resolution for an LHS binding assignment which makes use of the moved property. The `NavigationOrAttribute CallExp nacx` referring to the property `p` being moved is extended by a `NavigationOrAttributeCallExp nacx2`

which extends the expression to navigate along the association to the new location of the property. This is illustrated by the example in listing 7.3.

In the example, the property `city` was moved from the class `Writer` to the class `Address`, which is linked to `Writer` by the association member-end property address. The value binding for the new instance is extended from the attribute call expression `writer.city` to navigate the association between *Writer* and *Address* as `writer.address.city`.

```
1  rule WriterToPerson {
2   from writer:emflib!Writer (
3     writer.address.city = 'Santa Fe'
4    )
5
6   to person:peopleDir!Person
7    (
8     fullName <- writer.firstName + ' ' + writer.surname,
9     city <- writer.address.city
10   )
11 }
```

Listing 7.3: Example for the impact of *Move Property* on a LHS property binding

7.6.2. Filter Conditions (LHS)

The impact for filter conditions using the moved property is similar to that on a binding assignment. The rule in the example in listing 7.3 is constrained to only match authors living in Santa Fe in line 3. After the resolution, the value of the moved property *city* is fetched from the new location by navigating over the *address* association member-end property in the constraining expression.

The relation in Figure 7.8 handles the resolution for filter conditions. Again a `NavigationOrAttributeCallExp` referring to the moved property is extended to navigate across the association between the two classes.

7.6.3. Binding Assignment (RHS)

If the *Move Property* operator is used on a metamodel on the RHS of a transformation, all value bindings for the moved property have to be adjusted to reflect the new location of the property. The precondition of the operator, that the source and target classes of the move are linked by a one-to-one association, dictates that any transformation that creates an instance of one of the two classes also has to create an instance of the other and create the link between the two.

This situation is modelled by the resolution relation in Figure 7.9, where any rule with an out-pattern element `outEl` for the source class `class` also has an out-pattern element for the target class `class2`. The instances of the two classes created by the rule are linked by the value binding `binding` for the member-end property of the association between the two. For the resolution, the value binding of the property that is moved (binding2)is shifted to the out-pattern of the target class. The value of the property can be defined by any OCL expression and is moved along with the binding as exp:OCLExpression.

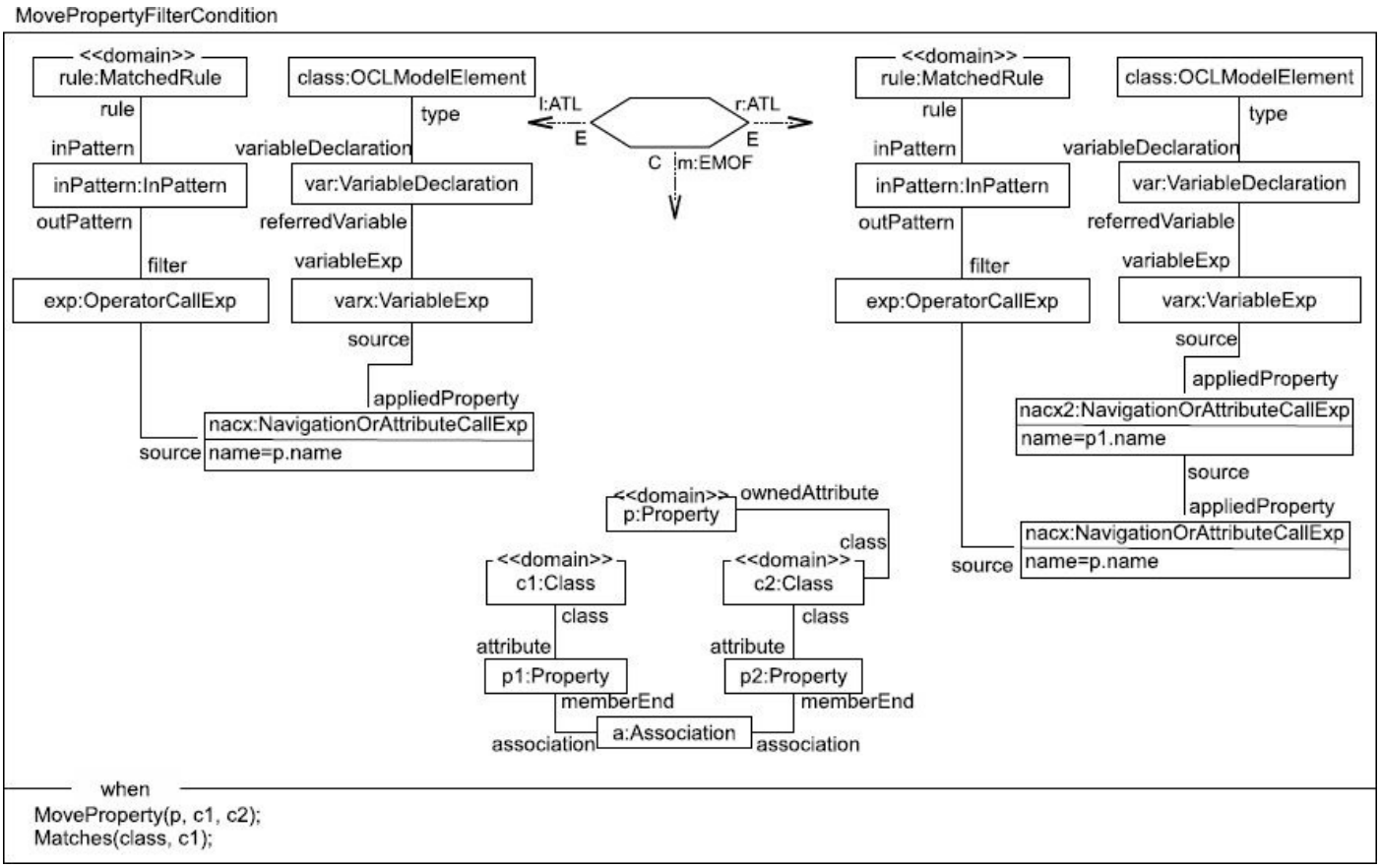


Figure 7.8.: *Move Property* filter condition impact

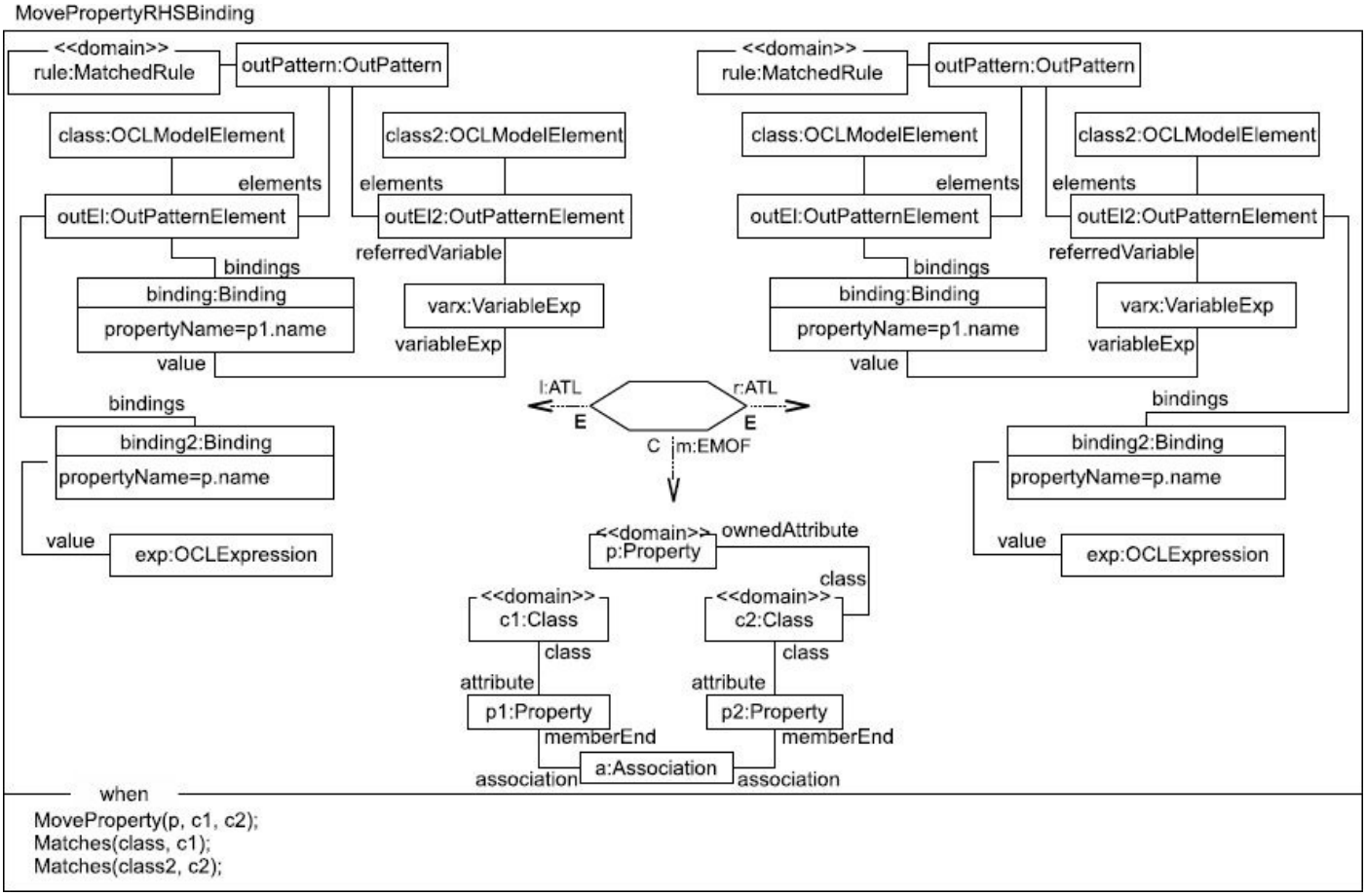


Figure 7.9.: *Move Property* RHS binding impact

```
1  rule WriterToPerson {
2   from person:peopleDir!Person
3
4   to writer:emflib!Writer (
5      surname <- person.fullName,
6     address <- address
7      ),
8
9      address:emflib!Address (
10    city <- person.city
11     )
12 }
```

Listing 7.4: Example for the impact of *Move Property* on a RHS property binding

In the example in listing 7.4, the property *city* was moved from the class *Writer* to the class *Address.* The value binding for the property is moved to the out-pattern element for the target class *Address. Writer* and *Address* are linked by the property binding for the *address* property of *Writer,* which is an expression pointing to the *address* variable. Thereby the correct instances are linked.

7.7. Push Simple Property

The *Push Simple Property* operator ‘pushes’ a property from a superclass to all its subclasses and removes it from the superclass. For its impact on transformations we need to look at all rules that make use of the pushed property either in the context of the superclass or in that of one of the subclasses.

For rules that use the property in the context of the subclasses, the impact is simple and no adaptation is needed as the property was previously available through inheritance.

For rules with a usage in the context of the superclass, we assume that the usage is no longer wanted as the property itself is no longer available in the classes’ context. The resolution can be achieved by delegating to the *Remove Property* impact resolution for the property and subclass (see Section 7.4.2). In fact, the operator itself already delegates to the *Remove Property* operator, so no further adaptation is needed (bar the alternative detailed below). In summary the resolution is:

For the use of the property in a binding assignment, the whole assignment is removed, if the RHS property is non-obligatory or if it has a default value. The user is warned and syntactic correctness would be preserved. Otherwise, human intervention is needed to create a new assignment for the obligatory value.

For the usage of the property in a filter condition, the whole constraint is removed, which makes the rule potentially match more often. The user is also warned of this behaviour. Here a more fine-grained resolution approach is possible in theory because of the nature of the *Push Property* operator, yet it is not possible for the current implementation of the ATL VM. This approach is detailed in the next section.

For the RHS of a transformation, rules can have a value binding for the pushed property. As it is no longer available, the binding assignment is removed in the resolution.

An Alternative Resolution Strategy for Filter Conditions

An alternative approach to removing the entire constraint in a filter condition which makes use of the pushed property would involve refactoring the constraint to match those instances that still have access to the property after it is pushed, being all instances of subclasses. Hassam et al. use a similar approach when refactoring OCL annotated class diagrams by extending the *PushDownProperty* operation for OCL constraints on MOF classes originally proposed by Marković [42, 65]. An expression attached to a class with a property which is pushed down in the form `oclExpression.p[RestExp]` is transformed into the expression:

```
oclExpression -> select(m| m.oclIsTypeOf(Son))-> forAll(m| m.p[RestExp])
```

Here `p` is the property, `Son` is the subclass to where the property is pushed, `oclExpression` some arbitrary leading part of the expression and `[RestExp]` the place holder for the expression part referring to `p`.

This approach would be transferable to ATL by using a type check and casting for the instances matched by the rule in question, to something like:

```
1 if not classA.oclIsTypeOf(left!ClassB) then false
2  else classA.oclAsType(left!ClassA).isMatched = true
3 endif
```

Listing 7.5: Possible constraint adaptation after *Push Down Property*

Unfortunately, ATL does not yet support the type-casting OCL operation `.oclAsType()` [5, p. 16]. So this approach, although preferable as it provides much more fine-grained resolution options, is not possible with the current design of the ATL VM.

7.8. Push Complex Property

The impact of pushing complex properties on ATL rules can be resolved in the same manner as that of simple properties for all rules that use the pushed property directly. If the pushed property is part of a bidirectional association however, rules that use the opposite property are also impacted and need resolution.

The *Push Complex Property* operator clones the association and the property on the opposite end of the association for each subclass and the cloned remote properties are renamed. For transformation rules that use the remote property, adjustments have to be made for the change in name and the changed set of instances navigable by the new associations.

7.8.1. Impact resolution for direct usage of complex properties

In the simple case where rules use the complex property in the context of the subclasses, no resolution is needed, as the properties were available through inheritance before the *Push Complex Property* operator was applied.

For use in the context of the superclass, the impact is also the same for all occurrences as that for simple properties and the resolution is the same: When used in filter conditions, in queries or in the navigation for value bindings on the LHS, or in a value assignment on the

RHS, the respective usage is removed by the applicable impact resolution relation (see 7.7 for more details). The same restrictions as for simple properties apply.

7.8.2. Impact resolution for opposite association-ends

Should the pushed property be part of a bidirectional association however, extra impact detection and resolution is required for the opposite association-end. The association and the opposite property are cloned by the operator for each subclass and the cloned properties are renamed to prevent name clashes. This is handled by the impact resolution as follows:

For any occurrence of the opposite property in OCL expressions for value calculation or navigation, the expression is adapted to cover the combined set of values of the new properties. For example, in the value binding `item <- statusReport.item` the value bound to the arbitrary property `item` is given by the attribute-call expression `statusReport.item`. As the property `item` is split up in example 6.14, the value-binding needs to be changed to `item <- statusReport.bookOnTape->union(statusReport.videoCassette).`

This makes it necessary to break down the resolution into a number of steps: one step to create a value-binding to any one of the new member-end properties (arbitrarily chosen) of the form `statusReport.bookOnTape` and subsequent steps for each further member-end property as `->union(statusReport .videoCassette).`

To assist in the chaining of these expressions, the black-box operations `Mark ()` and `isMarked()` are used (see figures 7.10 and 7.11). `Mark()` 'marks' the first and any subsequent expressions to be extendible by a `->union()` expression. As the *OppositeAssociationEndOther* relation is restricted to only apply to expressions that are marked by a call to `isMarked(),` the first unmarked expression is chosen as a chaining start and all other expressions are chained on.

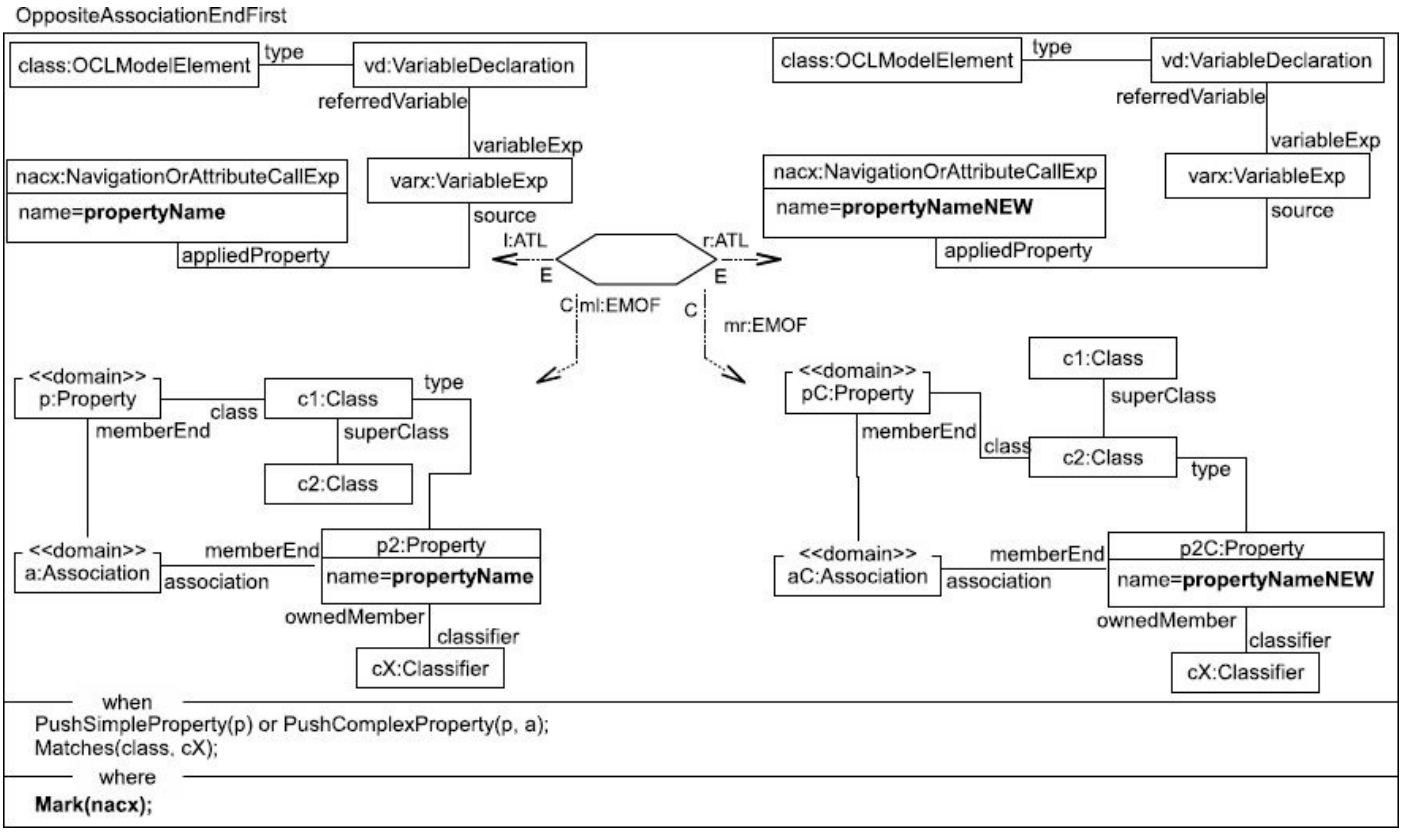


Figure 7.10.: The *Opposite Association-End First* relation

This resolution is provided by the relation in Figure 7.10 for the first step and the relation in Figure 7.11 for the subsequent steps. In the relation for the first step, the NavigationOrAttributeCallExp for the changed member-end property is changed to point to one of the (new) names of cloned properties. The marking mechanism is used to indicate that the expression can be extended by a subsequent step by a union expression. In the subsequent steps performed once for each further cloned association, the new member-end name is inserted as a NavigationOrAttributeCallExp (nacx2), as an argument of a new `union`-expression as given by the oc: CollectionOperationCallExp statement. This effectively chains together all new association-end properties into a combined `union`-statement.

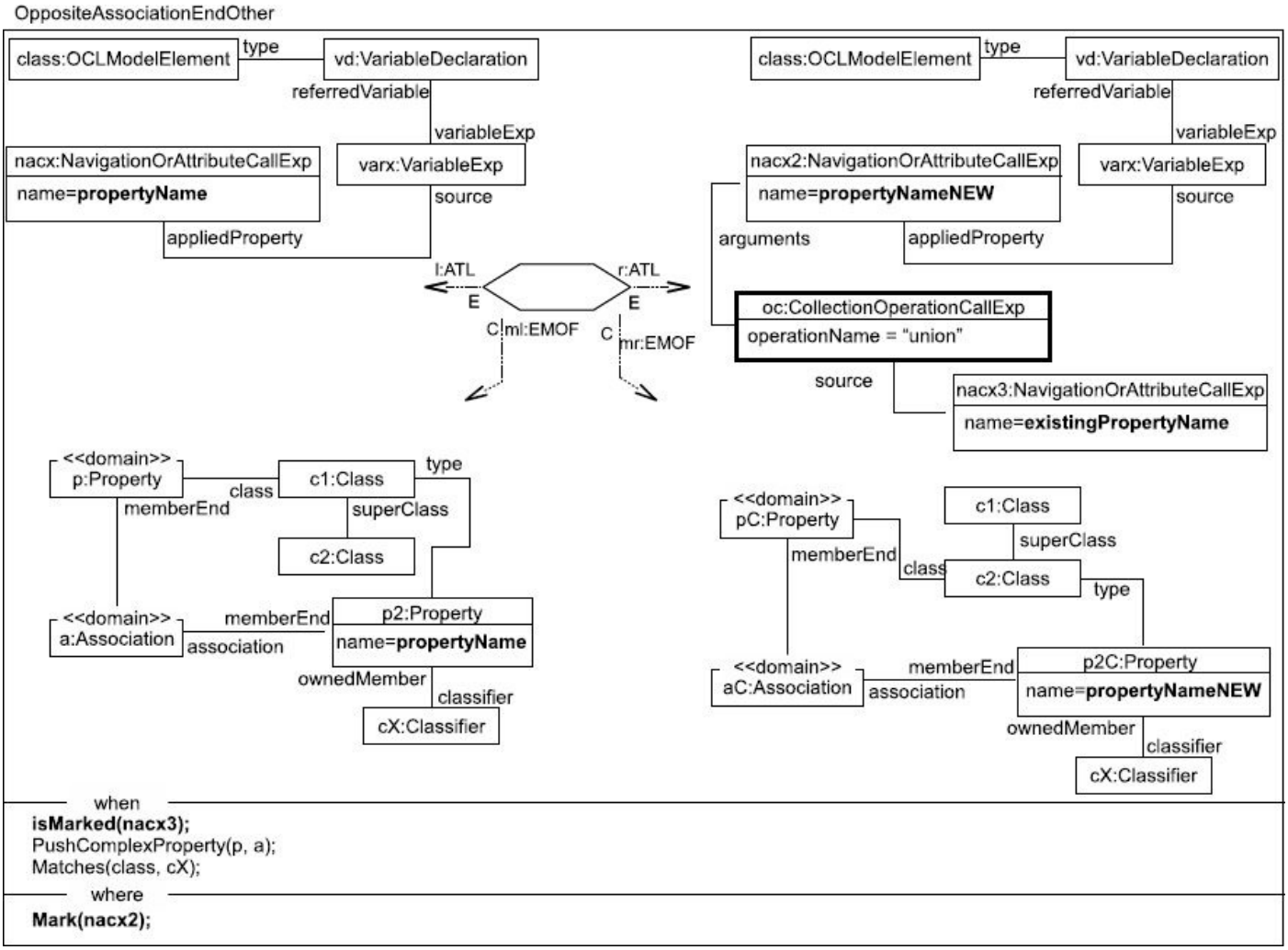


Figure 7.11.: The *Opposite Association-End Other* relation

7.9. Pull Simple Property

When a simple property is pulled from all subclasses to a common superclass, no existing ATL rules are affected. As one precondition, the *Pull Simple Property* operator of Section 6.7 ensures that all subclasses own the property to be pulled prior to execution, therefore the property was available directly before and is available now via inheritance. This means that no resolution is needed.

7.10. Pull Complex Property

As complex properties are linked to associations which are potentially bidirectional, the impact of pulling complex properties has to cover both the local property that is being pulled and the remote association member-end property.

For the local property, no change is needed, just as with resolving the *Pull Simple Property* operator described in Section 7.9.

The case of the remote property is not as simple if the association is bidirectional, as potentially different sets of instances previously available under different properties are now merged into one.

Furthermore, the type of the new combined remote property is that of the superclass to which the property is pulled, where previously the instances of each of the subclasses

where available through their own distinct property. Therefore, the user may wish to use some feature of the linked instances, to maintain the distinction. This could be achieved by type-checking for the subclasses and excluding unwanted ones or checking some tertiary property.

As an example, if the resolution for the *Pull Complex Property* operator were to fully reverse the impact resolution of the *Push Complex Property* operator (which is reasonable to expect) statements of the form `item <- statusReport.bookOnTape->union(statusReport.videoCassette)` should be combined into something like `item <- statusReport.item` (see Section 6.6).

Yet, as the make up of the sets of instances can only be determined at runtime and can differ for each model instance, no assumption can be made as to how statements using remote member-end properties should be refactored. Therefore, user intervention is needed.

7.11. Restrict (Unidirectional) Property

The impact of the *Restrict Property* operator on ATL rules is confined to places where rules make use of the property itself while the remote property can be disregarded as it is not navigable (and not owned by a class). The possible occurrences of the restricted property are the same as for the *Push Simple Property* operator: LHS filter conditions, value calculations in LHS binding assignments and targets of RHS value binding assignments.

On the LHS for both filter conditions and value calculations no resolution is required as the expressions remain syntactically valid and the reduction of the set of instances the property links is the intended outcome of restricting the property.

On the RHS, rules may exist that attempt to assign instances of the superclass to the restricted property. Here the user needs to intervene, as such a transformation rule would become invalid otherwise. This kind of impact can not be resolved automatically, as the intention can not be derived and a number of resolutions are sensible, like changing the matched type of such a rule or restricting the candidate instances using an `.oclIsTypeOf(subclass)` expression.

7.12. Generalise (Unidirectional) Property

The impact of the *Generalise Property* operator on ATL rules is confined like the *Restrict Property* operator to places where rules make use of the property itself: in LHS filter conditions, in value calculations in LHS binding assignments and as targets of RHS value binding assignments. None of these require resolution, as the set of possible value instances is extended by the operator and all old instance continue to match. After the use of the operator, the user can extend rules and employ the generalised property in its wider application.

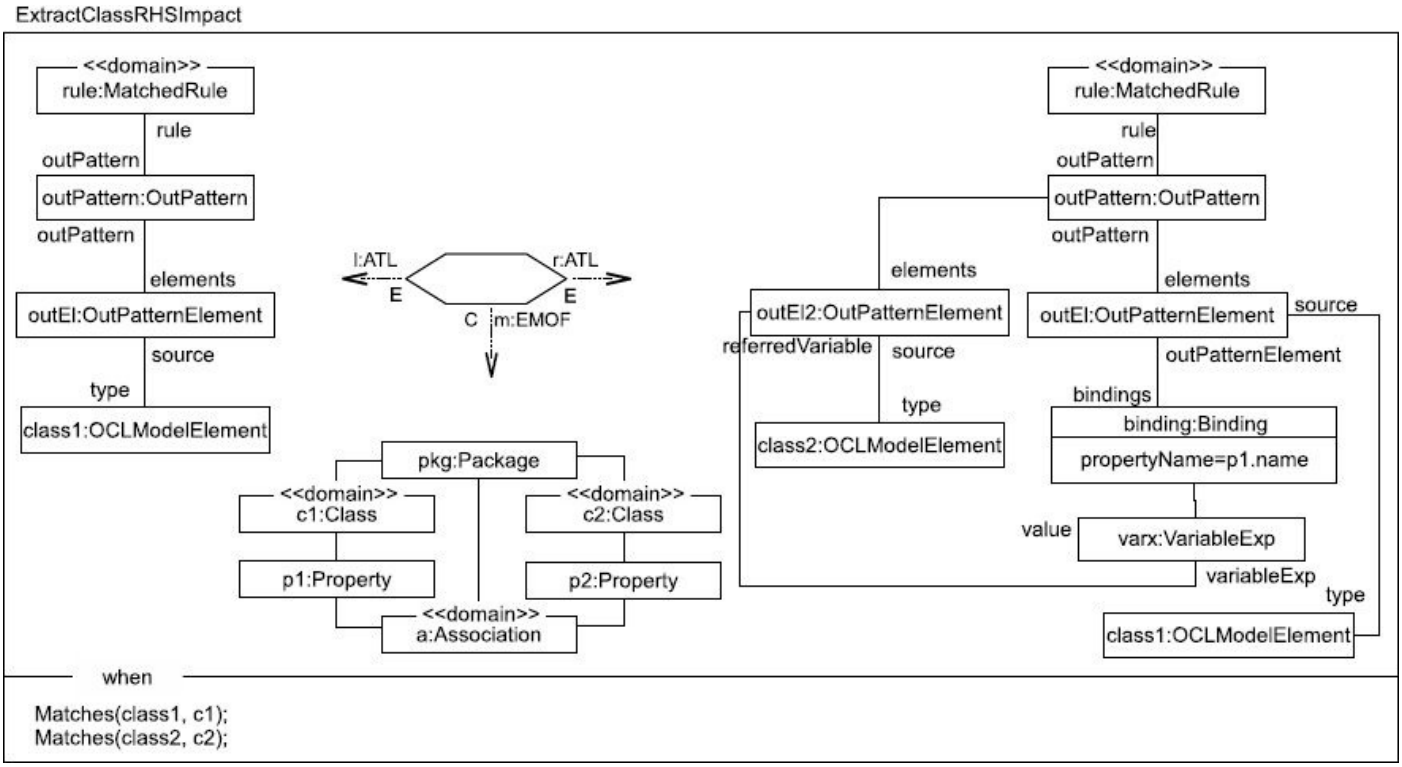


Figure 7.12.: Relation for the impact resolution of *Extract Class* on RHS metamodels

7.13. Extract Class

The impact of the *Extract Class* operator only needs to be resolved if the operator was used on the RHS metamodel of a transformation. For the LHS, all rules remain valid, as the class from which the new class is extracted does not change and is matched as before and all its properties remain accessible to be used in value bindings. Should the new, extracted class be adapted further, the impact resolution for operators like *Move Property* or *Add Element* can be handled in a subsequent step.

For the RHS, all rules which created instances of the class from which the new class is extracted need to be adapted to also create instances of the new class and to set the association link between the two. This impact resolution can be fully automated, see Figure 7.12 : For every rule with an out-pattern element matching the class c1 from which we extract, a new out-pattern element is introduced to create an instance of the new class c2, and a value binding to create the link for the new association.

```
1  rule AudioCassette2BookOnTape {
2   from cassette:inventory!AudioCassette
3
4   to bookOnTape:emflib!BookOnTape
5    (
6     title <- cassette.title,
7     damaged <- false,
8     status <- statusReport
9    ),
10
11    statusReport:emflib!StatusReport ()
12  }
```

Listing 7.6: Example for the impact resolution of *Extract Class* on a RHS metamodel

The example in listing 7.6 shows a rule after impact resolution, where a class `StatusReport` was extracted from the class `AudioCassette`. The rule now creates instances of both classes from one source instance. The association link between the two is set by the value binding on `BookOnTape`, where the property `status` is set to the variable `statusReport` of the introduced out-pattern element added in the impact resolution.

7.14. Inline Class

The impact of the *Inline Class* operator needs to be resolved both for metamodels that are used on the LHS and the RHS. Again we refer to the class that is merged as the 'merged' class and the class that remains as the 'target'.

The use of the operator effectively removes the merged class and the property member-end of the association between the two classes on the side of the target class. Therefore, the resolution has to handle the removal of a class and a property in a suitable manner.

For the LHS of a transformation and the removal of the merged class, the knowledge about the merge allows for rules that match the merged class to be diverted to match the target class instead. This resolution can be applied automatically by the relation given in Figure 7.13 .

For rules which match the target class, no change is needed as the class is not modified.

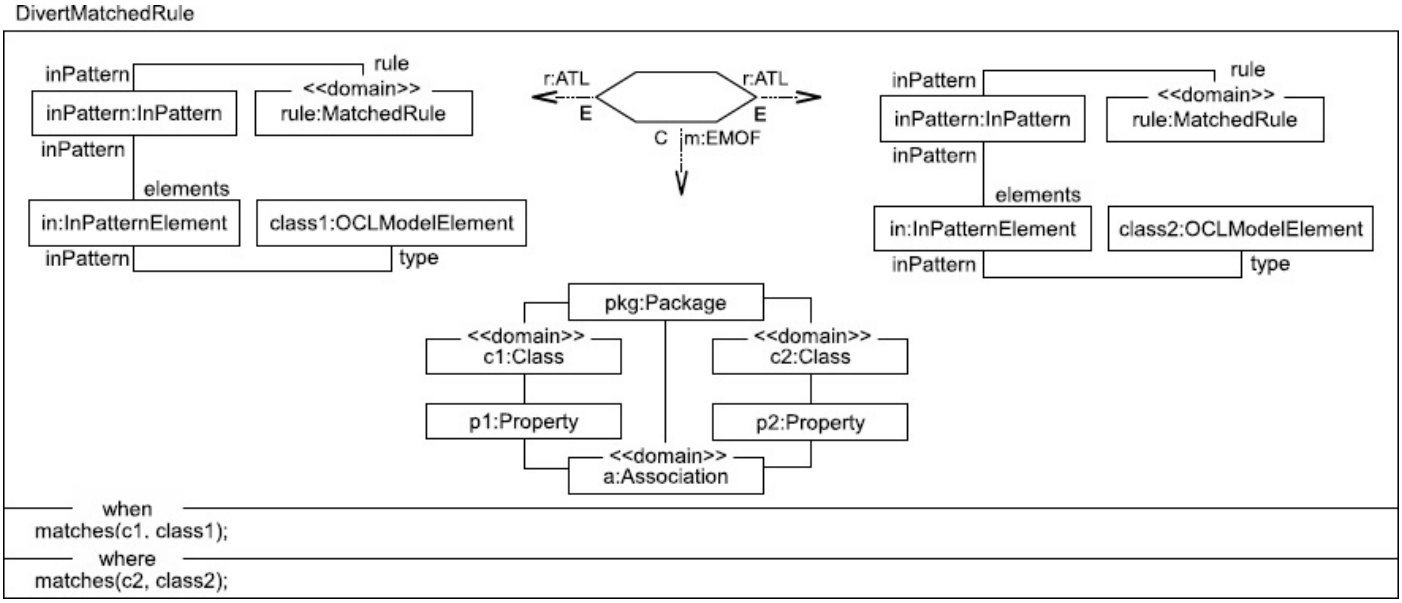


Figure 7.13.: Relation to divert a matched rule from a merged class for the *Inline Class* operator on the LHS

If the removed property is used to derive a value in a LHS value binding, user intervention is required, as some new way to derive the value is needed. This case may also indicate that the current rule and some other rule matching the merged class can be combined into a more concise version, which the user may wish to create.

For the RHS, as the multiplicity of the association between the merged and the target class must be one-to-one for the operator to be usable, every valid transformation which creates an instance of the merged class must also create an instance of the target class and set the association link between the two. This means that the rule must feature one out-pattern element for each class or refer to an instance created by another rule to link the two.

As both the merged class and the member-end property are removed, we can remove the value binding on the target class for the member-end property. We then need to remove all out-pattern elements which create instances of the merged class to cover both possible cases described above.

The removal of both the class and the properties can be achieved using relations for the general removal of these elements: the value binding is removed by the relation for the removal of properties on the RHS defined in Figure 7.6 . The removal of the out-pattern elements is handled by the *RemoveOutPatternElement* relation defined for the removal of classes, see Figure 7.2 .

7.15. Extract Superclass

A super class is extracted from one or more child classes. Common features of the children are pulled up into the new parent class. This operator has no effect on the applicability of rules, but rules can be simplified in one case: if two or more rules match two or more of the child classes and only the common features to be pulled to the parent class are used in the rules and the out-pattern of the rules match, the rules can be combined into one rule matching the new parent class instead. This behaviour is inverse to the duplication of rules in the *Flatten Hierarchy.*

7.16. Flatten Hierarchy

The *Flatten Hierarchy* operator (6.14) reduces the inheritance hierarchy between classes by pushing all properties from a parent class to all children and then removing the parent class. Although this is achieved by repeating the operators *Push Simple / Complex Property* (6.5, 6.6) and then removing the parent with *Remove Class* (6.3.3), the impact resolution for *Flatten Hierarchy* can be improved beyond the combined impact resolution of the operators involved in one case, wich is discussed in the next section.

7.16.1. Impact Resolution of Flatten Hierarchy on LHS Superclass Rules

Assuming *Flatten Hierarchy* is used on the LHS metamodel and there is a rule that applies to the superclass in the source pattern. Before the change, this rule would be executed for all instances of subclasses as well as for those of the superclass. If the rule is simply removed (as would be the impact of the *Remove Class* operator), target elements are no longer created where the rule previously matched subclass instances, although only the superclass was removed from the model. This is deemed an unexpected result.

```
1  -- Transform library items to catalogue entries:
2  module transformation;
3  create OUT: inventory from IN: emflib;
4
5  rule CirculatingItem2CatalogueEntry {
6   from item:emflib!CirculatingItem
7
8   to catalougedItem:inventory!CatalogueEntry
9    (
10    title <- item.title
11   )
12 }
```

Listing 7.7: Example for the impact of *Flatten Hierarchy* on the LHS (before)

We propose the following resolution instead: With the extra knowledge available when the *Flatten Hierarchy* operator is used explicitly, a copy of the rule that matches the superclass is created for each subclass and its source pattern is changed to match that subclass. Then the original rule for the superclass can be removed.

The resolution is illustrated by the following example. Before the operator is applied, the rule in listing 7.7 matches the superclass `CirculatingItem` to create elements of the type `CatalogueEntry`.

When the class `CirculatingItem` is now removed to flatten the hierarchy (as seen in Figure 6.23), the rule needs to be duplicated and adapted to continue to create `CatalogueEntrys` for all instances of the subclasses `Book` and `VideoCassette`. The resulting transformation is given in listing 7.8.

This resolution for superclass rules is given by the relation is Figure 7.14 . For every rule that matches the superclass `c` in an in-pattern, a new rule `(rule2)` is created, which matches the subclass `c2` instead. The cloning of the rule is performed by the relation in Figure 7.15 . This relation is defined between elements of the `ATL` domain on both sides, i.e. it is a helper-relation that solely modifies ATL statements without any reference to EMOF. It creates a deep copy of a matched-rule by recursively cloning the entire expression tree of the rule. In the overall process, two black-box operations are used. The `CloneSubTree()` operation creates a clone of an expression, including all contained expressions, i.e. a clone of the whole AST sub-tree of the expression. The `SpecialiseMatchedRule()` operation changes the in-pattern element of a rule to match another class. In this manner, the cloned rules are assigned to match the subclasses in the flattened hierarchy.

```
1  -- Transform library items to catalogue entries:
2  module transformation;
3  create OUT: inventory from IN: emflib;
4
5  rule Book2CatalogueEntry {
6   from item:emflib!Book
7
8   to catalougedItem:inventory!CatalogueEntry
9    (
10    title <- item.title
11   )
12 }
13
14 rule VideoCassette2CatalogueEntry {
15  from item:emflib!VideoCassette
16
17  to catalougedItem:inventory!CatalogueEntry
18   (
19    title <- item.title
20   )
21 }
```

Listing 7.8: Example for the impact of *Flatten Hierarchy* on the LHS (after)

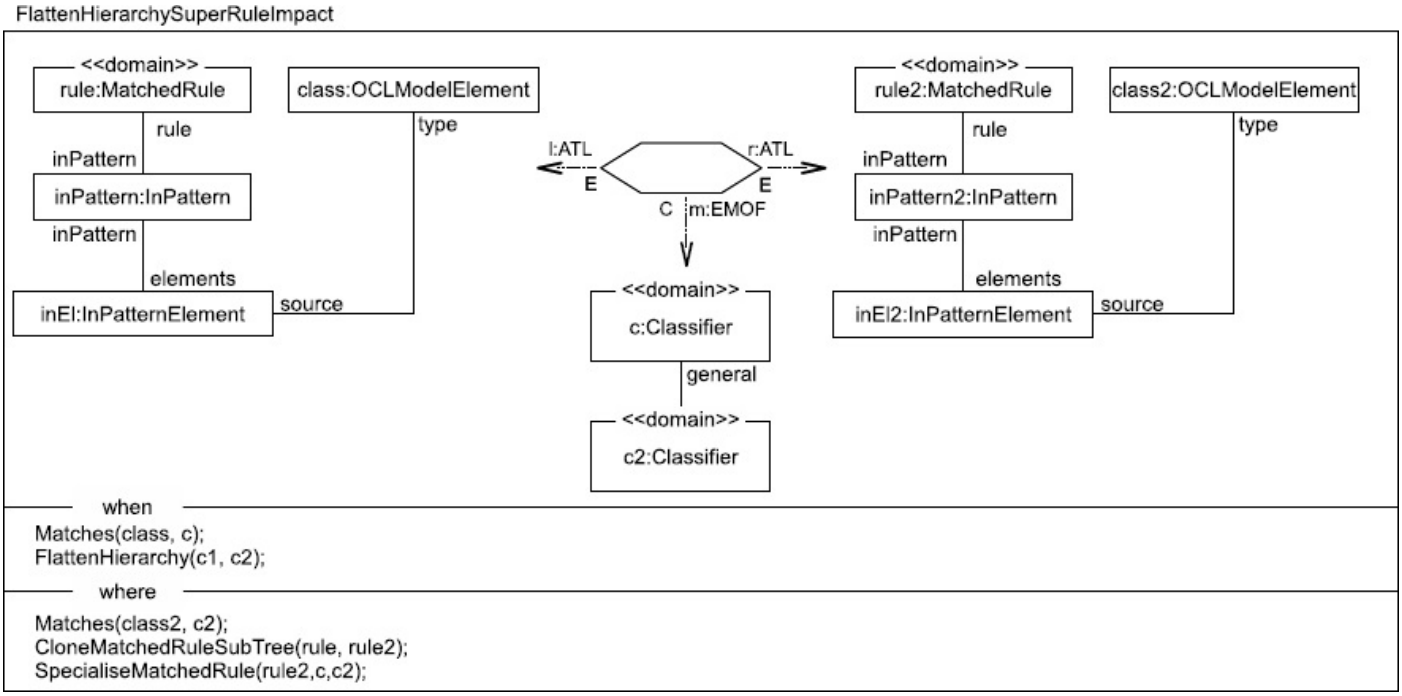


Figure 7.14.: Relation for the impact of *Flatten Hierarchy* on superclass rules

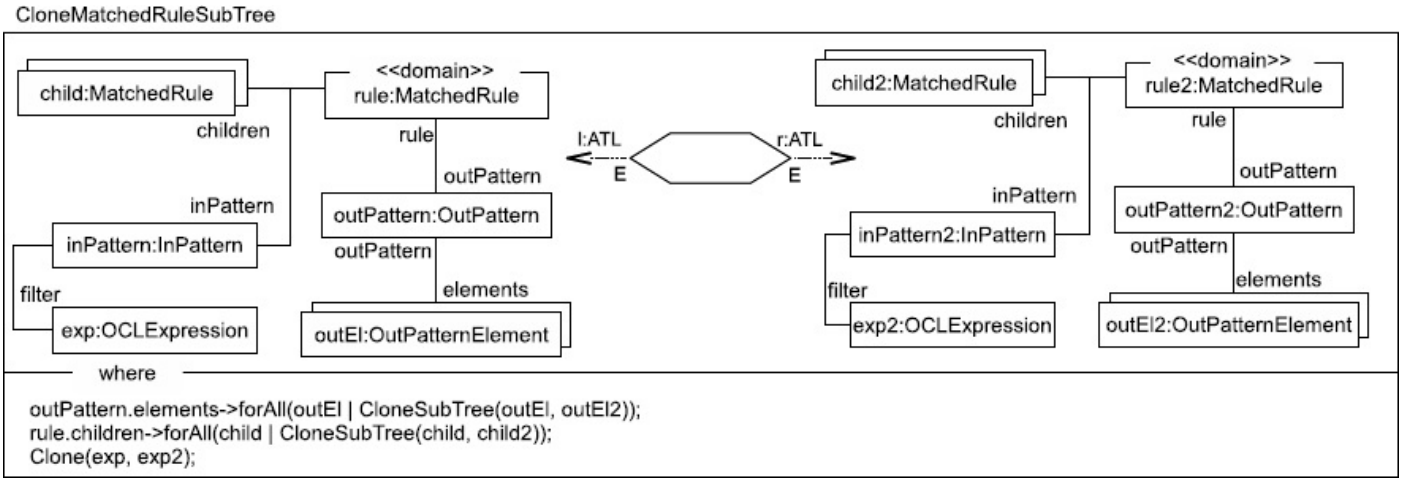


Figure 7.15.: Relation to clone a matched-rule and all child statements

7.16.2. Impact Resolution of Flatten Hierarchy for Other Cases

Rules that use the subclass are not impacted, neither for the LHS nor the RHS. The subclasses only receive features that were available previously through inheritance and that are not changed otherwise.

The final possible case for impact occurs on the RHS, where rules may create instances of the superclass. Here the resolution of the *Remove Class* operator can be applied, which removes the out-pattern element for that class and further removes rules that have empty out-patterns as a consequence (see Section 7.4.1).

7.17. Association to Class

The *Association to Class* operator replaces an association between two classes by a class connecting the two classes instead.

On the LHS, the operator impacts all parts of a transformation rule where selection or navigation over the replaced association occurs. This is expressed in statements that use the properties that served as member-ends of the association. As the association is replaced by a class that is connected to the original ends of the association by two new associations, the navigation or selection statements need to be extended by an extra step across the new class. The resulting Sequence needs to be flattened to receive the same Sequence as before. This is achieved by using the OCL `collect()` operation. An expression like `borrower.borrowed` would be modified to `borrower.borrowed-> collect(l | l.lendable)` (assuming that the property borrowed identifies the association that was replaced in the type of the variable `borrower)`. This resolution can be performed automatically both for LHS binding assignments and filter conditions.

For the impact resolution on metamodels used on the RHS, whenever an instance of the remote side of the replaced association was created, now an instance of the replacement class has to be created also.

7.18. Generalization to Composition

For the application of the *Generalization to Composition* operator on the LHS, we discern between rules that apply to the parent and those that apply to the child class.

Rules that apply to the parent class before the use of the operator do not need to be adapted as the set of features in the parent class and the number of instances remain the same after the operator usage.

For rules that apply to the child class, references to properties of the former parent class need to be redirected using the new association to point to the associated instance of the former parent class.

For the RHS, rules creating parent classes can remain unchanged. Rules which previously created child classes are changed to create a parent class for all parent properties and an associated instance of the new class for all properties that previously belonged to the child class.

7.19. Introduce Composite Pattern

The impact resolution for the use of the *Introduce Composite Pattern* operator on LHS models has to differentiate between rules that make use of the parent-child association and rules that don't. For the latter case, rules can remain unchanged as for both of the newly introduced classes *(leaf* and *composite)* all properties and relations remain available through inheritance.

For rules that make use of the association, we suggest the following resolution: The rule is duplicated and each copy is specialized to match a specific subclass *(leaf* or *composite)*. Then the association usage is removed from the rule matching the *leaf*, if possible. This is not the case if it is bound to a required property on the RHS and user intervention is needed.

For the RHS, all rules that create instances of the target class prior to the operator usage require resolution, as the class is made abstract by the operator and a decision has to made of whether to create an instance of the new *leaf* or *composite* classes. This 'nature' of the instances was previously determined at runtime: if an instance has children it acts like a *composite*, if not, like a *leaf*. This information has to be made explicitly at design time after the operator usage, so the user needs to provide further information.

CHAPTER 8

Related Work

In general, any co-evolution solution for two linked or dependent types of artefacts of the MDS can be considered related to the approach presented in this work. We found three general categories of approaches according to the type of dependent artefacts that are addressed: Co-evolution of metamodels and models, co-evolution of models and associated OCL constraints and approaches that like ours, deal with the co-evolution of metamodels and transformations. We discuss the three categories below, indicating their relevance for our approach. Section 8.4 provides an overall comparison of the related approaches.

8.1. Co-Evolution of Metamodels and Models

A variety of approaches in scientific research on MDE address the coevolution problem between metamodels and models. The aim is to reconcile broken models with changed metamodels with a varying focus on automation. Most are concerned with bulk adaptation to cater to the evolution of large numbers of models for the same metamodel. Those approaches that are operation based or that provide sets of complex adaptation or refactoring steps are of particular interest as they are closest to our approach. Of these, Herrmannsdoerfer et al. proposes the most comprehensive approach to our knowledge, providing a catalogue of 61 operators and tool support for the co-evolution of Ecore metamodels and models [44]. Another closely related approach is given by Cicchetti et al. [22], as they provide a classification for 16 metamodel modifications according to whether they can be resolved automatically or require human intervention in terms of the impact on dependent models. The approach by Wachsmuth is deemed closest as he proposes a stepwise approach and uses an QVT-like syntax in the operator formalisation [107]. Some of the operators introduced in chapter 6 are adapted from Wachsmuths work on metamodel and model co-evolution.

Wachsmuth proposes a manual, stepwise approach of metamodels and dependent models to handle co-evolution. The approach consists of a set of 16 adaptations for which the semantic and instance-preservation properties are discussed. The adaptations are taken from work on refactoring and grammarware engineering. The adaptations are provided in an QVT-R-like syntax and natural language for the metamodel part. For the model part, the impact of the adaptations is given as a ‘transformation pattern’, being a partial transformation that is to be instantiated for the concrete case. [107]

We base a number of our operators on Wachsmuth’s adaptations, modifying them to co-evolving transformations and formalizing them for EMOF using standard QVT-R graphical syntax. Section 6.2 provides an overview of our operators and their sources.

Cicchetti et al. provide a set of 16 common and complex modifications performed on metamodels and analyse the impact on dependent models. The changes are classified according to a scheme, establishing the class of 1) non-breaking changes, requiring no co-adaptation of dependent models; 2) breaking but resolvable changes for which the

conformance relation can be re-established automatically, and 3) breaking and unresolvable changes for which user intervention is required for adaptation. [21, 22]

They further propose the use of a model difference calculation to determine which of the changes has taken place. These changes are recorded using a difference model, which expresses for each metamodel entity, involved in a change, whether it was added, removed or modified. Each of these adaptations can be categorised as either *breaking resolvable* or *breaking unresolvable*. They propose the generation of ATL model transformations for the changes in both categories to automate the adaptation of dependent models. The generated transformations for the breaking unresolvable category are only 'stubs' to be extend by hand by the user. [23]

Herrmannsdoerfer et al. propose an operator- and recording-based approach for the co-evolution of metamodels and models. They provide an editor to adapt a metamodel using pre-defined operators and record the operator usage and resulting changes. The impact of each operator on conforming models is pre-defined and deterministic. In this manner, any number of conforming models can be adapted automatically. Since the nature of the co-evolution may be domain specific and is not pre-determinable in general, users can define new operators for specific cases. An example is the addition of properties with required values. Models have to provide values to be syntactically conformant and the values depend on the owning instance and the domain to be also semantically valid. The operator for this case could contain an algorithm to derive semantically valid values for all encountered cases, thus being domain specific in nature. The success of such an operator-based approach is dependent on the used library of (reusable) operators. [44]

The catalogue of operators is comprised of 61 operators for metamodel and model co-evolution and covers all our operators except for the *Introduce Composite Pattern* operator which reproduces the 'introduce composite' refactoring from OO refactoring work [34]. The operators are neither formalized for metamodels nor for models as such but implemented instead in tooling for Ecore. The metamodel formalism used is EMOF-like. The tooling is designed to be extendible so that domain specific, custom operators can be created to co-evolve models. [44, 45]

8.2. Co-Evolution of Models and OCL Constraints

The area of co-evolution of models and OCL constraints provides approaches for the validation and adaptation of OCL expressions linked to changing models (or metamodels). While Correa and Werner [25] and Cabot and Teniente [20] focus on the refactoring of OCL constraints in isolation, the relationship between a model and attached constraints can be treated as a co-evolution problem: when the model is changed the constraints need to be adapted accordingly to remain correct and useful [42, 65]. OCL constraints are text-based and refer to model elements and can be interpreted in the context of the elements (see Section 3.4).

The co-evolution problem for OCL constraints is very similar to that for model transformations, as:

1. OCL expressions and model transformations are textual statements that are attached to primarily graphical models,

2. OCL is declarative in nature, like the largest parts of the transformation languages ATL and QVT-R and

3. ATL and QVT both make use of embedded OCL expressions for a number of purposes so that OCL expressions inside the transformations need to be considered along with the transformations.

The impact of metamodel changes on the OCL concrete syntax in the context of model transformations is looked at in depth in chapter 5.6.2.

Marković and Barr present a set of refactoring rules for UML 1.5 Class Diagrams and linked OCL 2.0 expressions. The set contains 15 rules, where the three *push down* rules are only specified for cases with one subclass. The refactoring rules are provided as transformations in a QVT-R-like graphical syntax. The impact of the refactoring rules on the OCL expressions embedded into the UML diagrams is also provided in the QVT-R-like graphical syntax and discussed in natural language. [65]

Hassam et al. propose an assistance system for the co-evolution of OCL constraints and metamodels. The system is comprised of 19 operations of which 7 are defined by Hassam et al. using a QVT-R-like graphical syntax both for the evolution of the metamodel and the co-evolution of the linked OCL expression. [42]

A comparison of the features of the two OCL co-evolution approaches with the metamodel and model co-evolution approaches is provided in Table 8.1 below.

8.3. Co-Evolution of of Metamodels and Transformations

An overview of the related work for metamodel and transformation coevolution is given in Table 8.2.

Garcés et al. propose an approach that uses external transformation composition to tackle the problem of metamodel and transformation co-evolution [36, 37]. Instead of adapting existing transformations in response to a metamodel change, they semi-automatically generate a set of new transformations that are chained to the original transformation to produce a result in accordance with the metamodel evolution. The original transformation remains unchanged. The metamodel changes are not specified by the user but discovered semi-automatically. Their approach is based on EMF metamodels and ATL transformations. It covers a set of 10 change operators that are not explicitly specified.

Levendovszky et al. propose a co-evolution approach suitable for the coevolution of Domain Specific Language (DSL)s and model interpreters, which take the form of model transformations [62]. The transformations and the models and metamodels involved are based on the Graph Rewriting and Transformation (GReAT) language and tooling framework [8]. The changes made to the DSL metamodel are expected to be made available in a proprietary model which also servers as a basis for the co-evolution of dependent models. All changes made are captured by one such change model and executed in bulk to adapt dependent transformations. The approach covers fully automated and semi-automated changes but not those that are considered ‘fully semantic’. An example of a fully semantic change is the addition of a new model element, as the intended transformation target on the RHS is unknown. They discuss a set of 10 possible

metamodel changes of which 8 are addressed by their approach. The impact of the changes and the resolutions are not formalized but provided programmatically as part of the GReAT toolset.

Méndez et al. outline an operator-based approach for transformation migration [66]. It is not a fully implemented approach so that some details are not available. The set of ten available operators and the possible resolution for dependent ATL metamodels are described using natural language.

Whether the change operators are detected or their application recorded is not specified.

García and Díaz provide an approach based on the detection of simple and complex operators by model differencing [38]. The detection process occurs in two steps: first, simple metamodel changes are detected using a model differencing framework. In the second step, complex changes are detected on the basis of the simple changes and expressed as operators. The operators are then used to semi-automatically adapt dependent ATL model transformations in bulk. The impact of the detected complex operators is described using semi formal predicates and implemented using ATL transformations that take the impacted transformation and the detected operator as input and produce an updated transformation as output. Cases that require human intervention either lead to a warning during the process or leave the resulting transformation incomplete so that the user can perform manual changes afterwards.

Di Ruscio et al. propose an approach to perform metamodel and transformation co-evolution with a focus on predicting the significance of a metamodel change in terms of its impact on dependent model transformations before the change is performed [84]. The approach is demonstrated for EMF metamodels and ATL transformations. Like Levendovszky et al. they require the changes to be performed on a metamodel to be available in a suitable format before initiating the impact analysis. The suitability of a metamodel evolution is determined by calculating the cost of each evolution step in terms of the impact on a set of dependent transformations. The cost function considers a weighted sum of impact resolutions according whether they can be performed automatically, semi-automatically or solely by hand. They suggest the use of their *EMFMigrate* tool to perform the bulk adaptation of impacted transformations if the effort is deemed appropriate by the user. EMFMigrate uses migration rules to described changes performed on model transformations which can be customised by the user to perform manual adaptation.

8.4. Comparison

The features of the operator-based co-evolution approaches for models and OCL constraints and metamodels and models are summarized in Table 8.1 below. The following aspects were regarded:

- **metamodel** The metamodel column indicates which metamodelling language the operators of the approach are designed for. They can be applied to models conforming to these metamodels.

- **target artefact** This column indicates the types of artefacts that are to be co-evolved

in the approach.

- **formalism** The formalism column contains the language the operators were described in.
- **nr. of operators** This is the number of operator available in the approach.
- **automation** This column indicates whether or not the approach is aimed at automating the co-evolution task.
- **impact description** This column indicates how the impact of the operator on the dependent artefact is formalised.

For the operator-based approaches, we see a valuable contribution in our work in the formalisation of both the operator and the impact in QVT-R and the complete support of the EMOF metamodel. We expect that our stepwise approach can well be combined with approaches for other target elements, so that an overall co-evolution approach may be possible, using a single set of operators. We deem this to be future work (see Section 11.3).

Table 8.2 provides a comparison of the available approaches for the coevolution of metamodels and model transformations. The following aspects are detailed:

- **source/target** This column lists the side of the transformation the metamodel change can be addressed on. ‘S’ indicates source (LHS) and ‘T’ target (RHS) metamodels; ‘S & T’ both.
- **requires heuristics** Whether or not the approach depends on heuristics to determine the difference between the evolutionary states.
- **supported change types** This is the number of different types of changes are supported by the approach. For operator-based approaches, this is the number of operators.
- **tech. space** This column lists the technological space the approach is build for; i.e. which transformation framework and language is supported in the co-evolution.
- **change description** How are the changes that an operator makes formalised?
- **impact description** How are the impacts an operator has on a model transformation formalised?
- **mode of execution** This column indicates whether the approach is designed for the bulk adaptation of transformations after a number of changes occurred or whether a stepwise adaptation is intended.

For the related co-evolution approaches for model transformation, the greatest difference between our approach and the others lies in our focus on providing a stepwise method so that co-evolution and development activities can be undertaken closely together. A co-

evolution modification can be performed and all transformations updated to be consistent, after which forwardengineering tasks can follow immediately. We support this approach by demonstrating the close integration with common MDE tooling. Furthermore, our approach does not rely on heuristics and provides formalisation of both the operators and the impact on ATL transformations. We expect that a description of operators and impacts that is as precise as possible to be necessary, to accurately predict the behaviour of both and to be able to provide an unambiguous implementation. Furthermore, to our knowledge, our approach is the only one to support the complete EMOF metamodel.

approach	metamodel	target artefact	formalism	nr. of operators	auto-mation	impact description
Cicchetti et al.	KM3	models	own model	16	yes	ATL
Hassam etl al.	partial MOF	OCL expr.	QVT-R-like	7	yes	QVT-R-like
Herrmannsdoerfer et al.	own EMOF	models	none	61	yes	none
Marković & Barr	UML 1.5 Class	OCL expr.	QVT-R-like	~15	no	QVT-R-like
Wachsmuth	partial MOF	models	QVT-R-like	16	no	QVT-R-like
ours	EMOF	transformations	QVT-R	30	yes	QVT-R

Table 8.1.: Comparison of operator-based approaches for co-evolution

approach	source/target	requires heurist-ics	supported change types	tech. space	change descrip-tion	impact descrip-tion	mode of execu-tion
Di Ruscio et al.	S & T	n.a.	n.a.	EMF ATL	n.a.	own DSL	bulk
Garcés et al.	S & T	yes	10	EMF ATL	model diff.	transforma-tion	bulk
García and Díaz	S & T	yes	23	EMF ATL	model diff.	ATL	bulk
Levendovszky et al.	S	n.a.	8(10)	GReAT	own DSL	programmatic	bulk
Méndez et al.	S & T	n.a.	10	EMF ATL	operator	natural lang.	?
ours	S & T	no	30	EMF ATL	QVT-R	QVT-R	stepwise

Table 8.2.: Comparison of approaches for metamodel and transformation co-evolution

PART III

Validation and Conclusion

CHAPTER 9

Evaluation

This chapter covers the evaluation of the co-evolution approach and the impact resolution for ATL transformations as presented in chapter 5.4. Section 9.1 discusses aspects of completeness of the operators both on the metamodel and the transformation level. In Section 9.2 the use of operators in a co-evaluation scenario containing a stepwise metamodel evolution and the resolution of impacts on a dependent transformation is presented. Section 9.3 summarises the chapter. A proof of concept prototype of integrated tool support for the approach is discussed in detail in the next chapter.

9.1. Completeness

In terms of metamodel evolution, a set of operators is complete if any source metamodel can be evolved into any target metamodel [45]. This level of completeness is achieved by the operators presented in this thesis for EMOF metamodels. It can be demonstrated by reducing any source metamodel using the *Remove Element* operators until it is empty and then rebuilding it using the corresponding *Add Element* operators of Section 6.3:

We begin by removing all associations between classes from any given metamodel. Next, all properties and operations are removed and the generalisation hierarchy can be discarded. Then all types are removed, including enumerations and enumeration literals, data types and the classes. Finally, the empty packages can be removed and the metamodel is empty. The metamodel can be rebuild using the inverse approach and the *Add Element* operators: the packages are created first, then all types, the inheritance hierarchy and finally properties and associations.

Completeness on the side of the transformations is very difficult to achieve and possibly impractical. Here completeness would mean that any valid evolution of a metamodel and dependent transformation into another can be achieved using the operators. As transformation languages like ATL and QVT are Turing complete, the set of operators would need to be very powerful to achieve completeness. The practical use of such an operator set may also be doubtful, as it would need to cover the addition and removal of every possible transformation language construct and any valid combination of constructs. Such an operator set would inevitably be very large and present management issues of its own. Therefore the practical usefulness of the approach lies with the applicability of the operators, and the ability to use the co-evolution approach in close conjunction with other development and modelling activities and not completeness on the transformational level.

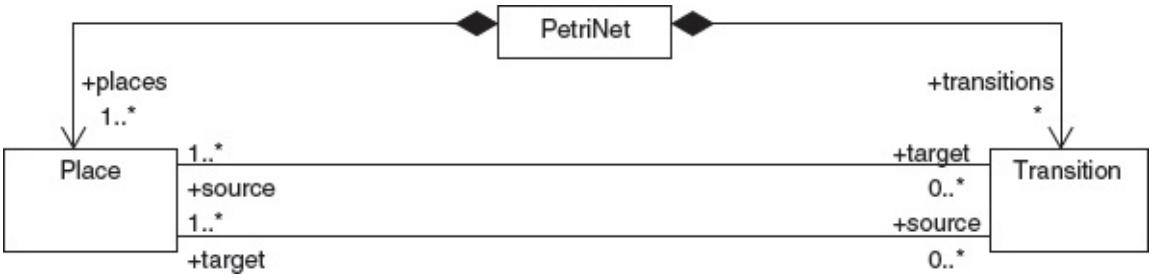
9.2. Petri Net Evolution Scenario

This section covers an evolution scenario of a metamodel evolution in combination with a co-evolving ATL model transformation scenario.

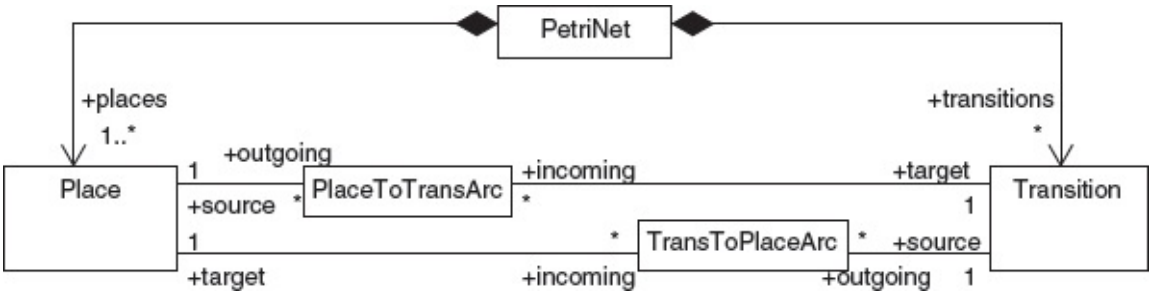
9.2.1. Metamodel Evolution

The evolution is performed in several steps from a simple to a more sophisticated metamodel for Petri nets [107]. The states of the metamodel during the evolution are given in Figure 9.1 . The first state *e0* describes a simple version of a metamodel for Petri nets; it consists simply of transitions and connected places so that each transition is required to have at least one source and target place. During the course of the evolution, the metamodel is extended by the concept of a weighted arc instead of simple connections.

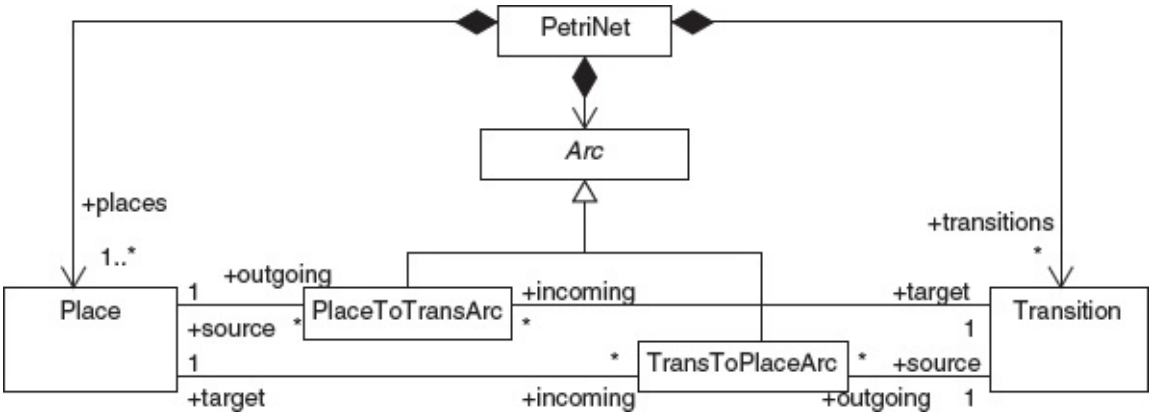
e0:



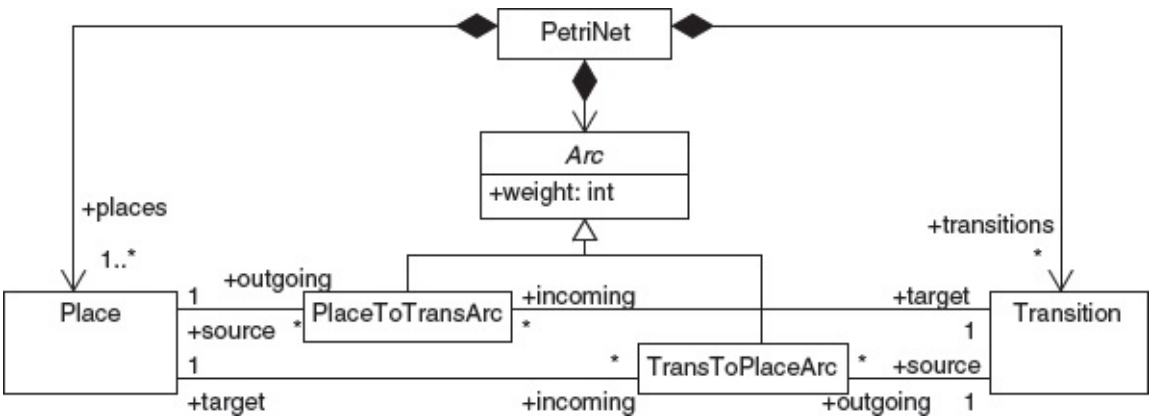
e1:



e2:



e3:



e4:

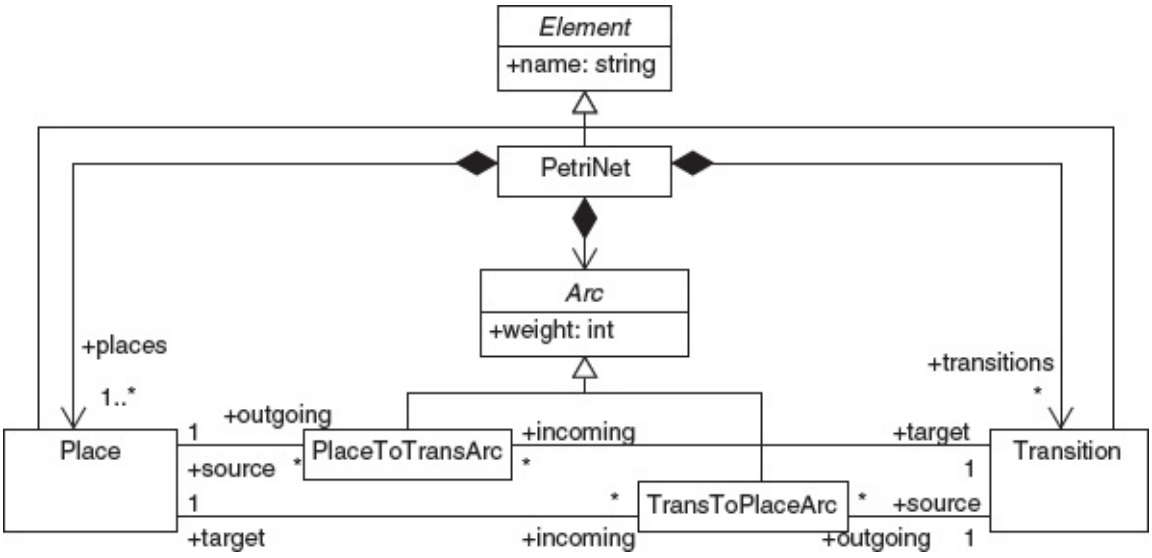


Figure 9.1.: Petri net metamodel evolution [107]

The arc is represented by a class with a weight property, which can be assigned for each instance, extending the Petri net concepts that can be modelled. Furthermore, a general superclass is added which provides a name property to elements. The final state is shown in *e4*.

This evolution from state to state can be reproduced using operators as follows:

`e0` This is the initial state.

`e1` The *Association To Class* operator is used on both associations between `Place` and `Transition` to introduce the two new classes `PlaceToTransArc` and `TransToPlaceArc` in the direction of `Place` to achieve this state. This means that for every instance of `Transition` before the evolution, one instance of each of the new classes `PlaceToTransArc` and `TransToPlaceArc` is expected. The operator introduces the extra associations needed and adjusts the multiplicity of the association-ends accordingly and the names of the association-end properties as provided by the software architect in the resolution.

`e2` A new abstract class `Arc` is introduced, from which both of the previously added classes `PlaceToTransArc` and `TransToPlaceArc` inherit. This can be achieved using the *Extract Superclass* operator. As the `Arc` class is to be contained by the `PetriNet` class, a new association between the two is introduced. This can be done using the *Add Association* operator.

`e3` The abstract class `Arc` receives a new property `weight` of type `int`. This can be performed by the *Add Property* operator.

`e4` The classes `Place, Transition` and `PetriNet` are given a common abstract superclass `Element` with a property `name`. This can again be achieved using the operators *Extract Superclass* and *Add Property.*

9.2.2. Transformation Impact Resolution

We chose a transformation from a transformation scenario in the ATL Transformation Zoo as a basis for an impact analysis [46], as it closely matches the final metamodel in state *e4*. The transformation scenario provides a set of transformations to convert between Petri nets and path expressions. Figure 9.3 contains an example of a Petri net and Figure 9.2 a matching path expression. Path expressions describe the directed graph underlying a Petri net in textual form as ordered edges while the Petri net model uses the common ‘place’ and ‘transition’ semantics.

```
path f;(g;h + k;m*;n);(p+q);s end
```

Figure 9.2.: Textual path expression [46]

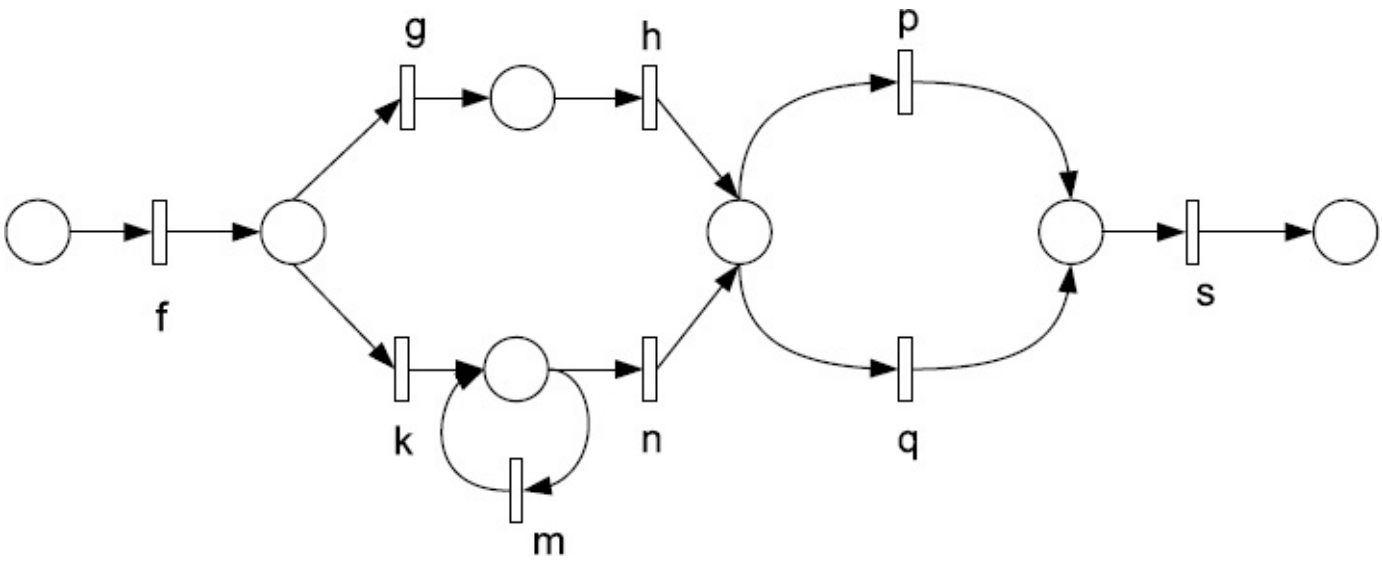


Figure 9.3.: Petri net example [46]

Figure 9.4 shows the metamodel that serves as AST for successfully parsed path expressions and is used on the LHS of the model transformations in this scenario. We chose the `PathExp2PetriNet` transformation to reconstruct the evolution impact of the metamodel evolution presented above. The transformation is given in listing 9.1 in abridged form[6].

For the offset for the evolution with state *e0,* we provide the transformation in listing 9.2 to convert between paths and instances of the simple metamodel.

The impact resolution for the evolution of the Petri net metamodel performed through operator application is as follows:

```
module PathExp2PetriNet;
create OUT : PetriNet from IN : PathExp;

helper def: allTransitions : Set(PathExp!Transition) =
 PathExp!Transition.allInstances();

rule Main {
 from
        pe : PathExp!PathExp
 to
   pn : PetriNet!PetriNet (
   name <- pe.name,
   places <- pe.states,
   transitions <- pe.transitions,
   arcs <- thisModule.allTransitions
  )
}

rule State {
 from
       pe_s : PathExp!State
 to
  pn_p : PetriNet!Place (
   name <- '',
   incoming <- pe_s.incoming,
   outgoing <- pe_s.outgoing
  )
}

rule Transition {
 from
       pe_t : PathExp!Transition
 to
  pn_t : PetriNet!Transition (
   name <- pe_t.name,
   incoming <- pn_ia,
   outgoing <- pn_oa
  ),

  pn_ia : PetriNet!PlaceToTransArc (
   source <- pe_t.source,
   target <- pn_t,
   weight <- 1
  ),

  pn_oa : PetriNet!TransToPlaceArc (
   source <- pn_t,
   target <- pe_t.target,
   weight <- 1
  )
}
```

Listing 9.1: Abridged PathExp2PetriNet transformation [46]

```
module PathExp2PetriNet;
create OUT : PetriNet from IN : PathExp;

helper def: allTransitions : Set(PathExp!Transition) =
 PathExp!Transition.allInstances();

rule Main {
 from
        pe : PathExp!PathExp
 to
   pn : PetriNet!PetriNet (
   places <- pe.states,
   transitions <- pe.transitions
  )
}

rule State {
 from
       pe_s : PathExp!State
 to
  pn_p : PetriNet!Place (
   incoming <- pe_s.incoming,
   outgoing <- pe_s.outgoing
  )
}

rule Transition {
 from
       pe_t : PathExp!Transition
 to
  pn_t : PetriNet!Transition (
   incoming <- pe_t,
   outgoing <- pe_t
  )
}
```

Listing 9.2: Primary transformation state matching *e0*

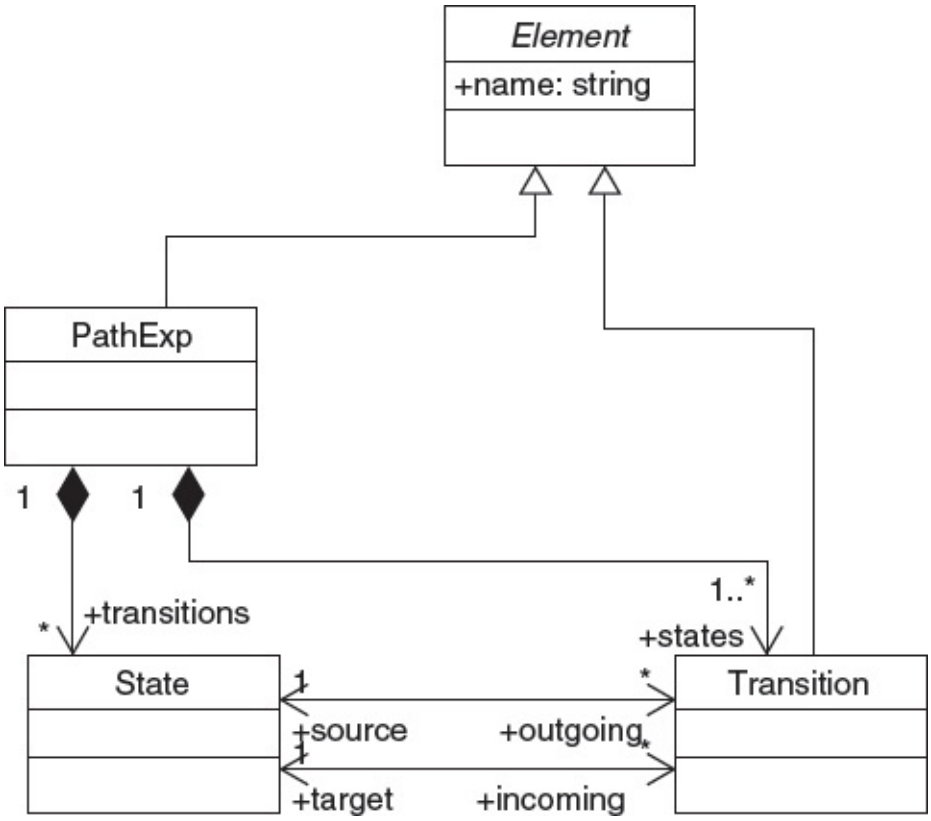


Figure 9.4.: The path expression metamodel [46]

i1 The *Association To Class* operator for the associations between Place and Transition introduces two new classes PlaceToTransArc and `TransToPlaceArc`. Impact is detected for the `Transition` rule. In the resolution, new out-pattern elements are created for instances of the classes `PlaceToTransArc` and `TransToPlaceArc` along with the Petri net `Transition`. The new instances are linked to the `Transition` using the original source and target `States` of the path expression `Transition`. This concludes the impact resolution for the first evolution step.

i2 First, a new abstract class `Arc` is introduced using the *Extract Superclass* operator. No impact occurs as there are no rules in the transformation that match the classes `PlaceToTransArc` and `TransToPlaceArc`. Otherwise, such rules could have been merged. In addition to the introduction of the class `Arc`, a new association is added. This addition has no detectable impact. The user has to extend the `Main` rule to provide the containment condition for all arcs: `arcs<-thisModule.allTransitions`. This value binding makes use of the helper `allTransitions` which is also added manually.

i3 During this step, the abstract class `Arc` receives a new property `weight` of type `int` using the `Add Property` operator. The transformation isextended to provide a default value of ‘1’ for all weights. This is a semantic change and is performed manually.

step	operator	impact	automatic resolution	manual addition
i1	Association to Class	yes	yes	no
i1	Association to Class	yes	yes	no
i2	Extract Superclass	no	n.a.	no
i2	Add Association	no	n.a.	yes
i3	Add Property	no	n.a.	yes
i4	Extract Superclass	no	n.a.	no
i4	Add Property	no	n.a.	yes

Table 9.1.: Result overview of the evolution and the matching impact resolution

i4 The classes `Place`, `Transition` and `PetriNet` are given a common abstract superclass `Element` using the operator *Extract Superclass* and a new property `name` using *Add Property*. Both operators have no impact. The transformation again has no rules that are the same for any combination of the three classes so that *Extract Superclass* has no impact. The `name` property is optional and requires no default value. Instead, the user can extend the rules to provide a binding for `name` using values from the path expression metamodel: `name <- pe t.name` for the rule `Transition` or a default value `name <- ''` for `State`.

Table 9.1 provides a summary of the operators used, their impact and the impact resolution performed.

9.3. Summary

This chapter covers the evaluation of our approach. While the completeness of the set of operators on the model level can be shown and is useful, completeness on the transformation level is difficult to achieve and most likely impractical in use. Instead, the close integration with other MDE tooling and activities is important. How such integration can be achieved is demonstrated in the next chapter in detail. The Petri net scenario discussed in Section 9.2 demonstrates how the operators and the impact resolution can be applied to the stepwise evolution of a metamodel and a dependent model transformation. The evolution contains both restructuring and extension steps. The proof of concept of the overall tool support is discussed next.

9.2.3. Discussion

As can be seen in Table 9.1, the entire evolution of the metamodel can be performed using operators. Furthermore, the impact on the dependent transformation was detected and resolved automatically. Some of the changes performed were extensions of the semantics of the metamodel and, although not creating an impact to the syntactic correctness of the transformation, required manual additions to the transformation to be complete. This

suggests that a close integration of co-evolution tooling and development tooling is sensible so that the two activities can be performed in easy succession. The other advantage of the approach can be seen in the reliable impact detection, as only the operators for the addition of elements required user attention, potentially reducing the time needed to search for impacts on all rules.

The next chapter covers the design and implementation details of integrated tooling for the operators, the impact detection and the approach as a proof of concept. It demonstrates how the close integration with common MDE tooling can be achieved and that the stepwise approach is suitable to be used in conjunction with other MDE development and modelling activities.

---

[6] The transformation was shortened by removing `thisModule.resolveTemp()` statements and comments.

CHAPTER 10

Proof of Concept Prototype

This chapter covers the prototypical tooling for the co-evolution approach for metamodels and model transformations. The purpose of the tooling is the demonstration of the feasibility of the approach in combination with common MDE tooling. The next section introduces the general aspects of the implementation and Section 10.2 discusses the main prerequisites for an integrated tooling. In Section 10.3 the details of the ATL textual editor and in Section 10.4 those of the graphical metamodel editor are covered. Sections 10.5 and 10.6 cover the details of the implementation for the operators and the impact resolution respectively. In addition, Appendix A provides a set of complementary screen-shots of the operator application and impact resolution in the prototype, starting on page 223.

10.1. Overview

This section provides a general overview of the projects that make up the prototypical implementation of our approach and the tool integration. Figure 10.1 contains a UML 2.5 Model Diagram of the project structure. Each component of the prototype is discussed in more detail in the rest of this chapter.

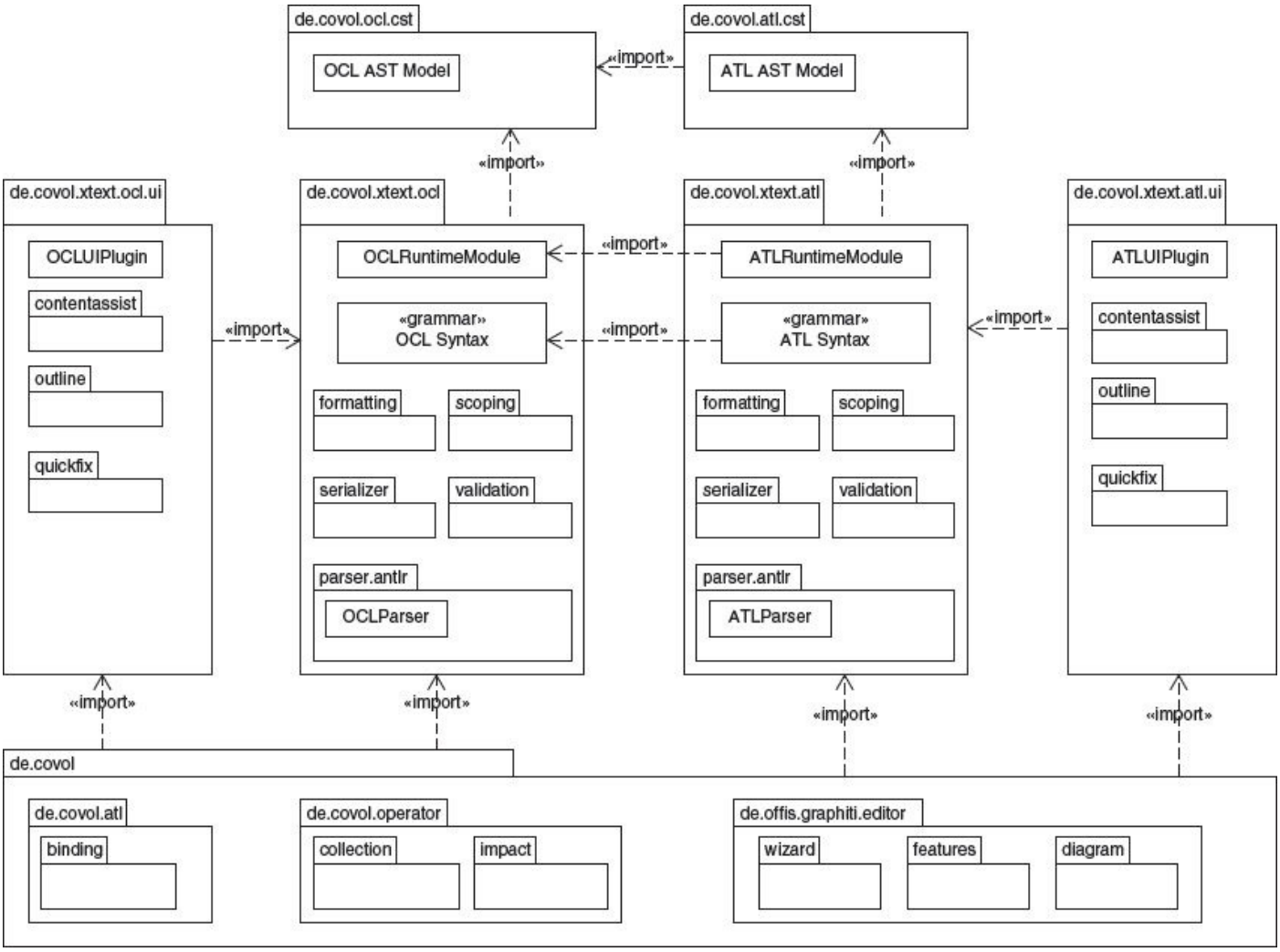


Figure 10.1.: The major components and relationships of the prototypical implementation

To provide editing support for ATL transformations, an integration with an ATL textual editor is needed. The editor needs to support the modification of the AST behind the model transformation code by an impact resolution and mirror the changes back to the textual representation. This is achieved in the prototype by generating a tool-set for ATL using the Eclipse Xtext Project framework for the Eclipse IDE. (The details are discussed in Section 10.3 below.) For this purpose, an Xtext grammar for both the ATL and the ATL dialect of OCL were created. From the grammars, Ecore models of both the OCL AST and the ATL AST were created. The Xtext generator framework was then used to generate the tooling and the ANTLR parser [81] for the conversion between the textual representation and the AST. The generated tooling provides basic support for features like content assist, an outline view, quick-fixes, a formatter, a serialiser, scoping and validation support.

The de.covol projects contain the central parts of the prototype and import the ATL editor projects. The de.covol.atl project contains binding support to map the model types used in the ATL transformation code to actual model elements in memory. The de.covol.operator project contains the implementation of our operators for Ecore models (Section 10.5) and the impact detection and resolution for ATL transformations(Section 10.6). The de.offis.graphiti.editor project contains a graphical editor for Ecore models based on the Eclipse Graphiti Project. It is discussed in detail in Section 10.4.

10.2. Preliminary Considerations

Two general design decision were made for the prototypical implementation. The first is concerned with the choice of MDE tooling to integrate our approach into. This choice is discussed in the next section. In Section 10.2.2 the consequential adaptation of the operators for the EMF implementation of EMOF are laid out. From Section 10.3 onwards, the implementation details of the prototype are discussed.

10.2.1. Tool Integration

Requirement 7 calls for the close integration of the tool support for coevolution with that common to metamodel and transformation development and to follow similar usage patterns (see page 72). For this reason, the tooling provided by the Eclipse ATL Project for the development of ATL transformations and the (meta-)modelling tools provided by the Eclipse Modeling Project were chosen as a starting point. The required integration is twofold; for one, the user should be able to apply the operators in the editor used for the editing of metamodels and second, the impacts on transformations and their resolutions should be displayed in the editor used for ATL transformations.

For the metamodel part, the editor must support the selection of elements and the selection of the operator to apply to these elements. The choice was made to extend an existing graphical Ecore model editor to support the usage of co-evolution operators. The details of the graphical editor are descried in Section 10.4.

The textual editors available in Eclipse are commonly backed by a model of the AST and convert between the concrete textual syntax and the AST on the fly. This means that a good integration approach for the impact resolution is the manipulation of the AST model used by the editor to perform the co-evolution, so that changes are immediately reflected

in the textual view. Unfortunately the AST used by the ATL editor provided by the Eclipse ATL Project proved inaccessible so that the decision was made to create a prototypical ATL editor using the Eclipse Xtext Project. Other benefits of using Xtext are the availability of syntax-highlighting of text snippets, which we use for previews of the impact resolution (see Section 10.6.2 below) and better support for editing features like code completion and error highlighting [102]. The ATL textual editor is detailed in Section 10.3.

10.2.2. Implementation for Ecore

As the available ATL tooling and especially the ATL VM is based on the EMF implementation of EMOF, the operators needed to be translated to the Ecore metamodel for the prototype. The Ecore implementation is close to EMOF but has some differences that needed to be taken into account (see Section 2.4).

First off, Ecore knows the elements `EAttribute` for simple and `EReference` for complex properties instead of the combination of properties and associations as in EMOF. Therefore, operators manipulating complex properties were implemented to handle `EReference` elements and those for simple properties `EAttribute` elements.

Secondly, in Ecore, bi-directionality of complex properties is modelled using two `EReferences` with an `opposite` property pointing to the pairwise other property, instead of using association-ends. This is regarded for implementations of operators for complex properties. The simple types defined for EMOF are mapped to equivalent Java types in Ecore so that those are used instead in the implementation.

Finally, Ecore requires that only one root element exists in each model so that an XMI serialisation is always possible. This is taken care of in the implementation by always choosing a containing element when new elements are created.

10.3. ATL Concrete and Abstract Syntax Implementation

The ATL textual editor used to demonstrate the tooling integration was created using the language tooling and generators provided by the Eclipse Xtext Project. Xtext allows the definition of a language grammar which defines the concrete syntax of a DSL and provides different generators for the creation of an AST model, editors for full or partial texts in the DSL with features like syntax highlighting, code-completion, error detection and highlighting and more [102].

Although the ATL AST model released with the (current) version of the Eclipse ATL Project (Eclipse Helios ATLAS Transformation Language SDK 3.2.1) is based on Ecore and would be suitable as a basis for an Xtext grammar definition, it is not accompanied by a EMF Genmodel (which is needed for the Xtext generators) for the generated AST Java classes, nor did the include Ecore model permit the creation of such an EMF Genmodel due to containment errors in the model. Therefore a new ATL ecore model was build for the prototype in close alignment to the official one and a custom version of the ATL AST was generated using the tools provided by Xtext. Since the structure is almost the same, converting between the official AST model and the custom AST model should be

possible. We refer to the custom AST implementation from now on as ‘the ATL AST’ unless otherwise noted.

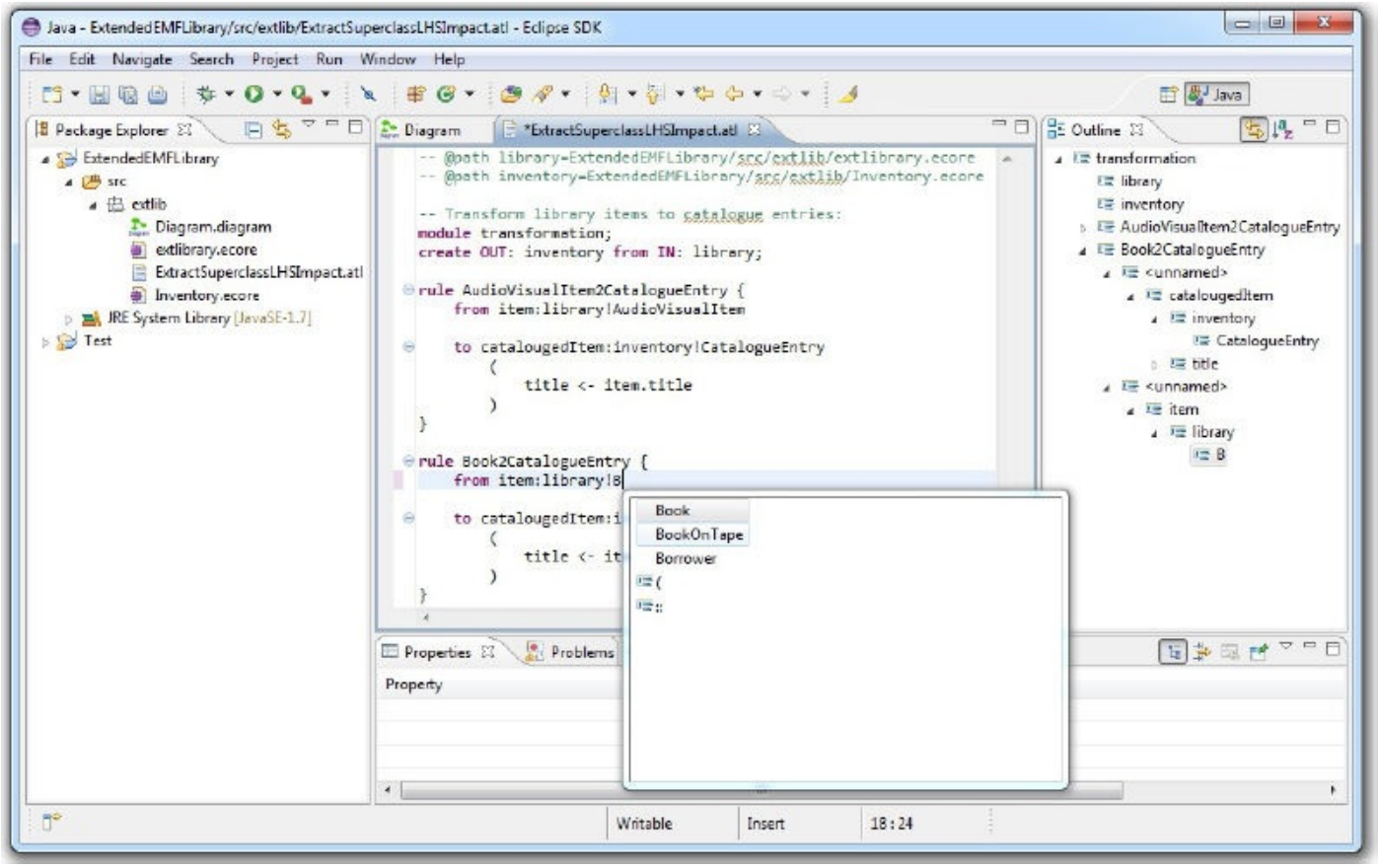


Figure 10.2 .: Screenshot of the custom ATL editor implementation

The grammar created for the ATL editor is split into two parts, one for the OCL dialect used by ATL and one for the ATL syntax itself which imports the OCL part. The full grammar definitions are given in Appendix B, beginning on page 227. The finished ATL editor implementation with features such as an outline view, syntax highlighting and code completion is shown in Figure 10.2 .

Some of the primary differences to the official ATL syntax and AST model are:

- The AST does not support the ATL `Query` construct, as queries are used primarily to reason about models, not for transformation definitions and are therefore of lesser concern for co-evolution.

- The `Library` construct is missing, as it is an alternative structuring mechanism for `Helpers` which can be used in a `Module` directly instead.

```
1  abstract class ATLVariableDeclaration
2  {
3    attribute varName : String {ordered};
4    property type : ATLType[?] {ordered composes};
5  }
6  class ATLType
7  {
8    attribute modelName : String[?] {ordered};
9    property type :
10       ocl::TypeNameExpCS[?] {ordered composes};
11 }
```

Listing 10.1: New ATL AST model elements

- Support for comments in LocatedElement was removed, as Xtext provides a different mechanism for comments.
- A new AST model element for variable definitions was introduced for ATL, to better separate the ATL and OCL AST models. It is shown in listing 10.1.
- Some further changes were made to the usage of OCL in ATL to improve the conformance to the OCL standard.

In sum these omissions and changes would need to be addressed to provide fully functional tooling for ATL or the integration of the impact resolution into the official ATL tooling could be attempted. Non-the-less, the current prototypical implementation is deemed suitable to demonstrate that the integration of our co-evolution approach with common MDE tooling is feasible.

10.4. Graphical Model Editor

The implementation of our approach is build into a graphical editor for metamodels to integrate as close as possible with established workflows for modelling. This fulfils Requirement 7 for common tool integration. For this purpose we chose a model editor based on the Eclipse Graphiti Project framework [98]. It was developed by Klaas Oetken to demonstrate different refactorings for Ecore models [79] and extended and adapted to be able to select and perform the co-evolution operators. Figure 10.3 shows the extended EMF-Library Metamodel example introduced in Section 1.1 in the Covol Model Editor.

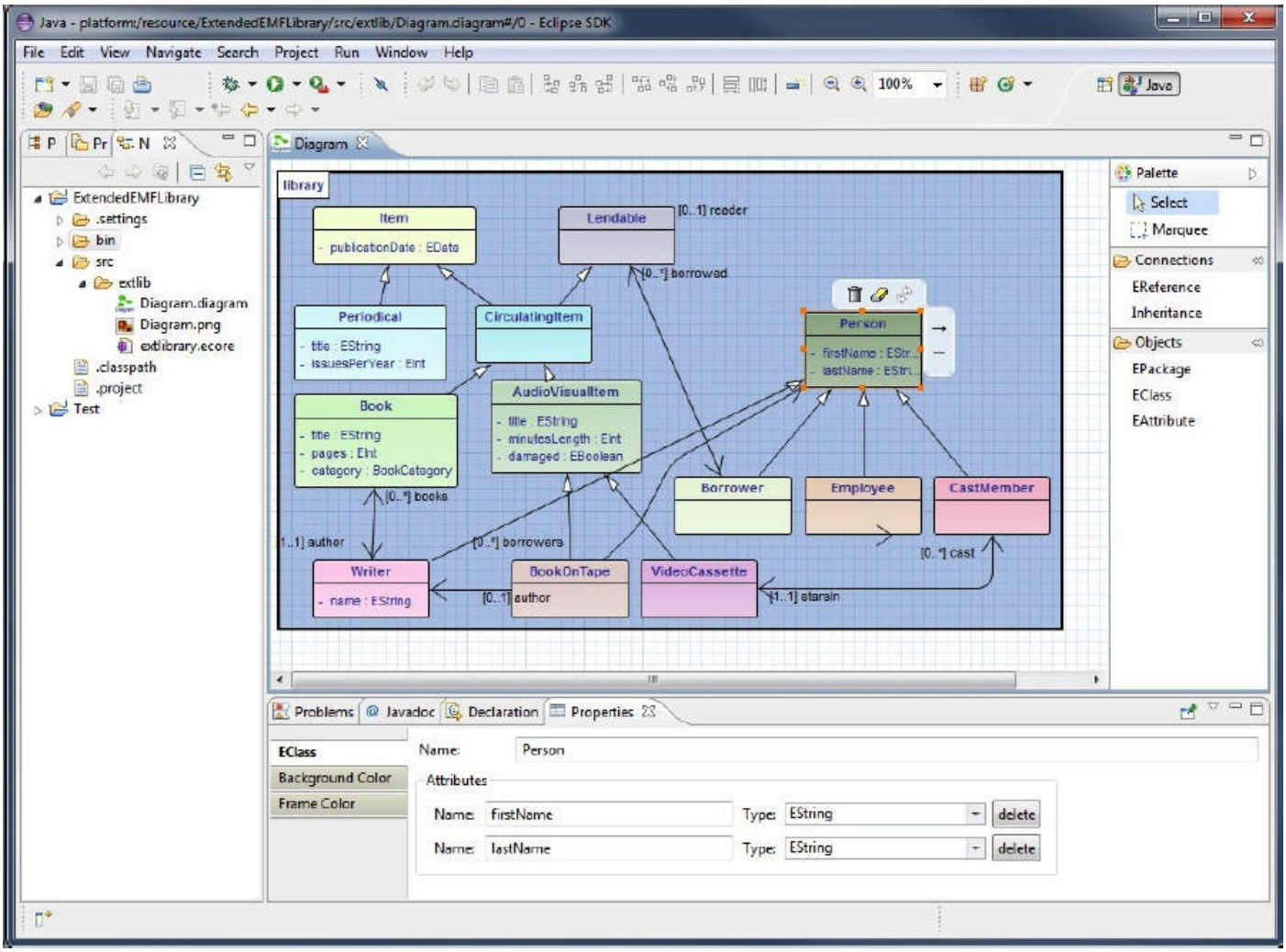


Figure 10.3.: Screenshot of the integrated Covol Model Editor

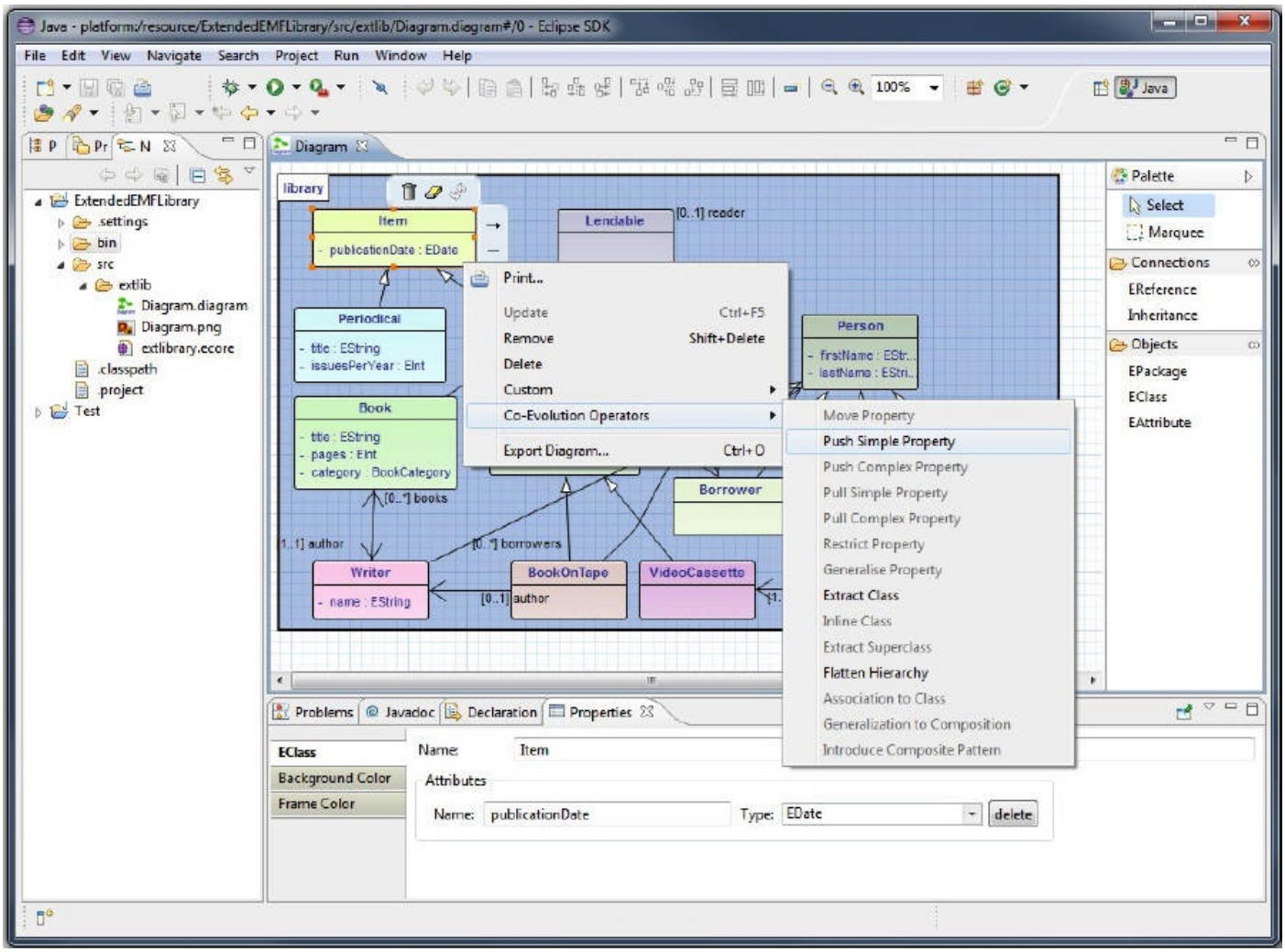


Figure 10.4.: Screenshot of the application of operators in the graphical editor

The editor provides the basic functionality to create, edit and manipulate a graphical representation of an Ecore model file. This graphical representation is an alternative view to the standard, table-based representation of Ecore models provided by the EMF. The Eclipse Graphiti Project provides a framework of basic functionality common to graphical editors that can be extended and customized for graphical DSLs. The Graphiti extension mechanism to interact with the graphical elements of the editor relies on so called *features*, where each feature encapsulates an action that the user can perform on the model. The integration of the operators into the editor is achieved by creating a custom feature for each operator.

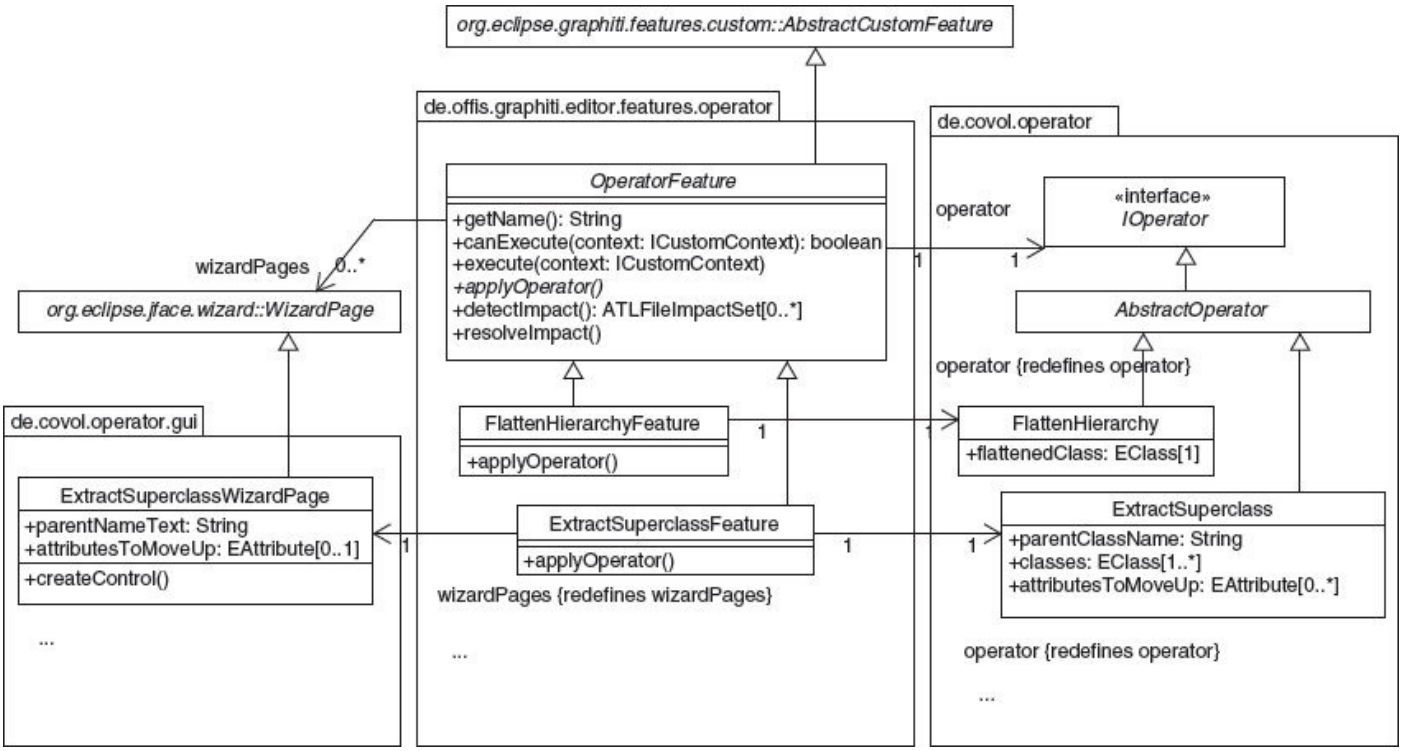


Figure 10.5.: Details of the *feature.operator* package

The available operator features are provided as a pop-up context menu as shown in Figure 10.4 . The menu is context sensitive to the currently selected elements in the graphical model editor. Whether or not an operator feature is enabled depends on the selected elements and the pre-conditions defined by each operator so that only those operators can be executed for which all constraints are met. For instance, in Figure 10.4, the operators *Push Simple Property, Extract Class* and *Flatten Hierarchy* are applicable to the selected `Item` class.

The execution details of the operator are delegated by the features to individual operator implementations. The user is guided through the coevolution process by a dialog which contains a number of individual pages for each step. This type of dialog is also known a wizard.

The relationship between the operator implementations, the operator features and individual wizard pages is shown in Figure 10.5 . The relevant classes are:

OperatorFeature The abstract class `OperatorFeature` provides the bridge between the graphical interface of the model editor and the operator implementations. It extends the `AbstractCustomFeature` extension mechanism of the graphiti framework for custom features. The term *feature* refers here to buttons or widgets that the user can interact with in the editor. The operator features are provided as menu items in the context menu of diagram elements of the editor (see the pop-up menu in Figure 10.4) and extend the basic functionality of `OperatorFeature` as subclasses for each operator available. Each subclass handles the specifics of its operator by translating the currently selected model elements into the parameters needed by the operator and by providing the specialized wizard pages for the operator. The general functionality provided by `OperatorFeature` for all operator features is:

- **getName(): String** Returns the name of the operator for use in the context menu entry.

- **canExecute(context: ICustomContext): boolean** This method is prescribed by `AbstractCustomFeature`. It checks whether this operator feature can be executed under the current circumstances as provided by the given `ICustomContext`. The `ICustomContext` provides the currently selected model elements, for instance.

- **execute(context: ICustomContext)** Executes this operator feature for the context provided. This entails starting the coevolution wizard and initializing the list of changes the operator performs on the selected model elements.

- ***applyOperator()*** After the user provides all further information required in the corresponding wizard and accepts the changes proposed by the operator to the current model, this method is called to perform the changes. This method is abstract and has to be specialized for each operator type.

- **detectImpact(): ATLFileImpactSet[0..*]** This method is called to detect all impacts on all ATL files currently active in the user workspace. These are returned as an `ATLFileImpactSet` for each ATL file.

- **resolveImpact()** If the user accepts the impacts and their resolution offered by the operator, this method is called to adapt the ATL files accordingly.

Implementations of OperatorFeature Each operator has a matching implementation of `OperatorFeature`. These handle the extraction of the parameters needed by each operator from the currently selected model elements and the wizard pages unique to the operator.

Implementations of WizardPage Some operators require further information from the user to perform their function besides the currently selected model elements. For instance, the *Extract Super Class* operator (see page 126) requires a new name for the extracted class and a selection of attributes to extract. These are obtained from the user by the specialized `ExtractSuperclassWizardPage` page. The specialized wizard pages are integrated into the co-evolution wizard when the operator feature is executed. The details of the wizard pages are discussed further in Section 10.5.1.

Implementations of IOperator The behaviour and the impact detection and resolution of the different operators is handled by implementations of the `IOperator` interface contained in the package `de.covol.operator`. The details of the operator implementations are discussed in the next section.

10.5. Operator Implementation

The operators introduced in chapter 6 are implemented as subclasses of the `IOperator` interface in the `de.covol.operator` package. The package contents, the relevant attributes and operations and the relevant relations to the `de. covol.operator.impact` package are shown in Figure 10.6 .

IOperator The interface `IOperator` prescribes the minimal set of operations required by an operator implementation to function in the current model diagram editor. These are:

- **getName(): String** provide the unique name of the operator.

- **setWorkspaceContext(context: WorkspaceContext)** The operator implementations require information on a general context to operate, like all ATL files in use that may be impacted by an application of the operator. In the current implementation, this information is provided by an implementation of `Workspace Context` and set using this method.

- **canApply(eObjects: EObject[1..*]): boolean** This method encapsulates the checking of pre- and post-conditions of the operator. It returns `true` when the operator can be applied to the given set of model elements. This method is used by the view to enable or disable features based on the currently selected set of model elements.

- **apply()** This method is called when the operator is meant to be applied. It is assumed that each operator implementation provides its own specific methods to set the model elements to which the operator is applied.

- **detectImpact(): ATLFileImpactSet[0..*]** This method is called to determine the potential impact this operator has on all the ATL files provided by the current `WorkspaceContext`.The impact is returned as a list, with one `ATLFileImpactSet` for each ATL file.

- **resolveImpact()** When the user accepts the determined impacts and the resolution suggested, this method is called to perform the resolution and update the ATL files in the current `WorkspaceContext`.

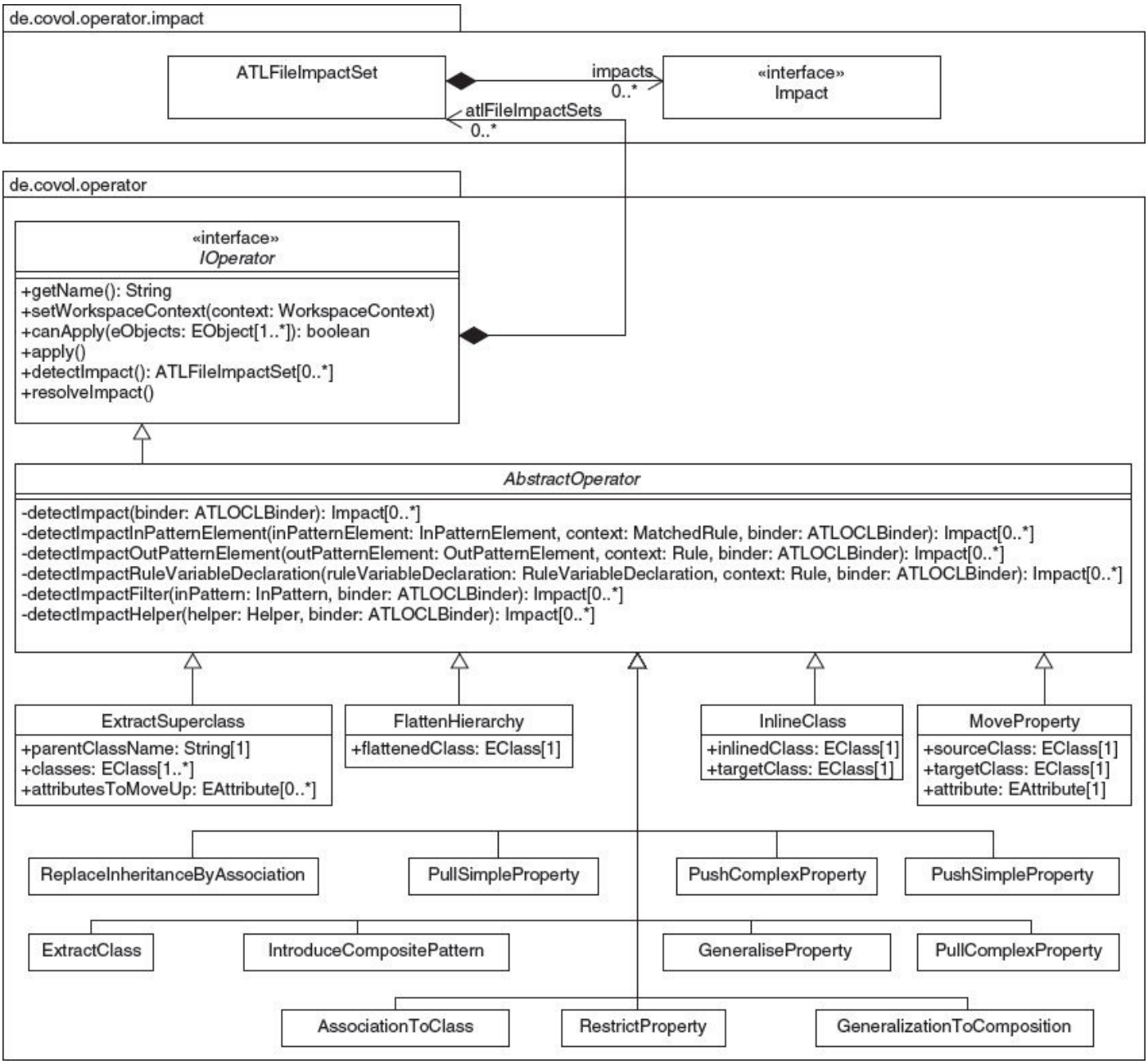


Figure 10.6.: Details of the *operator* package

AbstractOperator The `AbstractOperator` implements the `IOperator` interface and provides basic and common functionality for concrete operator implementations. The main focus lies on an impact detection mechanism that walks over the ATL of a potentially impacted ATL transformation and calls individual methods for the different parts of a transformation that may be impacted. The concrete operator implementations that extend this class only need to implement the abstract methods for the types of the transformation relevant to them. For instance, to handle the impact of the *Inline Class* operator (see page 161), only `Matched Rules` need to be considered so that only the `detectImpactMatchedRule` method needs to be overwritten and implemented by the concrete `InlineClass` operator as a subclass of `AbstractOperator`.

The impact detection requires type binding of the variables encountered in the ATL transformations under inspection and the relevant metamodels. This is performed by the class `ATLOCLBinder` which is passed into every method. The methods are as follows:

- **detectImpact(binder: ATLOCLBinder): Impact[0..*]** A call of this method

initiates the walking process. A correctly bound ATL AST is provided by the given `ATLOCLBinder`. The method returns the list of detected `Impact` entries, collected from implementations of the methods below.

- **detectImpactInPatternElement(inPatternElement: In-PatternElement, context: MatchedRule, binder: ATLOCLBinder): Impact[0..*]** This method is called for each `InPatternElement` encountered in the bound ATL AST of the transformation under inspection. The owning `MatchedRule` is provided as context.

- **detectImpactOutPatternElement(outPatternElement: OutPatternElement, context: Rule, binder: ATLO-CLBinder): Impact[0..*]** This method is called for each `OutPatternElement` encountered in the bound ATL AST of the transformation under inspection.

- **detectImpactRuleVariableDeclaration(ruleVariable Declaration: RuleVariableDeclaration, context: Rule, binder: ATLOCLBinder): Impact[0..*]** This method is called for each `RuleVariableDeclaration` encountered in the bound ATL AST of the transformation under inspection.

- **detectImpactFilter(inPattern: InPattern, binder: AT-LOCLBinder): Impact[0..*]** This method is called for each `InPattern` encountered in the bound ATL AST of the transformation under inspection. It should return any impacts due to an occurance in the `Filter` of the `InPattern`.

- **detectImpactHelper(helper: Helper, binder: ATLOCL Binder): Impact[0..*]** This method is called for each `Helper` encountered in the bound ATL AST of the transformation under inspection.

10.5.1. Operator-specific Wizard Pages

Some operators require further information beyond a set of selected model elements to be performed. This is the case for example for the *Extract Superclass* operator (as defined in Section 6.13 on page 126), for which the name of the extracted class and the selection of attributes to pull up are needed. This information is different for each operator so that the coevolution wizard can be extended by a set of extra pages for each operator implementation. These are contained in the `de.covol.operator.gui` package as shown on the left-hand side in Figure 10.5 . The wizard page for the *Extract Superclass* is shown in Figure 10.7 .

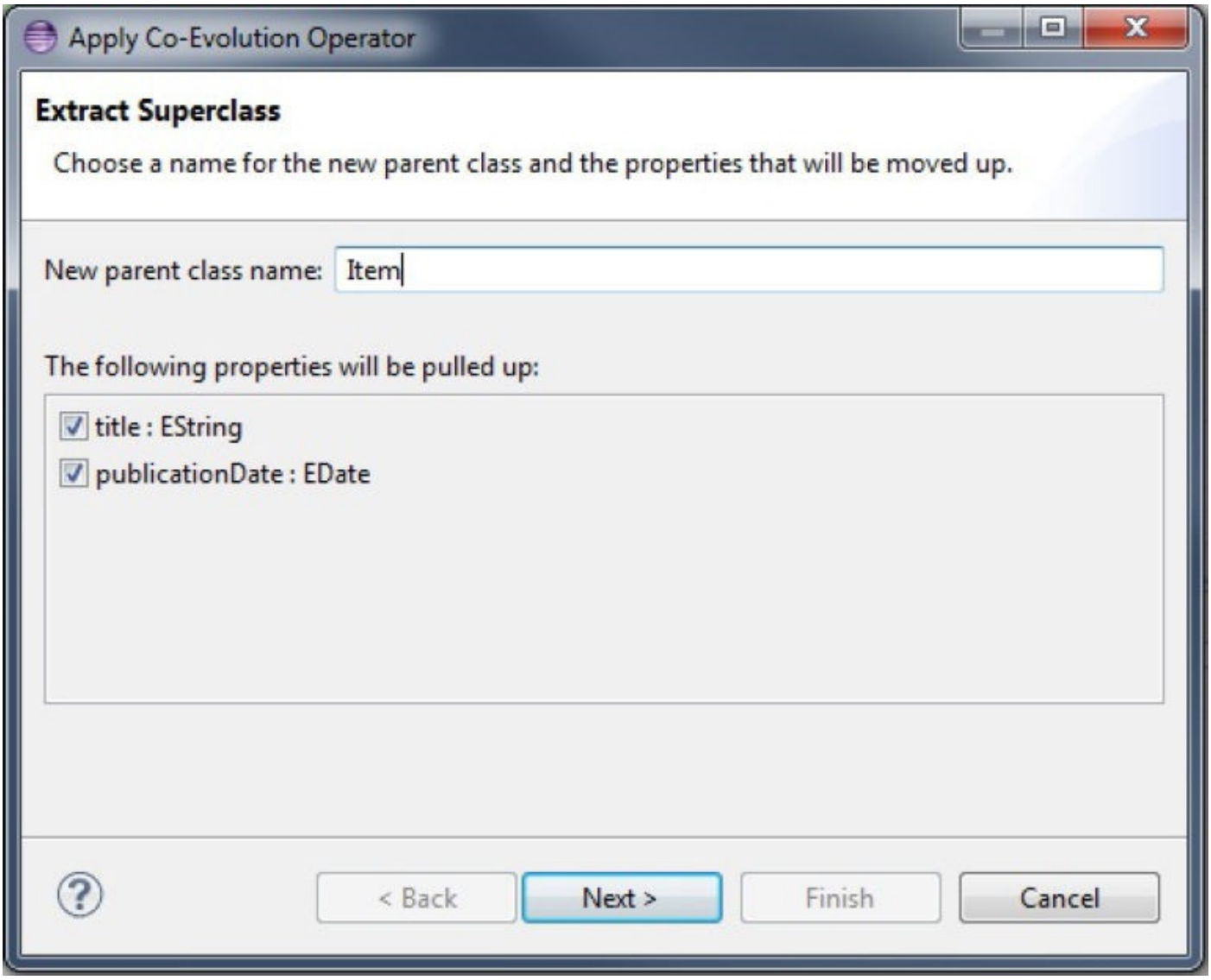


Figure 10.7.: The operator-specific wizard page for *Extract Superclass*

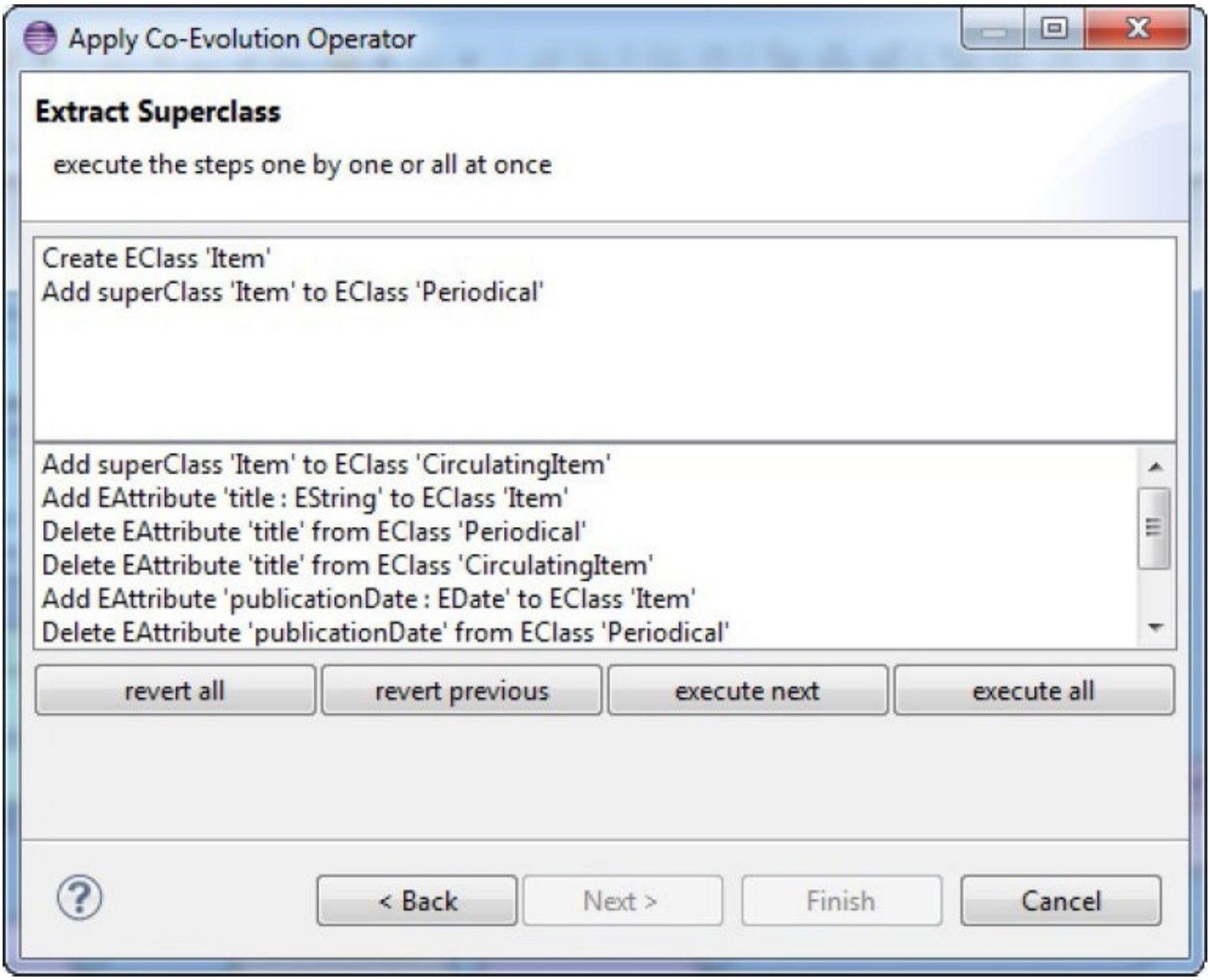


Figure 10.8.: The stepwise execution of operators

10.5.2. Operator Execution

The individual changes an operator makes to the selected model elements are provided to the user in the form of a list so that the result can be reviewed. This is implemented as a wizard page for the co-evolution wizard (see Figure 10.8 for an example of the changes made by the *Extract Superclass* operator). The user is given the option of inspecting each change by executing the changes individually or all at once. The steps are executed top-down in the order given by the operator, where the completed ones are shown in the top half of the wizard page and the remaining ones in the bottom half. The effect of the change can also be seen in the current model view, as it is performed. Should the user abort the operator, the model is reverted into its original state and all changes are undone. (For a complete step-by-step run-down of an operator execution please refer to section A starting on page 223 in the appendix.)

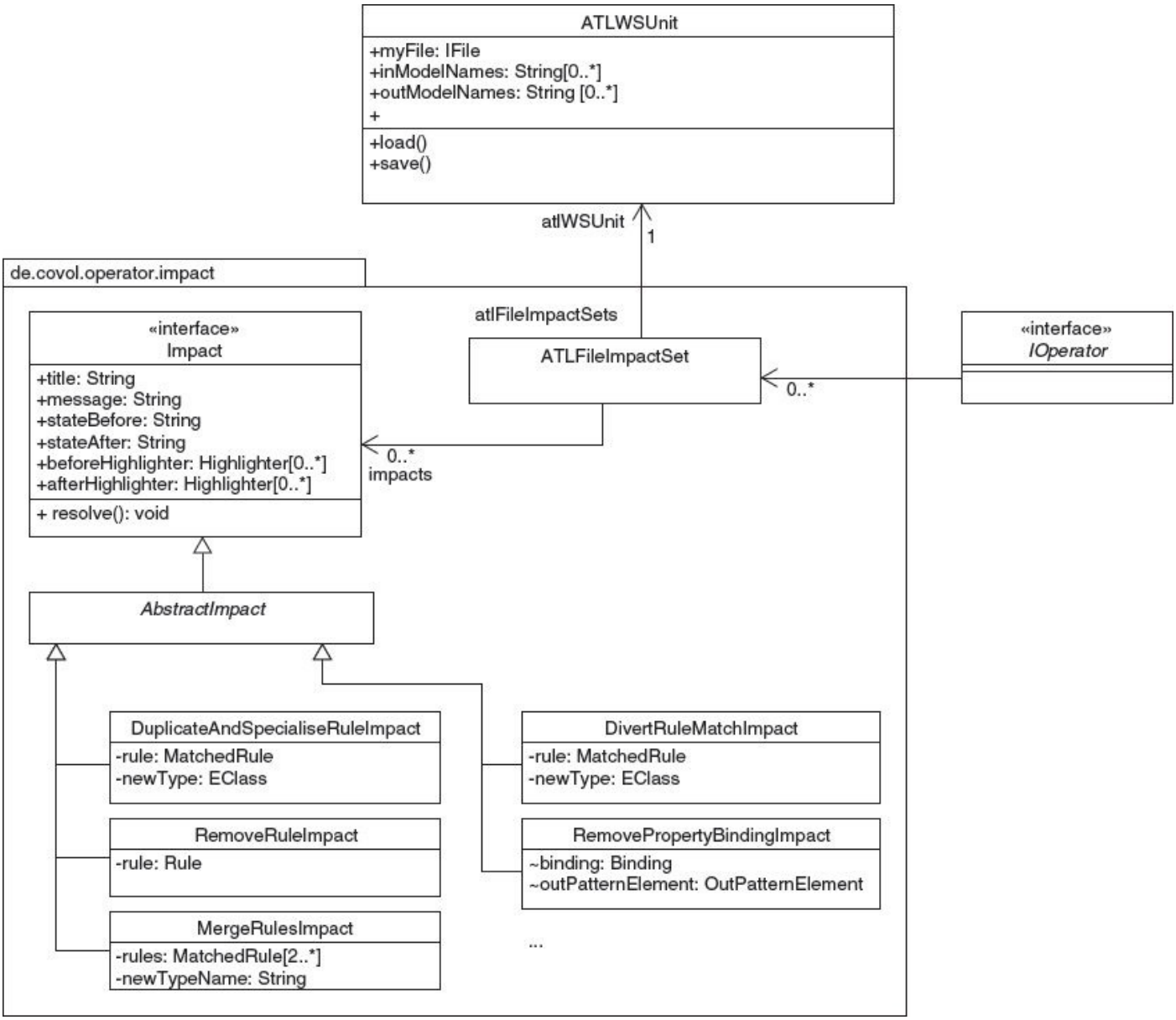


Figure 10.9.: Details of the *impacts* package

10.6. Impact Detection and Resolution

10.6.1. Impact Detection

The impact detection is initiated by a call to the `detectImpact()` method defined in the `IOperator` interface (see Figure 10.6). The `AbstractOperator` class provides a tree-walking implementation to detect impacts on the AST of the transformation in question, as described above. As a result, it returns a `ATLFileImactSet` instance for each ATL transformation that was found to be impacted by the operator. The `ATLFileImpactSet` in turn provides a list of instances of implementations of the `Impact` interface, where each implementation represents a different type of impact. The impact implemntations are contained in the package `de.covol.operator.impact` as shown in Figure 10.9 .

The classes relevant to the impact detection are as follows:

ATLWSUnit This class represents an ATL transformation in the current work space of the user. It references an actual file containing the transformation and a set of in and out

model names, that are used in the transformation to reference the domain models of the transformation.

ATLFileImpactSet The `ATLFileImpactSet` holds all the impacts detected for one `ATLWSUnit`. These are provided by the detection process performed by the operator implementations of `IOperator`.

Impact The `Impact` interface prescribes all the features necessary for a review by the user of the potential impact resolution and the `resolve()` method which is called to initiate the resolution process. The features are:

- **title: String** The name of the impact.
- **message: String** A description in natural language of the nature of the impact.
- **stateBefore: String** A `String` containing the relevant part of the transformation as it is before the resolution is performed in concrete syntax (ATL).
- **stateAfter: String** A `String` containing the state of the transformation after the resolution is performed.
- **beforeHighlighter: Highlighter[0..*]** The `Highlighter` contains one or more regions of characters in the concrete representation of the transformation as given by `stateBefore` that are highlighted to indicate the exact parts of the transformation that are affected by the resolution.
- **afterHighlighter: Highlighter[0..*]** As with the `beforeHighlighter`, the `afterHighlighter` highlights the exact parts of the transformation that are to be changed by the impact resolution.

AbstractImpact and its subclasses The different impacts and their resolutions are implemented as subclasses of `AbstractImpact`, where each class contains the specifics of one type of impact. For instance, the `MergeRulesImpact` contains two `MatchedRules` that are merged into one as a result of the impact resolution. The resolution is performed on the transformation given by the associated `ATLWSUnit` when the `resolve()` method is called.

- **getName(): String** provide the unique name of the operator.
- **setWorkspaceContext(context: WorkspaceContext)** The operator implementations require information on a general context to operate, like all ATL files in use that may be impacted by an application of the operator. In the current implementation, this information is provided by an implementation of `WorkspaceContext` and set using this method.
- **canApply(eObjects: EObject[1..*]): boolean** This method encapsulates the checking of pre- and post-conditions of the operator. It returns `true` when the operator can be applied to the given set of model elements. This method is used by the view to enable or disable features based on the currently selected set of model elements.
- **apply()** This method is called when the operator is meant to be applied. It is assumed

that each operator implementation provides its own specific methods to set the model elements to which the operator is applied.

- **detectImpact(): ATLFileImpactSet[0..*]** This method is called to determine the potential impact this operator has on all the ATL files provided by the current `WorkspaceContext`. The impact is returned as a list, with one `ATLFileImpactSet` for each ATL file.

- **resolveImpact()** When the user accepts the determined impacts and the resolution suggested, this method is called to perform the resolution and update the ATL files in the current `WorkspaceContext`.

10.6.2. Impact Resolution

The proposed impact resolutions are provided to the user using the last page of the co-evolution wizard. The ATL files in the current workspace that are impacted are shown as a list with the detected impacts as sub items (see Figure 10.10). For each impact, the proposed resolution is provided in a split screen, showing the state of the transformation before the resolution on the left and the state of the transformation after the resolution on the right. The user can review each resolution in detail and accept or reject the application of the operator.

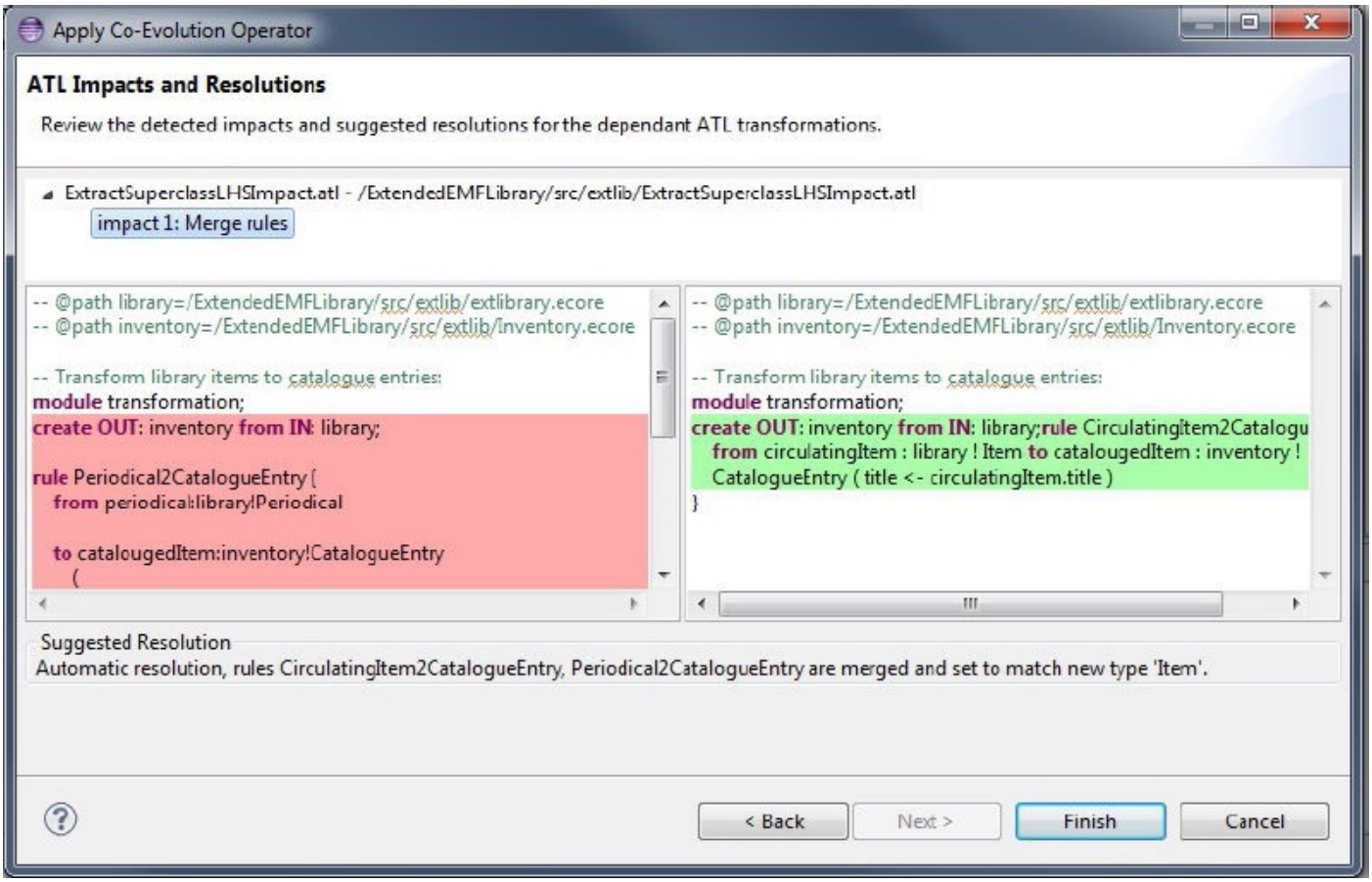


Figure 10.10.: The wizard page for the proposed impact resolution on transformations

CHAPTER 11

Conclusion

This chapter concludes the thesis, summarising the main contributions in Section 11.1. The benefits of this work for software architects and MDE in general are described in Section 11.2. Finally, Section 11.3 provides an overview of issues and challenges for future work.

11.1. Summary

This thesis provides an approach to the semi-automatic co-evolution of metamodels and model transformations. The approach supports software architects in evolving dependent artefacts of the MDS in stepwise fashion by applying predefined operators on EMOF-based metamodels and detecting and resolving the possibly occurring impacts on model transformations.

The main contributions of this work are as follows:

Approach The approach presented in chapter 5 allows the stepwise coevolution of metamodels and model transformations. It consists of three phases, starting with the adaptation of metamodels using an operator selected from a set of operators. In the second phase, the impact on an arbitrary number of model transformation that depend on the metamodel is detected. All possible impacts are collected. In the third phase, the resolutions for the impacts are suggested to the software architect and performed either automatically, if possible, or the software architect is supported in providing a manual resolution. With the conclusion of the approach, the MDS is left in an updated and valid state. The software architect can perform the next evolution step by choosing and applying the next operator.

Operator set Chapter 6 introduces a set of operators adapted from related work on co-evolution to be suitable to our approach. The operators are based on the UML EMOF metamodel and are formalised as relations using the OMG standard QVT-R graphical notation. Pre- and postconditions are used to ensure the applicability of the operators.

Impact resolution The impact detection and resolution for the set of operators is defined for ATL in chapter 7. The resolutions are also formalised as QVT-R relations for the ATL abstract syntax. The impact resolution can be automated where possible, to update the impacted transformation rule or rule part in accordance with the operator used. If an automatic resolution is not possible, the software architect is warned of the impact and can perform a manual adaptation.

Implementation The prototypical implementation of the approach, the operators and the impact detection and resolution is described in chapter 10. The implementation is integrated into common MDE tooling for the creation and editing of Ecore metamodels and ATL model transformations. The implementation provides feedback to the software architect about the changes made by applying an operator to a metamodel, the resulting impact on model transformations and the resolutions that are available.

In general, the approach contributes to the solution for the co-evolution problem for MDS. It addresses the problem of complexity and the tightcoupling of dependent artefacts in the context of software evolution and reduces the effort needed to prevent the occurrence of inconsistencies due to software evolution tasks for metamodels and model transformations.

11.2. Benefits

The results of this thesis supports software architects when performing evolutionary changes to an MDS, specifically when metamodels and dependent model transformations undergo co-evolution.

The benefit of the approach presented here is the reduced effort when performing commonly occurring changes due to the support provided to perform such changes through the application of operators and the reduced effort in resolving the impact on dependent model transformations. Depending on the change performed and the resulting impact, a range of resolutions can be performed automatically, while the software architect can be supported when performing manual resolution for the rest. Furthermore, the approach allows the automatic detection of possible impact on dependent transformation caused by metamodel evolution and narrows down the number of transformation rules that require work to be done.

As a distinctive feature compared to existing solutions, our co-evolution approach is designed to be stepwise and is directly integrated into tooling commonly used for the creation of transformations and metamodels. We furthermore provide formalized definitions of both operators and impact resolutions in QVT-R graphical syntax and support the full EMOF standard metamodel.

The individual benefits of our work are as follows:

Set of operators The operators presented in this thesis can be used for common co-evolution tasks and are complete on the metamodel level. Each operator defines a valid state before and after the application and ensures that the metamodel remains valid.

Formalised operators Formalising the operators using QVT-R and representing them in the graphical notation has the potential advantage of precision over informal methods or natural language. This advantage may come into play when the exact behaviour and impact of an operator for an implementation is needed. Furthermore, by providing a formal foundation, the impact of operators on different transformation languages or other metamodel-dependent elements of the MDS may become easier to compare.

Impact resolution for ATL The impact resolution presented in this thesis supports the software architect in handling the co-evolution problem by providing a means to predict the impact a change to a metamodel may have on a dependent ATL transformation and providing the means for automatic resolution in many cases. This greatly reduces the effort as the exact parts of a transformation that are impacted are known so that it is no longer necessary to check every part of every transformation for consistency after an

evolution step. Furthermore, the automatic resolution prevents the software architect from implementation work needed after a resolvable impact is detected.

Stepwise approach The stepwise nature of our approach ensures that it can be integrated into the normal working environment and tooling used by software architects in MDE. This means that co-evolution and development tasks can be performed in quick succession and supporting rapid development approaches.

To summarise, the approach presented in this thesis helps software architects by (1) reducing the effort needed for common co-evolution tasks, (2) by providing a means to predict the impact of different metamodel evolution steps on transformations and by (3) reducing the number of transformation rules that need to be validated after an evolution is performed.

Furthermore it advances the state-of-the-art in MDE and benefits researchers in software evolution by (1) providing a formalized set of co-evolution operators for EMOF metamodels, by (2) providing a set of formalised impact resolution relations for ATL and by (3) demonstrating the integration of our approach in common MDE tooling.

11.3. Future Work

This section discusses ideas and open issues summarized as future work. Short term future work requires smaller conceptual contributions and implementation work, while long-term future work requires in-depths new concepts and may for example be tackled in future research or industry projects.

Full scale integrated tooling While the prototypical implementation presented in chapter 10 shows how the integration of the approach with other MDE tooling could be done, industrial use of the approach requires full integration with the standard EMF tools, like the Ecore reflective editor. To this end, the interaction with model elements in the tree structure provided by the editor would need to be investigated so that software architects can select the elements on which to apply an operator in a similar fashion as in the prototype. Further application patterns may also be needed, as Ecore metamodels can also be created in textual editors for instance. Furthermore, to support the standard ATL editor, it would need to be extended with functionality to access and modify the AST underlying the textual view and to mirror changes performed to the AST back into the textual view. Furthermore, functionality to highlight ATL code regions that are impacted is needed. This can mainly be seen as an industry project.

Operator application extension The current set of operators covers the EMOF metamodel of MOF and thereby Ecore as an implementation. The completeness for EMOF was discussed in Section 9.1. The operators could be extended to cover all of MOF, i.e. to also include the CMOF metamodel. This would make them potentially more widely applicable and introduce the possibility of providing impact detection and resolution for a larger variety of transformation languages overall. The kind of operators needed for a coverage of CMOF remains a research question.

Extend the set of operators Although operator completeness in terms of the metamodel is reached for EMOF, completeness for the transformation part is relative to the application scenario and can be improved. Which operators are needed and should sensibly be added (or are unnecessary and can be removed) requires long term experience with the tool support in a practical setting. Furthermore, the frequency of use of the operators and the occurrence of impact may also depend on the kind of metamodel and dependent transformations and the nature of the evolution taking place, so that one operator may prove more useful than another depending on the project at hand. To pinpoint such factors and their relevance would require a large-scale and long-running industrial case study. Such results would furthermore be helpful to research on other cases of MDS co-evolution.

Support for other transformation languages The approach could be adapted to other transformation languages, as this would extend the practical use of the approach and the tooling. QVT-R is a promising next candidate as it is structurally similar to ATL and some tool support for Eclipse IDE exists. Defining the impact detection and resolution for QVT-O may yield further insight, as it is an imperative and not a declarative transformation language. Before a practical integration of other transformation languages can be achieved, further research on the operator impact on such languages is needed.

Integration with other co-evolution As related research covers the use of operators for other cases of MDS co-evolution, such as metamodel and model co-evolution or model and OCL co-evolution, the investigation of the applicability of the same operators in all cases holds promise. To our knowledge, no such overall approach exists so far. Such an approach could provide holistic support for the co-evolution of the MDS, by providing one overall set of operators and determining and resolving the impact on all types of dependent artefacts. For instance, after modifying a metamodel, dependent transformations and at the same time the set of models conforming to the metamodel could be updated in one step. This would allow the immediate testing of the result, as the updated transformation could be executed on the updated set of models to see the result. This is expected to require a large research project as the results of different areas need to be incorporated.

Reversing the approach The approach presented in this thesis assumes a change originates in a metamodel and a dependent model transformation needs to be updated in response. Reversing the approach may also yield a valuable result. The changes that a software architect makes to a model transformation could be recorded or also provided as operators and the metamodels of the transformation then be updated accordingly. How many use cases of this approach exist would have to be determined first but could provide further insight to the co-evolution problem.

Transformation 'bad smells' Wachsmuth suggests the use of metamodel adaptation to remove structural bad smells from architecture [107] as an extension of the code-centric bad smells suggested by Fowler for OO refactoring (see Section 4.2). Bad smells indicate parts of code that should be improved by the application of a refactoring [34]. Co-evolution operators could serve the same purpose for dependent artefacts of an MDS. In the case of metamodels and transformations, the use of our co-evolution

operators may indicate a bad smell in the structure of the MDS – as an extension of the concept of code bad smells and architecture bad smells. We see an opportunity for future research in the area of quality of MDS here.

To conclude, this thesis is a step towards addressing the co-evolution problem for MDS. Software architects are supported in co-evolution tasks and the effort to resolve impacts on transformations is reduced. Further research in the area of co-evolution for other transformation languages or co-evolution in other areas of the MDS may provide wider support for the co-evolution problem and strengthen the applicability in practice.

Appendix

APPENDIX A

Execution of Extract Superclass

Appendix A contains a full run-down in screen-shots of the application of the *Extract Superclass* operator on the extended EMF-Library Metamodel example of the application scenario introduced in Section 1.2. The screen-shots are of the prototypical implementation discussed in chapter 10.

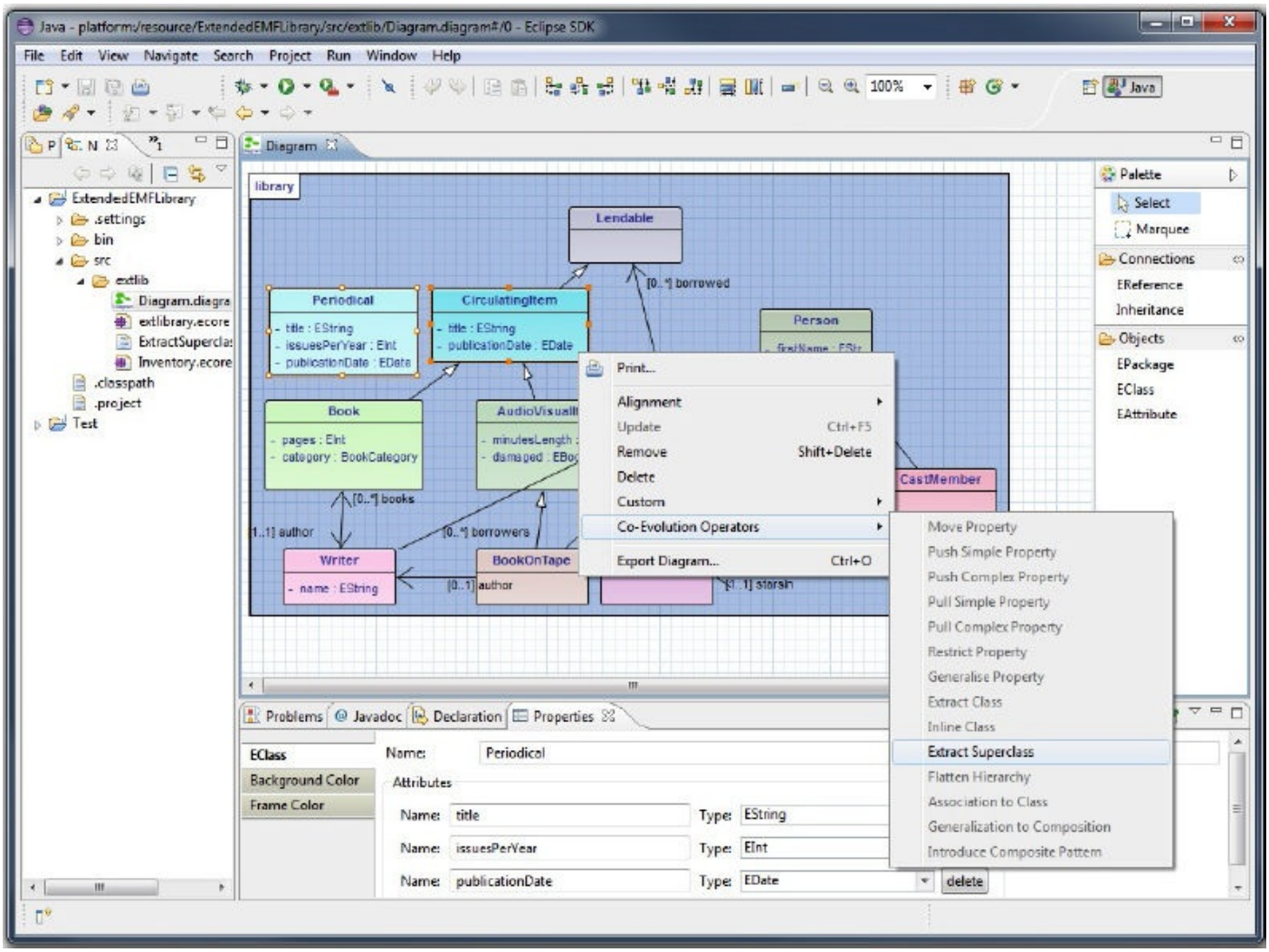


Figure A.1.: Choosing operator for selected elements

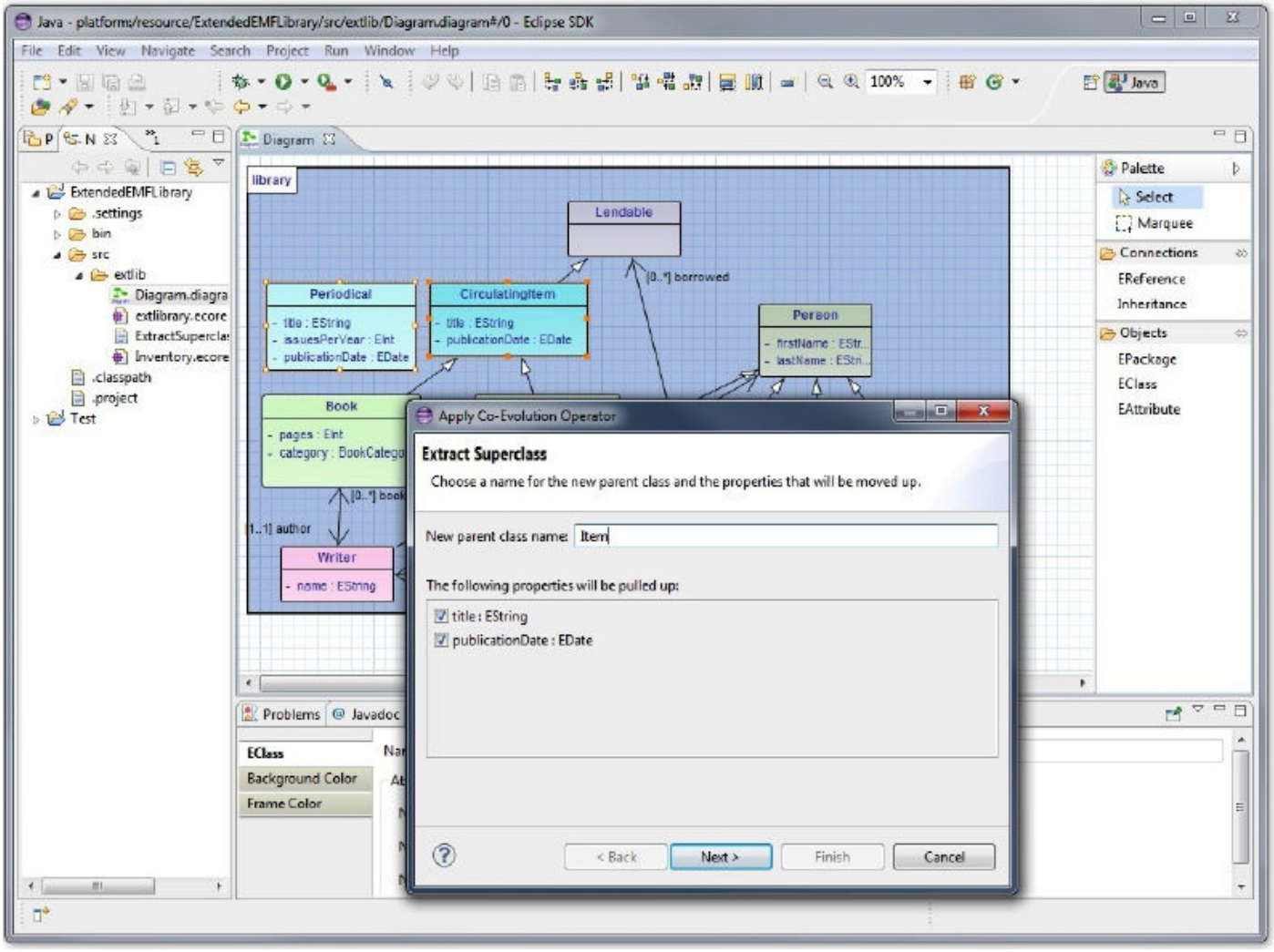


Figure A.2.: Providing additional information for operator

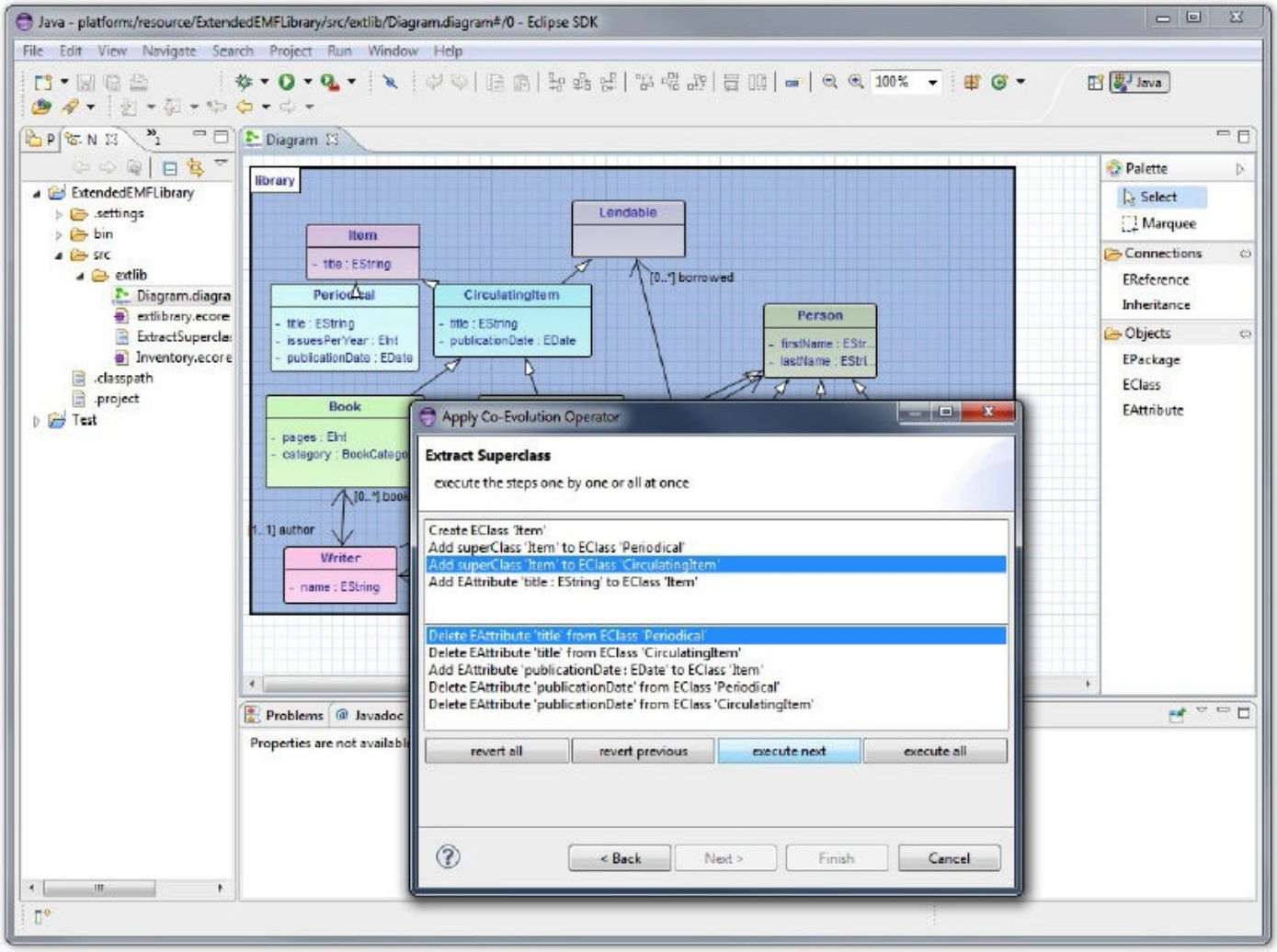


Figure A.3.: Performing individual model changes

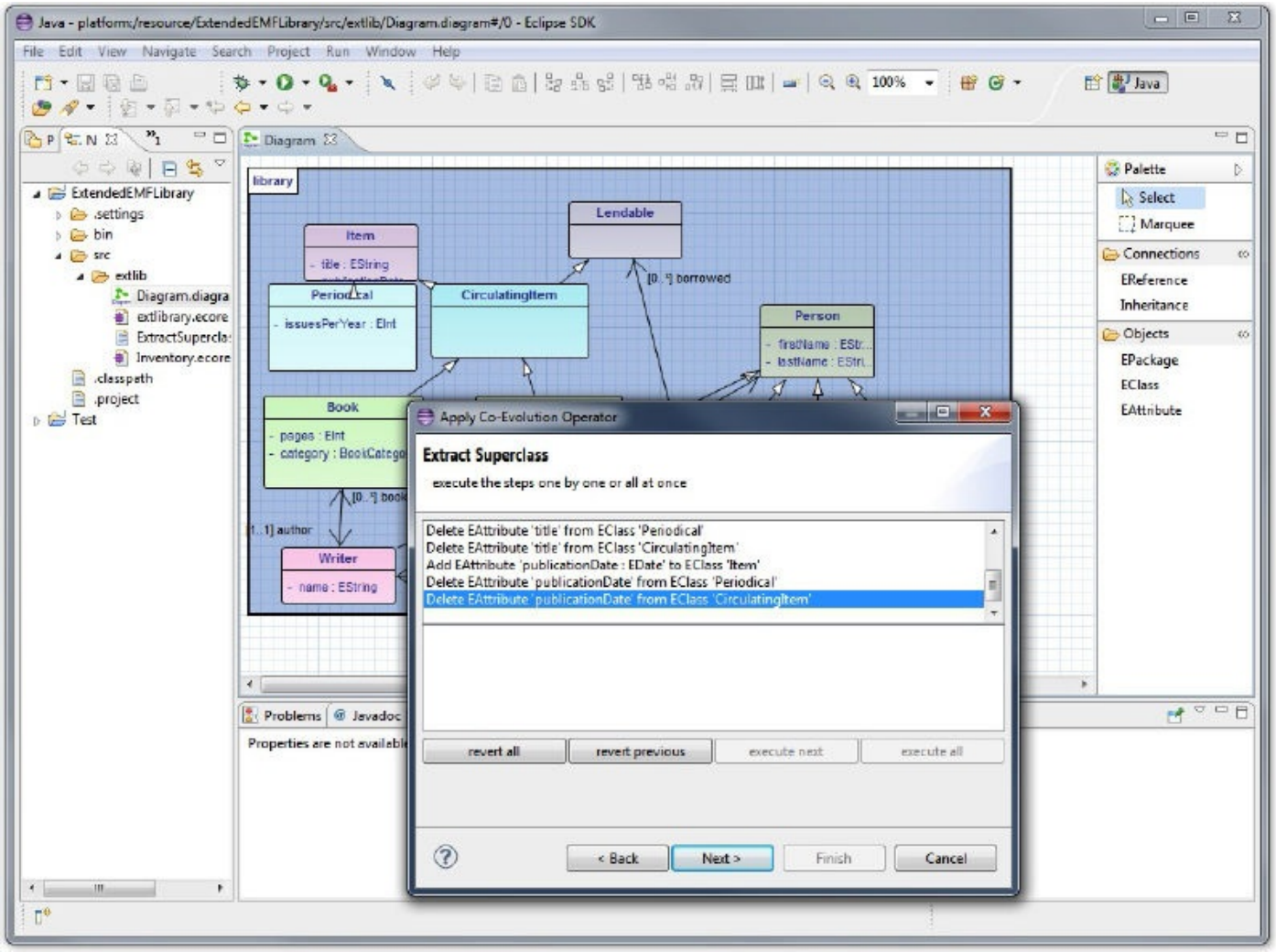


Figure A.4.: Completed set of operator changes

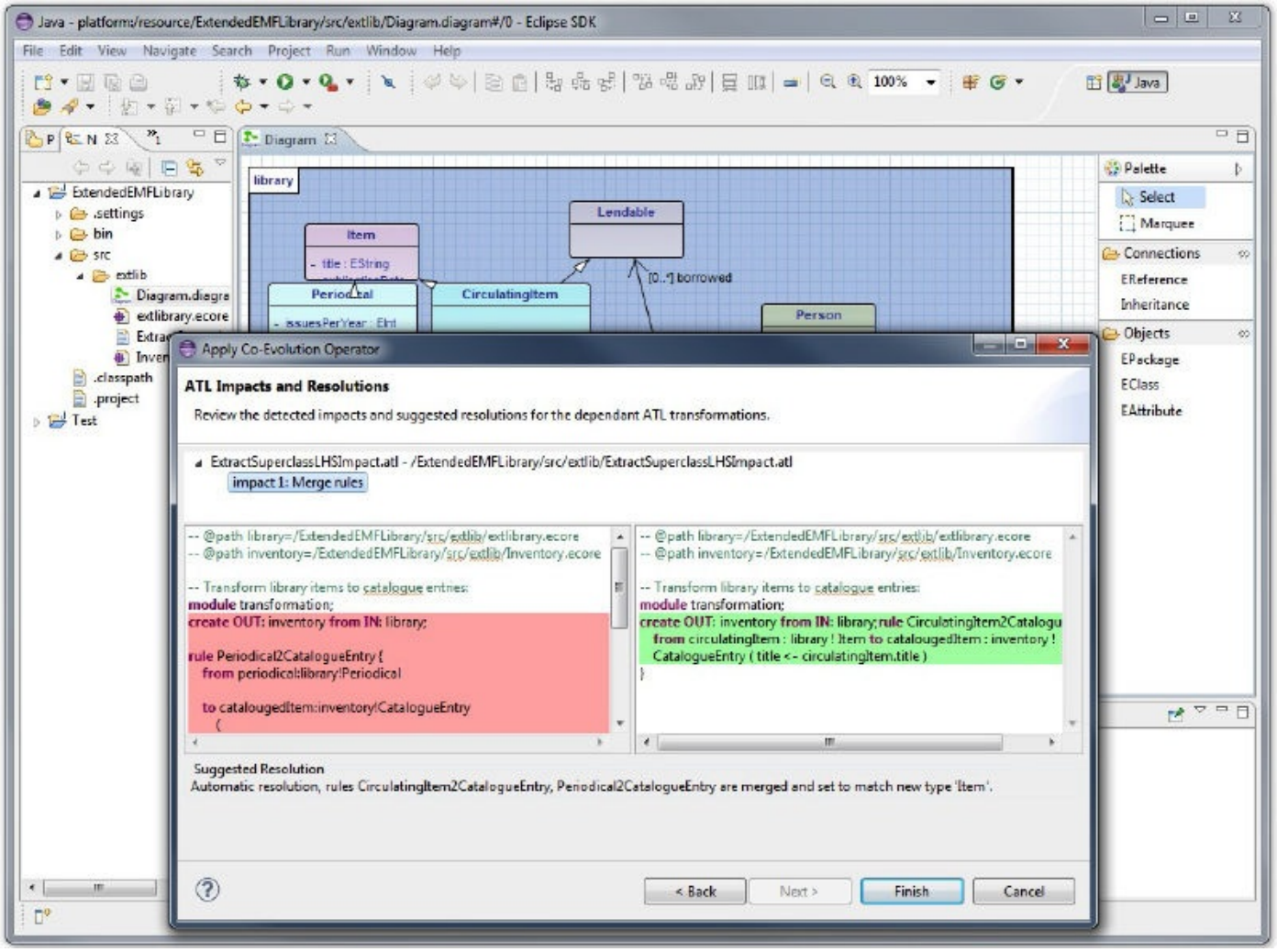


Figure A.5.: Preview of impacts and resolution

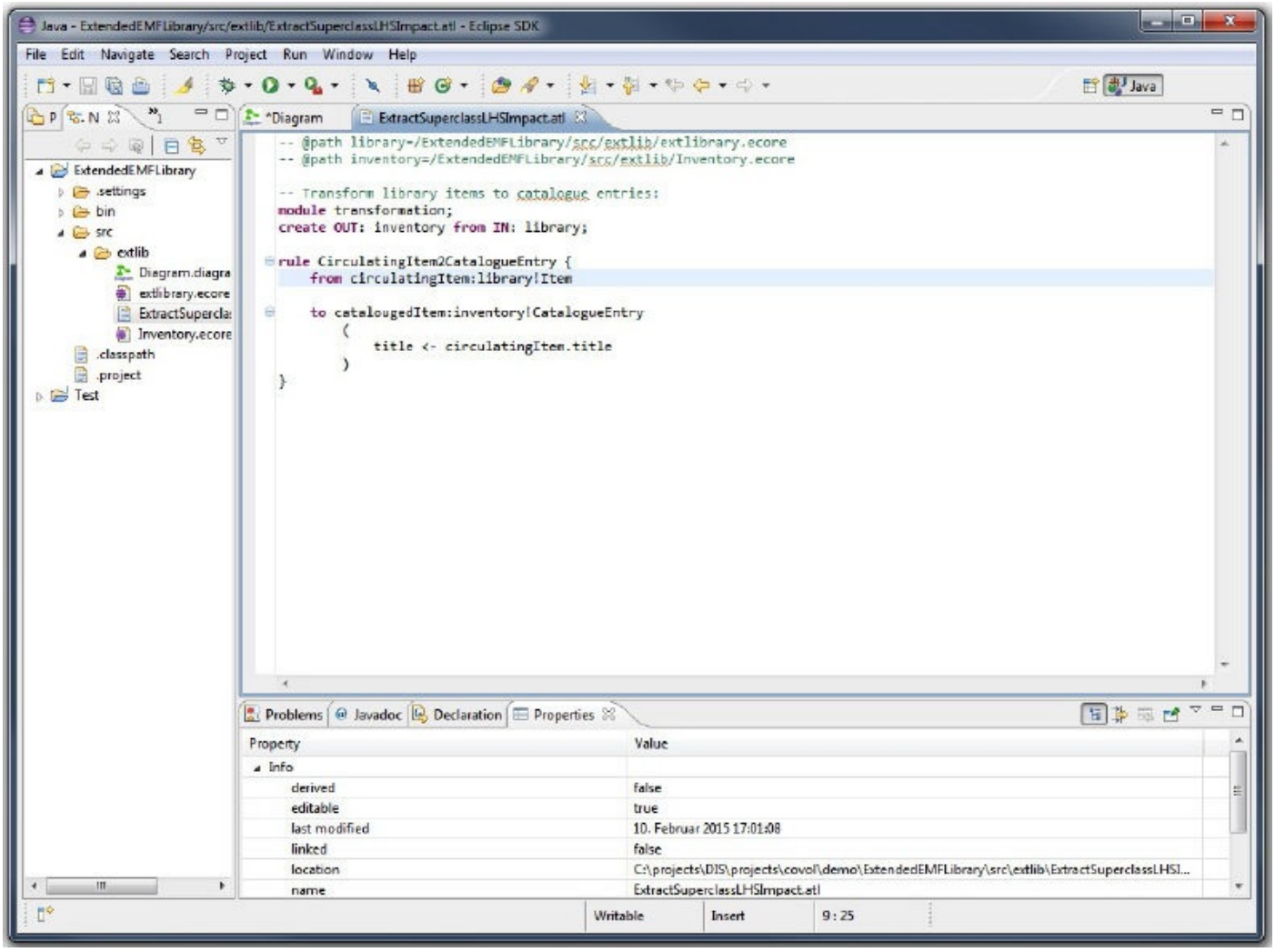


Figure A.6.: Updated model transformation after impact resolution

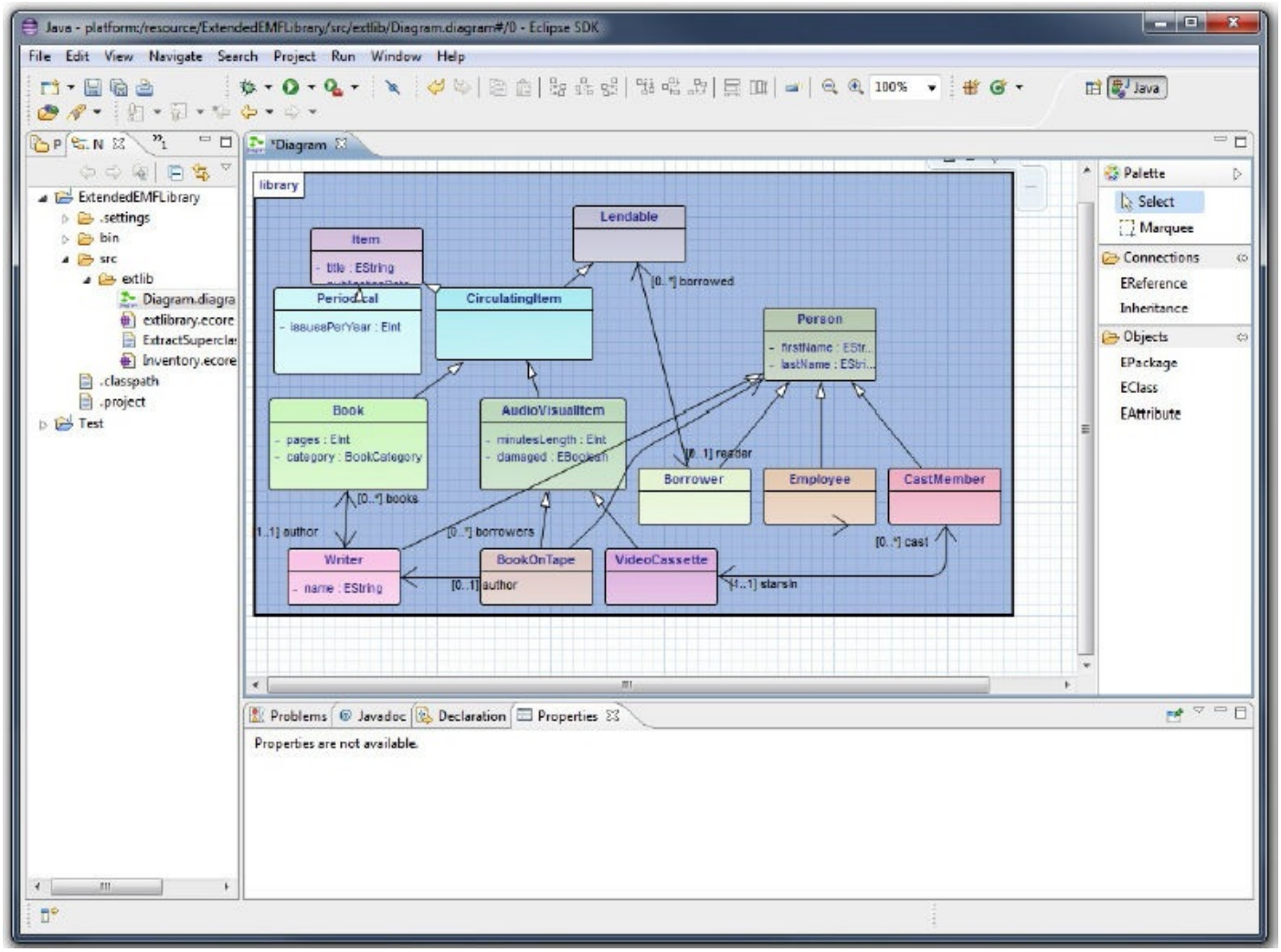


Figure A.7.: Updated model after operator application

APPENDIX B

ATL Implementation in Xtext

Appendix B contains the Eclipse Xtext Project grammar for the ATL and OCL textual syntax used to demonstrate the tooling integration as discussed in chapter 10.

Grammar of ATL

```
grammar de.covol.xtext.atl.ATL with de.covol.xtext.ocl.OCL

import
    "platform:/resource/de.covol.atl.cst/model/ATL.ecore"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "platform:/resource/de.covol.ocl.cst/model/
                        EssentialOCLCST.ecore" as ocl


/*
 * module transformation;
 * create OUT: right from IN: left;
 */
Module returns Module:
    'module'
    name=UnrestrictedName';'
    'create' 'OUT' ':' outModels+=NameExpCS ( ","
        outModels+=NameExpCS)*
    'from' 'IN' ':' inModels+=NameExpCS ( ","
        inModels+=NameExpCS)* ';'
    elements+=ModuleElement*
;

ModuleElement returns ModuleElement:
    Helper | MatchedRule | CalledRule;

MatchedRule returns MatchedRule:
'rule' name=UnrestrictedName '{'
        'from'
        inPattern=InPattern
        ('using' '{'
        variables+=RuleVariableDeclaration*
```

```
30          '}')?
31          'to'
32          outPattern=OutPattern
33          ('do' '{'
34              actionBlock=ActionBlock
35          '}'
36          )?
37      '}'
38  ;
39
40  CalledRule returns CalledRule:
41      ('entrypoint')? 'rule' name=UnrestrictedName
            ('('')')? '{'
42          ('using' '{'
43          variables+=RuleVariableDeclaration*
44          '}')?
45          'to'
46          outPattern=OutPattern
47          ('do' '{'
48              actionBlock=ActionBlock
49          '}'
50          )?
51      '}'
52  ;
53
54
55  /*
56   * helper [context context_type]? def :
          helper_name(paramters) : return_type = exp;
57   */
58  Helper returns Helper:
59      'helper' ('context' contextClassifier=ATLType)?
            definition=ATLDefCS
60  ;
61
62  /*
63   * Implementation of Complete OCL Parts for ATL.
64   * Adaption, ATL has some differences; 'context' is not
          optional in OCL,
65   * OCL DefCS requires 'static' key word.
66   */
67
68  ATLDefCS returns ATLDefCS:
69      'def' ':' varName=UnrestrictedName
70      ('('
71          (parameters+=ATLParameterCS (','
                parameters+=ATLParameterCS)*)?
72      ')')?
73      ':' type=ATLType
74      '=' initExpression=ExpCS ';'
75  ;
76
77  ATLParameterCS returns ATLParameterCS:
78      varName=UnrestrictedName ':' type=ATLType
79  ;
80
81
82  RuleVariableDeclaration returns RuleVariableDeclaration:
83      varName=UnrestrictedName ':' type=ATLType '='
            initExpression=ExpCS ';'
84  ;
```

```
85
86  /*
87   * InPattern
88   *
89   */
90  InPattern returns InPattern:
91      elements+=InPatternElement ('('
92          filter=ExpCS?
93      ')')?
94  ;
95
96  InPatternElement returns SimpleInPatternElement:
97      varName=UnrestrictedName ':' type=ATLType
98  ;
99
100
101 /*
102  * OutPattern
103  *
104  */
105 OutPattern returns OutPattern:
106     elements+=OutPatternElement (','
            elements+=OutPatternElement)*
107 ;
108
109
110 OutPatternElement returns OutPatternElement:
111     SimpleOutPatternElement
112     | ForEachOutPatternElement
113 ;
114
115
116 SimpleOutPatternElement returns SimpleOutPatternElement:
117     varName=UnrestrictedName ':' type=ATLType
118     ('('
119         (bindings+=Binding (',' bindings+=Binding)*)?
120     ')')?
121 ;
122
123
124 ForEachOutPatternElement returns ForEachOutPatternElement:
125     'xxx'
126     collection = ExpCS
127     'yyy'
128 ;
129
130
131 Binding:
132     propertyName=UnrestrictedName '<-' value=ExpCS
133 ;
134
135 ActionBlock returns ActionBlock:
136     statements+=Statement+
137 ;
138
139 Statement returns Statement:
140     BindingStat
141 ;
142
143 BindingStat returns BindingStat:
144     source=ExpCS '.' propertyName=UnrestrictedName '<-'
            value=ExpCS ';'
```

```
145 ;
146
147
148 /*************************************************/
149 /* ATL Specific Expressions
150  *
151  */
152
153 // Element to bridge ATL and EssentialOCL. This replaces
        the OCLType Element in ATL.
154 // Example: "left!AClassLeft"
155 ATLType returns ATLType:
156     (modelName = ATLModelName '!')? type = TypeExpCS
157 ;
158
159
160
161 // The variable-like name given to ecore models in ATL,
162 // as in the "right" and "left" in "create OUT: right
        from IN: left;"
163 // or "left" in the In/OutPatternElement:
        "left!AClassLeft"
164
165 ATLModelName returns ecore::EString:
166     UnrestrictedName
167 ;
168 }
```

Listing B.1: Xtext implementation of the ATL concrete and abstract syntax

Grammar of ATL-OCL

```
grammar de.covol.xtext.ocl.OCL hidden(WS, DOCUMENTATION,
    ML_COMMENT, SL_COMMENT)


import "platform:/resource/de.covol.ocl.cst/model
                                /EssentialOCLCST.ecore"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Model returns ContextCS:
    ownedExpression=ExpCS;

terminal DOUBLE_QUOTED_STRING:
    '"' ('\\' ('b' | 't' | 'n' | 'f' | 'r' | 'u' | '"' |
        "'" | '\\') | !('\\' | '"'))* '"';

terminal SINGLE_QUOTED_STRING:
    "'" ('\\' ('b' | 't' | 'n' | 'f' | 'r' | 'u' | '"' |
        "'" | '\\') | !('\\' | "'"))* "'";

terminal ML_SINGLE_QUOTED_STRING:
    "/'"->"'/";

terminal SIMPLE_ID:
    ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' |
        '_' | '0'..'9')*;

//ESCAPED_ID:
//  "_" SINGLE_QUOTED_STRING;

//ID: SIMPLE_ID | ESCAPED_ID;

terminal BIG_INT returns ecore::EBigInteger:
    ('0'..'9')+;

/*
NUMBER_LITERAL returns BigNumber: // Not terminal to
    allow parser backtracking to sort out "5..7"
    INT ('.' INT)? (('e' | 'E') ('+' | '-')? INT)?;
 */

terminal DOCUMENTATION:
    '/**'->'*/';

terminal ML_COMMENT:
    '/*' !('*')->'*/';

terminal SL_COMMENT:
    '--' !('\n' | '\r')* ('\r'? '\n')?;

terminal WS:
    (' ' | '\t' | '\r' | '\n')+;

terminal ANY_OTHER:
    .;

EssentialOCLReservedKeyword:
    'and'
    | 'else'
    | 'endif'
```

```
55      | 'if'
56      | 'implies'
57      | 'in'
58      | 'let'
59      | 'not'
60      | 'or'
61      | 'then'
62      | 'xor';
63
64
65  EssentialOCLPrefixOperator:
66      '-' | 'not';
67
68  EssentialOCLInfixOperator:
69      '*' | '/' | '+' | '-' | '>' | '<' | '>=' | '<=' | '='
            | '<>' | 'and' | 'or' | 'xor' | 'implies';
70
71  EssentialOCLNavigationOperator:
72      '.' | '->';
73
74  StringLiteral:
75      SINGLE_QUOTED_STRING;
76
77  PrefixOperator: // Intended to be overridden
78      EssentialOCLPrefixOperator;
79
80  InfixOperator: // Intended to be overridden
81      EssentialOCLInfixOperator;
82
83  NavigationOperator: // Intended to be overridden
84      EssentialOCLNavigationOperator;
85
86  //------------------------------------------------------
87  //  Names
88  //------------------------------------------------------
89
90  EssentialOCLUnrestrictedName returns ecore::EString:
91      SIMPLE_ID;
92
93  UnrestrictedName returns ecore::EString: // Intended to
        be overridden
94      EssentialOCLUnrestrictedName;
95
96  EssentialOCLUnreservedName returns ecore::EString:
97      UnrestrictedName
98  |   CollectionTypeIdentifier
99  |   PrimitiveTypeIdentifier
100 |   'Tuple'
101 ;
102
103 UnreservedName returns ecore::EString: // Intended to be
        overridden
104     EssentialOCLUnreservedName;
105
106
107 //------------------------------------------------------
108 //  Types
109 //------------------------------------------------------
110 PrimitiveTypeIdentifier:
111     'Boolean'
112     | 'Integer'
```

```
113     | 'Real'
114     | 'String'
115     | 'UnlimitedNatural'
116     | 'OclAny'
117     | 'OclInvalid'
118     | 'OclVoid';
119
120 PrimitiveTypeCS returns PrimitiveTypeRefCS:
121     name=PrimitiveTypeIdentifier;
122
123 CollectionTypeIdentifier returns ecore::EString:
124     'Set'
125     | 'Bag'
126     | 'Sequence'
127     | 'Collection'
128     | 'OrderedSet';
129
130 CollectionTypeCS returns CollectionTypeCS:
131     name=CollectionTypeIdentifier
132     ( ('(' ownedType=TypeExpCS ')')
133     | ('<' ownedType=TypeExpCS '>')
134     )?;
135
136 TupleTypeCS returns TupleTypeCS:
137     name='Tuple'
138     ( ('(' (ownedParts+=tuplePartCS (','
            ownedParts+=tuplePartCS)*)? ')')
139     | ('<' (ownedParts+=tuplePartCS (','
            ownedParts+=tuplePartCS)*)? '>')
140     )?;
141
142 tuplePartCS returns TuplePartCS:
143     name=UnrestrictedName ':' ownedType=TypeExpCS;
144
145 //-------------------------------------------------
146 //  Literals
147 //-------------------------------------------------
148 CollectionLiteralExpCS returns CollectionLiteralExpCS:
149     ownedType=CollectionTypeCS
150     '{' (ownedParts+=CollectionLiteralPartCS
151     (',' ownedParts+=CollectionLiteralPartCS)*)?
152     '}';
153
154 CollectionLiteralPartCS returns CollectionLiteralPartCS:
155     expressionCS=ExpCS ('..' lastExpressionCS=ExpCS)?;
156
157 PrimitiveLiteralExpCS returns PrimitiveLiteralExpCS:
158     NumberLiteralExpCS
159     | StringLiteralExpCS
160     | BooleanLiteralExpCS
161     | UnlimitedNaturalLiteralExpCS
162     | InvalidLiteralExpCS
163     | NullLiteralExpCS;
164
165 TupleLiteralExpCS returns TupleLiteralExpCS:
166     'Tuple' '{' ownedParts+=TupleLiteralPartCS (','
            ownedParts+=TupleLiteralPartCS)* '}';
167
168 TupleLiteralPartCS returns TupleLiteralPartCS:
169     name=UnrestrictedName (':' ownedType=TypeExpCS)? '='
            initExpression=ExpCS;
```

```
170
171 NumberLiteralExpCS returns NumberLiteralExpCS:
172     //name=NUMBER_LITERAL;
173     name=BIG_INT;
174
175 StringLiteralExpCS returns StringLiteralExpCS:
176     name=StringLiteral;
177
178 BooleanLiteralExpCS returns BooleanLiteralExpCS:
179     name='true'
180     | name='false';
181
182 UnlimitedNaturalLiteralExpCS returns
        UnlimitedNaturalLiteralExpCS:
183     {UnlimitedNaturalLiteralExpCS} '*';
184
185 InvalidLiteralExpCS returns InvalidLiteralExpCS:
186     {InvalidLiteralExpCS} 'invalid';
187
188 NullLiteralExpCS returns NullLiteralExpCS:
189     {NullLiteralExpCS} 'null';
190
191 TypeLiteralCS returns TypedRefCS:
192     PrimitiveTypeCS
193     | CollectionTypeCS
194     | TupleTypeCS;
195
196 TypeLiteralExpCS returns TypeLiteralExpCS:
197     ownedType=TypeLiteralCS;
198
199 TypeNameExpCS returns TypeNameExpCS:
200     (((namespace+=UnrestrictedName '::'
            (namespace+=UnreservedName '::')*)
201     element=UnreservedName)
202     | element=UnrestrictedName);
203
204 TypeExpCS returns TypedRefCS:
205     TypeNameExpCS
206     | TypeLiteralCS;
207
208 //-------------------------------------------------
209 //  Expressions
210 //-------------------------------------------------
211 // An OclExpressionCS comprising one or more LetExpCS
212 // is kept separate to ensure that let is right
213 // associative, whereas infix operators are left
        associative.
214 // a = 64 / 16 / let b : Integer in 8 / let c : Integer
        in 4
215 // is
216 // a = (64 / 16) /
217 //(let b : Integer in 8 / (let c : Integer in 4 ))
218 ExpCS returns ExpCS:
219     InfixedExpCS;
220
221 InfixedExpCS returns ExpCS:
222     PrefixedExpCS
223     ({InfixExpCS.ownedExpression+=current}
            (ownedOperator+=BinaryOperatorCS
            ownedExpression+=PrefixedExpCS)+)?;
224
```

```
225 BinaryOperatorCS returns BinaryOperatorCS:
226     InfixOperatorCS | NavigationOperatorCS;
227
228 InfixOperatorCS returns BinaryOperatorCS:
229     name=InfixOperator;
230
231 NavigationOperatorCS returns NavigationOperatorCS:
232     name=NavigationOperator;
233
234 PrefixedExpCS returns ExpCS:
235     PrimaryExpCS
236     | ({PrefixExpCS} ownedOperator+=UnaryOperatorCS+
            ownedExpression=PrimaryExpCS);
237
238 UnaryOperatorCS returns UnaryOperatorCS:
239     name=PrefixOperator;
240
241 PrimaryExpCS returns ExpCS:
242     NavigatingExpCS
243     | SelfExpCS
244     | PrimitiveLiteralExpCS
245     | TupleLiteralExpCS
246     | CollectionLiteralExpCS
247     | TypeLiteralExpCS
248     | LetExpCS
249     | IfExpCS
250     | NestedExpCS;
251
252 NameExpCS returns NameExpCS:
253     (((namespace+=UnrestrictedName '::'
            (namespace+=UnreservedName '::')*)
254     element=UnreservedName)
255     | element=UnrestrictedName);
256
257 IndexExpCS returns NamedExpCS:
258     NameExpCS
259     ({IndexExpCS.namedExp=current}
260     '[' firstIndexes+=ExpCS (',' firstIndexes+=ExpCS)* ']'
261     ('[' secondIndexes+=ExpCS (',' secondIndexes+=ExpCS)*
            ']')?)?;
262
263 NavigatingExpCS_Base returns NamedExpCS:
264     IndexExpCS
265     //  ({PreExpCS.name=current} '@' 'pre')?    --
            defined by Complete OCL
266 ;
267
268 // For Xtext 1.0.0, this rule is very sensitive to the
        65536 byte limit, so
269 //  keep it as simple as possible and avoid backtracking.
270 NavigatingExpCS returns NamedExpCS:
271     NavigatingExpCS_Base
272     ({NavigatingExpCS.namedExp=current}
273     '(' (argument+=NavigatingArgCS
            (argument+=NavigatingCommaArgCS)*
274     (argument+=NavigatingSemiArgCS
            (argument+=NavigatingCommaArgCS)*)?
275     (argument+=NavigatingBarArgCS
            (argument+=NavigatingCommaArgCS)*)?)?
276     ')')?;
277
```

```
278 NavigatingArgCS returns NavigatingArgCS:
279     name=NavigatingArgExpCS (':' ownedType=TypeExpCS)?
            ('=' init=ExpCS)?;
280
281 NavigatingBarArgCS returns NavigatingArgCS:
282     prefix='|' name=NavigatingArgExpCS (':'
            ownedType=TypeExpCS)? ('=' init=ExpCS)?;
283
284 NavigatingCommaArgCS returns NavigatingArgCS:
285     prefix=',' name=NavigatingArgExpCS (':'
            ownedType=TypeExpCS)? ('=' init=ExpCS)?;
286
287 NavigatingSemiArgCS returns NavigatingArgCS:
288     prefix=';' name=NavigatingArgExpCS (':'
            ownedType=TypeExpCS)? ('=' init=ExpCS)?;
289
290 NavigatingArgExpCS returns ExpCS: // Intended to be
        overridden
291     ExpCS
292     //  '?' -- defined by Complete OCL
293 ;
294
295 IfExpCS returns IfExpCS:
296     'if' condition=ExpCS
297     'then' thenExpression=ExpCS
298     'else' elseExpression=ExpCS
299     'endif';
300
301 LetExpCS returns LetExpCS:
302     'let' variable+=LetVariableCS (','
            variable+=LetVariableCS)*
303     'in' in=ExpCS;
304
305 LetVariableCS returns LetVariableCS:
306     name=UnrestrictedName (':' ownedType=TypeExpCS)? '='
            initExpression=ExpCS;
307
308 NestedExpCS returns NestedExpCS:
309     '(' source=ExpCS ')';
310
311 SelfExpCS returns SelfExpCS:
312     {SelfExpCS} 'self';
313 }
```

Listing B.2: Xtext implementation of the OCL concrete and abstract syntax

Glossary

ANTLR is a parser generator for textual (programming) languages [81].

ATL Transformation Zoo A collection of 103 model transformation scenarios in ATL (as of this writing) [100].

co-evolution the synchronized evolution of linked artefacts to maintain the consistency of the MDS. The *co-evolution problem* refers to inconsistencies that may occur when artefacts are changed individually.

conformsTo The relationship between a model and its metamodel.

consistency In the case of metamodels and model transformations, consistency is the suitability of the metamodel to represent the input and output models of the model transformation.

Eclipse ATL Project The Eclipse IDE project for the ATL language implementation and transformation engine [95].

Eclipse Graphiti Project An Eclipse project that provides an extendable framework for graphical editors for Eclipse IDE [98].

Eclipse IDE Eclipse is an integrated development environment (IDE) for Java and other programming languages and DSLs. It provides a pluin infrastructure and plug-ins for many different development purposes, among them tooling for MDE projects.

Eclipse Modeling Project The Eclipse IDE project for MDE related technologies of the Eclipse platform.

Eclipse Xtext Project An Eclipse IDE project for the generation of textual DSLs and accompanying tooling [102].

Ecore Ecore is an implementation of the Essential MOF (EMOF) for Java. It is part of the Eclipse Modeling Framework (EMF).

EMF Genmodel A type of model used in the EMF to customise the code generation of Java source code from Ecore models [94].

grammarware engineering Engineering discipline for the development and maintenance of grammars and grammar-dependent software, where *grammar* ‘is used in the sense of all established grammar formalisms and grammar notations including context-free grammars, class dictionaries, XML schemas as well as some forms of tree and graph grammars’ [53].

higher-order transformation Higher-order transformations represent transformation functions with transformations as input or output models.

metametamodel The representation of a language designed for metamodelling.

metamodel A metamodel is a representation of a set of models, which can be called a modelling language. If a model is a member of this set, it conforms to its metamodel.

model An abstract system is a model of another system, if it is a simplification of the other system, created for a given purpose.

Model Driven System A *Model Driven System (MDS)* is the collection of artefacts and tools used for the development of a software system in MDE and the dependencies between them.

Model Transformation A model transformation is the information on a mapping between two sets of models, to be executed by a transformation engine. It consists of a set of rules that describe how elements of the target language can be created from elements of the source language automatically.

Object Constraint Language A formal textual language to describe invariant conditions or queries for model elements of the OMG family of languages.

operator A mapping from one set of metamodel elements to another set of metamodel elements. For the purpose of co-evolution, the operator describes how metamodel elements are adapted to perform a predefined type of change.

real-world software system A software system that provides a solution to an acceptable approximation of a real-world problem or that it is embedded in the real world as a universe of discourse. Also called *E-type system* by Lehmann [61].

refactoring ‘Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure’ [34, p. xvi].

representationOf ‘The relationship between the model and the system it represents.

reusability Reusability is the property or properties of a development artefact that enhance its suitability for reuse.

system The ‘combination of interacting elements organized to achieve one or more stated purposes’[47].

technological space ‘A technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities.’ [58].

Unified Modeling Language A modelling language for software systems.

wizard A usage interface dialog consisting of a number of different pages to perform an overall task.

Acronyms

AMMA	ATLAS Model Management Architecture.
ANSI	American National Standards Institute.
AST	abstract syntax tree.
ATL	ATLAS Transformation Language.
ATL VM	ATL Virtual Machine.
CDIF	CASE data interchange format.
CMOF	Complete MOF.
DSL	Domain Specific Language.
DSM	Domain Specific Modelling.
EMF	Eclipse Modeling Framework.
EMOF	Essential MOF.
GReAT	Graph Rewriting and Transformation.
IDE	integrated development environment.
IMDB	Internet Movie Database.
IRDS	Information Resource Dictionary System.
LHS	left-hand-side.
MDE	Model Driven Engineering.
MDS	Model Driven System.
MOF	Meta-Object Facility.
OCL	Object Constraint Language.
OMG	Object Management Group.
OO	object-oriented.
QVT	Meta Object Facility (MOF) 2.0 Query/View/ Transformation.
QVT-C	QVT Core.
QVT-O	QVT Operational Mappings.
QVT-R	QVT Relations.
RHS	right-hand-side.

UML	Unified Modeling Language.
URI	Uniform Resource Identifier.
XMI	XML Metadata Interchange.
XML	Extensible Markup Language.

List of Figures

List of Listings

Index

Bibliography

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Pearson Addison Wesley, 2007. isbn: 0-321-48681-1.

[2] Carsten Amelunxen and Andy Schürr. 'Formalising model transformation rules for UML/MOF 2'. In: *IET Software* 2.3 (2008), pp. 204–222. ISSN: 1751-8806. DOI: 10.1049/iet-sen:20070076.

[3] Dave Astels. 'Refactoring with UML'. In: *Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering* (2002), pp. 67–70.

[4] Colin Atkinson and Thomas Kühne. 'Model-Driven Development: A Metamodeling Foundation'. In: *Software, IEEE* 20.5 (2003), pp. 36–41. DOI: 10.1109/MS.2003.1231149.

[5] ATLAS group LINA & INRIA Nantes. *ATL: Atlas Transformation Language, ATL User Manual - version 0.7 -*. 2006. URL: http://www.eclipse.org/atl/documentation/old/ATL_User_Manual[v0.7].pdf (visited on 01/02/2010).

[6] ATLAS group LINA & INRIA Nantes. *ATL: Atlas Transformation Language, Specification of the ATL Virtual Machine - version 0.1 -*. 2005. URL: http://www.eclipse.org/atl/documentation/old/ATL_VMSpecification%5Bv00.01%5D.pdf (visited on 01/02/2010).

[7] Thomas Baar and Slaviša Marković. 'A Graphical Approach to Prove the Semantic Preservation of UML/OCL Refactoring Rules'. In: *Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference, PSI 2006, Novosibirsk, Russia, June 27-30, 2006. Revised Papers*. Lecture Notes in Computer Science 4378 (2007). Ed. by Irina Virbitskaite and Andrei Voronkov, pp. 70–83. DOI: 10. 1007/978-3-540-70881-0_9.

[8] Daniel Balasubramanian et al. 'The Graph Rewriting and Transformation Language: GReAT'. In: *Electronic Communications of the EASST - Proceedings of the Third International Workshop on Graph Based Tools (GraBaTs 2006)* 1 (2007). issn: 1863-2122. URL: http://www.isis.vanderbilt.edu/node/3505.

[9] Mikaël Barbero, Frédéric Jouault and Jean Bézivin. 'Model Driven Management of Complex Systems: Implementing the Macroscope's Vision'. In: *15th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*. ECBS '08. Washington, DC: IEEE Computer Society, 2008, pp. 277 – 286. DOI: 10.1109/ECBS.2008.42.

[10] Keith H. Bennett and Václav T. Rajlich. 'Software Maintenance and Eevolution: A Roadmap'. In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE '00. New York, NY: ACM, 2000, pp. 73–87. DOI: 10.1145/336512.336534.

[11] Jean Bézivin. 'From Object Composition to Model Transformation with the MDA'. In: *TOOLS '01 Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)* I (Aug. 2001), p. 350. DOI: 10. 1109/TOOLS.2001.10021.

[12] Jean Bézivin. 'On the Unification Power of Models'. In: *Software and Systems Modeling* 4.2 (2005), pp. 171–188. issn: 1619-1374. DOI: 10.1007/s10270-005-0079-0.

[13] 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 2001)* (Nov. 2001), pp. 273–280. DOI: 10.1109/ASE.2001.989813.

[14] Marko Boger, Thorsten Sturm and Per Fragemann. 'Refactoring Browser for UML'. In: *Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays, NODe 2002 Erfurt, Germany, October 7–10, 2002 Revised Papers*. Lecture Notes in Computer Science 2591 (2003). Ed. by Mehmet Aksit, Mira Mezini and Rainer Unland, pp. 366–377. DOI: 10.1007/3-540-36557-5_26.

[15] Grady Booch, James Rumbaugh and Ivar Jacobson. *Unified Modeling Language User Guide, The (2Nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005. ISBN: 0321267974.

[16] Frederick P. Brooks. 'No Silver Bullet — Essence and Accident in Software Engineering'. In: *Proceedings of the IFIP Tenth World Computing Conference*. Ed. by H.-J. Kugler. Amsterdam: Elsevier Science B.V., 1986, pp. 1069–1076.

[17] Manfred Broy et al. '2nd UML 2 Semantics Symposium: Formal Semantics for UML'. In: *Models in Software Engineering*. Ed. by Thomas Kühne. Vol. 4364. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 318–323. ISBN: 978-3-540-69488-5. DOI: 10.1007/978-3-540-69489-2_39.

[18] Achim D. Brucker, Jürgen Doser and Burkhart Wolff. ‘Semantic Issues of OCL : Past, Present, and Future’. In: *Electronic Communications of the EASST* 5 (2006). ISSN: 1863-2122.

[19] Erik Burger et al. ‘View-based Model-driven Software Development with ModelJoin’. In: *Software and Systems Modeling* (2014), pp. 1–24. DOI: 10.1007/s10270-014-0413-5.

[20] Jordi Cabot and E Teniente. ‘Transformation Techniques for OCL Constraints’. In: *Science of Computer Programming, Special Issue on Model Transformation* 68.3 (2007). Ed. by Alfonso Pierantonio, B. Selic Vallecillo and J. Gray, pp. 179–195. ISSN: 0167-6423. DOI: 10.1016/j.scico.2007.05.001.

[21] Antonio Cicchetti, Davide Di Ruscio and Alfonso Pierantonio. ‘A Metamodel Independent Approach to Difference Representation’. In: *Journal of Object Technology* 6.9 (2007), pp. 165–185. URL: http://www.jot.fm/issues/issues_2007_10/paper9.

[22] Antonio Cicchetti, Davide Di Ruscio and Alfonso Pierantonio. ‘Managing Dependent Changes in Coupled Evolution’. In: *Theory and Practice of Model Transformations, Second International Conference, ICMT 2009, Zurich, Switzerland, June 29-30, 2009. Proceedings*. Lecture Notes in Computer Science 5563 (2009). Ed. by Richard F. Paige, pp. 35–51. DOI: 10.1007/978-3-642-02408-5_4.

[23] Antonio Cicchetti et al. ‘Automating Co-evolution in Model-driven Engineering’. In: *Twelfth IEEE International EDOC Enterprise Computing Conference*. Los Alamitos, CA: IEEE Computer Society, 2008, pp. 222–231. doi: 10.1109/EDOC.2008.44.

[24] James Clark. *XSL Transformations (XSLT) Version 1.0*. 1999. url: http://www.w3.org/TR/1999/REC-xslt-19991116 (visited on 11/10/2013).

[25] Alexandre Correa and Cláudia Werner. ‘Refactoring Object Constraint Language Specifications’. In: *Software and Systems Modeling* 6.2 (July 2006), pp. 113–138. doi: 10.1007/s10270-006-0023-y.

[26] Krzysztof Czarnecki and Simon Helsen. ‘Classification of model transformation approaches’. In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. Citeseer, 2003, pp. 1–17.

[27] Krzysztof Czarnecki and Simon Helsen. ‘Feature-based Survey of Model Transformation Approaches’. In: *IBM Systems Journal* 45.3 (2006), pp. 621–645. issn: 0018-8670. doi: 10.1147/sj.453.0621.

[28] Gregor Engels et al. ‘Consistency-Preserving Model Evolution through Transformations’. In: *<<MML>> 2002 - The Unified Modeling Language*. Lecture Notes in Computer Science 2460 (Sept. 2002). Ed. by Jean-Marc Jézéquel, Heinrich Hussmann and Stephen Cook, pp. 212–227. doi: 10.1007/3-540-45800-X_18.

[29] Andy Evans et al. ‘The UML as a Formal Modeling Notation’. In: *The Unified Modeling Language. <<UML>> ’98: Beyond the Notation*. Lecture Notes in Computer Science 1618 (1999). Ed. by Jean Bézivin and Pierre-Alain Muller, pp. 336–348. doi: 10.1007/978-3-540- 48480-6_26.

[30] Jean-Marie Favre. ‘Megamodelling and Etymology’. In: *Transformation Techniques in Software Engineering*. Dagstuhl Seminar Proceedings 05161. Dagstuhl: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. url: http://drops.dagstuhl.de/opus/volltexte/2006/427.

[31] Jean-Marie Favre. ‘Meta-model and Model Co-evolution within the 3D Software Space’. In: *Proceedings of the ELISA Workshop on Evolution of Large-scale Industrial Software Applications* (Sept. 2003), pp. 98–109. URL: http://soft.vub.ac.be/FFSE/Workshops/ELISA-submissions/16-Favre-full.pdf.

[32] Jean-Marie Favre. ‘Towards a Basic Theory to Model Model Driven Engineering’. In: *In Proc. of the 3rd UML Workshop in Software Model Engineering (WiSME’2004)*. 2004. URL: http://www-adele.imag.fr/Les.Publications/intConferences/WISME2004Fav.pdf.

[33] Jean-Marie Favre and Tam NGuyen. ‘Towards a Megamodel to Model Software Evolution Through Transformations’. In: *Electronic Notes in Theoretical Computer Science* 127.3 (Apr. 2005), pp. 59–74. ISSN: 15710661. DOI: 10.1016/j.entcs.2004.08.034.

[34] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison Wesley Longman, Inc., 1999. ISBN:0- 201-48567-2.

[35] Erich Gamma et al. *Design Patterns: Elements of Reusable Objectoriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[36] Kelly Garcés et al. *Adaptation of Models to Evolving Metamodels*. Tech. rep. RR-6723. 2008. URL: https://hal.inria.fr/inria-00338695/document.

[37] Kelly Garcés et al. ‘Adapting Transformations to Metamodel Changes via External Transformation Composition’. In: *Software and Systems Modeling* 13.2 (2014), pp. 789–806. ISSN: 1619-1366. DOI: 10.1007/ s10270-012-0297-1.

[38] Jokin García, Oscar Diaz and Maider Azanza. ‘Model Transformation Co-evolution: A Semi-automatic Approach’. In: *Software Language Engineering - 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*. Lecture Notes in Computer Science 7745 (2013). Ed. by Krzysztof Czarnecki and Görel Hedin, pp. 144–163. ISSN: 03029743. DOI: 10.1007/978- 3-642-36089-3_9.

[39] Roxana Giandini, Claudia Pons and Gabriela Pérez. ‘A two-level formal semantics for the QVT language.’ In: *Memorias de la XII Conferencia Iberoamericana de Software Engineering (CIbSE 2009)*. Medellín, 2009, pp. 73–86.

[40] Boris Gruschko, Dimitrios S. Kolovos and Richard F. Paige. ‘Towards Synchronizing Models with Evolving Metamodels’. In: *Proceedings of the International Workshop on Model-Driven Software Evolution held with the ECSMR*. Citeseer, 2007.

[41] Esther Guerra and Juan de Lara. ‘An Algebraic Semantics for QVT-Relations Check-only Transformations’. In: *Fundamenta Informaticae* 114.1 (2012), pp. 73–101. issn: 1875-8681. doi: 10.3233/FI- 2011-618.

[42] Kahina Hassam et al. ‘Assistance System for OCL Constraints Adaptation during Metamodel Evolution’. In: *2011 15th European Conference on Software Maintenance and Reengineering* (Mar. 2011), pp. 151–160. doi: 10.1109/CSMR.2011.21.

[43] Wilhelm Hasselbring et al. ‘Projekt-orientierte Vermittlung von Entwurfsmustern in der Software Engineering Ausbildung’. In: *Engineering*. Ed. by Andreas Zeller and Marcus Deininger. dpunkt Verlag, 2007, pp. 45–58.

[44] Markus Herrmannsdoerfer, Sebastian Benz and Elmar Juergens. ‘COPE - Automating Coupled Evolution of Metamodels and Models’. In: *ECOOP 2009 – Object-Oriented Programming*. Ed. by Sophia Drossopoulou. Vol. 5653. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 52–76. DOI: 10 . 1007 / 978 - 3 - 642 - 03013-0_4.

[45] Markus Herrmannsdoerfer et al. ‘An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models’. In: *Software Language Engineering, Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*. Springer-Verlag, 2010, pp. 1–20.

[46] Inria. *PathExpression to PetriNet & PetriNet to PathExpression*. 2005. url: http://www.eclipse.org/atl/atlTransformations/PathExp2PetriNet/ExamplePathExp2PetriNet[v00.01].p (visited on 23/02/2015).

[47] *ISO/IEC/IEEE 15288:2008 Systems and Software Engineering - System Life Cycle Processes*. Tech. rep. Geneva, Switzerland: International Organization for Standardization (ISO)/International Electrotechnical Commission (IEC), Institute of Electrical and Electronics Engineers, 2008.

[48] *ISO/IEC/IEEE Std 42010:2011 – Systems and software engineering – Architecture description*. Los Alamitos,CA: IEEE, 2011.

[49] Frédéric Jouault et al. ‘ATL: a QVT-like Transformation Language’. In: *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006*. Portland, Oregon: ACM Press, 2006, pp. 719–720. DOI: 10.1145/1176617.1176691.

[50] Lucia Kapová et al. ‘Evaluating Maintainability with Code Metrics for Model-to-Model Transformations’. In: *Research into Practice – Reality and Gaps*. Ed. by George T. Heineman, Jan Kofron and Frantisek Plasil. Lecture Notes in Computer Science. Prague: Springer Berlin Heidelberg, 2010, pp. 151–166. DOI: 10.1007/978- 3-642-13821-8_12.

[51] Stuart Kent. ‘Model Driven Engineering’. In: *Integrated Formal Methods, Third International Conference, IFM 2002 Turku, Finland, May 15–18, 2002 Proceedings*. Lecture Notes in Computer Science 2335.2 (Apr. 2002). Ed. by Michael Butler, Luigia Petre and Kaisa Sere, pp. 286–298. DOI: 10.1007/3-540-47884-1_16.

[52] Anneke G Kleppe, Jos Warmer and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 032119442X.

[53] Paul Klint, Ralf Lämmel and Chris Verhoef. ‘Toward an Engineering Discipline for Grammarware’. In: *ACM Transactions on Software Engineering and Methodology* 14 (2005), pp. 331–380. DOI: 10.1145/1072997.1073000.

[54] Maximilian Koegel et al. ‘Comparing State- and Operation-Based Change Tracking on Models’. In: *2010 14th IEEE International Enterprise Distributed Object Computing Conference* (Oct. 2010), pp. 163–172. DOI: 10.1109/EDOC.2010.15.

[55] Dimitrios S. Kolovos et al. ‘Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing’. In: *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models* (2009), pp. 1–6. doi: 10.1109/CVSM.2009.5071714.

[56] Thomas Kühne. ‘Matters of (Meta-) Modeling’. In: *Software and Systems Modeling* 5.4 (July 2006), pp. 369–385. issn: 1619-1366. doi: 10.1007/s10270-006-0017-9.

[57] Thomas Kühne et al. ‘Explicit Transformation Modeling’. In: *Proceedings of the 2009 International Conference on Models in Software Engineering*. MODELS’09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 240–255. doi: 10.1007/978-3-642-12261-3_23.

[58] Ivan Kurtev, Jean Bézivin and Mehmet Aksit. ‘Technological Spaces: An Initial Appraisal’. In: *International Conference on Cooperative Information Systems (CoopIS), DOA’2002 Federated Conferences, Industrial Track* (2002), pp. 1–6.

[59] Juan de Lara and Esther Guerra. ‘Formal Support for QVT-Relations with Coloured Petri Nets’. In: *Model Driven Engineering Languages and Systems*. Ed. by Andy Schürr and Brian Selic. Vol. 5795. Lecture Notes in Computer Science. Berlin Heidelberg: Springer Berlin Heidelberg, 2009, pp. 256–270. isbn: 978-3-642-04425-0. doi: 10.1007/978-3-642-04425-0_19.

[60] Meir M. Lehman. ‘Laws of Software Evolution Revisited’. In: *Software Process Technology 5th European Workshop, EWSPT ’96 Nancy, France, October 9–11, 1996 Proceedings*. Lecture Notes in Computer Science 1149 (1996). Ed. by Carlo Montangero, pp. 108–124. doi: 10.1007/BFb0017737.

[61] Meir M. Lehman. ‘Programs, Life Cycles, and Laws of Software Evolution’. In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076. doi: 10.1109/PROC.1980.11805.

[62] Tihamer Levendovszky et al. ‘A Novel Approach to Semi-automated Evolution of DSML Model Transformation’. In: *Software Language Engineering*. Lecture Notes in Computer Science 5969 (2010). Ed. by Mark van den Brand, Dragen Gašević and Jeff Gray, pp. 23–41. doi: 10.1007/978-3-642-12107-4_4.

[63] Ernst Lippe and Norbert van Oosterom. ‘Operation-based Merging’. In: *Newsletter ACM SIGSOFT Software Engineering Notes* 17.5 (1992). Ed. by Peter G. Neumann, pp. 78–87. doi: 10.1145/142868.143753.

[64] Mohamed Maddeh, Mohamed Romdhani and Khaled Ghedira. ‘Classification of Model Refactoring Approaches’. In: *Journal of Object Technology* 8.6 (Sept. 2009), pp. 121–126. issn: 1660-1769. DOI: 10.5381/jot.2009.8.6.a3.

[65] Slaviša Marković and Thomas Baar. ‘Refactoring OCL Annotated UML Class Diagrams’. In: *Software and Systems Modeling (SoSyM)* 7.1 (2008), pp. 25–47. doi: 10.1007/s10270-007-0056-x.

[66] David Méndez et al. ‘Towards Transformation Migration After Metamodel Evolution’. In: *Model and Evolution Workshop*. inria-00524145. Oslo, 2010, pp. 1–6. url: https://hal.inria.fr/inria-00524145.

[67] Tom Mens, Gabriele Taentzer and Dirk Müller. ‘Challenges in Model Refactoring’. In: *1st Workshop on Refactoring Tools*. Vol. 98. University of Berlin, 2007.

[68] Tom Mens and P. Van Gorp. ‘A Taxonomy of Model Transformation’. In: *Electronic Notes in Theoretical Computer Science* 152 (2006), pp. 125–142. issn: 1571-0661. DOI: doi : 10 . 1016/j . entcs . 2005 . 10.021.

[69] Tom Mens et al. ‘Challenges in software evolution’. In: *Computer* April (2005). issn: 1550-4077. url: http://www.computer.org/portal/web/csdl/doi/10.1109/IWPSE.2005.7.

[70] Tom Mens et al. ‘Formalizing Refactorings with Graph Transformations’. In: *Journal of Software Maintenance and Evolution: Research and Practice* 17.4 (2005), pp. 247–276. doi: 10.1002/smr.316.

[71] Parastoo Mohagheghi et al. ‘MDE Adoption in Industry: Challenges and Success Criteria’. In: *Models in Software Engineering*. Lecture Notes in Computer Science 5421 (2009). Ed. by Michel R. V. Chaudron, pp. 54–59. doi: 10.1007/978-3-642-01648-6_6.

[72] Tien N. Nguyen et al. ‘An Infrastructure for Development of Objectoriented, Multi-level Configuration Management Services’. In: *Proceedings of the 27th international conference on Software engineering - ICSE ’05*. New York, NY: ACM Press, 2005, pp. 215–224. isbn: 1595939632. doi: 10.1145/1062455.1062504.

[73] Object Management Group OMG. *Meta Object Facility (MOF) 2.0 Query/ View/ Transformation Specification Version 1.1 - January 2011*. 2011.

[74] Object Management Group OMG. *OCL : Object Constraint Language Version 2.3.1*. 2012.

[75] Object Management Group OMG. *OMG Meta Object Facility (MOF) Core Specification Version 2.4.1*. 2013. URL: http://www.omg.org/spec/MOF/2.4.1/.

[76] Object Management Group OMG. *OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.4.1*. Aug. 2011. URL: http://www.omg.org/spec/UML/2.4.1/.

[77] Object Management Group OMG. *OMG Unified Modeling Language (OMG UML), Superstructure v2.4.1*. Aug. 2011. URL: http://www.omg.org/spec/UML/2.4.1/.

[78] Object Management Group OMG. *OMG Unified Modeling Language (OMG UML) Version 2.5*. Sept. 2013. URL: http://www.omg.org/spec/UML/2.5/Beta2/.

[79] Klaas Oetken. 'Übertragung von objektorientierten Refactoring-Patterns in die Software-Modellierung'. Bachelorarbeit in Informatik. Carl von Ossietzky Universität Oldenburg, 2012.

[80] WF Opdyke. 'Refactoring Object-Oriented Frameworks'. PhD thesis. University of Illinois at Urbana-Champaign, 1992, p. 197.

[81] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, 2007. ISBN: 0-9787392-5-6.

[82] Ralf Reussner and Wilhelm Hasselbring. *Handbuch der Software-Architektur*. 1st ed. Heidelberg: dPunkt.verlag, 2006. ISBN: 3-89864-372-7.

[83] Bernhard Rumpe and Robert France. 'Variability in UML Language and Semantics'. In: *Software and Systems Modeling* 10.4 (Aug. 2011), pp. 439–440. ISSN: 1619-1366. DOI: 10.1007/s10270-011-0210-3.

[84] Davide Di Ruscio, Ludovico Iovino and Alfonso Pierantonio. 'A Methodological Approach for the Coupled Evolution of Metamodels and ATL Transformations - 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings'. In: *Theory and Practice of Model Transformations*. Lecture Notes in Computer Science 7909 (2013). Ed. by Keith Duddy and Gerti Kappel, pp. 60–75. DOI: 10.1007/978-3-642-38883-5_9.

[85] Andy Schürr. 'Specification of Graph Translators with Triple Graph Grammars'. In: *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94 Herrsching, Germany, June 16–18, 1994 Proceedings*. Lecture Notes in Computer Science 903 (1995). Ed. by Ernst W. Mayr, Gunther Schmidt and Gottfried Tinhofer, pp. 1–17. DOI: 10.1007/3-540-59071-4_45.

[86] Ed Seidewitz. *UML 2.5: Specification Simplification - Presented at the Third Biannual Workshop on Eclipse Open Source Software and OMG Open Specifications*. URL: http://de.slideshare.net/seidewitz/uml-25-specification-simplification (visited on 05/10/2014).

[87] Ed Seidewitz. 'What models mean'. In: *Software, IEEE* 20.5 (2003), pp. 26–32. ISSN: 0740-7459. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1231147.

[88] S. Sendall and W. Kozaczynski. 'Model Transformation: The Heart and Soul of Model-driven Software Development'. In: *Software, IEEE* 20.5 (Sept. 2003), pp. 42–45. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231150.

[89] Ian Sommerville. *Software Engineering (7th Edition) (International Computer Science Series)*. Seventh Edition. Addison Wesley, May 2004. ISBN: 0321210263. URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0321210263.

[90] Jörn Guy Süss et al. *ATL/User Guide - The ATL Language*. last seen: 2013-10-02. The Eclipse Foundation. Nov. 2012. URL: http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language.

[91] Herbert Stachowiak. *Allgemeine Modelltheorie*. Wien, New York: Springer-Verlag, 1973. ISBN: 3-211-81106-0.

[92] Thomas Stahl et al. *Modellgetriebene Softwareentwicklung - Techniken, Engineering, Management*. 2. Auflage. Heidelberg: dpunkt.verlag, 2007. ISBN: 978-3-89864-448-8.

[93] Jim Steel and Jean-Marc Jézéquel. 'On Model Typing'. In: *Software and Systems Modeling* 6.4 (Jan. 2007), pp. 401–413. ISSN: 1619-1366. DOI: 10.1007/s10270-006-0036-6.

[94] Dave Steinberg et al. *EMF: Eclipse Modeling Framework*. Ed. by Erich Gamma, Lee Nackman and John Wiegand. 2nd ed. Addison-Wesley, 2009. ISBN: 0-321-33188-5.

[95] The Eclipse Foundation. *ATL*. 2014. URL: http://eclipse.org/atl/ (visited on 20/10/2014).

[96] The Eclipse Foundation. *ATL/Developer GuideTitle*. 2014. URL: http://wiki.eclipse.org/ATL/Developer_Guide (visited on 05/11/2014).

[97] The Eclipse Foundation. *EMF Refactor*. 2015. URL: https://www.eclipse.org/emf-refactor/ (visited on 12/01/2015).

[98] The Eclipse Foundation. *Graphiti - a Graphical Tooling Infrastructure*. 2015. URL: https://eclipse.org/graphiti/ (visited on 14/02/2015).

[99] The Eclipse Foundation. *Java Development User Guide: Refactoring Support*. 2015. URL: http://help.eclipse.org/luna/index.jsp?topic=/org.eclipse.jdt.doc.user/concepts/concept-refactoring.htm (visited on 12/01/2015).

[100] The Eclipse Foundation. *The ATL Transformations Zoo*. 2015. URL: http://www.eclipse.org/atl/atlTransformations/ (visited on 23/02/2015).

[101] The Eclipse Foundation. *The Eclipse Modeling Framework (EMF) Overview*. 2005. URL: http://help.eclipse.org/kepler/topic/org.eclipse.emf.doc/references/overview/EMF.html (visited on 13/08/2013).

[102] The Eclipse Foundation. *Xtext*. 2014. URL: http://eclipse.org/Xtext/ (visited on 20/10/2014).

[103] Massimo Tisi et al. 'On the Use of Higher-Order Model Transformations'. In: *Model Driven Architecture - Foundations and Applications 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings*. Vol. 5562. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 18–33. DOI: 10.1007/978-3-642-02674-4_3.

[104] Ragnhild Van Der Straeten, Viviane Jonckers and Tom Mens. 'Supporting Model Refactorings Through Behaviour Inheritance Consistencies'. In: *<<UML>> 2004 — The Unified Modeling Language. Modeling Languages and Applications*. Lecture Notes in Computer Science 3273 (2004). Ed. by Thomas Baar et al., pp. 305–319. DOI: 10.1007/978-3-540-30187-5_22.

[105] Ragnhild Van Der Straeten, Tom Mens and Stefan Van Baelen. 'Challenges in Model-Driven Software Engineering'. In: *Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers*. Ed. by Michel R V Chaudron. Vol. 5421. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 35–47. DOI: 10.1007/978-3-642-01648-6_4.

[106] Hans Vangheluwe. 'Invited Talk: Promises and Challenges of Model-Driven Engineering'. In: *15th European Conference on Software Maintenance and Reengineering (CSMR), 2011*. Ed. by Tom Mens, Yiannis Kanellopoulos and Andreas Winter. Los Alamitos, CA: IEEE, 2011, pp. 3–4. ISBN: 978-1-61284-259-2. DOI: 10.1109/CSMR.2011.62.

[107] Guido Wachsmuth. 'Metamodel Adaptation and Model Co-adaptation'. In: *ECOOP 2007 – Object-Oriented Programming*. Ed. by Erik Ernst. Vol. 4609. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 600–624. ISBN: 978-3-540-73588-5. DOI: 10.1007/978-3-540-73589-2.

[108] Dennis Wagelaar. 'Composition Techniques for Rule-Based Model Transformation Languages'. In: *Theory and Practice of Model Transformations*. Lecture Notes in Computer Science (2008). Ed. by Antonio Vallecillo, Jeff Gray and Alfonso Pierantonio, pp. 152–167. DOI: 10.1007/978-3-540-69927-9_11.

[109] Manuel Wimmer and Gerhard Kramler. 'Bridging Grammarware and Modelware'. In: *Satellite Events at the MoDELS 2005 Conference*. Lecture Notes in Computer Science 3844 (2006). Ed. by Jean-Michel Bruel, pp. 159–168. DOI: 10.1007/11663430_17.

[110] Alanna Zito, Zinovy Diskin and Juergen Dingel. 'Package Merge in UML 2: Practice vs. Theory?' In: *Model Driven Engineering Languages and Systems*. Ed. by Oscar Nierstrasz et al. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 185–199. DOI: 10.1007/11880240_14.

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über www.dnb.de abrufbar.

Herstellung und Verlag:
BoD – Books on Demand GmbH, Norderstedt
ISBN 978-3-7392-5722-8