A Declarative Language for Bidirectional Model Consistency

Master’s Thesis of

Dominik Werle

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

Reviewer: Prof. Dr. Ralf H. Reussner
Second reviewer: Jun.-Prof. Dr.-Ing. Anne Koziorek
Advisor: Dipl.-Inform. Max E. Kramer
Second advisor: M.Sc. Michael Langhammer

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

Karlsruhe, 08. April 2016

........................................................
(Dominik Werle)
Abstract

In model-driven engineering, the system under development is described and analyzed using models. Different models provide different abstractions of the system for different purposes.

If multiple modeling languages are used, their models can contain overlapping information about the system. Then, they can become inconsistent after changes and need to be synchronized.

The complexity of synchronizing models can be reduced by specifying consistency relationships in specialized languages and by automating the synchronization based on this specification. The languages can hide complexity that is not specific to the domain of the modeling languages, for example by deriving the operations needed for propagating changes from a model to another model for all pairs of models instead of requiring explicit specification for each pair and direction.

In the course of this thesis, we designed the mapping language for specifying these consistency relationships and implemented a framework that maintains consistency based on the specification.

The mapping language allows the specification of patterns in two related modeling languages, and constraints on the relationship of the patterns. Imperative Java code is generated from the specification and executed in case of a change to a model to reestablish consistency with associated models. This is done by creating or deleting corresponding structures and by propagating attribute values between the models.

To allow a distinction from other approaches, we provide a comparison to state of the art transformation languages.
Zusammenfassung


Falls mehrere Modellierungssprachen benutzt werden, können ihre Modelle überlappende Informationen über das System enthalten. Ist dies der Fall, können sie nach Änderungen inkonsistent werden und müssen synchronisiert werden.

Die Komplexität des Modellsynchronisierens kann reduziert werden, indem spezialisierte Sprachen verwendet werden, um Konsistenzbeziehungen zu beschreiben, die als Grundlage für eine automatisierte Synchronisierung benutzt werden können. Die Sprachen verstecken Komplexität, die nicht für die Domäne der Modellierungssprachen spezifisch ist, zum Beispiel, indem Operationen für die Übertragung von Änderungen in einem Modell auf ein anderes Modell für jedes Paar von Modellen abgeleitet werden, anstatt eine explizite Spezifikation pro Paar und Richtung zu erfordern.

In dieser Arbeit haben wir die Mapping-Sprache, mit der diese Konsistenzbeziehungen spezifiziert werden können, und ein Rahmenwerk, das basierend auf der Spezifikation Konsistenz aufrechterhält, entwickelt.


Um eine Abgrenzung und Einordnung gegenüber verwandten Ansätzen zu erlauben, vergleichen wir unseren Ansatz mit modernen Transformationssprachen.
# Contents

<table>
<thead>
<tr>
<th>Abstract</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zusammenfassung</td>
<td>iii</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Motivation ........................................ 1
   1.1.1 Models as Abstractions .......................... 1
   1.1.2 Relationship of Different Models .................. 2
   1.1.3 Resolution of Inconsistencies ..................... 3
   1.1.4 The Mapping Language ............................. 3

1.2 Goals ........................................... 4

1.3 Structure ........................................ 4

1.4 Notation ......................................... 5

## 2 Foundations

2.1 Model-Driven Engineering ............................ 7

2.2 Models, Meta Models and Languages .................. 8

2.3 Domain-Specific Languages ........................... 8

2.4 Model Transformations ................................ 9
   2.4.1 Model Transformation Languages and Problems ....... 9
   2.4.2 Declarative and Imperative Programming Languages ... 10
   2.4.3 Model Transformation Properties ....... 10
   2.4.4 Different Execution Modes of Model Transformations ... 11
   2.4.5 Consistency Preservers ........................... 12
   2.4.6 Witness Structure ............................... 13
   2.4.7 Triple Graph Grammars ............................ 13

2.5 The Eclipse Modeling Framework ..................... 14

2.6 Xtext Language Engineering Framework ............... 15
   2.6.1 Xtext ........................................ 15
   2.6.2 Xbase ........................................ 15
   2.6.3 Xtend ......................................... 16

2.7 View-Centric Model-Driven Engineering and Vitruvius .... 16
   2.7.1 Problems with Heterogeneous Models ............... 16
   2.7.2 Orthographic Software Modeling .................. 16
   2.7.3 Vitruvius ...................................... 16

2.8 Roles in the Model-Driven Engineering Process ....... 18
## 3 Aspects of Consistency Preservation

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Surveys on Model Transformation</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>Basic Definitions</td>
<td>20</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Meta Model</td>
<td>20</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Consistency Relation</td>
<td>20</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Consistency Transformation</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>Characterization of the Meta Models</td>
<td>20</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Heterogeneous Representation of Information</td>
<td>21</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Heterogeneous Meta Model Design</td>
<td>21</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Derived Meta Models</td>
<td>22</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Validity</td>
<td>22</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Scope</td>
<td>22</td>
</tr>
<tr>
<td>3.4</td>
<td>Characterization of the Relation</td>
<td>22</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Bijectivity and Left- and Right-Totality</td>
<td>23</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Validity and Scope</td>
<td>23</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Additional Information</td>
<td>24</td>
</tr>
<tr>
<td>3.4.4</td>
<td>N-ary Consistency Relations</td>
<td>24</td>
</tr>
<tr>
<td>3.5</td>
<td>Specification of the Relation</td>
<td>25</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Relating Substructures</td>
<td>25</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Graph Patterns</td>
<td>26</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Meta Level of Pattern Description</td>
<td>26</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Mapping of Patterns</td>
<td>27</td>
</tr>
<tr>
<td>3.5.5</td>
<td>Mapping of Attribute Values</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>Characterization of the Transformation</td>
<td>29</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Dedicated Target Model</td>
<td>29</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Validity and Scope</td>
<td>29</td>
</tr>
<tr>
<td>3.6.3</td>
<td>User Interaction</td>
<td>30</td>
</tr>
<tr>
<td>3.7</td>
<td>Specification of the Transformation</td>
<td>30</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Constraint Solving</td>
<td>30</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Deriving Imperative Code</td>
<td>30</td>
</tr>
<tr>
<td>3.7.3</td>
<td>Consistency of Transformation Parts</td>
<td>31</td>
</tr>
<tr>
<td>3.7.4</td>
<td>Well-Behavedness of Attribute Propagation</td>
<td>32</td>
</tr>
<tr>
<td>3.8</td>
<td>Characterization of Multiple Transformation Steps</td>
<td>33</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Size of Deltas</td>
<td>33</td>
</tr>
<tr>
<td>3.8.2</td>
<td>Caching of Information</td>
<td>33</td>
</tr>
<tr>
<td>3.8.3</td>
<td>Oscillations</td>
<td>35</td>
</tr>
<tr>
<td>3.8.4</td>
<td>Undoability</td>
<td>35</td>
</tr>
<tr>
<td>3.9</td>
<td>Consistency Specification in the Model-Driven Engineering Process</td>
<td>36</td>
</tr>
<tr>
<td>3.9.1</td>
<td>Model Transformation and Meta Model Co-Evolution</td>
<td>36</td>
</tr>
<tr>
<td>3.9.2</td>
<td>Manual and Automated Consistency Preservation</td>
<td>37</td>
</tr>
<tr>
<td>3.9.3</td>
<td>Partial and Comprehensive Automated Consistency Preservation</td>
<td>37</td>
</tr>
<tr>
<td>3.9.4</td>
<td>Recognizing Uncertainties through Inconsistencies</td>
<td>38</td>
</tr>
</tbody>
</table>
## Contents

### 5.4 Language Implementation

#### 5.4.1 Editor

#### 5.4.2 MIR Base Language

#### 5.4.3 Generation of Responses

### 6 Formulation

#### 6.1 Consistency Relation

#### 6.2 Mapping

##### 6.2.1 Signatures

##### 6.2.2 Required Mappings

##### 6.2.3 Signature Conditions

##### 6.2.4 Body Condition

#### 6.3 Mapping Relation

##### 6.3.1 Correspondence Relation

##### 6.3.2 Implications of Correspondences

##### 6.3.3 Enforcing of Correspondences

##### 6.3.4 Consistency Violations and Repair

#### 6.4 Change Driven Consistency Transformation

##### 6.4.1 Changes

##### 6.4.2 Consistency Transformation

#### 6.5 Specification of Conditions in our Approach

##### 6.5.1 Signature Conditions

##### 6.5.2 Body Conditions

#### 6.6 Implemented Set of Expressions

##### 6.6.1 Signature Conditions

##### 6.6.2 Body Conditions

### 7 Related Work

#### 7.1 Bidirectional Transformation Languages

#### 7.2 Triple Graph Grammars

#### 7.3 .NET Modeling Framework Synchronization

#### 7.4 Basic Object-oriented Transformation Language

#### 7.5 Queries, Views, Transformations (QVT)

#### 7.6 ATL Transformation Language

#### 7.7 Constraint Solving

#### 7.8 Lenses

#### 7.9 Reactive Programming

#### 7.10 Change Impact Analysis

### 8 Comparison with a Triple Graph Grammar Tool

#### 8.1 Introduction

#### 8.2 Rule Specification

#### 8.3 Code Generation

#### 8.4 Development Environment

#### 8.5 Other Triple Graph Grammar Approaches
9 Conclusion and Future Work
  9.1 Conclusion ......................................................... 117
  9.2 Future Work ....................................................... 118

Bibliography ......................................................... 119
List of Figures

1.1 Semantic overlaps in an model-driven engineering environment with different modeling languages ........................................ 2
2.1 Different methods for retaining existing information in models when applying model transformations ................................... 12
2.2 Workflow for using VITRUVIUS with the MIR language family .......................... 18
3.1 Reduced consistency requirements for meta models that are reduced to a specific scope .......................................................... 23
3.2 Direct and indirect consistency relationships of meta models ............... 24
3.3 Patterns for structures in meta models and examples for consistent and inconsistent states ...................................................... 28
3.4 Consistency between parts of the derived code ................................ 32
3.5 The synchronization of different sizes of deltas can result in different synchronization results .................................................. 33
3.6 Example for a change that can be two atomic changes or a composite change 34
3.7 Undoability of a consistency transformation ................................ 36
3.8 Different levels of automation for consistency preservation .............. 38
3.9 Different levels of completeness of the consistency specification ........ 39
4.1 Graphical illustration for Listing 4.1 ................................................. 41
4.2 The consistent states (a) and (b) are created by the transformations generated from Listing 4.1 and Listing 4.2 respectively. ......... 42
4.3 Graphical illustration for Listing 4.2 ............................................... 43
4.4 Different languages in the MIR language family ................................ 46
4.5 Scope of the mapping language ...................................................... 47
4.6 Different structures in meta models, that can contain the same values .. 54
4.7 Possible consistent states while changing a model instance A .......... 56
4.8 Possible consistent states while changing a model instance B .......... 57
4.9 A meta model with structures that need transitive patterns for matching 63
5.1 State machine for the change handling protocol in VITRUVIUS .......... 71
5.2 Workflow for handling a change in the mapping language framework .... 72
5.3 Workflow for creating signatures .................................................... 72
5.4 Workflow for the generation of candidates ..................................... 73
5.5 Workflow for the handling of candidates ....................................... 74
5.6 Simplified class diagram of the code generated from the mapping specification in Listing 5.1 ...................................................... 76
5.7 Screenshot of the editor for the mapping language .......................... 84
List of Figures

5.8 Screenshot of mapping language validation and quick-fix .............. 84
8.1 TGG rule BoxToDictionary ..................................................... 109
8.2 TGG rule CardToEntry ............................................................ 111
8.3 Example for the editing of attribute constraints inside the graphical editor of eMoflon .......................................................... 112
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>A simple example mapping of two meta classes</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>An example mapping with a nested mapping</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>General structure of a mapping</td>
<td>48</td>
</tr>
<tr>
<td>4.4</td>
<td>An example for a mapping with chains of meta classes with unclear semantics</td>
<td>52</td>
</tr>
<tr>
<td>4.5</td>
<td>Exemplary mapping language code for mapping the meta model mmA to the meta model mmB</td>
<td>55</td>
</tr>
<tr>
<td>4.6</td>
<td>Sketch for the language feature for mapping between enumerable values and meta classes</td>
<td>60</td>
</tr>
<tr>
<td>4.7</td>
<td>Sketch of equivalent code to Listing 4.6 in our approach</td>
<td>60</td>
</tr>
<tr>
<td>4.8</td>
<td>Sketch for the language feature for specifying reusable bidirectional constraints</td>
<td>61</td>
</tr>
<tr>
<td>4.9</td>
<td>Sketch of equivalent code to Listing 4.8 in our approach</td>
<td>61</td>
</tr>
<tr>
<td>4.10</td>
<td>Sketch for the language feature for reference paths of arbitrary length</td>
<td>62</td>
</tr>
<tr>
<td>5.1</td>
<td>A mapping language specification that relates a Component with a contained ProvidedRole and a Class with a contained Method</td>
<td>75</td>
</tr>
<tr>
<td>5.2</td>
<td>Example for the generated code for synchronizing an incoming change eChange in a UML model to a PCM model.</td>
<td>79</td>
</tr>
<tr>
<td>5.3</td>
<td>Signature of the MatchUpdate constructor</td>
<td>79</td>
</tr>
<tr>
<td>5.4</td>
<td>checkPCM method generated from the example mapping specification</td>
<td>82</td>
</tr>
<tr>
<td>5.5</td>
<td>enforceCorrectInitializationOnPCM generated from the example mapping specification</td>
<td>82</td>
</tr>
<tr>
<td>5.6</td>
<td>propagateAttributesFromPCM generated from the example mapping specification</td>
<td>82</td>
</tr>
<tr>
<td>8.1</td>
<td>Sketch of the BoxToDictionary TGG rule as a mapping</td>
<td>109</td>
</tr>
<tr>
<td>8.2</td>
<td>Sketch of the CardToEntry TGG rule as a mapping</td>
<td>110</td>
</tr>
<tr>
<td>8.3</td>
<td>Propagating the name of a box to the title of a dictionary in eMoFlon</td>
<td>113</td>
</tr>
<tr>
<td>8.4</td>
<td>Propagating the name of a box to the title of a dictionary in the mapping language</td>
<td>114</td>
</tr>
<tr>
<td>8.5</td>
<td>Creation of a box and its partitions in eMoFlon</td>
<td>115</td>
</tr>
<tr>
<td>8.6</td>
<td>Creation of a box and its partitions in the mapping language</td>
<td>116</td>
</tr>
</tbody>
</table>
1 Introduction

In this chapter, we will first motivate the usage of models in engineering processes and our approach for resolving inconsistencies between models. Following the motivation, we will summarize our research goals and questions. Then we will describe the overall structure of this thesis and provide an overview on the notation that we use.

1.1 Motivation

We first motivate the usage of models in engineering processes. Then we will introduce the problems of fragmentation and inconsistencies, if multiple models are used, and motivate the automated resolution of the latter. This leads to the introduction of the topic of this thesis, the declarative specification of model consistency relationships.

1.1.1 Models as Abstractions

Models are widely used in engineering disciplines as abstractions of the system under development. Suitable abstractions are chosen depending on the task and the involved roles to support the development process by emphasizing relevant aspects of the system, its requirements, and the domain the system built for.

Because models represent some specific aspect of the system in an abstract and often formalized way, they can be used to analyze properties of the system before a full implementation, which allows developers to make informed design decisions instead of relying on experience or intuition. For example, a mathematical model for the capacity of a bridge can be used to make design decisions that align the dimensions of the bridge with the possible use cases.

Domain models help to persist and formalize knowledge about the problem domain of a software system and for establishing a common understanding between the different stakeholders of the system under design. By separating the solution domain and the problem domain, the requirement engineering for a system is simplified: the abstractions for the problem domain does not necessitate a very technical understanding of implementation details from the customer.

Additionally, models can be used to derive other artifacts that are used in the development process. For example, graphical representations of software systems can be used to generate documentation, or code stubs can be generated from an architectural outline of a system. The programs that implement those generations have models as input and output and are called model transformations. They are written in model transformation languages.
There exist various sophisticated frameworks and workbenches for creating software tools, such as graphical or textual editors, analysis tools and model transformations. The created tools can then be tailored to the specific abstraction a model provides.

In model-driven engineering, every artifact that describes a system is considered a model, including, for example, code and natural language documents. In principle, all models can be changed during the design and development process, finally arriving at a finished product.

### 1.1.2 Relationship of Different Models

Figure 1.1 shows an example of different modeling languages that could be used for developing a software system.

A model for the component based architecture of the software system is used by a software architect for making decisions on the overall structure of the system without regarding implementation details. In this example, the architectural models are instances of the Palladio Component Model (PCM) [12, 75]. They can be used to make decisions about the software architecture based on performance simulations of the architectural model. Another model in this context is the concrete Java source code that implements the functionality of the components. Additional models in the Unified Modeling Language (UML) can be used for describing, for example, the dynamic behavior and the requirements of the system.

The different models that are used in this example have common information. The project context might require that each of the components has a corresponding Java facade class that represents the component and has the same name as the component.

There is also information that is unique to each model: the Java source code contains the concrete implementation of the methods that are specified in an interface, while
the component model might contain additional information about the composition or deployment of components that does not directly influence the implementing code.

1.1.3 Resolution of Inconsistencies

In the model-driven engineering environment, there are redundancies between models, because the modeling languages are often not disjunct: the same information is contained in different models. If one of the models is changed, the overlapping models need to be synchronized to avoid inconsistencies.

The resolution of inconsistencies is, however, not a trivial task. If developers avoid changing models because of potential inconsistencies and because of the manual overhead for resolving them, the basic idea of model-driven engineering – to use relevant models to change aspects of a system – is threatened.

Model-driven engineering environments can automate parts of the process of keeping models with semantic overlap in a consistent state. The level of automation can range from providing developers with an estimation of parts of models that are possibly affected to automatic repairs that are triggered by changes and alter other models to restore a previously established notion of consistency.

1.1.4 The Mapping Language

Today, various approaches to the problem of consistency preservation exist. They allow people involved in the model-driven engineering process to specify rules for maintaining the consistency between models in different forms. However, there is not one common understanding on a way of solving the problem, but different perceptions of consistency, and of appropriate ways to configure consistency maintenance systems.

For existing approaches, it is often not simple to comprehend how the specified rules are operationalized, i.e., how the steps that are taken in case of a change to resolve inconsistencies are derived.

In this thesis, we provide an overview of problems that are associated with the consistency preservation problems in development environments that involve multiple models of a system. We give indicators on how the presented problems and questions can influence consistency maintenance systems.

Furthermore, we present our own approach to specifying consistency rules, the mapping language. The mapping language allows the declaration of patterns in modeling languages which consist of a set of meta classes and constraints on them. Furthermore, constraints on a pair of structures that are mapped to each other can be specified.

We also provide an implementation for our language. The implementation consists of a framework for the execution, editors and code generators. The generators derive (imperative) Java code from the declarative specification that is operationalized. That means, that we come from a declarative description of the desired consistent state to imperative code that repairs the consistency in case of a change. The generated code is designed to be comprehensible for the developer and aligned with the structure of the mapping declaration. The result of the code generation is a plug-in for the model-driven engineering environment VITRUVIUS.
In our implementation, if a match is found in a model after a change, or an existing match is altered to not fit the pattern anymore, a corresponding structure is created, or respectively destroyed, in a corresponding model. While structures are mapped to each other, we propagate attributes and alter the models after changes to arrive in states for which the additional constraints on the pair of models hold.

To make the behavior of our approach and the semantics of the mapping language easier to comprehend, we provide a formalization, and also discuss the differences and commonalities to a similar approach, eMo/lon. eMo/lon uses a different formalism for describing bidirectional model transformations, Triple Graph Grammars.

1.2 Goals

For this thesis, we have the following research goal:

\[ G_1 \] Support developers and domain experts in defining bidirectional mappings between elements of different modeling languages by hiding complexity of consistency preservation in a declarative language.

We aim to achieve this goal with the following research questions:

\[ Q_1 \] What are aspects of the bidirectional consistency preservation problem and of existing solutions that could be hidden from developers?

\[ Q_2 \] Which of these identified aspects can be encapsulated in a generator for a domain-specific language for bidirectional declarative consistency mappings?

We will also – more informally – provide lessons learned from our own approach and from the implementation using state-of-the-art tools for developing model transformations and tools for manipulating and analyzing models.

1.3 Structure

In this chapter, the problem of consistency maintenance has been introduced and motivated shortly, and we summarized the goal of this thesis. The conceptual and technical foundations for our work are introduced in chapter 2.

We describe different interesting aspects of the consistency preservation problem that have been recognized in different approaches and also while developing our own approach to the problem in chapter 3.

After this conceptual chapter on the consistency problem, we introduce the concept of our mapping language in chapter 4. After two introductory examples, we describe the syntax of the mapping language and its semantics. Additionally, we describe how our approach interacts with the model-driven engineering environment it is part of, and we present some possible extensions, limitations and future work for the language.

After the conceptual considerations, we present technical details of the implementation in chapter 5. Here, we shortly describe interesting aspects of the implementation of the
framework for executing the generated transformation, the generated code, the code
generation and the editors for the mapping language.

Subsequently, a more formal and precise description of our notion of consistency is
presented in chapter 6, including the conditions on pairs of models whose violation we
automatically repair, and in which way.

An overview about relevant related approaches to consistency preservation and model
transformations is given in chapter 7.

After we described related approaches, we discuss the commonalities and differences of
our approach and a particular approach, eMoflon, in chapter 8. This chapter aims to make
the classification and distinction of our approach easier.

We conclude this thesis in chapter 9 with two sections that provide a summary of this

1.4 Notation

In this section, we will shortly list the notation we used in different chapters in this thesis.

\( A, B, \ldots \)

Meta models

\( A, B, \ldots, A', B', \ldots, A'', B'', \ldots \)

Models. \( A' \) is usually a slightly changed version of \( A \), but may also be equal to \( A \),
depending on the context.

\( R, R_1, R_2, \ldots \)

Consistency relations between models.

\( R^{\rightarrow} \)

A model transformation that takes a pair of models as input and transforms it into a
pair of models that is consistent according to \( R \).

\( R^{\leftarrow}, R_1^{\leftarrow}, R_2^{\leftarrow}, R^{\rightarrow}, \ldots \)

Model transformations that change the “left” or respectively the “right” model if
needed for consistency. It should be clear which model is meant from the context.

\( \Delta, \Delta_A, \)

A change, a change in model \( A \).

\( \Delta_A^{+}, \Delta_A^{-} \)

A pair of changes in model \( A \), that are opposites. \( \Delta_A^{-} \) is the “reverse” change (the
undo) of a change \( \Delta_A^{+} \).

\( (a_j), (b_k), (a_j)_{j=1,\ldots,n}, (b_k)_{k=1,\ldots,m} \)

Model element sequences.

\( \top, \bot \)
The boolean truth values true and false respectively.
Additionally, we used different font families, styles and decorations in the text as follows:

repo2pkg, ChangeSynchronizing
References to elements from a code listing (mappings, named meta classes), or references to Java classes.

LearningBox
Model elements

name, signature condition
Placeholders in a listing, otherwise reference to the placeholder.

mapping, map
Keywords in the mapping language listings.

notnull, default-contain
Keywords in constraint language in the mapping language listings.

“Logger”, “AccountManaging”
A string-typed value for an attribute of a model element.
2 Foundations

In this chapter we will describe the foundations for our approach. First, we will introduce the field of model-driven engineering and the terminology and techniques associated with this field. Then we will introduce the technical foundations and frameworks for model-driven engineering that we have used in our implementation. We will conclude this chapter with a short introduction of a specific form of model-driven engineering, view-centric model-driven engineering.

2.1 Model-Driven Engineering

Models are widely used in software engineering for describing the requirements, architecture and implementation details of the software project. They are used to allow manipulating and reasoning about systems at a more abstract level and thus handling complexity. The same is done in other engineering disciplines for providing means of planning, validating or analyzing systems.

Models are used for example for code generation, for deriving other artifacts, or as documentation. For example, Unified Modeling Language (UML) [71] diagrams can be used for the documentation of a system. If, however, the derived artifacts diverge from the source models, the latter are not necessarily updated, since it would require a considerable additional effort to keep all models consistent manually. Therefore, if software systems are subject to change, the models and keeping them synchronous with the evolving system can be seen as a burden to the developer.

In model-driven engineering (MDE), often also called model-driven development (MDD), models are are used to achieve a higher level of abstraction in the engineering process [9]. Furthermore, there is the notion that “everything is a model” [16], meaning that all artifacts that are used to describe and develop the system – including code and natural language descriptions – are considered and treated as models. In particular, this means that there is not a distinction between a set of artifacts – for example the source code – that are the system and a set of models that represent the system. The tools and processes from model-driven engineering are leveraged to handle all involved artifacts.

Since all artifacts that are involved in the engineering process are considered to be equal, the fitting model for a task at hand can be used to manipulate the system. Since there are usually overlaps in the information that is represented in the different models, if they abstract from the same “information” for different purposes and in different ways, there needs to be some mechanism for propagating the changes to other models. Models are updated if other models that contain overlapping information change. This approach is also called round-trip engineering [44] and enables the developer to continuously analyze and understand the changing system from relevant viewpoints.
In complex systems, views or models which are tailored to the needs of a specific role or task allow focusing on the task at hand without having to understand the whole system, at least not the details that are irrelevant to the current task. They can be analyzed and validated, for example to ensure that there are no unreachable states in a finite state machine, that timing constraints of the software running on a specific hardware model are not broken, or to detect outages in a model of the smart grid [66].

2.2 Models, Meta Models and Languages

A model is an abstract representation of a system for a specific purpose. A model always represents an original, but only includes relevant information about it, i.e., it never fully describes the original. The purpose the model is created for is known before creating the model and dictates the information that is included in the model [81, sec. 2.1.1]. A model can also be prescriptive or descriptive [81, p. 129]. A prescriptive model describes how the modeled system should behave or how modeled processes are supposed to be, while a descriptive model describes the status quo of the system.

A meta model describes what constitutes a valid model. It is a model that describes another model, hence the name “meta” model. It consists of the following parts [82]:

- an abstract syntax describing the elements a valid model can consists of, independent of the representation,

- at least one concrete syntax that specifies how to express a model, for example in a textual language or as a graphical diagram,

- static semantics that further describe the form of valid instances and do not contain information about the usage or the transformation of the model instances. The static semantics can for example be expressed as constraints.

In this thesis, we use the terms meta model and language interchangeably, because a meta model can also be considered a language used for describing aspects of a system or process that are relevant to a specific domain. A model can conform to a meta model. We also say the model is an instance of a meta model interchangeably.

We refer to the elements of the abstract syntax as meta classes (classes of the meta model) to avoid confusion with classes in an object-oriented model of the software system.

2.3 Domain-Specific Languages

A domain-specific language (DSL) is a “a computer programming language of limited expressiveness focused on a particular domain” [39].

In the context of model-driven software engineering, the term DSL is often used to describe the meta models that are used. The line between meta models and DSLs is not strictly defined. For many purposes, the separation is also not necessary. We consider DSL engineering and DSLs to be processes and models, respectively, where the language is
created and adapted in a more iterative approach, during the process of understanding a domain and its requirements.

The DSL approach is often also more “pragmatic” than to model-driven engineering as for example envisioned by the Object Management Group\(^1\) (OMG) in the model-driven architecture (MDA): they are often build directly as APIs for general-purpose programming languages that are used in a specific “syntactic style” (internal/embedded DSLs) or only provide a separate syntax (external DSLs) \([39]\). There is often no distinction between abstract and concrete syntax for DSLs. For internal DSLs, existing General Purpose Languages (GPLs) like Java or Ruby can used as host languages, meaning that the DSL is expressed as a framework or library that can be used with language features of the GPL.

An example for an internal DSL is the configuration of the Dependency Injection framework Google Guice\(^2\), which can be done with a fluid application programming interface (API) in Java instead of using a specific language for the composition and configuration of different Guice modules. This allows the use of the often more mature tooling of the host language for editing and debugging or removes the need for independently developed tooling for the DSL completely.

The development of DSLs is facilitated by language workbenches which are tools for creating DSLs and incorporating them into the development process \([39]\).

In the context of model-driven development, the role of DSLs is twofold: on one hand, domain-specific languages can be created for the development domain and can be used for the generation of software artifacts or for the analysis and the validation of the system under development; on the other hand, the languages used for model-driven development are often DSLs for the model-driven engineering domain – for example model transformation languages, the consistency specification language family MIR that is topic of this thesis, or constraint languages such as the Object Constraint Language (OCL) \([49]\).

## 2.4 Model Transformations

A model transformation is a program that has models as input and output. In the following we mostly talk of a single input and output model. The considerations are, however, also applicable to multiple input and multiple output models.

### 2.4.1 Model Transformation Languages and Problems

Model transformations (the “heart and soul of model-driven software development” \([79]\)) are used to derive other artifacts from a model. Other artifacts can for example be analyses, documentation, executable code or views that encompass information from multiple models. Special programming languages, model transformation languages, are used for specifying model transformations. The development of specialized languages allows the usage of syntax for tasks that often occur in the context of model transformations, such as traversing of models or the tracing between model elements in the source and the target of a model transformation.

\(^1\)http://www.omg.org/
\(^2\)https://github.com/google/guice
Biehl offers a taxonomy of model transformation languages [17]. The work contains a comprehensive overview of the terminology used in MDE [17, sec. 2] and an introduction about the typical use cases in which model transformations are necessary. There, the term model transformation is understood as the “automatic generation of a target model from a source model, according to a transformation description” [17, p. 7], following the definition by Mens et al. [65]. The model transformation problem is the “problem that we would like to solve using a model transformation” [17, p. 12] and model transformation languages offer different capabilities that can make them more or less appropriate for a specific model transformation problem. There exist various studies that classify model transformation languages in general [27] or that give a more fine-grained classification for positioning model transformation problems according to their possible incrementality and the symmetry of the information that is represented in the models [65, 30].

Naturally, there is not one model transformation for a specific input (and output) meta model, but many model transformations for the same input meta model and – more generally – for the same input-output pair of meta models can exist for different purposes. Which information is transformed can be specific to the purpose and the parametrization of a model transformation. For example, if an architectural description is transformed into Java code that can be executed for measuring the predicted performance of the system, different model transformations could result in different Java code, depending on which system aspect the architect wants to consider in the performance evaluation. Another model transformation between the architectural description and Java code could create stubs that are used as a base for the implementation.

### 2.4.2 Declarative and Imperative Programming Languages

In this thesis, we often differentiate between declarative and imperative languages. Declarative languages declare rules that have to hold for the result of a program, for example for a model transformation. They specify properties of the desired result, without specifying the exact steps for reaching this result. The declared rules are then operationalized by the language framework, for example by code generation or interpretation, to derive concrete steps that have to be taken to reach this result.

An example for a programming paradigm that uses declarative programming is logic programming. In the programming language Prolog, for example, the developer specifies a series of elements, relations on those elements and logical rules for deriving relations. The user can then query the system whether elements are contained in derived relations and the Prolog engine reasons about the state of the system to derive an answer.

In imperative programming, the state of a programming is changed step by step following the control flow specified by the developer in the imperative programming language using commands. The statements that are used describe how to reach the state and do not specify the desired result explicitly.

### 2.4.3 Model Transformation Properties

If a model transformation has the same input and output model, and therefore also meta model, we call it in-place [27].
Another property of model transformations that is relevant for our work is directionality [27] or more specifically bidirectionality [83]. Many model transformation languages describe how to arrive at a dedicated target model from a dedicated source model. A separate description is then used for the other direction of the model transformation, if this transformation is needed. If a model transformation specification is bidirectional, both directions of a transformation can be derived from the same specification. If a model transformation explicitly only describes one direction, we call it unidirectional [27].

Even though model transformation languages are usually designed for specific contexts in the engineering process, they also often have to tackle “problems that have been studied in other contexts for decades” [83, sec. 1] regarding the transformation of data in general. This means that there are many insights – for example from the database community – that have to be considered when studying the problem of model consistency. The view-update problem [11], which is the problem of allowing updates to a view of a database and the back-propagation of changes, is similar to the problem of updating views on a system that is expressed in a view-specific meta model and models.

2.4.4 Different Execution Modes of Model Transformations

If a model transformation is executed in batch mode, we start with an empty target model, and derive all content of the target model from the input model.

In a batch mode model transformation, the previous target model is discarded and a new target model is built from scratch considering all the information from the source model. This can – depending on the concrete case – of course be more costly than choosing the appropriate adjustments based on the change.

Incremental model transformations [27] do not create a model from scratch, but also directly considers the state of the model transformation. Assuming that a model $B$ was already the result of the transformation from a model $A$, the transformation engine only considers the elements that need to be changed in the target. The needed elements are derived from the change from $A$ to $B$. For large models this can reduce the runtime of model transformations because not the whole input model has to be inspected. Then, the complexity of the needed change does not depend on the size of the input model, but on the size of the change that needs to be synchronized [43].

If there is an existing target model and information that cannot be derived from the input should be preserved in the newly created model, there are different general methods for keeping the existing information. In the following we will describe two approaches based on Hearnden et al. [43, sec. 1.1].

Figure 2.1a shows how information can be preserved using a model merge approach. After a change to model $A$ by a user, the resulting model $A'$ is transformed in batch mode using the transformation description $t_{AB}$. The model $B'$ that is the result of this transformation is then merged with the previously existing model $B$. The merge process can be done manually, for example with a tool such as EMF Compare3 or in a more automated way, for example by using information from the model transformation description.

3https://www.eclipse.org/emf/compare
2 Foundations

(a) Information that cannot be derived from $A'$ using the transformation can be retained by merging the result $B'$ of the transformation with the previous result $B$.

(b) If changes in the source model ($\Delta_A$) are translated directly into changes in the target model ($\Delta_B$) with a change-driven transformation, unrelated information in the target model is preserved.

Figure 2.1: Different methods for retaining existing information in models when applying model transformations (based on [73, 43, 14])

Figure 2.1b shows a process where the information from the previously existing model $B$ is considered directly in the model transformation process. As opposed to the batch execution, the model transformation engine does not create a new instance $B'$, but alters the model $B$ directly. If the model transformation engine can assume that $B$ is already the result of the transformation of $A$ from the same specification, only the changes that are needed to create a $B'$ that contains the same information as a model that would result from a batch transformation of $A'$ are applied to $B$ instead of creating a new model. This can be done, while leaving information that can not be derived from the model $A$ and that is irrelevant to the change $\Delta_A$ in the target model untouched. We call this kind of transformation change-driven, based on Bergmann et al. [14].

2.4.5 Consistency Preservers

In this thesis, we are concerned with model transformations that reestablish consistency between models after changes have been made to a model and the notion of consistency that can be automatically restored, using such model transformations. We call those transformations consistency preservers, consistency (preserving model) transformations, or (heterogeneous) synchronizers (the latter after Antkiewicz and Czarnecki [4]).

The systems that provide mechanisms for building and repairing consistency preservers are called consistency (repair) approaches, mechanisms, or systems.
2.4 Model Transformations

2.4.6 Witness Structure

The witness structure of a model transformation records information about its execution state. This is necessary for incremental model transformations, since they need to keep track of relationships between model elements that cannot easily be deduced from the models themselves – or in general information that is needed to find the correct target elements in case of a change in the source model. Information that is supplied manually to the model transformation system such as a choice between transformations that is not persisted in one of the models can also be contained in the witness structure.

Witness structures are not only needed for bidirectional consistency maintainers, but can also be used in unidirectional, batch-executed model transformations. For example, in the model transformation language QVT operational, during the execution of a model transformation, tracing information is stored that relates the source and the target elements that are created. The transformations can explicitly query and manipulate the tracing information using special keywords of the transformation language.

Model transformation languages, whether they are written for incremental or batch execution, can provide means of referring to the concrete witness structure. Czarnecki and Helsen discuss this in their classification of model transformation approaches [27] and call this information intermediate structures. Making this intermediate structures referenceable in the language may however make the writing of transformations more complex, if the user needs to understand the execution engine and the point of time and the order in which different rules are executed.

2.4.7 Triple Graph Grammars

Triple Graph Grammars (TGGs) are a declarative way of specifying how three interconnected typed graphs can be created consistently. They were introduced by Schürr [77]. For the context of this thesis we will introduce them as means for bidirectional model transformations, which is a common and well-studied use case for TGGs. TGGs are, however, not restricted to the domain of model transformations and were also not originally devised for model-driven engineering, but more generally for transforming graphs.

The three eponymous graphs that are maintained by a TGG are two graphs that are to be synchronized and an explicit correspondence graph. This correspondence graph acts as a witness structure and explicitly models the elements that are already corresponding according to TGG rules. All three graphs are usually models in our model-driven engineering case, and have defined meta models: the two sides of the TGG have the meta models of the models that are to be synchronized, and a meta model for the correspondence model is defined by the TGG.

The three graphs that are maintained by a TGG can be changed according to TGG rules that are contained in the TGG. A TGG rule specifies a precondition pattern that comprises all three graphs. Additionally the rule specifies a replacement for this precondition pattern. Rules that have an empty precondition, and can therefore be used as a “starting point” for the production, are called axioms. Model pairs are consistent, if they can be created by sequential production through TGG rules. Similarly to other model transformation approaches, there exist TGG implementations that only allow batch execution, and ap-
proaches that allow the incremental and change-driven production of consistent model pairs. Additionally, some approaches allow for the integration of a pair of models, which means the reconstruction of the correspondence graph, if possible.

2.5 The Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF)\(^4\) is part of the Eclipse project\(^5\). Eclipse is a project that comprises tools and projects for developing software. The most prominent part of the Eclipse project is the Eclipse Integrated Development Environment (IDE) for Java\(^6\).

EMF facilitates the use of meta models and models in Eclipse applications and – more generally – Java applications. The meta models and models used in EMF are based on the Ecore meta model which is an implementation of the Meta-Object Facility (MOF) standard [\(\text{[50]}\)] for MDE by the Object Management Group (OMG)\(^7\). In the MOF, models are persisted in the XMI (XML Metadata Interchange)\(^8\) file format.

In the Ecore meta meta model, the specified meta models are very similar to class structures in Java. The meta model specifies meta classes that have features and operations. The features have a name and are single- or multi-valued with a specific lower and upper bound and are ordered or unordered. Features can be attributes, which have a data type, and references, whose type are other meta classes. References can be explicitly specified as containments and for each model element, there can only be one other model element that contains it. Meta classes can furthermore be abstract, which means that they cannot be instantiated, and they can have inheritance relationships to each other, as in normal object-oriented programming languages.

EMF contains facilities for creating Java APIs for representing Ecore meta models and model instances of those meta models as class and object structures. Furthermore there is a common interface for building editors that manipulate those models. The specific differences between Ecore and the generated Java class structures are not important for the context of this thesis.

Besides providing a mechanism for mapping meta models and models to class and object structures respectively in form of EMF, the Eclipse Modeling Project\(^9\) also contains various other projects that are in some way related to bringing modeling and model-driven engineering to the Eclipse development environment, and therefore also to Java.

Graphical and textual syntaxes for meta models can be created (partially again using model-driven engineering techniques) by generating Eclipse plug-ins. The generated editors can be used for creating and manipulating models. Their code can be changed manually to offer richer and more specialized editing capabilities for the language under development. Textual editors can be created with Xtext, which will be described in the following section.

\(^4\)https://www.eclipse.org/modeling/emf/
\(^5\)https://www.eclipse.org/
\(^6\)http://www.eclipse.org/ide/
\(^7\)http://www.omg.org/
\(^8\)http://www.omg.org/spec/XMI/
\(^9\)https://www.eclipse.org/modeling/
Additionally, tools for validating, comparing and merging models, for specifying queries and transformations and for providing distributed and scalable storage are provided in the Eclipse Modeling Project.

2.6 Xtext Language Engineering Framework

2.6.1 Xtext

Xtext\(^{10}\) is a framework that allows the description of the textual syntax in a grammar language that is based on the extended Backus-Naur Form (EBNF). The grammar can reference existing meta models or a meta model can be inferred from the grammar. Textual editors that plug into the Eclipse IDE can be generated from the grammar. Those editors can then be used to create and manipulate model instances.

The generated code offers state of the art tool support out of the box, for example code completion, outlines and syntax highlighting. Validation that is displayed directly in the editor can be implemented by the developer. Likewise, the developer can implement Java code for determining which entities can be referenced how in the textual syntax, for example registered EPackages in the current Eclipse instance, or content of other models in the workspace.

The whole development process of the tooling is model-based itself. If the grammar changes, the generation can be rerun and the new plug-ins are integrated with the previously generated ones, requiring manual adaption in the manually changed code.

It also contains mechanisms for defining model transformations and integrating them into the editing workflow in Eclipse.

Xtext also allows the developer to directly associate model transformations (or “generators”) with the editors that are automatically triggered if the models are changed. This can be used for directly generating code from the changed files. Furthermore, Xtext contains a sophisticated framework for the generation of Java code that allows users to specify a class model that is to be created during the generation process in an internal DSL for Java. In this framework, Java types in the class path of the project the code is generated for can be directly referenced.

2.6.2 Xbase

Using Xtext, it is very easy to define simple domain-specific languages. Additionally to the basic grammar functionality, developers can reuse a Java-like expression language, Xbase, inside their languages [33]. If the developer decides to use Xbase, so called XExpressions can be referenced in the grammar description. The expression language is statically typed and uses the Java types that are available in the current context (more specifically in the Java class path of the code that is generated from the expressions). The scope (local variables and imports) can be defined by the developer by changing generated code in the plug-ins for the Eclipse IDE. The framework also includes a compiler that allows the

\(^{10}\)https://www.eclipse.org/Xtext/
generation of Java code from XExpressions, allowing for an integrated generation of Java code from the whole model specification.

2.6.3 Xtend

Xtend\(^{11}\) is a Java-like language that is developed with Xtext and Xbase. It is compiled to Java and can be used productively, and particularly independent from Xtext, in the Eclipse IDE. We use it for the code generation in the mapping language and in the underlying framework, since it provides some mechanisms that are convenient for model-driven software engineering techniques, such as a sophisticated mechanism for specifying template expressions\(^{12}\), which we use for generating some of our Java code, and dynamic dispatch\(^{13}\), which makes the definition of code that needs to distinguish between different meta classes at runtime more convenient.

2.7 View-Centric Model-Driven Engineering and Vitruvius

2.7.1 Problems with Heterogeneous Models

The basic idea in model-driven engineering is the usage of appropriate models for the tasks that occur during an engineering process. If different modeling languages (meta models) are used in the context of the process, the information that is needed for a role or task is often spread across different heterogeneous models, and the relationships between the different models that are related to a problem is not obvious or made explicit. This problem is called fragmentation [21].

Another problem that arises if heterogeneous models are used in the context of a model-driven engineering process are redundancies between the different models [21]. If the information that is shared between heterogeneous models is changed only in one of the models, inconsistencies are introduced into the system description.

2.7.2 Orthographic Software Modeling

One possibility of tackling those two problems is to build a Single Underlying Model (SUM) that contains all the information that is needed, and to project the information from it into different views that are tailored to the involved roles and tasks. This is for example done in the Orthographic Software Modeling (OSM) approach [7]. The SUM is then designed to avoid redundancies and fragmentation of information.

2.7.3 Vitruvius

Vitruvius (View-centric engineering using a virtual underlying single model) [56] is an approach to model-driven engineering using multiple meta models that describe an under-

\(^{11}\)http://www.eclipse.org/xtend/
\(^{12}\)http://www.eclipse.org/xtend/documentation/203_xtend_expressions.html#templates
\(^{13}\)http://www.eclipse.org/xtend/documentation/202_xtend_classes_members.html#polymorphic-dispatch
lying system and flexible views that select a subset of a meta model or of the combination of multiple meta models [20]. VITRUVIUS is based on the OSM approach, but is designed to allow the integration of existing meta models into a modular SUM, as opposed to a fixed SUM that is part of the approach. By allowing the usage of existing meta models, existing tools, for editing and analysis, can be used in the engineering process.

The approach is called view-centric, because all the information is presented to the users in views that do not necessarily encompass exactly the information in one model. Views can partially select the information from one or multiple models. They are tailored to the tasks and roles in the process. Views are described via ModelJoin [22], a DSL for describing how to select relevant information from one or multiple meta models, for handling the problem of fragmentation.

Editors for EMF models are monitored and changes are passed into the VITRUVIUS framework. Consistency-conserving transformations can be registered for pairs of meta models and are triggered if a model that conforms to one of the meta models changes. These transformations are used to handle the problem of redundancy between models.

This interplay of flexible views and consistency-preserving model transformations provides a virtual SUM as opposed to a manifestation of the SUM as a meta model and corresponding model that unites the information from all meta models and models respectively.

Before the work in this thesis, general purpose model transformation languages and general purpose programming languages such as Java have been used to express the needed consistency preserving model transformations [53]. Those transformation languages generally try to cover a broad spectrum of the problem scenarios described in subsection 2.4.1. VITRUVIUS supports the execution of in-place transformations for a specific meta model for incoming changes. The transformations can be written in any programming language, as long as they can be called from the technical space of VITRUVIUS, the Java programming language.

The VITRUVIUS idea also encompasses a family of domain-specific languages, MIR, which is used to describe the consistency preservation rules. This language family is currently under development, and one of its parts, the mapping language, has been studied and implemented in this thesis. The MIR language aims to separate understanding and specifying the consistency requirements of the used models on one hand, which is a task for a domain expert, and implementing the transformations on the other hand, which requires knowledge in model transformation languages and frameworks [53]. The MIR language can be used to handle the second task automatically, i.e., to generate change-driven consistency preservation transformations from the specification [53]. The workflow of using the MIR language family in VITRUVIUS is depicted in Figure 2.2. First, the methodologist specifies the consistency relationships using the languages in the MIR language family. Then the specification as a whole is used to generate incremental transformations. Those incremental transformations are triggered after a user makes changes in a view, and appropriately reestablish consistency between associated models. They maintain the state of the transformation and correspondences between model elements in an instance of a correspondence meta model.
The **VITRUVIUS** framework contains code for managing different meta models and models and correspondences between model elements from different meta models. It defines an interface that must be implemented by consistency transformations.

Our mapping language framework is designed to encapsulate functionality that is generic to all generated transformations (but not to all transformations usable in the **VITRUVIUS** framework) in form of abstract classes. The generated code inherits from those abstract classes and implements the functionality for checking mapping conditions, updating mutual information, selecting the correct mapping candidates to check for state change on the correct model elements based on the incoming change, etc.

### 2.8 Roles in the Model-Driven Engineering Process

There are different stakeholders involved in a model-driven engineering process. The OSM approach to view-based development defines two roles [7]: the **methodologists**, who specifies the used meta models and views for the model-driven engineering environment, and the **developer**, who uses the models in the development process.

Furthermore, we also consider that there might not be one methodologist that knows about the relationships and views of all the involved meta models, but rather multiple methodologists that know about a subspace of the modeling environment and define relationships for the meta models in that subset but do not know (in detail) about the other involved meta models in the process context.

---

**Figure 2.2**: In the workflow for using **VITRUVIUS** with the MIR language family, a MIR specification is used for generating incremental model transformations that react to changes in views in the system. Adapted from [53].
In this chapter, we will introduce different aspects of the problem of consistency preservation. The following considerations come from problems we faced during the design and implementation of our approach, and from existing literature on the topic.

To better position the following considerations, we will shortly describe our own approach. It is based on patterns that are defined on the meta model level. If matches for the patterns are found in a model after a change, a corresponding structure is created. If structures do not conform to a pattern anymore, corresponding structures are destroyed. Structures that correspond have additional constraints that we enforce after changes. The description is declarative and describes the desired state, not how the state is enforced after a change. Furthermore, the description does not have a direction. It can be applied for propagating changes from each involved meta model to the other.

When we talk about changes and their synchronization, we call the model that is changed by a user or the meta model the changed model conforms to the source (model) or meta model respectively and the (meta) model(s) the changes are propagated to the target(s), target (meta) model(s) or opposing (meta) model(s).

This chapter is structured as follows. First, we provide a short listing of existing surveys on model transformation. Then we define meta models, the consistency relation and consistency transformations. In the following sections, we describe characteristics of each of those concepts that affect possible consistency relationships. We additionally describe aspects of the specification of the consistency relation and the transformation. Furthermore, we consider the combination of multiple transformation steps. The final section of this chapter discusses the integration of consistency transformations into the model-driven engineering process.

### 3.1 Surveys on Model Transformation

There are existing surveys by Czarnecki et al. that classify model transformation approaches [27] and bidirectional model transformations in particular [28].

Biehl categorizes the model transformation problem space and approaches to model transformation and classifies existing tools in a literature study on the state of the art of model transformations [17].

Antkiewicz and Czarnecki describe properties of heterogeneous synchronizers, which are essentially the bidirectional transformations described here, and classify the design space for such mechanisms in different dimensions [4].

Diskin describes an algebraic framework for model synchronization based on the composition of diagram operations and covers some of the problems described here for the
integration into the framework [29]. Recently, Diskin et al. also provided a taxonomy for classifying bidirectional model transformation types [30].

### 3.2 Basic Definitions

We will first introduce the basic notion of meta models, consistency, and the consistency transformation for this chapter. The following considerations are based on the relational definition of consistency by Stevens [83] and Antkiewicz et al. [4].

#### 3.2.1 Meta Model

If we talk about meta models in the following, we write them as $A$ and $B$. In this chapter, a meta model is a set of all syntactically correct instances of the meta model. We therefore disregard static semantics of the meta models and point them out explicitly when they are relevant.

The different models for a meta model $A$ can be seen as states of a model instance. When we change a model $A$, we are moving through the state space of the meta model $A$ to the next model state $A'$.

The set of all pairs of models from the meta models $A$ and $B$ is $A \times B$. It consists of all (binary) tuples of the models. If $n$ meta models are involved, we accordingly consider $n$-tuples.

In the following, we refer to the information that can be expressed by the instances in a meta model $A$ by the notation $A_I$. The overlap of the information that can be described by two meta models is referred to by $A_I \cap B_I$.

#### 3.2.2 Consistency Relation

Two models $A$ and $B$ that are instances of the meta models $A$ and $B$ respectively are consistent, if they are in a consistency relation $R$. $R$ is a subset of all pairs of models: $R \subseteq A \times B$. In principle, more than two meta models could be involved in the relation, but we will first consider the binary case (two meta models).

In this chapter, we will call the consistency relation simply the relation $(R)$.

#### 3.2.3 Consistency Transformation

A consistency transformation $R^+ : A \times B \rightarrow R$ takes a pair of models $A$ and $B$ that adhere to the meta models $A$ and $B$ respectively, from an inconsistent state $(A, B)$ to a consistent state $(A', B')$ in $R$. $A'$ and $B'$ do not necessarily have to differ from $A$ and $B$ respectively.

### 3.3 Characterization of the Meta Models

We first want to give an overview of characteristics of meta models that affect their consistency relationships.
3.3 Characterization of the Meta Models

### 3.3.1 Heterogeneous Representation of Information

When using modeling in an engineering process, meta models that fulfill the needs of the different stakeholders, roles, and tasks need to be chosen. Since a meta model is built for a purpose and has a planned level of abstraction, there are multiple ways in which the meta models can differently represent information.

Different syntactical representations of information can be caused, for example, by a differing level of abstraction, which often result in a loss of information between the different models. Possible differences are, for example:

- Flattening of hierarchies of model elements,
- Different levels of information about data structures, i.e., ordered/unordered, distinct/non distinct values,
- Aggregation or filtering of lists of attribute values,
- Aggregation on more complex structures such as sub-graphs, for example models for route-planning which only contain information needed for calculating shortest paths versus models which contain all necessary information for drawing a map.

There is no fixed abstraction level for meta models. There exist, for example, meta models for the description of software architecture that can be used for a very broad range of architectures, such as the Unified Modeling Language (UML). On the other hand, there are also architecture description languages that dictate a specific architectural pattern, such as a component-based architecture in the Palladio Component Model (PCM).

### 3.3.2 Heterogeneous Meta Model Design

The same information, with a similar purpose and level of abstraction, can also be represented differently in a meta model.

There is no fixed reference structure or design guideline for meta models, the design of meta models is an active field of research [85]. Meta models are often evolved and changed as the understanding of the problem domain and the requirements and functionality of the system change. Furthermore, there may be need for using an existing meta model because of existing tooling for editing and analysis or existing expertise in the project or team.

Different technical representation of information can also cause meta model heterogeneity, for example:

- Different encoding of integer and float values, different endianness, different precision of float values,
- Information saved in an enumeration attribute versus in the type information of a meta class,
- Expressing of information in object structures or attribute values or lists of attribute values of a meta class.
3.3.3 Derived Meta Models

In some cases, meta models are derived in a structured way from other meta models, either automatically or manually.

For example, consider the idea of flexible views for model-driven engineering [21]. The meta model of the view is derived from the query specification in the language ModelJoin. It has a defined overlap in meta classes with the source meta model(s). The query also implies a combination of models of the input meta models to a model of the synthetic output meta model.

Consequently, there is a need for keeping the source models and the output view of this query transformation consistent. Even though this problem could be solved with additional, manually specified consistency transformations in principle, we typically want to derive the consistency transformations directly from the meta model combination specification.

3.3.4 Validity

We recognize different basic levels of validity for a model.

Most fundamentally, models can be syntactically valid instances of the associated meta model that can be serialized. Those models make up the set of instances of a meta model. For example, this encompasses syntactically correct Java code or an Ecore model that has a correct XMI (see section 2.5) representation. We call this property serializability. In the following, we only consider serializable models. Those are also the models that make up the meta model $A$ in our definition.

Since meta models can also include constraints, for example OCL (Object Constraint Language) expressions in UML models, a serializable model can furthermore be valid, if it does not violate constraints of the meta model. We define $A^V \subseteq A$ as the set of all valid instances of a meta model $A$.

3.3.5 Scope

Meta models are used in a specific scope that can differ from the scope the meta model is designed for. For example, this can be the context of a project or team.

In general, not all features of a meta model will be used in a specific scope. The restriction can be either explicit, in form of specified invariants, or implicit through usage. Therefore, the actually used subset of model instances is a subset of $A$. We call this subset $A^S \subseteq A$. Note, that $A^S$ is not necessarily contained in the previously defined set $A^V$ of valid models or vice versa.

3.4 Characterization of the Relation

In the previous section, we introduced characteristics of the meta models. In this section, we will describe consistency of pairs of instances of the meta models.
3.4 Characterization of the Relation

Figure 3.1: Meta models $A$ and $B$ are constrained implicitly or explicitly to a subset $A^S$ and $B^S$ respectively in the context of a project or team. Only the overlap $A^S \cap B^S$ of the possible expressed information of the meta models needs to be synchronized.

3.4.1 Bijectivity and Left- and Right-Totality

Basic questions for the relation $R$ are the following:

- Is the relation bijective? If $R$ is bijective, then for each $A$ in meta model $A$ there exists exactly one $B$ in meta model $B$ so that $(A, B) \in R$ and vice versa. This is normally not given in our context, because only parts of the models make up the semantic overlap. If we can, however, restrict the meta models in a way that makes the relation bijective, we can uniquely identify a consistent state for each model.

- Is the relation left-total and right-total? Then, for each $A$ there exists at least one $B$ in meta model $B$ so that $(A, B) \in R$ (left-total) and vice versa (right-total). If the relation is both, every state can be synchronized by only changing one of the involved models.

3.4.2 Validity and Scope

We defined the subsets $A^V$ and $A^S$ in subsection 3.3.4 and subsection 3.3.5 respectively. A natural constraint on the relation is to only consider elements for consistency, if they are valid and/or in the current scope. The specification of the consistent pairs of models is simplified, if less pairs have to be considered and distinguished.

If only a subset of the instances of a meta model is used, only this subset must be kept consistent. We illustrate this in Figure 3.1: the meta model $A$ is effectively constrained to $A^S$. The same is done with the other meta model $B$, resulting in $B^S$.

Only the overlap of the constrained meta models needs to be synchronized in case of changes. Approaches can explicitly incorporate the restrictions to avoid the specification of consistency relationships that are out of scope. Furthermore, they can avoid the creation of models that are valid, but not in the scope.

If meta models are constrained in a specific scope, the usage of a general consistency specification might not be sufficient for the scope. Consistency specification approaches can support the refinement of the specification for different, more specific scopes.
3 Aspects of Consistency Preservation

Figure 3.2: Multiple models can have both direct and indirect consistency relationships. If information is also propagated transitively, we need to avoid conflicts between the different paths of propagation.

3.4.3 Additional Information

A basic question about the realization of consistency specification is, whether we need additional information for deciding if two models are in the relation. This can encompass decisions that the realization makes about the mapping of structures inside the models, or user decisions.

We introduced this information as the witness structure of a model transformation in subsection 2.4.6. If this witness structure must also be considered for finding consistent states, then we cannot automatically derive which elements are used to describe some common information, for example by matching identical names or other attribute values.

In this case, there is an explicit representation for describing the additional needed matching information. The witness structure can contain additional information on the correspondence of model elements which are not otherwise distinguishable.

3.4.4 N-ary Consistency Relations

Usually, we need to consider consistency relationships between more than two meta models.

Consider Figure 3.2, which is an extended version of the illustration we used to motivate the problem in our introductory chapter (Figure 1.1). In a real world application, it might not suffice to only consider the relationships (a) and (b) and require that the relationship (c) can be automatically established using models that conform to the UML as a “hub” for the information between the PCM instance and the Java code.

This is already the case, if some information is represented in both the PCM and Java code, but not the UML model. One solution would be to find a different spanning tree that allows for this transitive propagation of information. This might, however, not always be possible.
3.5 Specification of the Relation

Then, it is beneficial, if the relationships (a), (b) and (c) are all specified, while explicitly referencing each other and only propagating the information in (c) that is not already covered by the transitive connection of (a) and (b).

For describing the relationship between \( n \) meta models, we must extend our binary relation \( R \) to a \( n \)-ary relation. For example, consider a ternary consistency relation \( R_{ABC} \subseteq A \times B \times C \). For this relation, we can inspect more properties:

- Assume we have described the relation \( R_{AB} \subseteq A \times B \) and \( R_{BC} \subseteq B \times C \). If we define

  \[
  R'_{ABC} := \left\{ (a, b, c) \mid (a, b) \in R_{AB} \land (b, c) \in R_{BC} \right\},
  \]

  then how does \( R'_{ABC} \) relate to \( R_{ABC} \)? Can we explicitly describe \( R_{ABC} \setminus R'_{ABC} \), and therefore simplify our specification through composition of the two relations?

- If we explicitly specify \( R_{AB} \), \( R_{BC} \) and \( R_{AC} \), and derive

  \[
  R'_{AC} := \left\{ (a, c) \mid \exists b : (a, b) \in R_{AB} \land (b, c) \in R_{BC} \right\},
  \]

  then do conflicts exist if \( R_{AC} \neq R'_{AC} \)?

3.5 Specification of the Relation

In the previous section, we described characteristics of the consistency relation. In this section, we want to discuss, how the pairs in the relation can be specified and characterized.

3.5.1 Relating Substructures

The most simple approach for specifying the consistency relation would be to explicitly state pairs of complete models that are consistent. However, this is not possible for more complex consistency relationships and meta models.

Usually, the user wants to explicitly separate parts of the meta models that are relevant for the consistency to another meta model from parts that are irrelevant. Therefore, \( (A, B) \in R \) is often decided by identifying substructures in \( A \) and \( B \) and deciding whether they have corresponding counterparts in the opposite model.

Subsequently, the relation \( R \) can be considered a composition, for example a conjunction or disjunction, of multiple subrelations \( R_i \) that relate substructures of the models. This division allows to handle the complexity of specifying the relation.

However, the relationship of those subrelations can also cause conflicts. For example, if two relations that require the existence and non-existence of a pattern at the same time, then their conjunction can never be true. Static and dynamic analyses are needed to warn users of such a situation.
3.5.2 Graph Patterns

A common approach to specifying these substructures is to interpret the models as graphs and to define graph patterns for the involved meta models that are in some way related to each other.

Independently from the domains of model transformations and consistency preservation, there exist various languages in the database community for specifying graph patterns. There, queries on graphs are important for retrieving relevant data and recognizing interesting structures in large graphs and databases of graphs. The query languages that are used in the database community, such as Gremlin\(^1\) or Cypher\(^2\), allow the declarative specification of graph patterns.

In the context of Triple Graph Grammars, beside the static specification of a set of meta classes and their attributes and references, there exist extensions for allowing the structures to be dynamic, including collection operators [42] and a star operator [61] for allowing a more expressive specification of patterns. Multi-amalgamation allows the combination of rules which also allows the definition of dynamic structures [59].

Usually, graph patterns are not specified by identifying concrete instances of model elements, but by providing constraints that refer to concepts in the meta model. This is of course similar to the way arbitrary imperative programs are specified, for example in Java, without knowledge of the actual input at runtime. However, if models are not expected to change to a great amount in specific contexts, it might be a viable alternative to refer to concrete instances of meta classes in a specification for consistency preservation.

Another possibility to define how some information can be found in one of the models and propagated to an element in the opposing model is to write (imperative) code. In the code, the source model is traversed, the information is located, processed appropriately, and the correct set of affected model elements in the target model is identified and altered. A similar transformation is then specified for the opposite direction, forming a pair of transformations which fulfill some notion of consistency.

This way of approaching the problem of consistency specification does, however, force the developer to understand how to traverse graphs in code, how to define structures that represent some information and match them in the source graph.

3.5.3 Meta Level of Pattern Description

In strict meta modeling, the separation of the different meta levels implies that there can only be an “instantiates”-relation between a model element and another model element on the meta level directly above [5] (cited after [6]). There are, however, mechanisms for breaking the strict meta levels, for example UML stereotypes or powertypes.

Languages for the specification of the relation can allow referring to those breaks of meta levels. This is particularly important, if patterns have to be specified that refer to elements on conceptually different meta levels. This can also manifest in meta meta model- or meta model-specific extensions of the pattern language.

---

\(^1\)http://gremlin.tinkerpop.com

\(^2\)http://neo4j.com/developer/cypher/
One example for how concepts can be expressed on different meta levels is *enumerations*, and model "libraries". Many meta meta models, including MOF, provide a mechanism for specifying data types with a fixed set of possible values, often called *enumeration*. The data type and its values are therefore provided in the meta model. Another method of expressing a set of values that are referenceable from the model level is to provide a model library, that is automatically referenced from all model instances. The mechanisms for referencing the different concepts are different, even if they are used for expressing similar information.

### 3.5.4 Mapping of Patterns

Previously, we have described, that it is a pragmatic approach to specify the relation \( R \) through a set of subrelations \( R_i \). Now we want to give a more concrete example for this relation of patterns, which we call *mapping*.

We tie the existence of a substructure that fulfills certain conditions in one model to the existence of an according structure in the opposite model. This concept is illustrated in Figure 3.3. The patterns for the structures are defined on the meta model level and we say that the patterns are mapped to each other.

We define two models as consistent, if for each match for a pattern \( P \) there exists a corresponding match for the pattern \( P' \) that \( P \) is mapped to (c). We also require that this correspondence is explicitly stated, and do not simply require the existence of a structure that adheres to the pattern. We consider this to be the difference between simply matching a corresponding structure and explicitly creating and maintaining a correspondence to a structure.

Two models are also consistent, if no patterns are matched at all (a). If a match for a pattern exists, that has no corresponding structure, the state is inconsistent (b).

Even though the illustration firstly displays states that are consistent or inconsistent, the different states might also occur in the order from top to bottom in a model-driven engineering process. First the models in the environment are in a consistent state (a). Then a user changes a model and creates an inconsistent state (b). Finally, consistency is reestablished by an automated process, that creates a corresponding structure for the match that was newly created by the user (c).

### 3.5.5 Mapping of Attribute Values

In addition to the previously described information that is contained in structures that we identify by patterns, we want to make constraints on attribute values on a corresponding pair of structures.

The patterns only cover one meta model and describe how to identify a structure. Attribute relations describe constraints on the pair of corresponding structures. They do not influence whether two structures are mapped to each other, however, they are relevant for propagating attribute values from one structure to the other. Because they apply after a corresponding pair has been created, we call them *postconditions*. In contrast, the conditions on the structures (the patterns) are called *preconditions*. 
Figure 3.3: Patterns for structures are specified on the meta model level. If matches are found in model instances that do not have corresponding structures, there exists an inconsistency.
3.6 Characterization of the Transformation

In the previous two sections, we described the consistency relation between model instances and how it can be described by relating patterns for substructures in the involved meta models. In this section, we first want to describe properties of a transformation that conserves this consistency relation in case of changes to a consistent pair of models.

For the consistency transformation $R' \leftrightarrow$ to be useful, there need to be constraints on the changes the transformation performs. For example, a transformation that only revokes changes after a user introduces an inconsistency would be useless in practice, even though it reestablishes consistency.

3.6.1 Dedicated Target Model

A common constraint on the changes is that the source model is not changed by the consistency transformation. This is also the notion of Stevens [83], that associates the two directional transformations $R^- : A \times B \rightarrow B$ and $R^+ : A \times B \rightarrow A$ with the relation $R$.

This means our generalization can be aligned with the formalization by Stevens [83] in the following way, where $\Delta$ is the change that triggers the synchronization:

$$
R'^+ (A, B) := \begin{cases} 
(R^- (A, B), B) & \text{\(\Delta\) changed } A \\
(A, R^+ (A, B)) & \text{\(\Delta\) changed } B
\end{cases}
$$

3.6.2 Validity and Scope

Independently from reducing the description of the consistency relation to a subset of the involved meta models regarding the validity and the scope (c.f. subsection 3.4.2), we can constrain our transformation.

A transformation that does not need to create valid or in-scope states can be easier to specify. However, if the transformation only creates valid and/or in-scope models, the space of consistent models to choose from is reduced. Furthermore, the transformation can yield more desirable results. For example, if we only create in-scope models, the results can be less surprising and more understandable for the users.

If a valid target model must be constructed, it may also not be sufficient to leave out information that cannot be deduced from the source but would be enough to create a serializable (syntactically correct) target. User interaction could then force the user to ensure that the target is valid.
3.6.3 User Interaction

If a change is to be propagated to a target meta model that is less abstract and thus contains more information that cannot be obtained from the source meta model, the missing information could be requested from the user, if it is needed to construct a syntactically valid target model. This can, for example, be the case if the type and the attributes of a source element that needs to be created can be deduced from the source, but not where in the target model it has to be referenced.

The user might decide between different possibilities that are encoded in automatic transformations but for which the information about which option to use is not automatically deduced. Czarnecki and Helsen describe this as interactive rule application and scheduling [27].

3.7 Specification of the Transformation

After describing characteristics of the transformation, we now want to describe how we want to specify this transformation. First, we will introduce constraint solving, a method that we did not choose for our approach and motivate why. Then we will describe our approach, deriving imperative code from the declarative specification of the consistency relation, and the relationship of different parts of the consistency transformation.

3.7.1 Constraint Solving

One possibility of implementing the consistency transformation is to use constraint solving, similar to logic programming. When using constraint solving, the space of possible consistent solutions is described in form of logical clauses (constraints) on the target model. Then, a mechanism for choosing the correct solutions must be implemented.

If there exist multiple possible consistent models for a given model, we need to decide on one of the possible consistent states. For example, we can derive the decisions from the triggering change, ask the user for a decision, or rank the states according to a metric such as least change or least surprise [23].

There are multiple approaches that use constraint solvers to derive correct target models of model transformations, or changes that restore consistency [25, 63].

If constraint solving or other approaches are used, where rules are interpreted and executed by an engine, the user additionally needs to understand this engine to correctly predict how the transformation behaves.

3.7.2 Deriving Imperative Code

Another approach to deriving a transformation from the consistency specification is to derive imperative code. In case of a change, the appropriate entry point into the code is chosen and the code is executed. This is an advantage, if the user is familiar with imperative programming languages. Furthermore, debugging of this kind of code is straightforward, because the user can simply pass through the code statement by statement. Imperative
model transformation code has a dedicated *direction* in form of a specific input and output model. Different blocks of code are executed, if the other direction is needed.

In the following, we will reflect on different pieces of imperative code that are derived from the consistency relation and how they relate to each other. First, we need to introduce a definition for *bidirectionality*. According to Stevens,

> A bidirectional model transformation is some way of specifying algorithmically how consistency should be restored, which will (at least under some circumstances) be able to modify either of the two models. [83, p. 410]

The author further recognizes that it is not necessary that there is a single specification from which both functions are derived [83, sec. 3.2], because one could also achieve that both explicitly specified directions are consistent in some other way. If the consistency between the transformations is, however, manually established and maintained there is the danger of inconsistencies between the directions.

Additional mechanisms for ensuring that the two independently specified directions of a model transformation are consistent to each other are also thinkable. There are existing approaches to “faking” bidirectional transformations using two unidirectional transformations and a constraint language which describes the consistency they should preserve [72].

### 3.7.3 Consistency of Transformation Parts

If we derive imperative code from a relation on meta models that is structured in sub-relations, as we described in subsection 3.5.1 and subsection 3.5.4, there are different transformations that need to be consistent to each other. In the following we will describe those relationships. The different transformations that we need in this context are displayed in Figure 3.4.

We need to be able to both check if a substructure inside a model conforms to our pattern and to create substructures that conform to a pattern. Therefore, we need to establish consistency between the *checking* and *enforce* of a pattern on a structure. This pattern spans only one of the meta models. The *check* method checks if the pattern applies to the structure. The *enforce* *true* method makes sure that a structure is matched by the pattern, or might create a structure that is matched by the pattern. The *enforce* *false* method ensures that a structure does not match the pattern anymore, possibly by deleting the structure.

There is not necessarily only one possibility for deriving consistent methods from a declaration of a structure. If the meta meta model allows a concept similar to inheritance of object-oriented programming languages, the pattern might include abstract meta classes. Then, a decision has to be made when creating (or enforcing) this precondition, regarding which concrete meta class to instantiate. Similarly, there are other conditions on attributes or structures for which more than one configuration that evaluates to *true* exists: for example, if a precondition dictates the length of a string-valued attribute to be fixed to a constant value, the *enforce* or *create* methods must choose a specific value, possibly considering other constraints that involve the attribute.

In addition to a pattern that matches a structure in one model, we also specify post-conditions on a pair \((A, B)\) of corresponding structures, as described in subsection 3.5.5.
3 Aspects of Consistency Preservation

The parts of derived imperative code have to be consistent. Structures that are modified to match (enforce true) or not match (enforce false) must create an appropriate result of the check method. The attribute propagating methods \( R_1^- \) and \( R_1^+ \) need to preserve information in a round-trip from \( \mathcal{A} \) to \( \mathcal{B} \) and back.

The unidirectional transformations \( R_1^- \) and \( R_1^+ \) propagate attribute values from \( \mathcal{A} \) to \( \mathcal{B} \) and from \( \mathcal{B} \) to \( \mathcal{A} \) respectively, to reestablish consistency to the postcondition. They also have to be consistent to each other. We will describe one notion of consistency of \( R_1^- \) and \( R_1^+ \) in the following section.

### 3.7.4 Well-Behavedness of Attribute Propagation

A characterization of consistent behavior of two directions of attribute propagation are the *lens laws*. They essentially describe that there is no change in \( \mathcal{A} \), if we first apply \( R_1^+ \) to propagate its values to \( \mathcal{B} \) and subsequently propagate them back via \( R_1^- \).

A *lens* [37] is a view transformation that propagates changes in the view back to the source. Because lenses have been originally proposed for an abstract source and a concrete view, they are inherently *asymmetric*. In the context of lenses, there are two *round-tripping laws* that have to apply for a lens to be *well-behaved* [37]. If the view is extracted from the source (via the so-called *get* function), and the unchanged view is immediately *put* back into the source, the source must not be changed (the *GetPut* law). If a (possibly changed) view \( \nu \) is *put* back into the source, extracting the view immediately afterwards via *get* must return the view \( \nu \) (*PutGet*). A lens is called *very well-behaved*, if two consecutive *puts* have the same result as only executing the second *put* (*PutPut*).
3.8 Characterization of Multiple Transformation Steps

The previous considerations covered the transformation from an inconsistent to a consistent model state. In the following, we want to describe properties of multiple consistency transformation steps that are triggered consecutively.

3.8.1 Size of Deltas

Before discussing properties of multiple transformation steps, we must first consider that it is not trivial to decide the size of the changes after which we trigger the consistency transformation. It might be necessary, to consider compound changes and to translate them in to compound changes, instead of only synchronizing atomic changes on the models.

The general concept is displayed in Figure 3.5. If we achieve a different result if an aggregated change is synchronized as such instead of as a series of atomic changes, this must be clear to the user of the consistency synchronization approach. Depending on the consistency preserver, different sizes of deltas can result in different synchronization results, if $\Delta_1^A + \Delta_2^B \neq \Delta_1^{\text{sum}}$, where $\Delta_1^A$ is the result of the consistency transformation applied to $\Delta_1^A$.

3.8.2 Caching of Information

It must be clear to the user of a consistency transformation whether the transformation removes information from the opposing model that can not be reconstructed from the
source model. In this section we want to discuss when this problem arises and how it can be handled by a consistency transformation.

As we already described previously, our notion of consistency ties the existence of a structure that fulfills a set of constraints in one model to the existence of a corresponding structure in another model. After a change by the user, a structure in the source model might not fulfill the conditions that previously implied the existence of a corresponding structure in the target model anymore, and which previously triggered the creation of this structure.

When we remove elements from the target model in the transformation, we can remove additional elements or attributes from the target model, if they are attached to or contained in the removed elements. If in a subsequent step, another change brings the source model to a state in which the previously removed model elements are created and associated with the source structure anew, the information that was only contained in the target model is effectively lost.

This is illustrated in Figure 3.6. The pattern for \( A \) consists of a model element \( Ac \) and its relation to a model element of type \( A \). For the consistency preservation mechanism it might be a different result if multiple changes are considered at once (as a “update” of \( ac \rightarrow a1 \) to \( ac \rightarrow a2 \)) or if the removal of the reference \( ac \rightarrow a1 \) is synchronized, and consequently the creation of \( ac \rightarrow a2 \) is synchronized.

If we simply let the user change the structure in the source model, and remove the information about the mapping of the two structures, there is now a synchronizable structure left in the target model, because it was just in a correspondence relationship. Therefore, the previously corresponding structure in the target model still implies the existence of a corresponding structure in the changed source model.

There are two possibilities for resolving this inconsistency. One possibility would be to create a new corresponding structure in the source model. We consider this to be unintuitive as a change propagation, since it effectively revokes the change the user made.
If we restrict the impact of the consistency transformation to the target model, we need to enforce that the target structure is not matched anymore. There are in general many possibilities to alter the target model elements to achieve this, from the removal of the whole structure, to minimally altering only one constrained attribute to negate the pattern match. If we destroy the target structure, we lose the information that is attached to or contained in it and that is not derivable from the source model.

If a consistency mechanism actively takes precautions by analyzing which information cannot be deduced automatically and caching it in an additional data structure, or by actively warning the user if information is lost, accidental misuse and erroneous or incomplete specification of consistency rules and a negative impact on the model-driven engineering process can be prevented.

This however also requires a strategy for maintaining this cache of model information that is removed by a model transformation, since there is also deliberate removal of information from the system.

The information described here is also similar to the complement function as it is used for formalizing symmetric lenses [48]. This complement is the information in a transformation that is not contained in the target of a model transformation and is needed to reconstruct the source from the target.

Greenyer et al. describe a method of preserving information while synchronizing models using Triple Graph Grammars, which avoid the deletion and subsequent recreation of model elements by reusing elements of TGG rules that are deleted and created during the same synchronization step [41].

### 3.8.3 Oscillations

If model transformations are composed or multiple transformations are applied at once, a consistency preserving system might need to also synchronize the changes that are made by the consistency transformation. If those synchronizations, in turn, generate changes that contradict the original changes or require further propagation, then loops which result in an oscillation between different inconsistent states or “ripple” effects might be generated.

Static analysis of the transformations can help in preventing those effects or managing the impact of changes.

### 3.8.4 Undoability

After a change $\Delta_A^\to$ that takes a pair of models from $(A, B)$ to $(A', B)$ where the latter state is inconsistent, the consistency transformation will create a state $(\bar{A}, \bar{B})$. If we then apply $\Delta_A^\leftarrow$ (i.e., the reverse of $\Delta_A^\to$) on $A$ and achieve a state $(\bar{A}', \bar{B})$, an additional property of the transformation is, whether the original state $(A, B)$ will be reached after we apply it. If a different state is reached, this could come as a surprise to the user and weaken the understandability of the consistency repair process. Xiong et al. call this property undoability [88].
Figure 3.7: Undoability of a consistency transformation is given, if we arrive at the same state $B'' = B$ after synchronizing a change $\Delta_A^+ \rightarrow$ and subsequently synchronizing the reverse change $\Delta_A^-$.

Figure 3.7 illustrates this property for an unidirectional model transformation $t_{AB}$. In general, the transformation might also take more input than only the change $\Delta_A$, for example the changed model $A$ and the previous target model $B$.

### 3.9 Consistency Specification in the Model-Driven Engineering Process

In the previous sections, we have described a notion of metamodels, the consistency relation and consistency transformations. We introduced characteristics regarding consistency and specification and their interplay. In the following, we want to give some indicators on properties of consistency specifications and transformations that are relevant if they are used in a model-driven engineering process.

#### 3.9.1 Model Transformation and Meta Model Co-Evolution

The meta models that are used in a model-driven engineering process can be subject to change. Model transformations, and therefore also model consistency preservers, also need to be synchronized with the changing meta models. This is needed both on a syntactic level, for example when we change the name of a meta class, and on a semantic level.

Some model transformation languages also allow the developer to treat the transformations as models. Then, techniques for model-driven engineering can be used for analyzing and transforming the transformations via so-called higher-order transformations. Particularly, approaches for consistency, can then also be used for synchronizing model transformations with changing meta models.

For example, model transformations that refer to changed meta model element could be marked, or simpler changes, such as the change of the name of a meta class, could be propagated directly to the consistency specification. Integrated development environments already offer refactorings that consistently change the code base, i.e. the renaming of a Java class in the Eclipse IDE.
3.9 Consistency Specification in the Model-Driven Engineering Process

3.9.2 Manual and Automated Consistency Preservation

Model-driven engineering is based on the idea that all artifacts that describe a system can change during the development process, which necessitates that changes are propagated to all other artifacts that are used. Therefore it is crucial that developers are not deterred from introducing changes to artifacts by possible overhead regarding the propagation of changes. However, a complete specification of the consistency requirements might be too much of an initial effort; additionally the understanding of the domain and the requirements of the system under development might change during the development process.

Some inconsistencies might be resolved fully automatically, while others only result in indicators on how to resolve them in form of a change impact analysis which marks possibly affected artifacts. There may also be inconsistencies for which no form of automated resolution is supported by the system (yet), but which result in a more general warning.

Figure 3.8 illustrates different levels of user involvement and automated consistency preservation that can be covered by consistency mechanisms. General warnings which do not indicate specific models or model elements can help a developer identify situations where additional effort is needed for consistency. A change impact analysis can make an overestimation of the possibly affected elements in other models and guide the developer when manually reestablishing consistency between different artifacts. In some cases, user interaction can be included in an automated consistency repair process, or the changes can be propagated fully automatically to other artifacts.

Note that we do not consider a consistency preservation approach to be located on exactly one of those levels. It might even be beneficial if the approach allows a specification of different levels of automation for different purposes, since we assume that a fully automated reconciliation might not be possible for all cases, and additionally is in general more complex to specify by a user of the system than for example a change impact analysis.

The aforementioned possible and dangerous tendency to avoid changes might, of course, also stem from aspects of the model-driven engineering environment that are not directly related to consistency between different models, such as transactions of changes to a model that are not easily revertible, versioning, or the usability of the tooling for editing.

3.9.3 Partial and Comprehensive Automated Consistency Preservation

In Figure 3.9, we illustrate the relationship of consistency requirements and the coverage by automatic consistency synchronizers. We assume that for non-trivial systems and meta models there is a gap between the actual semantic overlap of different models of the system (semantic overlap), the understanding of the semantic overlap by methodologists (known consistency requirements), and the implementation of automatic consistency preservers (specified consistency requirements). Additionally, the used system for consistency synchronization might not be able to cover all possible consistency requirements (specification possible).

Since there is an overhead associated with closing those gaps there is a trade-off between the accepted inconsistency and the cost of completely covering the consistency requirements of the project. It might be a pragmatic alternative for a model transformation.
Figure 3.8: Different levels of automation for consistency preservation

Recognizing Uncertainties through Inconsistencies

When changing a system by changing one of the constituting artifacts, it is not realistic to assume that the system only passes through states that we can automatically make consistent.

The insight that inconsistencies are introduced into a system description and that they can be useful during the design process for resolving uncertainties about the system under design, if they are known, is not new and is described by Balzer [10] or Nuseibeh et al. [69] and also central in the ViewPoints framework [68, 34]. Inconsistencies are introduced into the system and might be resolved in later steps, either by changing other parts of the system or by revoking the change that initially introduced the inconsistency.

For example, in test-driven development, tests that the system does not pass are created to represent missing or aspired functionality before changing the software to pass the tests. Similarly, contracts for interfaces may be specified to validate the code implementing them, for example by pre- and postconditions and invariants.
3.9 Consistency Specification in the Model-Driven Engineering Process

Figure 3.9: There are different levels of completeness of the consistency specification for automated consistency preservation. Not all of the possible consistency relationships are actually automated (cf. Figure 3.1).

Depending on the notion of consistency this might not be considered an inconsistency in the description of the system, or the goal of the software engineering process as a whole is to resolve this inconsistency. Resolving inconsistencies of the implementation with tests and contracts via automatic adaption in case the code changes might defy their original purpose.
4 The Mapping Language

In this chapter we present our approach to specifying consistency requirements between models with the mapping language. The mapping language is a declarative DSL which allows the definition of patterns in meta models that are synchronized by transformations that are generated from the specification. We will show the features of the language, how it can be used, and its limitations.

First we will give an introductory example for a simple consistency specification that is possible with our approach. Then we will introduce the different language features and features of the framework of the language. Finally, we will show the limitations of the language and how it is embedded in the VITRUVIUS framework and the MIR language family.

4.1 Two Introductory Examples

4.1.1 Simple One-to-One Mapping

An example for the simplest mapping specification possible is shown in Listing 4.1. The specification consists of two imports that specify the meta models that are used in the mapping specification on lines 1 to 4. In this case they are two very simple meta models that only contain a small subset of the meta classes in a Unified Modeling Language (UML) class diagram meta model and a meta model for a component-based software architecture, the Palladio Component Model (PCM), and are called UML mockup (muml) and PCM mockup (mpcm) respectively. We give each imported meta model an identifier with the as keyword. This identifier is used for referencing them in the rest of the specification.

```
import "http://edu.kit.ipd.sdq.vitruvius.tests.metamodels.pcm_mockup" as mpcm
import "http://edu.kit.ipd.sdq.vitruvius.tests.metamodels.uml_mockup" as muml

mapping repo2pkg:
map [ mpcm.Repository ]
and [ muml.UPackage ]
```

Listing 4.1: A simple example mapping of two meta classes

Figure 4.1
4 The Mapping Language

In lines 6 to 10 the mapping repo2pkg between the meta class Repository from the meta model mpcm and the meta class UPackage from the meta model muml is specified. The image on the right of the textual specification shows a simple graphical representation of the mapping for easier understanding of the mappings for this thesis. There is, however, no graphical syntax currently available for specifying mappings in an editor.

If the user generates code from this mapping specification and plugs it into the VITRUVIUS environment, changes in instances of each meta model are passed to the generated mapping code. If the user now creates a new instance of Repository, places it inside a model instance of mpcm and saves this change, the system will recognize the object creation and check the newly created model for applicable mappings. Because the mapping repo2pkg is applicable to the Repository and the model element is not mapped, the consistency transformation creates a UPackage. Because the location where the UPackage has to be persisted is not specified, the user is then prompted for a location for the newly created model element. The system saves the newly created model and additionally stores the information that the newly created Repository and UPackage are mapped by the mapping repo2pkg. The result of this simple example is displayed in Figure 4.2a.

Our user could also have initially created a UPackage instead of the Repository and arrived at the same result via the consistency transformation, because our mapping specification has no direction – it only declares that the two meta classes are mapped to each other.

4.1.2 Nested Mapping and Conditions

In the following example, we extend the example from subsection 4.1.1. We additionally map a Component that is contained in the previously mapped Repository to a UClass that is contained in the UPackage that corresponds with this Repository.

Listing 4.2 shows the code for the slightly more complex mapping specification that achieves this, again with a graphical representation in Figure 4.3. The first ten lines are equal to the specification of the first example (Listing 4.1). Lines 12 to 21 define the second

![Figure 4.2: The consistent states (a) and (b) are created by the transformations generated from Listing 4.1 and Listing 4.2 respectively.](image-url)
4.1 Two Introductory Examples

```python
import "http://edu.kit.ipd.sdq.vitruvius
    .tests.metamodels.pcm_mockup" as mpcm
import "http://edu.kit.ipd.sdq.vitruvius
    .tests.metamodels.uml_mockup" as muml

mapping repo2pkg:
    map
    [ mpcm.Repository ]
    and
    [ muml.UPackage ]

mapping component2class:
    when (repo2pkg as r2p)
    map [ mpcm.Component ]
    with [ in(component,
        r2p::repository.components) ]
    and [ muml.UClass ]
    with [ in(uClass, r2p::uPackage.classes) ]
    {
        [ equal(component.name, uClass.name) ]
    }
```

Listing 4.2: An example mapping with a nested mapping

![Diagram](image-url)
mapping named component2class. In this mapping the meta classes Component from the
meta model mpcm (line 14) and UClass from the meta model muml (line 17) are mapped.

In addition to the signatures containing the meta classes, the mapping specifies a required
mapping, signature conditions and body conditions. The mapping repo2pkg is required in
line 13. For referencing parts of the required mapping more easily, we associate the name
r2p with it.

In line 15 we specify a signature condition for the side of the mapping that belongs to
the mpcm meta model. The expression \texttt{in(component, r2p::repository.components)} requires
that the element \texttt{component} is referenced by the \texttt{repository} element in an instance of the
mapping repo2pkg.

The name \texttt{component} is automatically created for the meta class mpcm.Component which
is specified in line 14. We derive default names for meta classes, required mappings and
mappings for convenience reasons. All elements in signatures can also be named explicitly
using the keyword \texttt{as}. For example, the Component could be given the name \texttt{comp} and later
referenced by this name, by replacing line 14 with \texttt{map [ mpcm.Component as comp ]}.

Similarly, the condition in 18 requires that the UClass is referenced by the uPackage
in the same instance of repo2pkg. This essentially means that we do not only require the
Component and the UClass to be referenced by a Repository or a UPackage, respectively,
but that we also require that those two elements are already mapped by the repo2pkg
mapping.

Line 20 specifies a body condition for the mapping. This equals condition means that
when a Component and a UClass are mapped by the mapping component2class, their
respective name attributes have to be equal at all times.

Again, we can generate code from this specification and hook it into the \textsc{Vitruvius}
environment. If we start with the same change as in the previous example, creating a model
containing a Repository, and synchronize it, we again result in the state displayed in Fig-
ure 4.2a. If we then create an instance of mpcm.Component with the name “AccountManaging”
and synchronize, nothing will happen, since the signature condition specified in line 15 is
not fulfilled. Only when we add the Component to the reference components of the Repos-
itory already mapped by the mapping repo2pkg, the transformation will synchronize this
state by creating the signature for the other meta model for the mapping component2class.
In this case this means that we need to create an instance of muml.UClass. Furthermore,
we want to reach a state, where the condition in line 18 evaluates to true. For achieving
this, we have to add the newly created UClass to the classes reference of the UPackage
the previously determined Repository is mapped to.

As a last step, we need to ensure that the body condition in line 20 evaluates to true.
Because the change by the user was made in the instance of the mpcm meta model, and we
have as a result changed the muml instance, the system propagates the attribute value of
name from the Component to the newly created UClass.

A mapping can of course also be instantiated multiple times. Figure 4.2b shows the state
after a muml.Class with the name “Logger” has been added to the mapped Package. This
time the change was made in the opposite model.
4.2 The MIR Language Family

In the previous section, we introduced the mapping language in two simple examples. Before we describe the language features and syntax in detail in the next section, we will first describe the MIR language family to give an overview of the context of the mapping language.

4.2.1 Parts of the MIR Language Family

The MIR language family is a family of languages that can be used for the specification and generation of change-driven transformations. The generated transformations can automatically repair inconsistencies that are covered by the specification.

Figure 4.4 shows the three languages that are part of the MIR language family. MIR stands for mappings, invariants and responses.

The mapping language is a language for specifying a subset of the possible relationships between meta models declaratively and bidirectionally by describing structures that are mapped to each other. The mapping language is the focus of this thesis.

Using the invariants language, conditions can be specified (on the meta model level) which can be checked for instances of the meta model. The invariants are realized by XOCL [36], which is a language that is based on Xbase. XOCL is designed to be similar to the widely used Object Constraint Language (OCL) [49]. In XOCL, conditions for meta class can be specified. Since in many cases – and especially for the specification of transformations that repair (consistency) problems in models – developers need to find the elements that are responsible for the violation of an invariant, the approach also allows the derivation of queries that select violating elements from the code for checking an invariant. There is also an approach for deriving XOCL constraints from OCL constraints which enables the reuse of existing OCL invariants for the meta models that are used in the Vitruvius environment, while benefiting from the added functionality.

The response language is a language for specifying unidirectional, change-driven transformations. In Vitruvius, changes are recorded and passed to transformations as instances of a change meta model. When defining a response in the response language, developers can specify conditions on a change. This condition can for example specify the type of the affected meta class, the affected meta model or that it is a creation, update or deletion. Additionally, actions that are to be taken as a result of a change that fits the conditions can be specified via a range of basic actions like the creation of a corresponding element, or the deletion of such an element. Additionally, arbitrary code that is specified in Xbase can be attached to the response. Theoretically, all consistency transformations could be specified in the response language in an imperative and unidirectional way.

4.2.2 The Mapping Language

The mapping language does not cover all consistency relationships between meta models. In fact, our aim for the language is the opposite. Figure 4.5 illustrates this idea. General purpose programming languages, like Java, provide the developer with a very powerful set
of language features that can in principle be used to specify arbitrarily complex transformations of data, and therefore also of models. However, if a lot of code has to be written for a specific domain, domain-specific languages simplify this task by providing suitable abstractions for the domain. This can reduce the amount of similar code or slightly varied code that has to be written for similar tasks. As depicted, model transformation languages are domain-specific languages for the domain of model-driven engineering. They provide features for loading, storing, traversing models, or specifying relationships between model elements that are managed by a model transformation engine.

The mapping language is such a model transformation language. However, we think that the scope of the mapping language can be even narrower than that of arbitrary model transformation languages. Our aim is to provide powerful abstractions that make the consistency relationships between meta models easier to specify for a methodologist. Additionally, we want to make the effects of this specification regarding automated consistency repair clear.

In chapter 3, we gave an overview of the varied consistency problems that need to be tackled by a consistency preservation system. By providing abstractions that, for example, allow the domain expert to specify the information that has to be supplied by a user to the consistency preservation process, or how information is differently represented in heterogeneous meta models explicitly, the description of the consistency requirements could be made more declarative. The domain expert does not need to understand, how the solution of the consistency preservation problem looks like.

In the case of the mapping language as a part of the MIR language family, we do not need to cover all possible relationships between meta models, since all needed transformations that are not covered can be implemented using the response language. In fact, our code generator for the mapping language even generates responses from the mapping specification. Via this approach of specifying consistency requirements on different levels of abstraction using different paradigms (declarative and imperative) we aim to provide the users of our approach with the right tools for the right tasks.

Consistency relationships can be specified using the mapping language and its future abstractions for more high-level and bidirectional specification. Relationships that cannot be implemented in the mapping language in a declarative and bidirectional way can be implemented unidirectionally using responses. Therefore we do not lose expressiveness of the language family.

Additionally, in the future we also plan to integrate the different languages to a higher degree by making references between the different languages and their execution mechanisms possible. For example, violated invariants could be referenced in responses or reused
in mappings as conditions, or correspondences that are generated by a mapping could be used in a response to attach additional unidirectional code for propagating attributes. Furthermore, static code analysis can be used to prevent conflicts between the declarative mappings and the imperative responses from going unnoticed.

### 4.3 Language Features

In the previous sections, we first gave an introductory example to the basic functions the mapping language supports, and how Vitruvius, extended with the generated transformation, behaves in case models are changed. Furthermore, we shortly described the other languages in the MIR language family. In this section, we will introduce the syntax and semantics of the mapping language.

#### 4.3.1 Syntax

The mapping specification file first specifies two distinct meta models for which the consistency rules are specified. This is done with the `import` statement, followed by the namespace URI\(^1\) of the meta model and an optional name with which the import is referenced in the file. The namespace URIs are used to find the `EPackages` in the package registry of the EMF\(^2\). Here is an example for this `import section` which imports two meta models we used for testing purposes in our project:

```plaintext
1 import "http://edu.kit.ipd.sdq.vitruvius.tests.metamodels.pcm_mockup" as mpcm
2 import "http://edu.kit.ipd.sdq.vitruvius.tests.metamodels.uml_mockup" as muml
```

\(^1\)The namespace URI (Uniform Resource Identifier) is the value used to identify a meta model.

\(^2\)The `EPackage.Registry` holds the code of the meta models that are loaded in the current plugin context (the context of the Eclipse IDE).
4 The Mapping Language

Each of the mappings specified in the mapping language describes a structural consistency relationships between models in a bidirectional, declarative way on the meta model level. The general structure of a normal mapping is shown in Listing 4.3. Each mapping consists of the following elements:

- an optional `name`,
- a set of meta classes for each involved meta model with optional names. This set is called `signature`.
- An optional set of `signature conditions` that have to be true for each involved signature,
- an optional set of `body conditions` that have to be true for a concrete instance of the mapping and which can involve both sides of the mapping, and
- an optional list of (not necessarily unique) `required mappings`, which are uniquely named.

In our current implementation, the following statements are possible as signature conditions.

- `in(element, referencingElement.feature)`
  Checks whether `element` is referenced by `referencingElement` by the feature `feature`. If the feature is single valued, check for equality, otherwise check if the element is contained in the feature value.
  To enforce, if the feature is single valued, set it to `element`, otherwise add `element` to the multi-valued feature.

- `nonnull(element.attribute)`
  Checks whether the given attribute of the element is not equal to null.
  To enforce, set the attribute value to the default value of the attribute type.

- `equal(element.attribute, literal)`
  Checks whether the given attribute of the element is equal to the given literal value.
  To enforce, sets the attribute value to the given literal value.
4.3 Language Features

- **default-contain**(element, referencingElement.feature)
  Checks whether element is contained in *any* element, not necessarily referencingElement. Containment is a special form of reference in the framework we are using. Containment is similar to *composition* in the UML which is used when “the composite object has responsibility for the existence and storage of the composed objects” [71, p. 110]. In EMF, all model elements must be contained (either directly or indirectly) in a Resource. The elements that are directly contained in a resource are *root elements*. To enforce (if no containment is found), add the element to the given (containment) feature of referencingElement.

- **default-resource**(element, resourcePath relative-to element)
  Similarly checks whether element is contained in *any* element. To enforce (if no containment is found), put the element inside a new Resource which is located at the given path. Optionally, an element can be specified, whose storage path is used to calculate the location of the new element, if the given resource path is relative.

- **xbase check** { ... } **enforce** { ... }
  Uses the given code block (specified in Xbase) for checking if the signature condition is applicable. The code given in the second block (after the **enforce** keyword) is used to create a state where the condition evaluates to true.

The body conditions can currently only consist of the following two conditions:

- **equal**(elementA.attributeA, elementB.attributeB)
  If the elementA.attributeA is changed, then the change is propagated to elementB.attributeB, meaning that the value is set to the new value. The other direction happens, if elementB.attributeB changes.

- **xbase from** metamodel with { ... }
  If an element of the signature of the given metamodel is changed, the specified imperative code block is executed.

In addition to this very simple body condition and the fallback to a manual and specification of both directions for achieving a consistent state, there has been work in the context of VITRUVIUS for automatically deriving both directions from a declarative specification of much more complex attribute relationships of the two structures involved in a mapping. The approach has, however, not yet been integrated with our mapping language. Since it is based on the same underlying technology, we hope that we can in the near future integrate the two approaches to allow more expressive specification of body conditions.

4.3.2 Semantics

If, after a change, an allocation of the required mappings and a structure is found in the changed model that fulfills all signature conditions, but is not yet mapped, i.e., for which no corresponding signature of the other meta model exists, the opposite structure is created
4 The Mapping Language

by the consistency transformation. For this purpose, all of the aforementioned created enforce-methods are used.

We call the structures that correspond to each other according to a mapping specification, and for which a correspondence is persisted, mapping instances.

We do not create a correspondence between two existing structures, but always create the opposite side immediately after a match for a signature has been found. The semantics of a mapping are not to finding as much as possible of the opposite side in case of a match, and to extend or alter it to fit to the mapping specification, but to create an opposite structure from scratch and an correspondence, and then maintaining the newly created correspondence. This is done until one of the sides does not conform to the signature description anymore. In this case, the opposite side of the correspondence, i.e., the structure that corresponds with the structure that is not conformant anymore, is destroyed.

This is based on our assumption that changes are small enough that they do not contain the creation of corresponding structures in the two models that are to be synchronized. If this was possible, then we would also need an approach in the tooling or in our framework that allows the creation of correspondences for existing pairs of structures that are "detected" as mapped.

The system then ensures that all signature conditions of the opposite side hold, by applying all abovementioned enforce methods on the newly created structure. Then the information that the two structures correspond according to the mapping and with the given required mapping allocation is saved in the framework. Lastly, a state is created, where all body conditions evaluate to true, using the propagation method for each of the body conditions.

After changes, the system always ensures that the body constraints for mapping instances evaluate to true, as long as both structures individually fulfill the signature constraints, i.e., as long as there is a correspondence between the structures.

If the signature constraints are not fulfilled anymore for one of the structures, the system deletes the correspondence and the structure in the opposite model. Because our synchronization mechanism is triggered after a change in only one model, i.e., there are no changes that span multiple models, the signature conditions can only be violated in one of the models before synchronization.

4.3.3 Default Mappings

In addition to the normal mappings, we implemented a feature called default mappings. Default mappings are a special kind of mapping that only specifies a single signature. Our system ensures that one instance of the signature that is known to our system exists at all times. The instance can be referenced by other mappings by the normal mechanism for required mappings as described above, but does not have to be specified explicitly in the list of required mappings. A default mapping is specified as follows:

```
default [name]
create [signature]
with [signature conditions]
```

The different parts of the specification behave exactly as in the normal mapping case.
4.4 Design of the Mapping Language

In the current implementation, the mapping language is similar to other approaches for specifying bidirectional model transformations that relate model element structures that conform to defined patterns, such as Triple Graph Grammars (TGGs).

The initial aim for our mapping language and the scope of the prototype built in this thesis was, however, different from TGGs and the current implementation. The language evolved during the work in this thesis. Additionally, we want to add features to the language in the future that will make the different aim of our approach more apparent.

In this section we will first give a short introduction with a classification of our approach. Then we will describe the previous iteration of the design of the mapping language and give some examples that influenced the current design. Additionally, we provide an outlook for the future design iterations of the approach.

4.4.1 Language Classification

Regarding the classification by Biehl (see subsection 2.4.1 and [17]) the mapping language and the transformations generated from it have to be considered separately. The mapping language is designed to be a declarative DSL used to generate imperative, in-place (and therefore incremental) model-to-model transformation. The model transformation problem answered by the mapping language can – but does not have to – contain a change of the level of abstraction, contains a change of meta models, is bound by the current implementation to exactly two EMF models and preserves semantics which are shared by the two meta models. If changes are considered transformations themselves (i.e., the change made by the user is a small model transformation), the specification in the MIR language family describes a higher-order transformation (HOT) which is executed by the VITRUVIUS framework and which transforms the change in one meta model to a semantically consistent change in the opposite meta model.

4.4.2 Mappings for Single Meta Classes

Previous to the work in this thesis, the mapping language had been prototypically implemented in a basic form. Our first approach was to allow the declaration of mappings between meta model classes (map ... and ...), and conditions under which the mappings are in effect (when-where { ... condition ... }). Mappings could be nested (with featureA[TypeA] and featureB[TypeB] { ... }) which is equal to stating the condition that there is a parent of type A that points to the child and is already mapped by the outer mapping. Properties that must be kept consistent between the mapped elements could be stated in a similar fashion (with-block { ... }). Similarly to the current implementation, we also derived check and enforce code from the conditions on the containment, in this case only the containment condition.

The first extension of the concept was to allow the definition of “chains” of meta classes, that are mapped in one of the meta models instead of single meta classes. We initially planned this as “syntactic sugar” for specifying multiple nested mappings. This was done by specifying a meta class and a sequence of features. This way, simple graph patterns
(consisting only of simple paths) could be specified and mapped. While inspecting case studies that involved mappings which we also wanted to cover in our mapping language, the need for more complex graph structures quickly became apparent. For example, the approach could not be used for specifying something as simple as a pattern for a Class meta class that has a Feature and a getter and a setter method.

Additionally, the simple notion of providing chains of meta classes that are rooted in one of the specified classes, and an initial correspondence model that only allowed the mapping of exactly one model element to an opposite model element lead to the constraint that there is one “primary” model element for each mapping, in which additional nested mappings are rooted. Consider the mapping definition in Listing 4.4 in our previous approach, where A, Ac, B and Bc are meta classes and acs and bcs are features of A and B, respectively.

It is not really clear, if an instance of A should be mapped to an instance of B, even if there are no children, or if only the combination of an A and an associated Ac should be mapped to a corresponding B and a Bc. We considered different relationships between the parent and the nested mappings, but found the notation not intuitive and lacking in expressiveness.

### 4.4.3 Mappings for Model Element Structures

Based on those observations, we arrived at a generalization of the single mappings and chains of mappings, by specifying graph patterns in the signatures and allowing the user to explicitly state the relationship of the elements in a signature. Additionally we replaced the nesting of mappings by a requires-relationship that can be explicitly stated, therefore removing the restriction of having no or exactly one required mapping (the parent).

The specification of graph patterns is similar to the graph patterns that are specified for TGGs. A mapping relates two “signatures” which consist of a fixed sequence of meta classes and additional preconditions on the classes as described in the previous sections. The meta classes implicitly contain the condition, that an element of the specified type exists, and make the element referenceable in other conditions. Furthermore, a mapping can contain a body that contains conditions that involve both signatures.

The direction association of features (i.e., attributes and references) is replaced by a constraint language that can relate all the elements in the current scope instead of only allowing the definition of “mappings” for features of the meta class that is the current “primary element”.

Those considerations and the process of arriving at the current design of the mapping language are important for the following reason. We aimed to provide a language that
4.4 Design of the Mapping Language

is as simple as possible for describing simple rules. However, our initial prototype was not expressive enough for slightly more complex examples. In the future, we can aim to include the simple abstractions into the language again for specifying specific cases such as “chains” of meta classes, while having understood how the resulting generated transformations should behave in different change scenarios from our more sophisticated basic language.

4.4.4 Mappings of Structures and Values

While we were considering the consistency requirements that were elicited for models used in the context of cyber-physical systems [66] using an early sketch of the mapping language (for example [66, pp. 36ff.]), we discovered that in many cases, there needs to be a mapping of attribute values that are reachable in different ways from a meta class. Values can be either represented by attributes that are directly associated to a meta class, or by meta classes that can carry additional values. For example, consider a meta model that only saves an integer value for a measurement that represents the concrete value that has been measured. A meta model that is more concerned with the measuring process, includes additional information about the used measurement tools, the uncertainty of the measurement, or the time of the measurement. Therefore, this second meta model uses a dedicated meta class, that contains a value and additional information. Additionally, more hierarchy levels can be used to organize the data attached to an object.

Figure 4.6 shows a more generalized and abstract example for different structures in meta models that contain the same information in different ways. The three displayed meta models are called (from left to right) mmA, mmB and mmC in the following. In Figure 4.6a, the integer-valued attribute \(v_1\) and the string-valued attribute \(v_2\) are contained directly in the meta class \(A\). In Figure 4.6b, they are contained in dedicated meta classes which are specific to the type of the value (and could contain additional information, as mentioned above). In Figure 4.6c, there is an additional indirection for reaching the values from the meta class \(C\) they are semantically associated to, a container for the value objects.

Based on this generalized observation, we decided that our language should be able to map all of the displayed structures to each other.

Because Vitruvius, and our transformations, always consider atomic changes and therefore an atomic creation of the models, we need to consider the state in which models should first be mapped, resulting in the creation of the corresponding structure.

Figure 4.7 and Figure 4.8 show an example for possible consistent states our system must be able to establish between models for mmA and mmB. In the first column, the changed model instance is displayed. The different columns represent different mapping specifications (in a slightly abbreviated form) that are possible with our approach. The cells contain the state our system would create in \(B\) for mmB in case the changes are made in \(A\) for mmA as displayed in Figure 4.7, and the state of \(A\) in case the changes are made in \(B\) (Figure 4.8). In our example, we assume that every attribute can also have the value \texttt{null}.

Because there are space restrictions and we wanted to fit the different states into an overview table, all the attributes that are used in the bodies of the mapping code have an \textit{implicit not-null condition} in their signatures, i.e. the condition would have to be specified additionally in the specification but is omitted in the code in the table. Our approach
actually needs the user to state this *explicitly*. This is also the reason, why for example in the second column of Figure 4.7, the first two consistent states have empty models for B.

We think that using this approach for specifying mappings for attributes that are kept consistent independent from each other, and allowing the user to explicitly specify which attributes are required to have a value (i.e., that must not be null), the users can fine-grainedly tune how the transformation behaves, while having a reasonable and expected default behavior: the whole structure has to exist to be mapped.

We do not display the possible mappings for mmC. The signatures for mmC could be written like the signature for mmB, while also including the Container meta class. The resulting mapping code could be similar to Listing 4.5.

This would, however, imply that the whole structure for the right side, especially the Container instance, must already exist. If the container should be created automatically by Vitruvius for each C that could be expressed in an invariant and a response that fixes this invariant.

A problem of our current approach is that we do not provide an expression for matching the same Container element in a different mapping. A future extension of the mapping language could be able to express that an element must be either found or created, instead of always requiring that the structure is created newly by our generated transformation for the mapping at hand.

### 4.4.5 Default Containment

If a model transformation generates a new model, the model-driven engineering environment the transformation is executed in needs to know a storage location to persist the new model in the file system, or create an identifier by which it can be requested from the environment. Vitruvius uses the XMI format (see section 2.5) for persisting models and meta models. In the mapping language framework, model elements also need a storage location, an identifier, and a mechanism for finding model elements by the identifier.
4.4 Design of the Mapping Language

Listing 4.5: Exemplary mapping language code for mapping the meta model mmA to the meta model mmB

```plaintext
map { mma.A }
with {
  notnull(a.v1)
  notnull(a.v2)
} and {
mmc.C,
mmc.Container,
mmc.IntC,
mmc.StrC
} with {
  notnull(intC.valC)
  notnull(strC.valC)
  in(container, c.container)
  in(intC, container.values)
  in(strC, container.values)
} {
  [ 
    equal(a.v1, intC.valC)
    equal(a.v2, strC.valC)
  ]
}
```
Figure 4.7: Possible consistent states while changing a model instance A.
Figure 4.8: Possible consistent states while changing a model instance B
Our system always creates the new corresponding objects for a new match of a mapping. For persisting the new consistent state, the newly created objects have to be contained in some resource in the system (or in a model element that is contained).

This state can be achieved, by providing an \texttt{in(\ldots)} signature condition that positions the model elements in a containment. Then, however, the model elements are bound to a specific containment feature of a specific model element. We want to provide the developers with a convenient way of specifying how an initial containment can be determined automatically, while allowing the users to change this containment.

In the description of the language syntax and features in section 4.3, we also introduced the two signature conditions \texttt{default-contain} and \texttt{default-resource}. They do not enforce a specific containment (storage location), but provide the mapping language framework with enough information for finding a containment if one does not exist.

If no containment is specified for a structure, and the transformation detects model elements that are not contained, the mapping language framework asks the user (via an Eclipse dialog) for a resource the object is saved in.

Additionally, the responsibility of finding containments and handling non-contained objects could also be transferred to the VITRUVIUS framework in the future in form of a component that can be equipped with rules for finding suitable containments for model elements that are handled in the framework.

### 4.5 Possible Extensions

In this section we will give indicators on possible extensions we devised for the mapping language, but did not implement because of time constraints.

#### 4.5.1 Rule Composition and User Interaction

In the current prototypical implementation of the mapping language, we excluded mechanisms for the composition of mapping transformations that go beyond the “requires” dependency between mappings. As discussed in subsection 3.4.4, and also recognized by other researchers in the field of model transformations \cite{27, 28, 74} (both for unidirectional and bidirectional model transformation languages), more intricate mechanisms for composition can benefit the developer of model transformations.

On one hand, we would desire a refinement mechanism for mapping rules, where a (possibly abstract) base rule can be refined by altering the signature, adding additional constraints, etc. On the other hand, it would be beneficial to be able to explicitly specify a relationship of mappings where only one of a group of mappings can map a model element or a set of model elements. Then the exclusive matched rule could be chosen, for example by the user or by an additional set of constraints.

#### 4.5.2 Inheritance in Signatures

In the current implementation, our mapping language works with concrete meta classes only, i.e. classes that are not abstract and for which an instance can be created. Therefore,
there is an implicit check for the type of a model element encoded in the signature of the mapping, which has a very simple check (is the type equal to the specified type?) and a directly corresponding enforce method (creating an element of the given type).

In the future, we must, however, also consider the usage of inheritance in the signatures. The developer could specify a type in the signature (which can be abstract) and a second “instantiation type” that must be instantiable and able to substitute the first type. Then the resulting condition checks if the model element can substitute the first specified type. If the signature has to be created, the instantiation type is used as the concrete type. Alternatively, the user could be asked for a concrete type to create.

4.5.3 Class Hierarchy, Enumerables and Stereotypes

As described in subsection 3.5.3, different concepts that are located on different levels of abstraction can be used in meta models for describing things that have to be kept consistent by a model transformation. In one of the case studies we inspected during the course of this thesis, models that use the UML profile SysML were part of consistency relationships that were expressed using Triple Graph Grammars [40]. Profiles are a mechanism that allows meta model designers to constrain and extend meta models to customize them while staying in the four-level-hierarchy of the Meta Object Facility (MOF). Stereotypes that are specified in a profile can be attached to meta classes to give them additional semantics.

In the case study, the stereotypes were used to discern which elements the stereotyped meta classes should be mapped to. However, the stereotypes of SysML were referenced using constraint expressions that contain the name of the stereotype as a string and checks that use a mechanism similar to reflection in Java for getting information about the meta classes that elements are instances of. Therefore, there is no direct “type-safe” support of stereotypes. Similar problems arise, if other mechanisms are used to breaking the meta levels in the meta models, but are not represented in the transformation languages used to process instances of those meta models.

In future versions of the mapping language it could be possible to add language features that are specific to the used meta meta model or meta model, for example for the UML, for allowing the user to reference elements of the meta model (in this case concrete stereotypes) and their relationship to the elements to map in the signature in a type-safe way.

Another break of meta levels that could be relevant for future versions of the mapping language is also described in subsection 3.5.3. There possibly needs to be a mechanism for not directly specifying the type of a meta class in the signature, but rather of allowing the developer to specify a mapping between enumerable values and concrete types in the other meta model, thus allowing to bridge this gap in a concise way. A sketch of a language feature that could achieve that, and an equivalent but more verbose version that is closer to our current approach is given in Listing 4.6 and Listing 4.7 respectively.

Referencing concrete models is currently not supported in the mapping language. However, since some meta models, such as the Palladio Component Model (PCM) [75, 12], represent for example types not as meta classes or enumerables on the meta model level, but reference model instances of the type DataType, making it possible for the user of the meta model to extend the library of types dynamically while modeling, the mapping language (and other model transformation languages) might need a mechanism for explicitly
4 The Mapping Language

referencing such a “default” model instance. Then information that is represented on different meta levels in different meta models can be referenced in the transformation language while making this break of meta levels explicit via language constructs.

4.5.4 Reusable Mapping Functions

In the current implementation of the mapping language the user can provide manual implementations for the different directions of propagation of attributes. Additionally we provide a constraint language that provides declarative expressions for attribute relationships for which we automatically provide both directions of the propagation.

There are relationships that are part of the consistency requirements, but need to be referenced multiple times in the context of the specification, for example in different mappings and for different meta classes, it would be beneficial if the user could specify named and typed operations that can then be referenced in the mappings for separating the manual specification of the directions of a constraint and the usage of the resulting bidirectional declarative construct in the mappings. Additionally, parameters could be used to generalize multiple similar propagation operations.

A sketch of an appropriate language feature and how it would currently be implemented in our approach is displayed in Listing 4.8 and Listing 4.9 respectively. Note that the example excludes a check and enforce of a state where the attribute propagation is applicable (i.e. the needed suffix is existent) which would be needed additionally.

4.5.5 Complex Attribute Relations

Previous work by Rakhman et al. in the context of VITRUVIUS and MIR was concerned with the bidirectional specification of complex attribute relationships and the generation of
4.5 Possible Extensions

operation mapPrefix(String a, String b, String suffixA, String suffixB) 
check { (a.removeSuffix(suffixA) == (b.removeSuffix(suffixB)) } 
--- { b = a.removeSuffix(suffixA) + suffixB } 
<-- { a = b.removeSuffix(suffixB) + suffixA } 

map { a.Class } 
and { b.Component } 
[ { mapPrefix(class.name, component.name, "Component", ")" ) } ] 

Listing 4.8: Sketch for the language feature for specifying reusable bidirectional constraints

map { a.Class } 
and { b.Component } 
[ { xbase from b with { class.name = component.name + "Component" } 
xbase from a with { component.name = class.name.removeSuffix("Component") } } ] 

Listing 4.9: Sketch of equivalent code to Listing 4.8 in our approach

imperative unidirectional transformations from this specification [54, 55]. Similar concepts are also implemented for TGGs [3].

The integration of the approach of Rakhman et al. will be a natural next step in the extension of the mapping language, since the languages are based on the same language framework (Xtext) and use the same underlying modeling technology.

4.5.6 Transitive Closure and Dynamic Structures

Our current approach specifies a fixed sequence of types for each meta model for each mapping. It can also be useful to specify dynamically sized structures that are mapped to each other. An example for this also exists in one of the case studies we inspected as a base for our work. If the structure can be specified using collection operators [42] or a star operator [61], the mapped patterns become more expressive. There are approaches for TGGs that implement those and more complex and non-static model element structures [59].

A concrete example for the specification of a transitive structure is given in Figure 4.9 and Listing 4.10. The figure shows an excerpt of the meta model for modeling software in the AMALTHEA project3 [19]. AMALTHEA is an open source tool platform based on Eclipse for developing multicore embedded systems. It is used and developed by Bosch for the automotive sector. The namespace URI of the software meta model is http://www.amalthea.itea2.org/model/1.3.0/sw.

The example is adapted from a case study which was conducted in a different thesis in the VITRUVIUS project and which used an early sketch of the MIR language family for
describing consistency requirements between AMALTHEA models and another tool for modeling software components in the automotive sector, ASCET⁴.

In the example, we want to define a pattern for CallSequenceItems that are reachable from a model element in a CallGraph through different graphEntries and their references to other entries. We do not know in detail the lengths of the “paths” in the CallGraph, and do not need information about the elements that lay on a path to a CallSequenceItem.

For describing such a structure, the mapping language must allow the definition of signatures of arbitrary size (instead of fixing the number of meta classes in the signature). A sketch of a possible description of the consistency requirement that we described can be seen in Listing 4.10 with the .* operator (line 4).

The instance of the opposite structure that is created if a new mappable structure is found by the framework must then be a “minimal” instance that fulfills this dynamic signature description. Such a minimal instance would contain only a CallGraph with one contained CallSequence that in turn contains an arbitrary CallSequenceItem.

There are existing approaches that allow the matching of patterns that involve transitive closure of references in models, for example EMF-IncQuery. In the future of the mapping language, an appropriate language feature could be created for describing this condition and the existing approaches could be used for matching instances.

### 4.6 Limitations and Future Work

In the previous section we described concrete additional features that could be implemented for the mapping language. In this section, we want to give a broader overview of the limitations of the mapping language and associated future work.

#### 4.6.1 Language Expressiveness

We have built the current implementation of the mapping language as an extensible language that can be extended by further bidirectional language features regarding signature constraints and body constraints. The framework, the editor, and the generator in our implementation are extensible and will in the future be extended further.

---

Figure 4.9: An example for a meta model for which model instances can contain structures that need to be matched using a transitive pattern. Adapted from [64]
We inspected an array of examples from academia and industry to find a suitable extensible base language. In the future, additional language features for specifying patterns and signatures that are matched can be derived from the same and additional examples and case studies.

### 4.6.2 Consistency Requirements

While studying existing solutions to consistency problems, which we did for finding suitable features for our mapping language, we encountered the following problem.

In many cases, the description and the implementation that is provided in form of source code or as a scientific paper shows how specific consistency relationships have been expressed and what the system is able to repair automatically. However, it is often difficult to find the true limitations of the used approaches, and whether specific relationships are not part of the consistency requirements and therefore not specified for automatic consistency repair, difficult to specify using a given approach, or if they are not possible to formulate at all. There is, however, also no fixed schema or process for documenting consistency requirements independent from the technical solution.

The mapping language we built is less focused on problem “patterns” and requirements that could not be solved using existing approaches, but more on problems – and opportunities for simplification – we saw in the approaches that (successfully) solve a specific automated consistency problem.

Therefore, further research to derive additional language features should be focused on case studies that show consistency requirements that cannot be modeled using a given approach, or on a structured consistency requirement elicitation.

### 4.6.3 Batch Mode and Model Integration

As mentioned previously, we neither support a non-incremental execution of our transformation, nor an integration of existing models that have not been built incrementally with our system, and for which subsequently no correspondence structure exists. We chose to separate the tasks of incremental execution, batch execution and the integration of existing models. Our language is designed for the first task. However, we consider the latter two tasks (batch execution and integration) to be important features for the adoption of our tool in real contexts in the future.

One possibility to achieve a batch execution of our transformations would be a simulated atomic construction of the models, using the containment structure of the models. This has already been studied in the Vitruvius context [60].

For the integration of two existing models multiple we will shortly touch on two ideas. They are, however, not yet implemented or researched in detail. One idea is to execute the transformation in batch mode, as described above, for both existent models simulating an incremental transformation. As a result, there exist an original and a reconstructed version of both models, and two correspondence models. Tools for model diffing and merging, which are available in EMF, can then be used to integrate both versions.

Alternatively, an integration mode for executing the mapping transformations could be generated from the mapping specification. The integration mode then reconstructs a
suitable correspondence structure for existing models, either fully automatically, if possible, or in an interactive mode with user input.

We also plan to support manual overwrite of the rules for correspondence in the future. This means correspondences between arbitrary objects can be created by a user and mapping preconditions can be ignored if a user decides to apply a mapping regardless.

We gave a short introduction to the problem of model transformation evolution in subsection 3.9.1. This is not addressed in the current research prototype.

### 4.6.4 Pattern Matching and Change-Drivenness

The current implementation of the mapping language can be improved regarding its performance in various places. Currently, the pattern matching that is used for determining the instances of the structures in the signatures of a mapping, is based on a brute force approach.

Incremental approaches for finding model element sequences that fulfill a set of conditions, such as EMF-IncQuery\(^5\) allow for a much more performant and incremental matching of graph patterns in models.

Our mapping language framework currently generates responses that are executed for every change in any of the instances of models. Therefore, a lot of redundant checks happen on each model change for finding out if all conditions for a mapping still hold for signatures that cannot possibly be affected by the change. Furthermore, attributes are propagated for signature pairs that are already consistent (even though this could be prevented by deriving a consistency check method for a pair of signatures), possibly removing information from the models.

By performing a more intricate static analysis of the mapping rules, only changes that could actually change the truth value of either a signature precondition or a body condition could be used to trigger the corresponding propagation methods, resulting in a more performant implementation.

### 4.6.5 Validation of Mapping Specifications

More intricate static analysis of the mapping rule definitions could be used to find situations where mappings provide insufficient information for creating a valid target (EMF) model, for example, because no containment is set for a created model element. Furthermore, incomplete or contradictory rules could be identified and the developer could be warned while specifying the rules.

This static analysis would also make the possibly impacted mappings more apparent to the developer of the mapping language expressions. Currently, there is a lot of dynamic interpretation of mapping rules.

\(^5\)https://www.eclipse.org/incquery
4.6.6 The Mapping Language and the MIR Language Family

In the future of the implementation and design of the mapping language, its role in the MIR language family must be further understood and defined. Even though there are currently efforts to implementing the other parts of the language (i.e., the responses and invariants), all parts are implemented in relative isolation. However, the MIR language family has additional constraints on its parts as a whole: rules and responses must not interfere with each other’s execution, or at least report possible inconsistencies between the generated transformations to the user. Even though it is planned that the response language includes a possibility to react to the violation of invariants, the mapping language currently has – besides generating responses – no possibility to reference invariants or responses. This could be useful for reusing separately defined and maintained conditions on model structures, or for explicitly specifying that a mapping may or may not apply by sharing information from a response to a mapping. In the other direction, there possibly needs to be a way of reacting to a newly established mapping, a mapping that no longer applies, or in general to check whether there is a correspondence that belongs to a specific mapping between model structures.
5 Implementation

In this chapter we will describe the design and implementation of our consistency preservation approach. First we will introduce the framework our approach is a part of, VITRUVIUS.

5.1 VITRUVIUS

The VITRUVIUS modeling environment is based on the Eclipse IDE and realized as a set of plugins for this IDE. Changes to models that reside in the Eclipse workspace are recorded by VITRUVIUS using monitored editors, which are extensions to the Eclipse editors for observing the various artifacts that are included in a project. This includes (Java) source code editors, or EMF model editors, which can be graphical or textual, but can, in principle, be extended to all other editors in the framework.

There exist case studies for VITRUVIUS for which unidirectional transformations were implemented in Java or Xtend. They include for example a mapping between the Palladio Component Model (PCM) and Java code [57], and a mapping for the co-evolution of Java, the Java Modeling Language (JML) which is a "behavioral interface specification language"\(^1\), and unit tests [78].

5.1.1 Correspondences

Correspondences are a part of the VITRUVIUS framework and are the implementation of the witness structure for model transformations in the VITRUVIUS context. They can be used to store information about the consistency preservation state and process. A correspondence consists of a list of model elements of arbitrary size (which can also be empty) for each involved meta model. Correspondences can depend on other correspondences, but the consistency dependency graph has to be cycle free.

The correspondence model as a whole can therefore persist an arbitrary symmetric relation between sequences of objects.

VITRUVIUS does not automatically handle correspondences. This means that there is no meaning attached to correspondences by the framework – all functionality that depends on them, for example deleting corresponding model elements, has to be implemented by the consistency preservers. VITRUVIUS uses subclasses of a Correspondence meta class that are specific to the application. For example, for mappings, there exists a MappingCorrespondence meta class that contains additional information about which mapping instantiated the correspondence. The correspondences can then be filtered based on their type.

\(^1\)http://www.eecs.ucf.edu/~leavens/JML/index.shtml
In the case of mappings, the identity of the mapping that manages a correspondence is attached to each MappingCorrespondence. Note that this implies that there is a use to multiple correspondences between the same lists of objects and also between two empty lists of objects, since the correspondences can have different dependencies and also different managing mappings.

5.1.2 Temporary Unique Identifiers

TUIDs (temporary unique identifiers) are a mechanism for storing references to model elements that have been persisted on the file system. Each meta model that is managed by VITRUVIUS is equipped with a TUIDCalculatorAndResolver that can calculate TUIDs for every model element and can find a model element in the containment hierarchy of Ecore.

This mechanism is needed because one of the major use cases for the VITRUVIUS framework, the synchronization of PCM instances and Java code needs a mechanism for referencing elements in the model that is the result of parsing Java code with the Java Model Parser and Printer (JaMoPP). For example, methods do not have a unique identifier inside a class but have to be referenced by their name. The TUIDCalculatorAndResolver can then use the name of a method to resolve the model element representing the method. If the name of a method changes, which is a usual refactoring, all persisted correspondences that reference the method have to be updated accordingly. The TUID mechanism allows an efficient and quick update of the identifiers by keeping the TUIDs in a linked hash map.

Transient TUIDs can reference objects that are not currently persisted in the file system but only exist as Java objects on the heap of the Java Virtual Machine. This functionality is needed, when correspondences between objects are created before any containment has been established, and the containment is set in a later step.

5.1.3 Change Meta Model

The changes to models are recorded as instances of a VITRUVIUS-specific change meta model. The change meta model includes atomic change types, which represent minimal changes to model elements, such as changing the value of an attribute or setting, unseting, or updating a reference. Furthermore, compound changes are used to group transactions that are usually realized as single operations in tooling and for which consistency preservation mechanisms can specify special handlers that look ahead to prevent unnecessary or destructive changes to a model as described in subsection 3.8.2. The change meta model itself is also realized as an EMF meta model and instances can be handled as Java object structures.

In the current implementation, models have to be build by incremental change operations while monitored by VITRUVIUS when they are to be mapped to a corresponding model. Mechanisms matching existing models and finding (partial) correspondence relationships are, however, under consideration.
5.1 Virtual Single Underlying Model

The base of **VITRUVIUS** is the **VSUM**, the virtual single underlying model. The **VSUM** is a representation of the modeling environment and provides multiple functions:

- Manage meta models that are relevant for the environment. The meta models are referenced by *namespace URIs* in the models and transformations. Additionally, meta models have associated file extensions, which are used to identify the correct editors and transformations based on the file name of a model instance.

- Load and persist models for given URIs that adhere to the managed meta models, so called *ModelInstances*. The URIs reference files in the Eclipse workspace (so called *platform URIs*).

- Manage mappings between meta models. For each (unordered) pair of meta models, a *Mapping* is instantiated and saved. This *Mapping* is not a mapping in our mapping language, but persists that **VITRUVIUS** manages a relationship between two meta models.

- For each *Mapping*, the **VSUM** maintains a *CorrespondenceInstance*. It holds all correspondences for a pair of meta models. Depending on the transformation, the *CorrespondenceInstance* is wrapped with a filter that only returns *Correspondences* that are relevant to the context, for example *MappingCorrespondences*, if the transformation is generated from the mapping language.

- Persistence of its state, which includes all data structures that are needed for the functionality mentioned above. The **VSUM** can serialize its state when the host IDE is closed, and load the previously persisted state, when the IDE is newly started. Therefore, the editing process and the state of the *transformations* is also saved and restored to allow the continuous usage of the model-driven engineering environment.

5.1.5 Model Synchronization Process

The different stages of the process of changing models and applying appropriate transformations is implemented as a protocol that is displayed in Figure 5.1 in form of a state chart. This protocol is adhered to for each new change inside the **VITRUVIUS** environment. We create a new instance of our state machine for handling a change. Additionally, the state machine holds the data that is needed for the change. We will call this combination of state machine and data *protocol instance* in the following.

**VITRUVIUS** contains a *ChangeSynchronizing* component, which provides functionality for handling changes in the **VITRUVIUS** environment. Monitored editors record changes and provide them to the change synchronizer. In the following, we will describe, how changes are then passed through the framework according to the protocol.

After the creation of the protocol instance for a change and affected meta model pair (*Waiting4Changes*), changes are pushed to it. A *ChangePreparing* component creates composite changes, possibly removes redundant changes, creates correspondence instances
5 Implementation

between models if they are not existent yet, and maps changes in the file system to EMF model changes.

After the preparation (state Waiting4Transformation), an appropriate Change2CommandTransforming is chosen by the ChangeSynchronizing, and its transformChanges2Commands method is called. The Change2CommandTransforming is the interface for the synchronization transformations that can be registered in the Eclipse instance using a so-called extension point. They provide a list of pairs of mapped meta models and are called for every change in a model that conforms to a meta model in one of the involved meta model pairs. This is also the class for which we generate concrete implementations from our mapping specification (and the MIR language family specification as a whole).

The Change2CommandTransforming takes the changes from the protocol instance (resulting in state Waiting4Commands) and analyzes them and the changed models. As a result, a list of commands, the transformations on the target model, are pushed on the protocol instance data (state Waiting4Execution).

The commands that are created by a Change2CommandTransforming are of the type Command from the Eclipse Modeling Framework that encapsulates changes on EMF models in general. Vitruvius provides a specific implementation, VitruviusTransformationRecordingCommand, that can be used to record changes in a model by executing imperative Java code that returns a data structure (TransformationResult) that includes additional information about the changes that are needed for the VSUM: URIs of models that need to be deleted from the file system and URIs and content for models that are newly created.

In the next step, the CommandExecuting component takes the commands, executes them (Waiting4Check) and determines which model elements have been changed. In turn, all affected resources are saved. Furthermore, the protocol provides states for undoing and redoing transformations.

5.2 Change Synchronization for Mappings

In this section, we first describe how changes are handled in the mapping language framework in general. In the following sections, we will describe interesting aspects of the generated code and the code generation itself.

The overall process for an incoming change for a Change2CommandTransforming for each mapping that is generated from a mapping specification is displayed in Figure 5.2.

First, we generate signature instances for the current mapping (Signature Generation, Figure 5.3). For this purpose, we implemented a simple variant, which takes all resources that are affected by the change and creates possible signature instances by building the cross product for all model elements for each bound type in the mapping signature, because the pattern matching was not the focus of this thesis.

Then we build possible candidates from this set of possible signature instances and existing required mapping instances (Candidate Generation, Figure 5.4). Again, we simply build the cross product. A more intelligent handling for the candidate generation is future work for our implementation. Altogether, we build all candidates which could be newly mapped by our mappings, or for which the mappings do not longer apply. Additionally (not pictured in the figures) we include all existing instances of a mapping for rechecking,
5.2 Change Synchronization for Mappings

Figure 5.1: A state machine representing the protocol for handling a change in VITRUVIUS since our current implementation does not model elements that are deleted in a change easily, and therefore cannot create candidates for instances that are no longer part of the changed resource.

After we have built all candidates, we handle each candidate (Candidate Handling, Figure 5.5). All possible combinations of required mappings and elements in the signature of the model that was changed have been generated. For each candidate, the previous mapped
state can be acquired from the correspondence structure, by checking for a correspondence that is tagged with the mapping under observation and that has the same correspondences. The new mapped state is tested with the generated check-method. The figure displays the possible outcomes of the test and the action that is taken in each case:

- If the elements were not mapped before, and are not mapped anymore, do nothing.

- If the elements were not mapped before, and are now mapped, create a corresponding structure in the opposite model and create a correspondence. Then propagate all attributes from the source model.

- If the elements were mapped before, and are still mapped, propagate all attributes from the source model.

- If the elements were mapped before, and are still mapped, but the required mapping instances have changed, remove all references between the target structure and the target required structures that are declared in the signature conditions. Then create those references for the new required structures.

- If the elements were mapped before, and are not mapped anymore, destroy the structure in the target model, and remove the correspondence.
Candidate Generation

signature left

yes

get mapped correspondences

build cross product of signature with correspondences

build candidate

Figure 5.4: Workflow for the generation of candidates
Figure 5.5: Workflow for the handling of candidates
5.3 Code Generation

5.3.1 Generated Entities

An example for the code that is generated from the mapping specification displayed in Listing 5.1 is shown in Figure 5.6 (Generated Mapping Code). We omitted signatures in the framework and meta model classes. In the following we will describe the generated entities, their role and relation to the framework, and point them out in the given example.

For each mapping, the following two wrappers are generated:

- A signature wrapper for each signature of the mapping. The wrappers allow the type-safe and named access to the elements in the signature (we will go into more detail on this in the next section). The wrappers have the super type AbstractWrapper, which encapsulates a list of EObjects and Correspondences that are required.

Each wrapper has a static method for creating a wrapper from a Correspondence (the elements are then always fetched from the correspondence model), or by providing a list of (correctly typed) EObjects for the signature. The latter results in a so called transient wrapper.

In the example, those wrappers are UMLPCM_Wrapper_UML and UMLPCM_Wrapper_PCM. They have named references (in this case theClass, theMethod, and so on) to classes from the meta models.

- A correspondence wrapper for an existing mapped pair of signatures. This wrapper contains a reference to a signature wrapper for each involved meta model, thus allowing the type-safe and named access of all elements in its scope. The wrapper has the super type AbstractCorrespondenceWrapper, which knows a (VITRUVIUS) Correspondence and derives the signature wrappers which reference this correspondence.
Figure 5.6: Simplified class diagram of the code generated from the mapping specification in Listing 5.1
5.3 Code Generation

In the example, this wrapper is UMLPCM_CorrespondenceWrapper.

Furthermore, we generate a mapping class, that contains the functionality that is specific to a mapping. The interface for a mapping is MappingRealization. The abstract implementation, AbstractMappingRealization, from which all generated mapping classes inherit, contains functionality that is needed during a transformation:

- **Destroying** a collection of elements (destroy).
- Creating correspondences for two ElementProviders (in our case the signature representations).

In the example, this generated mapping class is called UMLPCM_Mapping. It contains the following static information about the mapping, that is available with the correct name and type in the code:

- An identifier for the mapping that is used to persist the association to a mapping in a Correspondence.
- A reference to each of the involved EPackages.
- A representation of each of the signatures as a (ordered) List of EClasses.
- A representation of the types of the required mappings as a List of Mapping.

Furthermore, the mapping class includes the following functionality for each of the signatures (and therefore for each of the packages). In the following, PackageName is a placeholder for the name of the Java package for the code that is generated from the meta model by EMF and imported via the import statement in the mapping language:

- A method for checking whether all of the signature constraints are fulfilled for a signature wrapper (checkPackageName).
- A method for enforcing all signature constraints after a signature has been created (enforceCorrectInitializationOnPackageName).
- A method for propagating the attributes of a correspondence wrapper from one package to the other package (propagateAttributesFromPackageName).
- A method for creating a correspondence wrapper (createMappedCorrespondenceWrapper). This method takes an element list for each meta model and creates a correspondence that references those elements, adds references to all required correspondences (mapping instances) and then adds the correspondence to the correspondence model.
- A method for transforming Candidates into typed wrappers (createPackageName-Candidate). Candidate is the data type used for representing potential candidates for wrappers. Each candidate contains a list of required correspondences and a list of EObjects, but no information about the type and name of the objects.
The actual code that is executed for a change in a model instance of one of the involved meta models is contained in the `applyEChangeForPackageName` methods. They have the return type `void`, because they are executed in a response and their effect is recorded and put into a `Command` by the framework (instead of returning a `Command` themselves). The generated `applyEChangeForUML` method is shown in Listing 5.2. In the following we will explain technical details of the implementation that we described in section 5.2. The lines in the following are in reference to Listing 5.2 if not declared otherwise.

The `MappedCorrespondenceInstance mci` that is set in line 4 is the part of `VITRUVIUS` that contains all the correspondences and additionally contains information about the association of correspondences to mappings and methods for querying correspondences for a specific mapping. The `MappingExecutionState` is a state of the mapping that contains a `TransformationResult` (see subsection 5.1.5) and additional information about TUIDs and model elements that have possibly been changed during the mapping execution.

Inside the change synchronizing method, first a `CandidateGenerator` is used for creating candidates for the signature (lines 6 and 7). `REQUIREMENTS` is a static variable of the mapping class that contains the (possibly empty) list of the required `MappingRealizations`. The candidates are typed as `Candidate`, but lack the concrete information about the type and name of the contained `EObject`s and the required correspondences.

This information is added in a subsequent step in the `MatchUpdate`. In Listing 5.3, the signature of the used constructor is shown. `W` is the generic type parameter that refers to the signature wrapper for the current match update (in the example `UMLPCM_Wrapper_UML`, because we are looking at the method for synchronizing a change in a UML model), and `CW` is the generic type parameter for the correspondence wrapper (here `UMLPCM_Correspondence_Wrapper`).

The `MatchUpdate` is populated (lines 8 to 12) by passing the current mapping (`INSTANCE`), the set of `Candidate`s, and the methods for turning a `Candidate` into a signature wrapper (`matchWrapper`), for checking whether a signature wrapper fulfills the conditions for being mapped (`checkFunction`), a method for turning a `Correspondence` into a correspondence wrapper (`correspondenceWrapper`), and a function for getting the signature wrapper for the current meta model (`correspondenceToMatchWrapper`).

In the example, mostly static methods are passed to the method: `UMLPCM_Mapping::createUMLCandidate` is the previously described method for wrapping a UML candidate into a signature wrapper, `UMLPCM_Mapping::checkUML` is the check method, which we will describe further in a later section (subsection 5.3.4), and `UMLPCM_Correspondence_Wrapper::new` is the constructor of the correspondence wrapper class. The Java lambda in line 12, `cw -> cw.getUML()`, takes an argument named `cw` of type `UMLPCM_Correspondence_Wrapper` and returns the UML wrapper from it.

The used constructor is a convenience constructor that passes the given methods to the `MatchUpdates populate` method, which classifies the candidates according to the existing mappings and the check method.

Subsequently, for new candidates an opposing signature is created (lines 15 and 16). Then a correspondence is created (lines 18 and 19), the signature is initialized (line 21), and attributes are propagated from the UML side of the new correspondence (line 22).

Correspondences that are not mapped anymore are destroyed (lines 25 to 28) and for correspondences that are still active, attributes are propagated (lines 30 to 33).
5.3 Code Generation

```java
public void applyEChangeForUML(EChange eChange, Blackboard blackboard,
                                MappingExecutionState state) {
  MappedCorrespondenceInstance mci = state.getMci();

  Set<Candidate> candidates =
      candidateGenerator.createCandidates(eChange,
                                          REQUIREMENTS, UML_SIGNATURE, INSTANCE, mci);
  MatchUpdate<UMLPCM_Wrapper_UML, UMLPCM_Correspondence_Wrapper> matchUpdate =
      new MatchUpdate<>
      (INSTANCE, mci, candidates, UMLPCM_Mapping::createUMLCandidate,
       UMLPCM_Mapping::checkUML, UMLPCM_Correspondence_Wrapper::new,
       cw -> cw.getUML());

  for (UMLPCM_Wrapper_UML newCandidate : matchUpdate.getNewCandidates()) {
    List<EObject> pcmObjects = MappingUtil.createSignature(PCM_PACKAGE,
                                                           PCM_SIGNATURE, state);
    UMLPCM_Correspondence_Wrapper currentCorrespondence =
        createMappedCorrespondenceWrapper(mci, newCandidate.getElements(),
                                            pcmObjects, state);
    enforceCorrectInitializationOnPCM(currentCorrespondence.getPCM(), state);
    propagateAttributesFromUML(currentCorrespondence, state);
  }

  for (UMLPCM_Correspondence_Wrapper voidedCorrespondence :
       matchUpdate.getVoidedCorrespondences()) {
    destroy(voidedCorrespondence.getPCM(), state);
  }

  for (UMLPCM_Correspondence_Wrapper currentCorrespondence :
       matchUpdate.getCurrentCorrespondences()) {
    propagateAttributesFromUML(currentCorrespondence, state);
  }
}
```

Listing 5.2: Example for the generated code for synchronizing an incoming change eChange in a UML model to a PCM model.

```java
public MatchUpdate(MappingRealization mapping, MappedCorrespondenceInstance mci,
                   Set<Candidate> candidates, Function<Candidate, W> matchWrapper,
                   Predicate<W> checkFunction, Function<Correspondence, CW> correspondenceWrapper,
                   Function<CW, W> correspondenceToMatchWrapper)

Listing 5.3: Signature of the MatchUpdate constructor
```
5.3.2 Types and Names in the Generated Code

One problem with the generic code in the VITRUVIUS framework is, that it is designed to work with any kind of transformation, not necessarily implementing our bidirectional notion of consistency that ties model element structures to each other. Therefore, correspondences do not know about mappings. They also do not have type information attached to the elements that are in correspondence, but operate on generic E0bjects.

For our implementation we chose an approach that generates wrappers for all necessary structures that casts those generic E0bjects to the types that are assumed in the mappings. By doing this, we aim to make the generated code easier to understand for users, both when reading as well as debugging.

Figure 5.6 shows an example for this kind of code generation. This kind of handling of the objects that are involved in a mapping is possible, since there is a layer of generated code that handles the creation and typing of all involved objects.

The actual code specified by a developer (in the Xbase expressions inside the language) never creates elements or changes the type of elements, but only works with the wrappers that have elements of a specific type. The same is done in a wrapper for correspondences. Similarly, the correspondence model is never changed by a developer directly, but the correspondences that are relevant to our mapping are only changed and created by the mapping code itself, which ensures that the correspondences create the correct number of elements with the correct types (by using the wrappers themselves).

By adding this additional generated layer, we can provide type-safety regarding types that are not known in the framework (in AbstractCorrespondenceWrapper and AbstractWrapper), but are known in the mapping specification and should therefore also be fixed in the generated code, thus simplifying the understanding of the code and avoiding manual casts by users.

5.3.3 Mapping Language Code Generator

For the mapping language, we build a code generator that consists of multiple steps.

During the editing of the mapping language file, the MappingLanguageJvmModelInferer infers a JVM class model that contains the generated wrappers described in subsection 5.3.2. Additionally, methods for each Xbase block that is specified manually by the developer instead of using our constraint language, are generated that have parameters that are typed with the generated wrappers.

By doing that, we allow the user to have code completion and type checking for the elements that are bound in the current context: the current signature in a manually specified check or enforce method (the signature constraint), and the pair of signatures in a pair of attribute propagation methods for a manually specified body constraint.

If the user explicitly triggers the generation of all MIR language family files in a project (by right-clicking the project in the Eclipse Package Explorer and choosing “Generate MIR”), the MIR framework collects all files (that reside in source folders of the project).

First, the framework then calls the MappingLanguageGenerator for each mapping file. The method generateAndCreateResponses takes a collection of EMF Resources as input and returns a collection of Responses for each Resource. Currently, our implementation
5.3 Code Generation

generates responses that react to all changes in a model instance of the mapped meta
models and calls all generated applyEChangeForMetamodel methods for the changed meta
model.

For generating the actual mapping code, we use the programming language Xtend.
Xtend is a language with Java-like syntax that compiles to Java code. We mainly rely on
a feature of the language which is called template expressions\(^2\). Template expressions are
delimited by triple single quotes and allow the embedding of Xtend code inside guillements
(«»). They also allow branching (using the keyword IF) and looping over collections with
FOR. Using template expressions, we can avoid building large strings in verbose Java using
a StringBuilder.

We built a helper for postponing the actual creation of source files called TemplateGen-
erator. In each method for creating a Java source file, the template generator is provided
with a fully qualified name of a Java class, and a method that returns the text of the source
file.

This method is provided with the template generator itself and can request code for
extension points that are identified by an arbitrary Java object (and therefore also a
collection of objects), resulting in a closure that can be called for creating a Java source
file, called a template java file. After all template Java files have been generated, we call
additional extensions, that can add source code to extension points. After all extensions
have been called, the actual generation methods are called and the source code is saved at
the correct location. By doing that, we can decouple different advanced features in our
mapping language and the generator.

For example, the feature of default containment is decoupled as follows. We specify an
additional generator (DefaultContainmentGenerator) that contains an additional method for
each import in the mapping class body (tryApplyDefaultContainmentForPackageName).
The generator injects code for calling this method at an extension point that is identified
by a list that contains the elements of the mapping language, the import, and the String
"defaultContainmentCheck". In the method that generates the mapping class, we simply
call the method expandTemplate with the same parameters at the appropriate position.

Additionally, we built a EMFGeneratorHelper. The helper is initialized at the beginning of
the generation process and generates a EMFWrapper at the end of the generation. The helper
is called, if specific EClasses need to be referenced by the generated code. The requested
entity is then saved inside the helper and a call to the EMFWrapper (which resides alongside
the generated code) is returned. At the end of the generation, the EMFWrapper is generated
in a way that provides all the needed entities.

5.3.4 Constraint Language Code Generator

Each of the signature constraints for a signature in the mapping specification is translated
into code that checks this constraint that is placed in the checkPackageName method, and
code that enforces this code after the elements have been created in enforceCorrectIni-
tializationOnPackageName.

\(^2\)https://eclipse.org/xtend/documentation/203_xtend_expressions.html#templates
5 Implementation

```java
public static boolean checkPCM(UMLPCM_Wrapper_PCM pcm) {
    if (!(((pcm.getPCM().getComp().contains(pcm.getPCM().getRole()))))
        return false;
    return true;
}
```

Listing 5.4: checkPCM method generated from the example mapping specification

```java
public static void enforceCorrectInitializationOnPCM(UMLPCM_Wrapper_PCM pcm,
                                                 MappingExecutionState state) {
    state.record(pcm.getPCM());
    state.record(pcm.getComp());
    pcm.getPCM().getComp().add(pcm.getPCM().getRole());
    state.updateAllTuidsOfCachedObjects();
    state.persistAll();
}
```

Listing 5.5: enforceCorrectInitializationOnPCM generated from the example mapping specification

Consider for example the signature constraint in(role, comp.providedRoles) in line 6 of Listing 5.1. In the checkPCM this is translated to the code displayed in Listing 5.4.

After line 3 in the example (Listing 5.4), additional code for checking constraints would be inserted following the same schema:

```java
if (!((check constraint)) return false.
```

We implemented a ConstraintLanguageGenerator that creates code for each of the available constraint types. The generator must ensure that names of the getters and the path to a specific element is constructed correctly. In the example the Component comp that references the element specified in line 4 in Listing 5.1 is translated to the expression

```java
public static void propagateAttributesFromPCM(UMLPCM_Correspondence_Wrapper umlpcm,
                                             MappingExecutionState state) {
    state.record(umlpcm.getUML().getTheMethod());
    // theMethod.name := role.name
    umlpcm.getUML().getTheMethod().setName(umlpcm.getPCM().getRole().getName());
    state.updateAllTuidsOfCachedObjects();
    state.persistAll();
}
```

Listing 5.6: propagateAttributesFromPCM generated from the example mapping specification
PCM.getPCM().getComp(). This translation is done in the method getJavaExpression-ThatReturns(Map<List<?>, String>, ContextVariable, Mapping) in the Constraint-LanguageGenerator. The method takes the current local context as a map, a variable that needs to be referenced (in this case comp) and the Mapping we are generating code for. The local context is passed the name of the parameter pcm for the current package pkg, through the entry that maps ['this', PCMImport] to 'pcm', where PCMImport is the mapping language AST element that declares the import of the PCM meta model.

The code for enforcing a correct initialization on a newly created PCM wrapper is displayed in Listing 5.5. First, all possibly affected objects are recorded in the MappingExecutionState (lines 3 and 4). Then the correct state is established (line 6). Subsequently, we update TUIDs (if necessary) of the previously recorded objects (line 8) and save all changed resources (line 9).

For propagating the attributes in an established mapping between two corresponding signatures, the two methods propagateAttributesFromPCM and propagateAttributesFromUML are generated. In our example, only one body constraint is propagated: equal(role.name, theMethod.methodName) (line 13, Listing 5.1).

The code for the method propagateAttributesFromPCM is displayed in Listing 5.6. The same techniques as for the check and enforce methods are used for determining the correct Java code for getting the affected elements and the correct getter and setter methods.

If Xbase code is used for manually specifying the corresponding check and enforce sides of a body constraint or both directions for the attribute propagation in a body constraint, we create static methods from that code that reside in an additional class for the constraints. Then calls to those static methods are embedded in the generated code for checking/enforcing/propagating. In a future version of the language, the code could also be directly embedded into the respective methods to avoid this indirection.

5.4 Language Implementation

5.4.1 Editor

Xtext allows the generation of an Eclipse editor from the language specification. A screenshot of the editor with a mapping specification similar to our previous example is displayed in Figure 5.7. Xtext automatically generates powerful features such as syntax highlighting, code completion, code folding and an outline by default. These features can be customized in the generated code.

Additionally, our editor automatically allows the addition of dependencies to plug-ins that provide packages that are referenced in the mapping specification via an import (Figure 5.8).

5.4.2 MIR Base Language

Initially, all languages that are part of MIR were implemented relatively independent from each other. However, during the implementation of the mapping language and the response
5 Implementation

Figure 5.7: Screenshot of the editor for the mapping language, including syntax highlighting, code completion, code folding and an outline.

Figure 5.8: Missing dependencies to Eclipse plug-ins that provide the Java code that is generated for an EMF package can be automatically added to the plug-in containing the mapping specification and code with Xtext validation and quick-fixes.
language, we identified similar functionality that was needed in both languages, and can also be leveraged when integrating the invariants language as a third part of MIR.

The following language functionality is needed in all the languages:

- The import of EPackages that are registered in the host Eclipse instance.
- Referencing imported EPackages.
- Referencing EClasses that are part of an imported EPackage.
- Referencing EStructuralFeatures that are part of an EClass.

Additionally, we also identified functionality that is used for the integrated generation of the languages:

- Validation and quick-fixes for checking and fixing, respectively, if the project that contains a MIR file is a Java plug-in project.
- Validation and quick-fixes for checking and fixing, respectively, if the bundles that export the generated code for the EPackage that are imported in MIR files are included in the dependencies in the manifest file of the plug-in project (the file, which configures the plug-in).
- An integrated MIR generator that can be called on a set of MIR files, and that calls the respective generators of the individual languages.
- A command for calling the MIR generator on a plug-in project (instead of a single file) which collects all necessary MIR files for generation and passes them to a generator.

To achieve a base for all MIR languages, we created an “abstract” base language, “Mir-Base”. We pulled all the common functionality mentioned above to the base language. Xtext provides a mechanism for reusing languages with so called grammar mixins\(^3\). The grammar, editors of the language that uses mixins can inherit, reuse and extend language elements and functionality defined in the base language.

We made the base language abstract by removing all generated extensions for Eclipse editors which normally register editors for the file extension that is defined in a Xtext grammar. In result, the editors, parsers and additional IDE features for the abstract MIR base language are not used directly, but only by the editors of the concrete MIR languages.

5.4.3 Generation of Responses

The response language is also part of the MIR language family. It can be used to specify imperative code that is executed in reaction to changes in models that are managed by Vitruvius. Responses are as of the end of this implementation under development in another thesis.

To integrate the mapping language and responses as parts of MIR, we decided to leverage the change-reaction functionality of the response language. Response language expressions

\(^3\)https://eclipse.org/Xtext/documentation/301_grammarlanguage.html#grammar-mixins
are composed of triggers and effects. Triggers specify the conditions under which code is to be executed, for example specific change types, the meta model that is affected, elements that are modified and additional constraints on those elements. Effects are the actions that are taken, if an appropriate change is identified.

The developer can provide arbitrary imperative code to execute in Xbase, which can also reference Java code in the current class path. Additionally, the response language provides developers with a set of effects that are built into the language framework and can be used in addition to manually specified code blocks. Those effects include the creation of a new target model, of a model element in the existing target model, the deletion of model elements, etc. Depending on the effect, the parameters of the code block the user can provide is different. For example, elements created in the effect can be referenced in the code block.

Since the mapping framework was initially designed independent from the response language, a simple mechanism for reacting to changes was already implemented. For the scope of this thesis, we decided to first only use the capabilities of the response language to react to changes in an instance of a meta model, without using response features for discerning between different change types. The responses did also not provide a concept of order for the executed responses. As a result, we generated two responses from the mapping specification, that execute a static method for reacting to an EChange for each mapping, in the order we derived.

All mapping expressions and all responses in a project are collected when the user triggers the creation of the code that can be hooked into the VITRUVIUS framework. First, all necessary responses are created from the mapping. The response generation framework creates one integrated Change2CommandTransforming from those responses, and the responses specified in response files.

This means that the current implementation only uses the responses as an entry point from the VITRUVIUS framework. Future work on the mapping language can implement parts for reacting to specific change types in a more fine-grained way in the response framework. While this functionality was initially pictured for the mapping language, we did not implement this. However, there exist extension points in the code, where more sophisticated approaches regarding the selection of model elements can be implemented based on the type of the change that happened in the framework, that can be implemented in the future.

The response language provides an API for creating responses programatically instead of creating response language files that need to be parsed and processed by the plug-ins generated from the Xtext grammar, which we use in our mapping language generator.
6 Formalization

In this chapter we present a more formal approach to defining our notion of consistency. In chapter 3, we already gave an informal description and pointed out some fundamental questions for the concept of consistency. This chapter aims to answer questions more precisely that might arise in the informal description provided previously regarding the behavior of our consistency repair approach. However, we do not draw logical conclusions about the correctness or completeness from the logical framework that we construct.

6.1 Consistency Relation

In our consistency approach, the user specifies a relation $R$ between instances of two meta models $A$ and $B$. For reasons of simplicity, we will define meta models as the set of all possible valid model instances that adhere to the meta model in the following. Then,

$$R \subseteq A \times B.$$  \hfill (6.1)

Furthermore we define that

$$(A, B) \in R \iff A \text{ and } B \text{ are consistent.}$$  \hfill (6.2)

This means that our notion of consistency is, that we define two models as consistent if they are in relation $R$, i.e. our consistency notion is normative. In turn, two models are inconsistent, if $(A, B)$ is not in $R$. Consequently, Equation 6.2, i.e., $(A, B) \in R$ is the condition that is violated, for example by user interaction, and is then repaired by our system. By repairing we mean that we change the involved models $A$ and $B$ (or one of the models) in a way, that the resulting models are in the relation – and therefore consistent – again.

We call the specification that describes the relation $R$ the mapping specification. It consists of one or multiple mappings. In the following, we will use $i$ as the identifier of a mapping. We refer to the mapping $i$ by $M_i$.

6.2 Mapping

Each $M^i_i$, specifies a mapping relation $R_i \subseteq A \times B$. The overall relation $R$ is composed conjunctively from all mapping relations as follows:

$$(A, B) \in R \iff \forall i : (A, B) \in R_i$$

For precisely defining which models are in the relation $R_i$, we need to first define what makes up a mapping.
Each $M^i$ is a tuple:

$$M^i = ((T_{j^1}, A^1), (T_{j^2}, A^2), \ldots, (T_{j^t}, A^t), (R^i), (c^i), (e^i))$$

The elements are described in detail in the following.

### 6.2.1 Signatures

$(T_{j^1}, A^1)$ is a sequence of (not necessarily unique) types which we call signature. In our formalization, a type $T$ is the set that contains all model elements that have this type. To simplify our formalization, we furthermore do not cover inheritance or abstract types. $t_{l^A}$ is the number of meta classes specified in the signature. $(T_{j^1}, A^1)$ is defined accordingly. We define:

$$T_{l^A} := T_{j^1} \times \cdots \times T_{j^t} \times T_{l^B}$$

Based on those types we can derive a signature type condition $\tau_{l^A}$ which tests if a sequence is correctly typed:

$$\tau_{l^A}(a) : \iff \begin{cases} 
  a_1 \in T_{l^A} \\
  a_2 \in T_{j^A} \\
  \vdots \\
  a_t \in T_{j^A} \\
  a_t \in T_{l^A} 
\end{cases}$$

$\tau_{l^B}$ is defined accordingly. The signature defines the types of the model elements that should be mapped to each other.

### 6.2.2 Required Mappings

$R^i$ is a sequence of indices of required mappings:

$$R^i = (R_{i^1}^i, R_{i^2}^i, \ldots, R_{i^t}^i)$$

The required mappings are mappings of which instances are required to exist and for which a relationship with possible instances of the current mapping can be tested, for example by expressing the condition that a reference to an already mapped element in an instance of a required mapping exists.

$R^i$ can also be empty and a mapping can be required multiple times. If a mapping $M^i$ specifies that it requires $M^j$ one or multiple times, then $(M^i, M^j)$ is in the requires relation. The transitive closure of this relation must be acyclic, i.e. there may not be pairs of mappings that (directly or indirectly) require each other.

We define $T_{R^i}$ as the transitive closure of all types that are required directly or indirectly by $M^i$. Because the requires-relationship between mappings is non-cyclic, we can define the transitive closure as the result of a depth first search of required mappings, where each time a mapping is passed for the first time, the list of its specified types is added to the
transitive closure. We do this to avoid the nesting of mappings and types in the following definitions.

For example, for mapping $M^0$ with one required mapping $M^1$, i.e. $R^0 = (1)$ (and therefore $R^1_1 = 1$) that does not require mappings itself, this means that

$$T^{R^0} = T^{1,A} \times T^{1,B}.$$ 

For example, if the required $M_1$ requires two mappings $10$ and $11$ that do in turn not require further mappings, then

$$T^{R^0} = (T^{1,A} \times T^{1,B}) \times T^{R^1} = (T^{1,A} \times T^{1,B}) \times (T^{10,A} \times T^{10,B}) \times (T^{11,A} \times T^{11,B}).$$

### 6.2.3 Signature Conditions

$c^{l,A}$ is a condition on the signature and is called the signature condition:

$$c^{l,A} : (T^{l,A} \times T^{R^l}) \to \{\top, \bot\} \quad (6.4)$$

where $\top$ and $\bot$ are the boolean truth values true and false, respectively. If there are no required mappings, Equation 6.4 reduces to

$$c^{l,A} : (T^1 \times \cdots \times T^{l,A}) \to \{\top, \bot\}.$$ 

Using Equation 6.4, we can test a sequence of objects that has a specific type signature.

Since $c^{l,A}$ is only defined on an already correctly typed sequence of objects, we additionally define a method $c^+_l$ in a way, that incorrectly typed and sized object sequences are mapped to $\bot$, and correctly typed sequences are mapped to $c^{l,A}(a_i)$. With this definition, we can write the condition on an arbitrary sequence of model elements, to which the signature Equation 6.4 is not applicable in general, since it requires elements that are correctly typed:

$$c^+_l(a_i, rs) := \begin{cases} \bot & \text{if } \neg \tau^{l,A}(a_i) \\ c^{l,A}(a_i, rs) & \text{else} \end{cases} \quad (6.5)$$

$\tau^{l,A}(a_i)$ is the signature type condition we defined in subsection 6.2.1.

For easier reference, we write $c_+$, if both mapping and meta model are obvious from the context. $c^+_l$ and $c^+_r$ are defined accordingly.

Together, the signature and the signature condition define the structures that are mapped by our approach.

### 6.2.4 Body Condition

In addition to the signature and signature condition, each mapping specifies a body condition $C^i$ that has to hold for an existing instance of a mapping:

$$C^i : (T^{l,A \times B} \times T^{R^i}) \to \{\top, \bot\} \quad (6.6)$$
Using this condition, we can test valid sequences (meaning they are correctly typed according to the signatures of the mapping and the required mappings).

This means that the body condition defines the relationship of the two signatures by specifying constraints on it. In contrast, the signature and signature constraint specify what makes up a structure that should enter this relationship.

### 6.3 Mapping Relation

#### 6.3.1 Correspondence Relation

We define:

\[(a_j) \overset{rs}{\Rightarrow} (b_k) \iff (a_j) \text{ corresponds to } (b_k) \text{ according to mapping } M^i, \quad (6.7)\]

where \((a_j)_{j=1,...,n}\) and \((b_k)_{k=1,...,m}\) are sequences of objects in the meta models \(A\) and \(B\) respectively.

\(rs\) is a fixed allocation of the signatures for each required mapping, i.e., it contains a fixed set of model elements for each signature of the required mapping.

Additionally, the required mappings also have fixed allocation of their required mappings. Therefore, \(rs\) can be seen as a tree of fixed allocations of model elements.

For accessing a specific signature (for navigating in the tree) we use a tuple notation, if necessary: \(rs.M_1.A\) is the allocation for the meta model \(A\) of the allocation of required mapping \(M_1\) for \(rs\); \(rs.M_1.M_3.A\) is the allocation of the required mapping \(M_3\) that is required by the allocation for its required mapping \(M_1\).

Furthermore, we define \(rs.M_j\) as the subtree of the allocations that is used for the mapping \(M_j\).

We define:

\[ (A, B) \in R^*_rs \iff rs.M_j \overset{rs}{\Rightarrow} (a_j, b_k) \quad (6.8) \]

#### 6.3.2 Implications of Correspondences

If two sequences of objects correspond, they need to be correctly typed and sized. Additionally they have to fulfill the signature conditions \(c_+\) and the body conditions \(C^i\). Using the definition in Equation 6.8, we therefore require:

\[ (a_j) \overset{rs}{\Rightarrow} (b_k) \implies c^+_{\overline{\overline{A}}}(a_j, rs) \wedge c^+_{\overline{\overline{B}}}(b_k, rs) \wedge C^i(a_j, b_k, rs) \wedge (A, B) \in R^*_rs \quad (6.9) \]

This implies a recursion between Equation 6.9 and Equation 6.8. The recursion ends, if a mapping \(M_j\) in Equation 6.8 does not require any mappings. Since the transitive closure of our requires relation is cycle free and finite, this is always the case for some mapping \(M_j\) in \(rs\).
Equation 6.9 is only an implication from the left to the right side and not an equivalence. This means that we conclude the conditions a specified correspondence relationship between two object sequences makes on object structures, but do not specify, when two object structures should correspond.

Regarding our change driven approach, this is beneficial, because we can separate deciding when two object structures should correspond, by manipulating the relation $\Rightarrow$, and the consequent conditions and their repair.

### 6.3.3 Enforcing of Correspondences

With the definitions up to now, we defined a framework for testing if two models, for which substructures are defined to be corresponding (by the relation $\Rightarrow$), can be tested for consistency according to a relation $R$, using a set of mappings that define conditions on the signatures and their relationship. Therefore, the mapping instances have to be provided and defined by the user of the formal framework. Using this framework (and particularly Equation 6.9), we now want to provide a schema for deriving, when correspondences ($\Rightarrow$) – and therefore structures in the opposite model – have to exist for structures.

Using the definitions and observations that we made in this chapter so far, we can now formally define the relation $R_i$ for two models $A \in \mathbb{A}$ and $B \in \mathbb{B}$:

\[
(A, B) \in R_i \iff \begin{cases} 
(a_j) \Rightarrow^*_i (b_k) \implies c^A_+(a_j, rs) \land c^B_+(b_k, rs) \land C^l(a_j, b_k, rs) \land (A, B) \in R^*_i \\
\lor \\
\forall (a_j) \in A, rs : c^A_+(a_j, rs) \implies \exists (b_k) \in B : (a_j) \Rightarrow^*_i (b_k) \\
\lor \\
\forall (b_k) \in A, rs : c^B_+(b_k, rs) \implies \exists (a_j) \in A : (a_j) \Rightarrow^*_i (b_k)
\end{cases}
\]  

(1) is the same as Equation 6.9. This is the core of the consistency specification and defines how models have to be manipulated in case a correspondence ($\Rightarrow$) between object sequences is required.

(2) specifies, when a correspondence ($\Rightarrow$) has to be established. Our notion of consistency requires the following: if a object sequence ($a_j$) in model $A$ is found, which is correctly typed, sized, and fulfills the signature condition $c_+$ (left-hand side of the implication in Equation 6.10, (2)), we require that there is a object sequence in model $B$ that is in correspondence with ($a_j$) (right-hand side). If such a structure ($b_k$) and the correspondence exist, this directly implies that ($b_k$) must fulfill $c^B_+$, and the corresponding structures in their entirety must fulfill $C^l$ (right-hand side of Equation 6.10, (1)). (3) is symmetric to (2).

In Equation 6.10, $R_i$ is defined as a conjunction of (1), (2) and (3). If either (2) or (3) is removed from the conjunction, we can cover cases in which one structure does not correspond to exactly one other structure. If (2) is left out, there can exist structures that fulfill $c^A_+$ and are not propagated to the target model $B$ (and vice versa if (3) is left out).

The case in which both (2) and (3) are left out, meaning that $(A, B) \in R_i$ is defined as Equation 6.10,(1), can be seen as a situation where correspondences are not automatically created. If the user manually specifies the correspondence structure, defining the relation $\Rightarrow$, the correspondences are, however, still maintained.
6.3.4 Consistency Violations and Repair

A consistency repair mechanism in our formal framework handles cases in which two models $A^{\text{pre}}$ and $B^{\text{pre}}$ are in a consistent state $(A^{\text{pre}}, B^{\text{pre}}) \in R$ and a change $\Delta$ changes model $A^{\text{pre}}$ to $A$, resulting in a state $(A, B^{\text{pre}})$. The aim of consistency repair in our framework is then (in general) to find a model $B$ so that $(A, B) \in R$. We constrain changes to achieve a consistent state to the model $B$ and explicitly specify that we do not alter $A$. In general, there must not be exactly one $B$ for which $(A, B) \in R$. In this section, we describe how our approach arrives at a specific $B$.

If $(A, B^{\text{pre}}) \in R$ we (trivially) define $B := B^{\text{pre}}$ and do not need to make changes to $B$. In the following, we specify the different cases that our approach can recognize after a change in $A$, for which $(A, B^{\text{pre}}) \notin R$. The opposite directions, for a change in $B$ are defined accordingly, by swapping $(a_j), A, A$ and $(2)$ with $(b_k), B, B$ and $(3)$ respectively.

For each case, we first specify a condition that is true for the pair of models and describe it informally. Then we describe which previously stated condition for consistency is violated by this, and which steps are taken to restore consistency.

### Case 1

$$
\exists(a_j), (b_k), rs : \quad (a_j) \overset{r_s}{\rightarrow} (b_k) \\
\land c_+(a_j, rs) \\
\land c_+(b_k, rs) \\
\land \neg C_i(a_j, b_k, rs)
$$

(6.11)

There are substructures in the models that are in a correspondence relationship, and which fulfill the respective signature conditions of the mapping. However, the body condition on them is violated. This violates Equation 6.10, (1).

Repair $C_i$ by changing $B$.

### Case 2

$$
\exists rs : \left[ \exists a_j \in A : c_+(a_j, rs) \\
\land \neg \exists (b_k) \in B : (a_j) \overset{r_s}{\rightarrow} (b_k) \right]
$$

(6.12)

A structure $(a_j)$ is a mappable instance of the signature for $A$, including a valid instantiation of the required mappings, but does not correspond with a structure $(b_k)$.

This violates Equation 6.10, (2).

Create a $(b_k)$ that fulfills $c_+(b_k, rs)$ and a correspondence $(a_j) \overset{r_s}{\rightarrow} (b_k)$. Subsequently, Equation 6.11 can be violated and needs to be repaired.

This also implies that the same structure $(a_j)$ must be mapped multiple times for different instantiations of $rs$, because the correspondence relation $\overset{r_s}{\rightarrow}$ is parametrized with the concrete instantiation of $rs$. 
6.3 Mapping Relation

Case 3

\[ \exists (a_j), (b_k), rs : \quad (a_j) \overset{rs}{\leadsto} (b_k) \]
\[ \quad \land \quad \neg c_+ (a_j, rs) \quad (6.13) \]

A structure \((a_j)\) is in correspondence relationship with a structure \((b_k)\), but does not no longer fulfill all necessary signature conditions. This violates Equation 6.10, (1). Since the change was made in \(A\), we remove \((b_k)\) from \(B\) and remove the correspondence \((a_j) \overset{rs}{\leadsto} (b_k)\).
We choose the most "drastic" change that repairs this condition. The change is, however, reasonable, because our system also creates the corresponding structure in case 1. A different possibility would be to remove the correspondence \(rs\), and to alter \((b_k)\) so that it no longer fulfills \(c_+ (b_k, rs)\) to avoid an instant recreation of a corresponding structure \((a'_j)\) because of a fallback to case 1.

Case 4

\[ \exists (a_j), (b_k), rs, rs' : \quad (a_j) \overset{rs}{\leadsto} (b_k) \]
\[ \quad \land \quad \neg c_+ (a_j, rs) \]
\[ \quad \land \quad \neg (a_j) \overset{rs'}{\leadsto} (b_k) \]
\[ \quad \land \quad c_+ (a_j, rs') \quad (6.14) \]

There is a different set of required mappings that can be used for \((a_j)\). This violates Equation 6.10, (1) and (2). First, we negate \(c_+\) for \((b_k, rs)\). Then we repair \(c_+\) for \((a_j, rs')\) and \((b_k, rs')\) and \(C^i\) for \((a_j, b_k, rs')\).

Case 5

\[ \exists (a_j), (b_k), rs : \quad (a_j) \overset{rs}{\leadsto} (b_k) \]
\[ \quad \land \quad \neg c_+ (b_k, rs) \quad (6.15) \]

A structure \((b_k)\) is in correspondence with a structure \((a_j)\) but does no longer fulfill all necessary signature conditions. This violates Equation 6.10, (1). However, this cannot happen, since the change was made in \(A\).
6.4 Change Driven Consistency Transformation

6.4.1 Changes

We define a change $\Delta$ in a model $A$ as a finite sequence of updates. The sequence has the size $|\Delta|$.

$$\Delta = (u_1, u_2, \ldots, u_{|\Delta|})$$

Each $u_i$ is an update. We define $\mathcal{U}_A$ as the space of all possible updates on model elements in a model $A$. Furthermore, we define $\mathcal{D}_A$ as the space of all changes $\Delta$ in a model $A$.

To refer to a feature $^1 f$ of a model element $a$, we will in the following use the notation $a.f$.

We write the elements in $\mathcal{U}_A$ in the following way:

<table>
<thead>
<tr>
<th>Element</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>set($a.f,v$)</td>
<td>set the value of $a.f$ to $v$.</td>
</tr>
<tr>
<td>create($T$)</td>
<td>create a model element of type $T$</td>
</tr>
<tr>
<td>delete($a$)</td>
<td>remove the model element $a$ from the model and remove all references to $a$</td>
</tr>
</tbody>
</table>

We define apply($A,\Delta$) as a function that returns the model that results if a change $\Delta$ is applied to $A$.

6.4.2 Consistency Transformation

Using the definitions in the previous sections, if $(A,B) \in \mathcal{R}$ and $(\tilde{A},B) \notin \mathcal{R}_i$ where $\tilde{A} = \text{apply}(A,\Delta)$ and $\Delta \in \mathcal{D}_A$, then the consistency transformation is a function

$$\text{ct}^{\rightarrow} : \mathcal{A} \times \mathcal{D}_A \to \mathcal{D}_B,$$

so that

$$(\tilde{A},\tilde{B}) \in \mathcal{R}_i,$$

where

$$\tilde{B} := \text{apply}(B, \text{ct}^{\rightarrow}(A,B)).$$

In the same way, we define $\text{ct}^{\leftarrow} : \mathcal{B} \times \mathcal{D}_B \to \mathcal{D}_A$. We choose exactly one $\mathcal{D}_A$ for each $\mathcal{D}_B$ (and vice versa).

6.5 Specification of Conditions in our Approach

For working with the consistency notion we described in the previous sections, for each mapping $\mathcal{M}$ we must specify:

$^1$Features can be references to other model elements or attributes, whose value are not model elements.
6.5 Specification of Conditions in our Approach

- the signature types $\mathcal{T}^{i, A \times B}$,
- the required mappings $R_1, R_2, \ldots, R_r$,
- the signature conditions $c^{i,A}$ and $c^{i,B}$,
- and the body condition $C_i$

In the following, we will describe how we arrive at a consistency transformation (see subsection 6.4.2) from the specification of those mapping parts.

### 6.5.1 Signature Conditions

$c^{i,M}$ is a conjunction of the signature type condition $\tau^{i,M}$ and user-defined signature conditions $c^{i,M}$. Therefore,

$$c^{i,M}_+(a_j, rs) :\iff \begin{array}{ll}
    a_1 & \in \mathcal{T}_{1,i}^{i,M} \\
    a_2 & \in \mathcal{T}_{2,i}^{i,M} \\
    \vdots & \\
    a_t & \in \mathcal{T}_{t,i}^{i,M} \\
    \end{array} \quad = \tau^{i,M}(a_j)$$

$$\begin{array}{ll}
    & \wedge \\
    & \wedge \\
    & \wedge \\
    & \wedge \\
    c^{i,M}_1(a_j, rs) & \\
    \vdots & \\
    c^{i,M}_{\#c^{i,M}}(a_j, rs) & =: c^{i,M}(a_j, rs) \\
\end{array}$$

Equation 6.20

Each of the conditions in $c^{i,M}_+$ must be checkable and enforceable. This means that it must be possible to check whether a model element of the correct type fulfills the conditions in $c^{i,M}$ and also it must be possible to

- find or create a model element of the correct type for each type condition in $\tau^{i,M}(a_j)$, and that
- the system must be able to manipulate a correctly typed $(a_j)$ in a way that $c^{i,M}_k(a_j, rs)$ evaluates to $\top$ for each $k \in \{1, \ldots, \#c^{i,M}\}$.

We formalize this method as the enforce function:

$$c^{i,M}_+ \uparrow : \mathcal{T}^{R_i} \rightarrow \mathcal{U}_M$$

Equation 6.21

When one of the sides is newly matched, a object sequence for the opposing side (meta model $M'$) is created that matches the types $\mathcal{T}^{i,M'}$. Then for each (independent) condition, the consistency transformation must enforce $c^{i,M'}_k$ for each $k \in \{1, \ldots, \#c^{i,M'}\}$ to $\top$. 

95
6.5.2 Body Conditions

$C^i$ is constructed similar to $c_+$:

\[
C^i(a_j, b_k, rs) : \iff \bigwedge_{q=1}^{[#C^i]} C^i_q(a_j, b_k, rs)
\]

\[
C^i_q : (T^{i,A} \times T^{R^i}) \rightarrow \{\top, \bot\} \quad (\forall q \in \{1, \ldots, [#C^i]\})
\tag{6.22}
\]

Each $C^i_q$ must be independently checkable.

Additionally if a state $(a_j, b_k)$ exists, where $C^i_q(a_j, b_k) = \bot$, we must be able to derive how to enforce a state $(a_j, b_k)$ so that $C^i_q(a_j, b_k) = \top$, and therefore how we can derive a $b_k$ from $\overline{b_k}$, and how they are related. This means that for each condition $C^i_q$ we must be able to derive:

\[
C^i_q : T^{i,B} \times T^{R^i} \rightarrow U_B
\tag{6.23}
\]

\[
C^i_q : T^{i,B} \times T^{R^i} \rightarrow U_A
\tag{6.24}
\]

6.6 Implemented Set of Expressions

The previous sections described which functions we need to derive in the different cases for restoring consistency as specified in our approach. In this section we will show the set of conditions for both signature and body for which we have implemented this derivation of

- $\Delta \in D_A$ (and $\Delta \in D_B$) as a result of $c^{i,A}_+$ (and $c^{i,B}_+$), and
- $\Delta^- \in D_A$ and $\Delta^- \in D_B$ as a result of $C^-_{i,q}$ and $C^-_{i,q}$, respectively.

6.6.1 Signature Conditions

The types $T$ in the signature might be abstract types for which it might not be trivial to decide for which concrete type an instance should be created to enforce the condition in $\tau$. Our formalization currently excludes abstract types and assumes that all types are concrete, but the signature must be changed accordingly, if abstract types are allowed.

$c^{i,M}$ is a conjunction of conditions that are independently defined and checked:

\[
c^{i,M}(m_j, rs) = \bigwedge_{p=1}^{[#c^{i,M}_p]} c^{i,M}_p(m_j, rs); \quad c^{i,M}_p : (T^{i,M} \times T^{R^i}) \rightarrow \{\top, \bot\}
\tag{6.25}
\]

\[
\forall p : c^{i,M}_p : (T^{i,M} \times T^{R^i}) \rightarrow \{\top, \bot\}
\tag{6.26}
\]

For our implementation, we consider the following single conditions $c^{i,M}_p$. In the following equations, both $a$ and $a'$ are correctly typed elements of the signature instance $(a_j)$. 
6.6 Implemented Set of Expressions

Model element is referenced

\[ a \in el.\ ref \] (6.27)

\( el \) can be either an model element in the same signature \((el \in (a_j))\) or in the allocation of the required mappings \((rs)\).

This condition is checked directly by getting the value \( v \) of the feature \( ref \) from \( el \) and either testing for equality between \( el \) and \( v \), if \( ref \) is single-valued, or for containment of \( el \) in \( v \), if \( ref \) is multi-valued.

If the check evaluates to \( \bot \), it can be enforced by the change

\[ \Delta = (\text{set}(a', ref, (a'.ref \cup \{a\}))) \] (6.28)

Attribute is not null

\[ a.\ attr \neq \text{NULL} \] (6.29)

This condition can be checked directly by getting the value \( v \) of the feature \( attr \) from \( a \) and testing for equality to NULL.

If the check evaluates to \( \bot \), we choose to enforce it by setting \( a.\ attr \) to the default value of its type \( T \), which we designate by default-value:

\[ \Delta = (\text{set}(a, attr, \text{default-value}(T))). \] (6.30)

If the default value cannot be determined, this condition cannot be formulated.

Attribute equals literal value

\[ a.\ attr = \text{LITERAL} \] (6.31)

This condition can be checked directly by getting the value \( v \) of the feature \( attr \) from \( a \) and testing for equality to the given literal.

If the values are not equal, we enforce it with

\[ \Delta = (\text{set}(a, attr, \text{LITERAL})). \] (6.32)

In addition to these three types of conditions, we considered the following two conditions, which we did not implement. We will shortly describe, why we do not implement them.
Formalization

**(Attribute equals value of other attribute)**

\[ a.\text{attr} = a'.\text{attr}' \]  \hspace{1cm} (6.33)

We do not implement this condition because it is not trivial to decide which default value to assume for the involved attributes, if both \( a \) and \( a' \) are newly created. If \( a' \) is a model element in a required mapping, this could however be a useful condition to implement.

**(Referenced by any model element of type \( T \))**

\[ a \text{ referenced-by t.ref; } t \text{ instanceof } T \]  \hspace{1cm} (6.34)

This condition means that there must be any \( t : T \) that references \( a \) by the reference \( t.\text{ref} \). The model element \( t \) must not necessarily be bound in the current context, meaning that it does not have to be part of the current signature or the signature of a required mapping. We do not provide a language mechanism for this possible condition, since it is not trivial to enforce. Possible enforce methods could include asking the user for a model element from which the reference is created, or creating a model element and giving the user options for deciding how to further process the created model element.

### 6.6.2 Body Conditions

Similar to the derived enforce methods for \( c_j^{M} \) we need to derive methods for establishing \( C \) while one of the sides \( a_j \) or \( b_k \) of the mapping is fixed.

**Attribute equality**

\[ a.\text{attr} = b.\text{attr}' \]  \hspace{1cm} (6.35)

The check for this condition is a direct check for equality.

\[ \Delta^{-} = (\text{set}(b, \text{attr}', a.\text{attr})) \]  \hspace{1cm} (6.36)
\[ \Delta^{←} = (\text{set}(a, \text{attr}, b.\text{attr}')) \]  \hspace{1cm} (6.37)

We could also consider to to require additionally that \( a.\text{attr} \neq \text{NULL} \) and \( a'.\text{attr}' \neq \text{NULL} \) in the signature if this is specified, which we do not do automatically (see Equation 6.29).
Additionally to the attribute equality, there has been work in the Vitruvius approach on more complex attribute relationships that could be specified in the body conditions [54, 55]. However, they have not yet been integrated with the mapping language prototype.

The bidirectionalization of attributes can, however, be integrated into this formal framework, because the approach also derives appropriate change propagation changes from deltas in an involved model.
7 Related Work

The related work discussed in this thesis will be focused on other approaches to specifying consistency rules for meta models. Related work for Vitruvius and view-centric modeling in general can be found in [21, 56, 53].

7.1 Bidirectional Transformation Languages

In many cases, general purpose languages, which are languages not specifically designed for change-driven consistency preservation, are used to keep models synchronous. If two models are in some kind of consistency relationship that can be established via a model transformation, and both models can possibly be changed by a user, it can be beneficial to use a bidirectional transformation [83]. In this paradigm, the two directions of a transformation do not have to be specified separately, which in turn requires effort to ensure that they are consistent with each other. Strictly bijective transformations, which means that there is a one-to-one mapping between instances of both meta models which express the same information, are “in most practical situations much too restrictive” [83, p. 413].

There is a lot of research regarding bidirectional transformations (shortened as bx) that is not related to a specific implementation. There is a community1 around bx that consists of researchers associated with different transformation languages and model-driven software engineering. The community also provides a repository of bx examples2 [24].

Stevens describes the “landscape” of bidirectional model transformations and introduces different challenges and aspects [83]. We reference to this work in several parts of this thesis and describe aspects of bidirectional transformations that we encountered in our research. Similarly, Antkiewicz et al. describe the “design space” for model transformations that synchronize heterogeneous models [4].

Diskin et al. describe how the consistency of a set of heterogeneous models can be checked by merging the overlapping parts of the models using a specification of the overlap of the models in form of so called spans [32], but do not specify how to enforce consistency using a specific implementation.

7.2 Triple Graph Grammars

Triple Graph Grammars (TGGs) have been studied thoroughly from a theoretical angle and have also been used in industry [40]. There have been restrictions regarding the

1http://bx-community.wikidot.com/
2http://bx-community.wikidot.com/examples:home
bidirectionality of attributes, which have been approached for TGGs recently [3]. We see TGGs as a major influence on our approach, because they are also used to relating patterns in meta models. Our basic concrete syntax is textual, as opposed to the graphical description of TGGs. We generate code that is aligned with the specification of a mapping and we allow the integration of imperative code into the declarative specification that is integrated directly in the generated imperative code. Our intention is for the generated code to be understandable for the user. We are not aware of a TGG tool that generates code for this purpose.

In chapter 8, we will compare one model-driven engineering approach that uses TGGs, eMoflon\(^3\), with our mapping language, to show some commonalities and differences between the two approaches, thus allowing an easier distinction, especially for those that are already familiar with TGGs.

There are multiple implementations for the concept of TGGs. eMoflon is an open source approach that provides plugins for the Eclipse IDE and for Enterprise Architect\(^4\) for specifying TGG rules graphically and generating synchronization code from the specified rules. For this purpose, rules are “operationalized”, i.e., each (bidirectional) rule is used to generate unidirectional rules that can be applied to models. The TGG Interpreter\(^5\) is, as suggested by the name, an interpretative approach to TGGs that are specified in graphical editors based on EMF. MoTE\(^6\) (Model Transformation Engine) uses Story Diagrams [35] as a base for model transformations. The differences between the different tools for realizing TGGs are discussed by Hildebrandt et al. [45] and incremental TGG tools in particular by Leblebici et al. [58].

### 7.3 .NET Modeling Framework Synchronization

The .NET modeling framework (NMF)\(^7\) [46, 47] is a modeling framework that uses the .NET software framework. NMF provides an internal DSL for C#, the NMF Transformations Language (NTL). NMF Synchronizations is an internal DSL for bidirectional synchronizations that can optionally propagate changes instead of deriving models in batch.

An internal DSL, essentially a framework that is accessible in C#, has the advantage of being well integrating into the existing mature tooling for C#. The languages has to always be correct syntax of the host language. In our language, we can separately express rules and generate code, which is not done in NMF Transformations.

### 7.4 Basic Object-oriented Transformation Language

The Basic Object-oriented Transformation Language (BOTL) [18] is a bidirectional model transformation language that also allows the graphical definition of rules that map struc-
tures similar to the patterns we use in our approach, or the graph patterns in TGGs, to each other. We are, however, only aware of implementations that derive models in batch-mode based on a BOTL specification, as opposed to our incremental and information preserving approach.

### 7.5 Queries, Views, Transformations (QVT)

The QVT (queries, views, transformations) standard [70] by the Object Management Group (OMG) also contains a transformation language for the relational description of bidirectional transformations, QVT relational (QVT-R). There are, however, known semantic inconsistencies [84] and we do not have knowledge of a stable and currently maintained implementation that adheres to the OMG standard.

There exist several implementations for QVT-R. The transformations can be run in both directions, but can mostly only be run in batch mode. There exist, however, semantics and implementations for incrementally running QVT-R transformations specifically for runtime models [80].

We do not know QVT-R implementations that use code generation to operationalize declarative QVT-R rules into imperative code similar to our approach with a special focus on the alignment of the generated code with the rules. Our approach only supports the incremental in-place transformation case, and does not cover the consistency checking between existing models, and supports no batch derivation of models. Furthermore we only relate exactly two heterogeneous meta models to each other.

### 7.6 ATL Transformation Language

The ATL (ATL Transformation Language)\(^8\) is a model transformation language that has been prototypically extended to work incrementally [51]. While bidirectional transformations are implemented as a pair of unidirectional transformations [52], there have been efforts to allow the bidirectionalization of existing unidirectional transformations by manipulating the byte code that results from an unidirectional model transformation specified in ATL [89] or by translating parts of the transformation into a bidirectional graph transformation language [76]. In contrast to our approach, these approaches do not use a declarative bidirectional model transformation language.

### 7.7 Constraint Solving

Echo is a framework for implementing least-change bidirectional model transformations written in QVT-R using a constraint solver for finding models that fulfill the QVT-R constraints additionally to OCL constraints of a meta model [63, 62].

Logic programming in form of Answer Set Programming (ASP) has been used for propagating changes between models via model approximations [26]. The approach can

\(^8\)http://www.eclipse.org/atl
find solutions for models that can not necessarily be created by a model transformation, by finding the nearest creatable model. The Janus Transformation Language (JTL) is a bidirectional model transformation language from which ASP programs are generated and then solved using an ASP solver [25].

7.8 Lenses

Lenses [37, 38] are a way of describing bidirectional model transformations for handling the view-update problem: reconciling changes in a view that has been generated from data by a transformation with the original data. The lenses approach is based on building an abstract view that contains the information that has to be synchronized with other related models. When a concrete format changes, the view is updated (via a function called \textit{get}) and if a view changes, the changes are propagated back to the concrete format (the function \textit{put} takes the original concrete data and the changed view to calculate the changed concrete data).

The approach provides a method of defining well-behaved [37, sec. 3] bidirectional model transformations. This notion of well-behavedness is also formalized in a set of rules the lenses have to adhere to that correspond with the intuitive notions, that (i) calculating a view and writing the unchanged view back to the original concrete data should not modify the original data (\textit{GetPut} rule) and that (ii) writing back a view and calculating the view immediately afterwards returns the same view (\textit{PutGet} rule).

Foster et al. describe a set of basic lenses and combinators that can be used to create complex lenses [37]. The set of well-behaved lenses is closed under those combinators (with the exception of recursion) which provides a robust framework for defining bidirectional transformation with well-defined behavior. The \textit{Boomerang} language\footnote{http://www.seas.upenn.edu/~harmony/} is a programming language for defining lenses.

Based on this basic assumption of an abstract and a concrete model (an asymmetric lens), lenses have also been applied to the symmetrical scenario [48], where two models that both have unique information are synchronized using a lens. Lenses can also be not only applied state-based but also be used for propagating deltas between models [31]. There is to our knowledge currently no implementation that realizes the delta-based propagation of changes in asymmetric lenses based on this theoretical observations.

Combinable bidirectional lenses for model synchronization as an internal model transformation language in the programming language Scala have been implemented by Wider [86].

Lenses provide many insights regarding the properties of “well-behaved” bidirectional model transformations and their composition and regarding change propagation in model transformations from a theoretical standpoint.

We are, however, also not aware of an implementation of lenses that allows the developer to directly see how the declarative specification is operationalized using code generation as in our approach. Additionally, we are not aware of a framework that implements lenses for model transformations with EMF, aside from the Scala-based implementation by Wider [87].
7.9 Reactive Programming

Reactive Programming [8] is a programming paradigm that can be used to develop event-driven applications, such as change-driven Vitruvius transformations. In the context of reactive programming, questions that are also relevant to the transformations that are generated from the MIR language family are examined: the value of which expressions can be impacted by the incoming change, which mapping preconditions can change their value, which attributes of opposing model elements must be adapted? Which structures must be adapted, deleted or created? The paradigm aims to free the programmer from explicitly specifying the impact of a change – “[a] change of value should be automatically propagated to all dependent computations” [8, sec. 3.2].

That is also the aim of the MIR language family in the context of model-driven engineering. For example, a common problem with reactive programming environments is the further propagation of changes that are the result of the reaction to a change. In the Vitruvius and MIR context it can also be the case that more than two models have to be kept consistent – either by specifying more than two transformations that have to be analyzed statically or dynamically for possible conflicts, or by propagating consistency preserving changes.

The projects ViATRA\textsuperscript{10} and EMF-IncQuery\textsuperscript{11} revolve around change-driven and reactive model transformations [13, 14, 15, 73]. A change-driven model transformation is a transformation in which a change model for one meta model is translated to a change model on another meta model, instead of manipulating the model directly. The ViATRA 3 model transformation platform [15] is built around the reactive programming paradigm. It is realized on top of an event-driven virtual machine that is domain-agnostic. The virtual machine allows the definition of actions that are executed when the state of the model is changed (similar to Vitruvius).

In the context of ViATRA and EMF-IncQuery, unidirectional and imperative transformations are triggered to react to specific changes incrementally. The used techniques for providing performant incremental transformations could be used as a future base for the result of the operationalization and generation based on the mapping specification in our language.

7.10 Change Impact Analysis

Change impact analysis (CIA) is a problem related to change propagation which aims to support developers in identifying parts of related software artifacts that are also affected by an incoming change. It is, however, mostly not concerned with fixing them automatically. CIA can be facilitated for model evolution scenarios by specifying impact rules in a DSL, which is in contrast to the MIR language family used to create hints for specific changes instead of propagating them to corresponding models. In addition to automatic model differencing, the developers can provide additional “user presettings” which describe the change of a model instead of recording atomic changes in models [67].

\textsuperscript{10}https://www.eclipse.org/viatra/
\textsuperscript{11}https://www.eclipse.org/incquery/
8 Comparison with a Triple Graph Grammar Tool

8.1 Introduction

In this chapter we want to show how the mapping language relates to Triple Graph Grammars (TGGs) and particularly to the TGG implementation that is used in the eMoflon\textsuperscript{1} tool suite for model-driven engineering. We chose eMoflon as a representative implementation based on the survey on TGG tools by Hildebrandt et al. \cite{45}, because eMoflon dominates in criteria that are relevant to our approach and that we will discuss in the following: application conditions and bidirectional constraints.

As reference, we used the eMoflon handbook\textsuperscript{2}, which describes in detail how to setup and use the eMoflon tooling and the involved languages. We focused in detail on the parts describing how the usage and evolution of models can be controlled using a technique called Story Driven Modeling (SDM) (part 3) \cite{1} and how models can be synchronized using TGGs (part 4) \cite{2}.

We used a SHARE virtual machine\textsuperscript{3}, that contains TGG examples, which is referenced on the eMoflon homepage\textsuperscript{4}. The provided virtual machine includes the eMoflon IDE plugin org.moflon.ide.feature.feature.group in version 1.7.18.201507192047.

8.2 Rule Specification

In eMoflon, bidirectional model transformations can be implemented using TGGs. The usual concrete syntax for TGGs is graphical across all TGG implementations that we are aware of, including eMoflon.

We have introduced the basic structure of TGGs in subsection 2.4.7. A rule consists of a pattern and a replacement. In eMoflon, a compact notation is used: the elements that are contained in both the pattern and the replacement are rendered with a black border, elements that are newly created in the replacement are rendered with a green border. When a rule is applied, the match for a pattern is replaced.

There is a basic difference in the specifications in TGGs in eMoflon and in the mapping language. The usual concrete syntax for TGGs is graphical. The rules are specified graphi-

\textsuperscript{1}http://www.emoflon.org
\textsuperscript{2}http://www.emoflon.org/emoflon/documentation/
\textsuperscript{3}http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_eMoflon-TGG-Examples.vdi
\textsuperscript{4}http://www.emoflon.org/emoflon/emoflon-in-action/
Comparison with a Triple Graph Grammar Tool

cally and an additional textual specification is the fallback, for example for constraints on model elements.

An example for the graphical syntax of a TGG rule that is part of the “Leitner’s Learning Box” example from the eMoFlon documentation is shown in Figure 8.1. The learning box models a system for memorizing information that is stored on cards, which have a front and a back side that contains the information that a user is expected to be able to associate with each other from memory. The cards are organized in different partitions inside a box, and are moved according to the current memorization status.

The example transformation in the eMoFlon documentation maps a learning box and a dictionary that contains a set of entries that contain the same information as the cards in the box, but as a concatenation of the back and front side of a card.

The basic rule for creating a corresponding box and dictionary is displayed in Figure 8.1. A box always contains three partitions that have a fixed index. A mapping rule which achieves a similar effect and which we have designed in accordance with the TGG from the example is displayed in Listing 8.1. The left side of the TGG that concerns the learning box language is expressed in lines 2 to 14. The right side is specified in line 15. The equality constraint of the name of the box and the title of the dictionary can be found in line 16.

Similarly, the rule for mapping individual cards to dictionary entries from the eMoFlon example is shown in Figure 8.2. The rectangle at the bottom contains the complex attribute constraints between a Card and an Entry that are mapped by the TGG rule. The three expressions that use the free variables word and meaning create a constraint solving problem that is solved depending on the direction the rule is applied. To simplify the example, we omitted an additional rule that ties the index of the partition of a card to the level of an entry, thus deciding which partition to use for a card.

Our mapping version is displayed in Listing 8.2. In our version, the two directions of the constraint solving problem are specified manually, since we do not have appropriate constraints in our language. Since our mapping does not allow us to match an arbitrary additional partition, we also specify manually the condition to check if the card is contained in the box (lines 4 to 6) and how to put the card in the first partition of the box on creation (lines 7 to 9).

This is one major difference between TGGs and the mapping language. Our approach does not currently allow the matching of arbitrary elements, but only considers elements that have been created inside a rule. However, the constraint language for specifying the signature constraints could be extended to allow additional constraints, such as the one used in the example. Alternatively, an invariant and a response could for example be used to ensure that a box always contains at least one partition, therefore preventing the get(0) in line 8 from returning a null pointer.

Our approach allows the specification of arbitrary manually specified constraints which consist of a check and enforce pair. In eMoFlon, values (either of literals or more complex) can be assigned to elements in the newly created side of a TGG rule. In rule Figure 8.1, each partition is assigned a literal value for the index attribute.

The tooling also allows for other operators for checking, such as equality or greater/-less than. Values that can be assigned are not limited to literals. An editor in Enterprise Architect inside the provided virtual machine for specifying this object attribute con-
Figure 8.1: Example for the TGG rule `BoxToDictionary` from the “Leitner’s Learning Box” example. Adapted from [2, p. 22]

```
mapping boxToDictionary:
map [ boxLang.Box,
     boxLang.Partition as p0,
     boxLang.Partition as p1,
     boxLang.Partition as p2 ]
with [
    equal(p0.index, 0)
    equal(p1.index, 1)
    equal(p2.index, 2)
    in(p0, box.containedPartition)
    in(p1, box.containedPartition)
    in(p2, box.containedPartition)
    in(p1, p0.next) in(p0, p1.previous)
    in(p2, p1.next) in(p0, p2.previous) ]
and [ dictLang.Dictionary ]
{ [ equal(box.name, dictionary.title) ] }
```

Listing 8.1: Sketch of the `BoxToDictionary` TGG rule as a mapping.
Listing 8.2: Sketch of the CardToEntry TGG rule as a mapping
8.3 Code Generation

In the code generated from the mapping language, we focused on making the generated code understandable for a developer. We explained different measures towards this in chapter 5, for example wrappers that contain methods for getting correctly typed model elements by name from a mapping instance. Additionally, we try to create code that has little indirection in control flow, regarding the propagation of attributes and the checking and enforcing of signature instances.

In eMoflon, the generated code is not specifically generated for this purpose. The code generator used for SDMs allows a flexible and efficient approach to solving the pattern matching and constraint solving problem that the TGGs are centered around.

An example for a generated method for propagating the name from a box to the title of a dictionary is displayed in Listing 8.3. The constraint solving problem (CSP) is created dynamically, and variables are added to it. The name of the box is bound in this direction.
Comparison with a Triple Graph Grammar Tool

Figure 8.3: Example for the editing of attribute constraints inside the graphical editor of eMoflon in the SHARE virtual machine

and the equals constraint (Eq) is added. In the mapping language framework, the generated code directly assigns the values (Listing 8.4).

The code for creating a correct instance of the learning box side of the box-to-dictionary rule is displayed in Listing 8.5. The corresponding code generated from the mapping language can be seen in Listing 8.6. The actual creation of the instances is done dynamically based on the signature that can also be found in the generated code. After generation, our concrete implementation for the box-to-dictionary mapping, including the generated types and named wrappers, comprises about 400 lines of code. Altogether the generated code contains around 30 methods. The mapping class comprises 13 methods with about 220 lines of code. The generated code from the eMoflon rule has over 6000 lines of code with over 300 methods.

The reason for this difference is that the code generated from the TGGs in eMoflon is not generated to be read by the user. Furthermore, it contains methods for matching all combinations of bound and unbound variables in a TGG rule and therefore also contains code used for pattern matching, which we omitted in our approach. We want to highlight, however, the difference in focus of the two approaches. In our approach, the language that is used to specify rules is much closer to the code that is generated from it, and the structures a user specifies can be traced back more easily to the generated code. The main focus is on coming from a bidirectional, declarative description to two consistent unidirectional imperative transformations.

The mapping language specification is closer to the actual operationalization of the rules, and therefore could be considered less “declarative”. However, we want to make the impact of the specified rules clear to the user, regarding the code that is executed when
8.4 Development Environment

```java
public CSP isApplicable_solveCsp_FWD(IsApplicableMatch isApplicableMatch, Box box,
Partition partition0, Partition partition1, Partition partition2) {
    // Create CSP
    CSP csp = CspFactory.eINSTANCE.createCSP();
    isApplicableMatch.getAttributeInfo().add(csp);

    // Create attribute variables
    Variable var_box_name = CSPFactoryHelper.eInstance
        .createVariable("box.name", true, csp);
    var_box_name.setValue(box.getName());
    var_box_name.setType("String");

    // Create unbound variables
    Variable var_dictionary_title = CSPFactoryHelper.eINSTANCE
        .createVariable("dictionary.title", csp);
    var_dictionary_title.setType("String");

    // Create constraints
    Eq eq = new Eq();
    csp.getConstraints().add(eq);

    // Solve CSP
    eq.setRuleName("");  
    eq.solve(var_box_name, var_dictionary_title);
}
```

Listing 8.3: Propagating the name of a box to the title of a dictionary in eMoflon

a pattern is newly matched, or when attributes are propagated if a matched instance is changed. We aim to achieve this in both the textual syntax of the language as well as in the code that is generated from the specification.

TGGs can be hard to understand in their effect, since the span from a TGG rule to the operationalization and the effect that follows a concrete change is wider and we also see a necessity for understanding the underlying algorithm in more complex cases.

### 8.4 Development Environment

eMoflon is based on the Eclipse IDE, as is VITRUVIUS, and uses the Eclipse Modeling Framework and the Ecore meta model. Eclipse contains graphical editors for manipulating models and meta models which are instances of Ecore based meta models or the Ecore meta model. Additionally, eMoflon provides editors for manipulating unidirectional model transformations (Story Driven Modeling, SDMs) and bidirectional model transformations (TGGs), in a graphical syntax. The editors are based on Enterprise Architect\(^5\), which is a

\(^5\)http://www.sparxsystems.com/products/ea/
public static void propagateAttributesFromBoxLang(
    BoxToDictionary_Correspondence_Wrapper boxToDictionary,
    MappingExecutionState state) {

    state.record(boxToDictionary.getDictLang().getDictionary());
    // dictionary.title := box.name
    boxToDictionary.getDictLang().getDictionary()
        .setTitle(boxToDictionary.getBoxLang().getBox().getName());

    state.updateAllTuidsOfCachedObjects();
    state.persistAll();
}

Listing 8.4: Propagating the name of a box to the title of a dictionary in the mapping
language

tool for model driven engineering. The transformations are models and code is generated
from the transformation models using CodeGen2, which is a part of the Fujaba Tool Suite6.

8.5 Other Triple Graph Grammar Approaches

There are different other implementations of TGGs that vary in different aspects. Their
individual strengths and differences in implementation and expressiveness are, however,
often not obvious. Some approaches allow for an incremental execution of rules. The
destruction of structures – which our approach does if a rule does not match anymore – is
also not done in all (incremental) approaches.

Hildebrandt et al. provide a survey of TGG tools which includes a framework for
comparing TGG tools both qualitatively and quantitatively and compare the tools MoTE7,
TGG Interpreter8, and eMoflon [45]. Leblebici et al. specifically focus on the incrementality
of TGG tools in a subsequent survey, again providing a set of qualitative and quantitative
comparison criteria and comparing the same concrete tools [58].

6http://www.fujaba.de/
7http://www.mdelab.de/mote/
8http://www-old.cs.uni-paderborn.de/en/research-group/software-engineering/research/
projects/tgg-interpreter.html
8.5 Other Triple Graph Grammar Approaches

```java
public static final Object[] pattern_BoxToDictionaryRule_37_6_greenFFFFFFBB(
    ModelgeneratorRuleResult ruleResult, CSP csp) {
    Box box = LearningBoxLanguageFactory.eINSTANCE.createBox();
    BoxToDictionary boxToDictionary =
        LearningBoxToDictionaryIntegrationFactory.eINSTANCE.createBoxToDictionary();
    Dictionary dictionary = DictionaryLanguageFactory.eINSTANCE.createBoxToDictionary();
    Partition partition0 = LearningBoxLanguageFactory.eINSTANCE.createPartition();
    Partition partition1 = LearningBoxLanguageFactory.eINSTANCE.createPartition();
    Partition partition2 = LearningBoxLanguageFactory.eINSTANCE.createPartition();
    Object _localVariable_0 = csp.getValue("box", "name");
    Object _localVariable_1 = csp.getValue("dictionary", "title");
    int partition0_index_prime = Integer.valueOf(0);
    int partition1_index_prime = Integer.valueOf(1);
    int partition2_index_prime = Integer.valueOf(2);
    boolean ruleResult_success_prime = Boolean.valueOf(true);
    int _localVariable_2 = ruleResult.getIncrementedPerformCount();
    ruleResult.getSourceObjects().add(box);
    boxToDictionary.setSource(box);
    ruleResult.getCorrObjects().add(boxToDictionary);
    boxToDictionary.setTarget(dictionary);
    ruleResult.getTargetObjects().add(dictionary);
    partition0.setBox(box);
    ruleResult.getSourceObjects().add(partition0);
    partition1.setBox(box);
    ruleResult.getSourceObjects().add(partition1);
    partition0.setNext(partition1);
    partition1.setPrevious(partition0);
    partition1.setBox(box);
    ruleResult.getSourceObjects().add(partition1);
    partition2.setBox(box);
    partition2.setPrevious(partition0);
    ruleResult.getSourceObjects().add(partition2);
    String box_name_prime = (String) _localVariable_0;
    String dictionary_title_prime = (String) _localVariable_1;
    partition0.setIndex(Integer.valueOf(partition0_index_prime));
    partition1.setIndex(Integer.valueOf(partition1_index_prime));
    partition2.setIndex(Integer.valueOf(partition2_index_prime));
    ruleResult.setSuccess(Boolean.valueOf(ruleResult_success_prime));
    int ruleResult_performCount_prime = Integer.valueOf(_localVariable_2);
    box.setName(box_name_prime);
    dictionary.setTitle(dictionary_title_prime);
    ruleResult.setPerformCount(Integer.valueOf(ruleResult_performCount_prime));
    return new Object[] { box, boxToDictionary, dictionary, partition0,
        partition1, partition2, ruleResult, csp };
}
```

Listing 8.5: Creation of a box and its partitions in eMoflon
public static void enforceCorrectInitializationOnBoxLang(
    BoxToDictionary_Wrapper_BOXLANG boxLang,
    MappingExecutionState state) {
    state.record(boxLang.getBox());
    state.record(boxLang.getP0());
    state.record(boxLang.getBox());
    state.record(boxLang.getP1());
    state.record(boxLang.getBox());
    state.record(boxLang.getP2());
    state.record(boxLang.getBox());
    state.record(boxLang.getP0());
    state.record(boxLang.getP1());
    state.record(boxLang.getBox());
    state.record(boxLang.getP1());
    state.record(boxLang.getP2());
    state.record(boxLang.getP0());
    state.record(boxLang.getP1());
    state.record(boxLang.getP2());
    state.record(boxLang.getP0());
    state.record(boxLang.getP1());
    state.record(boxLang.getP2());
    state.record(boxLang.getP0());
    state.reportIndexState(0);
    boxLang.getP0().setIndex(0);
    boxLang.getP1().setIndex(1);
    boxLang.getP2().setIndex(2);
    boxLang.getBox().getContainedPartition().add(boxLang.getP0());
    boxLang.getBox().getContainedPartition().add(boxLang.getP1());
    boxLang.getBox().getContainedPartition().add(boxLang.getP2());
    boxLang.getP0().setNext(boxLang.getP1());
    boxLang.getP1().setPrevious(boxLang.getP0());
    boxLang.getP1().setNext(boxLang.getP2());
    boxLang.getP2().setPrevious(boxLang.getP0());
    state.updateAllTuidsOfCachedObjects();
    state.persistAll();
}

Listing 8.6: Creation of a box and its partitions in the mapping language.
9 Conclusion and Future Work

9.1 Conclusion

In this thesis, we described our approach to specifying consistency relationships between models of different modeling languages that describe a system in a partially redundant way. Partially automating the repair of inconsistencies between models in a model-driven engineering context is important, because manual repair is time-consuming and requires a detailed understanding of the involved modeling languages. If changes are avoided because they entail an overhead that is too large, the original goal of using suitable abstractions for changing the system description is hindered.

In our mapping language, consistency relationships are specified as declarative mappings between patterns on the meta model level. From the specification, we generate imperative code that is executed after a change in a model and enforces our notion of consistency. If a match for a pattern is found in a model after a change, it is mapped to a corresponding, newly created structure. If a mapped structure is changed and does not match the pattern anymore, the corresponding structure is destroyed. Additionally, constraints on a mapped pair of structures can be formulated and are repaired, if they are violated after a change.

Our description of the different aspects of the consistency preservation problem provides an overview that was gathered from existing approaches and literature, and during the work on the mapping language. It can be used to understand our and other similar approaches better, and to simplify the categorization of mechanisms for consistency repair regarding their respective capabilities and foci.

Our implementation of the mapping language is built with the Xtext language framework and provides the syntax for specifying consistency relationships. We evolved the mapping language from a very basic and limited feature set to a more comprehensive specification language. For this purpose, we iteratively extended and tested the language prototype on the base of different examples from both academia and industry.

We built the mapping language framework and generators to be extensible, and provided some extensions that we deemed necessary in the VITRUVIUS context. Based on the work on the mapping language prototype, we also expect it to be possible for future developers in the VITRUVIUS project to provide additional language features.

Furthermore, we provide a generator that generates change-driven model transformations in Java from those mapping declarations.

As a short summary, we make following contributions:

- An overview of problems associated with specification and execution of consistency preserving transformations.
• A mapping language for specifying consistency requirements, described both informally and more precisely in a formalized form.

• An editor for this language.

• A framework for generating imperative code from mapping language specifications. The code can be plugged into the Vitrivius model-driven engineering environment and is executed if models are changed, thus maintaining consistency based on the specification.

## 9.2 Future Work

In future work, the expressiveness of the mapping language should be extended by implementing additional language features. These language features can provide more powerful abstractions for describing the patterns that are mapped to each other, and for describing constraints that affect the mapped pair of model structures. We have already provided some pointers for possible extensions in section 4.5, such as reusable bidirectional mapping methods or patterns that can match paths of references of dynamic length. Furthermore, we have discussed the limitations of the language in section 4.6, for example regarding static validation of the mapping specification and the integration of existing models.

For our implementation, we need to implement a more efficient mechanism for incrementally matching patterns in models, and for handling candidates that are newly matched, or not matched anymore to make our approach usable for larger models. For this purpose, existing techniques and implementations should be evaluated and integrated into our approach, if possible.

Apart from the previous future work for the mapping language, there is also future work for the integration of the mapping language in the context of the MIR language family. The mapping language should be integrated to a greater degree with the other languages in the MIR language family, to allow references between the languages and for static analysis for avoiding conflicts between consistency transformations that are specified using the different languages.

For the evaluation of the mapping language, and the MIR language family as a whole, case studies should be done in which the languages are used to express more complex meta model relationships to find out more about the limitations and the interplay of the languages in the MIR language family. At best, the case studies also include the elicitation of consistency requirements. Furthermore, experimental studies about the usability and comprehensibility of the MIR language family might provide additional insights about the strengths and weaknesses of the MIR language family compared to other approaches.


Bibliography


