Karlsruhe Institute of Technology

Software Design and Quality

Designing a Change-Driven Language for Model Consistency Repair Routines

Master Thesis of

Heiko Klare

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 Koziolek
Advisor:	Dipl.-Inform. Max. E. Kramer
Second advisor:	M.Sc. Michael Langhammer

14. December 2015 – 13. June 2016

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

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, 10. June 2016

. . . .

(Heiko Klare)

Abstract

A software system under development can be described by several models, which represent different concerns or abstractions of the system. These models can contain dependent or even redundant information.

Whenever a software developer changes such models without considering their dependencies, the models can get *inconsistent*. A developer can hardly maintain an overview of all models involved in a development process and the dependencies between them. Therefore, automated mechanisms for restoring the consistency after model changes are necessary.

Existing approaches provide declarative constructs for specifying the model dependencies and automatically derive consistency-restoring mechanisms from them. These approaches have *limited expressiveness* and only allow *restricted influence* on the way consistency is restored. They prescribe a certain way of restoring consistency although different possibilities exist.

In this thesis, we present the change-driven *response language* for the repair of model consistency. It allows to specify the way of repair *explicitly* in *imperative* routines, which are executed in reaction to specified changes. Besides, the language provides constructs that encapsulate recurring actions and make them reusable. We introduce a consistency definition focused on comprehensibility and a categorization of possible changes within models. The presented language reflects a generic structure for change-driven consistency repair, derived from our consistency definition.

We provide an evaluation of our approach in a case study realizing the consistency of architecture descriptions and their implementation in object-oriented code. The evaluation confirms the applicability of the proposed language for preserving consistency in that exemplary case and reveals benefits compared with a manual implementation.

Zusammenfassung

Ein Software-System kann während der Entwicklung durch verschiedene Modelle beschrieben werden, um unterschiedliche Teile oder Abstraktionen des Systems darzustellen. Diese Modelle können voneinander abhängige oder sogar redundante Informationen enthalten.

Wenn ein Softwareentwickler an solchen Modellen Änderungen vornimmt ohne diese Abhängigkeiten zu beachten, können die Modelle *inkonsistent* werden. Ein Entwickler ist kaum in der Lage einen Überblick über alle Modelle eines Entwicklungsprozesses und deren Abhängigkeiten zu behalten. Daher sind automatisierte Verfahren zur Wiederherstellung von Konsistenz nach Modelländerungen notwendig.

Existierende Methoden erlauben die deklarative Spezifikation von Abhängigkeiten zwischen Modellen, aus denen automatisiert Mechanismen zur Konsistenzerhaltung abgeleitet werden. Diese Ansätze sind in ihrer *Ausdrucksmächtigkeit beschränkt* und bieten nur *eingeschränkten Einfluss* auf die Art und Weise in der Konsistenz wiederhergestellt wird. Sie legen automatisiert eine Art der Wiederherstellung der Konsistenz fest, obwohl es verschiedene Möglichkeiten dafür gäbe.

In dieser Arbeit stellen wir die änderungsgetriebene *Response-Sprache* für die Konsistenzerhaltung von Modellen vor. Sie erlaubt es, die Art und Weise der Wiederherstellung von Konsistenz *explizit* in *imperativen* Programmen festzulegen, welche als Reaktion auf festgelegte Änderungen ausgeführt werden. Zusätzlich bietet sie Sprachkonstrukte an, die wiederkehrende Reaktionen kapseln und wiederverwendbar machen. Wir führen einen Konsistenzbegriff mit dem Fokus auf Verständlichkeit ein und stellen eine Kategorisierung von möglichen Modelländerungen vor. Die Sprache ist entsprechend einer allgemeingültigen Struktur für die änderungsgetriebene Wiederherstellung von Konsistenz aufgebaut, welche wir aus unserem Konsistenzbegriff herleiten.

Wir stellen eine Evaluation unseres Ansatzes anhand einer Fallstudie zur Konsistenzerhaltung von Architekturbeschreibungen und deren Implementierung in objektorientiertem Code vor. Die Evaluation zeigt die Anwendbarkeit der vorgestellten Sprache für die Sicherstellung von Modellkonsistenz in diesem konkreten Fall und verdeutlicht einige Vorteile gegenüber einer manuellen Implementierung der Mechanismen zur Konsistenzerhaltung.

Contents

Abstract i

Zusammenfassung iii

1 Introduction 1
1.1 Describing Software Systems with Models 1
1.2 Preserving Model Consistency 2
1.3 The Response Language 3
1.4 Structure of the Thesis 4

2 Foundations 5
2.1 Model-Driven Software Development 5
2.1.1 Metamodels 6
2.1.2 Meta-Levels 7
2.1.3 Domain-Specific Languages 7
2.2 Model Transformations 8
2.2.1 General Properties of Transformations 9
2.2.2 Transformation Execution Modes 10
2.2.3 Transformation Specification Approaches 11
2.3 The Eclipse Modeling Framework 12
2.3.1 Essential Meta-Object Facility and Ecore 13
2.3.2 Textual Domain-Specific Languages with Xtext 13
2.3.3 Reusable Programming Languages Xbase and Xtend 14
2.4 View-based Software Development 15
2.4.1 Projective and Synthetic Views 16
2.4.2 Orthographic Software Modeling 16
2.4.3 The Vitruvius Approach 17
2.5 Change-Driven Development 19

3 Running Example: Consistency of Software Architecture and Implementation 21
3.1 A Software Architecture Description Metamodel 21
3.2 A Metamodel for the Java Programming Language 23
3.3 Relations between Software Architecture Descriptions and Code Models 25
3.3.1 Repository Mapping 25
3.3.2 Component Mapping 26
3.3.3 Interface Mapping 27
3.3.4 Data Type Mapping 27

4 Model Changes and Model Consistency **29**
4.1 Model Consistency . . . 29
4.1.1 Consistency Overlaps . . . 29
4.1.2 Identifying Consistency Overlaps . . . 32
4.1.3 Dependency between Consistency Overlaps . . . 34
4.1.4 Granularity of Consistency Overlaps . . . 35
4.1.5 Prescriptive and Descriptive Consistency Overlaps . . . 37
4.1.6 Correspondence Models . . . 37
4.1.7 Preserving Model Consistency . . . 38
4.2 Model Changes . . . 39
4.2.1 Change Descriptions . . . 40
4.2.2 Atomic Model Changes . . . 40
4.2.3 Atomic Model Changes in the Essential Meta-Object Facility . . . 41
4.2.4 Atomic Model Changes in Ecore . . . 43
4.2.5 Composite Model Changes . . . 44
4.3 Change-Driven Model Transformation Environments . . . 45
4.3.1 Monitors for Model Changes . . . 45
4.3.2 Transformation Execution Engines . . . 46
4.3.3 Change-Driven Model Transformations . . . 47
4.4 Change-Driven Consistency-Restoring Transformations . . . 47
4.4.1 Atomic Changes as Triggers . . . 47
4.4.2 Intra-Model Consistency after Atomic Changes . . . 49
4.4.3 Responsibilities of Transformations . . . 50
4.4.4 Structure of Transformations . . . 51
4.4.5 Change-Driven Consistency Repair Languages . . . 53

5 The Response Language **55**
5.1 Introduction to the Response Language . . . 55
5.1.1 Language Structure . . . 55
5.1.2 Basic Language Constructs . . . 56
5.1.3 A Metamodel for Change Descriptions in Ecore . . . 58
5.2 Triggers: Identifying Potential Inconsistencies . . . 59
5.2.1 Specifying Change Type and General Preconditions . . . 60
5.2.2 Restricting Change Properties . . . 61
5.2.3 Change Type Generalization . . . 63
5.3 Matchers: Retrieving Elements of Consistency Overlaps . . . 65
5.3.1 Retrieving Corresponding Elements . . . 66
5.3.2 Restricting Retrieved Elements by Filter Functions . . . 67
5.3.3 Restricting Retrieved Elements by Correspondence Tags . . . 68
5.3.4 Retrieving Optional Elements . . . 70
5.3.5 Expecting the Non-Existence of Corresponding Elements . . . 71
5.3.6 Specifying Further Conditions . . . 71
5.4 Effects: Restoring Consistency . . . 72
5.4.1 Specifying General Effects . . . 73
5.4.2 Creating Model Elements . . . 74

5.4.3 Adding and Removing Correspondences 75
5.4.4 Deleting Model Elements 76
5.5 Separating Triggers from Repair Routines 77
5.5.1 Specifying Repair Routines with Matchers and Effects 78
5.5.2 Reusing Repair Routines 79
5.5.3 Iterating Repair Routine Calls 80
5.6 Language Properties and Responsibilities 82
5.6.1 Target Models of Responses 82
5.6.2 Transitivity of Consistency Repair Routines 82
5.6.3 Visibility of Model States 83
5.6.4 Persistence of Logical and Physical Models 84
5.7 Aspects of Restoring Consistency with Responses 85
5.7.1 Response Execution Order 85
5.7.2 Modifications of Bidirectional References 85
5.7.3 Interaction with the Model User 86
5.7.4 Meta-Level Breaks 87
5.8 Possible Extensions 89
5.8.1 Composite Change Triggers 89
5.8.2 Constraint Satisfaction Change Triggers 90
5.8.3 Explicit Consistency Overlap Type Specification 91
5.8.4 Extensions for User Interaction 91
5.8.5 Imperative Code Validation 92
5.8.6 Language Constructs for Element Modifications 92

6 Response Language Implementation 93
6.1 The Vitruvius Framework 93
6.1.1 Virtual Single Underlying Model 93
6.1.2 Correspondences, TUIDs and Changes 94
6.1.3 Model Synchronization 95
6.2 Runtime Environment and Transformation Structure 96
6.2.1 Structure of the Runtime Environment 96
6.2.2 Runtime Data Structures 97
6.2.3 Responses 99
6.2.4 Response Executors 99
6.2.5 Repair Routines 100
6.2.6 Repair Routine Facades 100
6.3 Response Language Specification and Code Generation 101
6.3.1 The MIR Base Language 101
6.3.2 Response Language Grammar 102
6.3.3 Code Generation 102
6.3.4 Scoping for Response Constructs 103
6.3.5 Scoping and Validation of Code Blocks 105
6.3.6 Identifier Validity 105
6.3.7 Runtime Environment Generation 105
6.3.8 Constructs for Debugging Responses 106

6.4 Possible Extensions 106
6.4.1 Restriction of Referenceable Metaclasses 107
6.4.2 Modularization of Responses Specification 107

7 Evaluation **109**
7.1 Functionality 109
7.1.1 Reaction to Possible Changes 110
7.1.2 Definition of Arbitrary Repair Logic 112
7.2 Applicability 113
7.2.1 Consistency of Architecture Description and Code 113
7.2.2 Limitations 116
7.2.3 Threats to Validity 116
7.3 Benefits 117
7.3.1 Trigger Specifications 117
7.3.2 Correspondence Handling 118
7.3.3 Relevance of Language Constructs 119
7.3.4 Reuse of Repair Routines 120
7.4 Evolution Scenarios 121
7.4.1 Transformation Environment Evolution 121
7.4.2 Metamodel Evolution 122
7.4.3 Consistency Overlap Type Evolution 123

8 Related Work **125**
8.1 Change-Driven Development and Reactive Programming 125
8.2 Consistency Specification and Validation 126
8.3 General Topics of Model Consistency Repair 126
8.4 Problem-Specific Model Consistency Repair 127
8.5 Declarative Model Consistency Repair 128
8.5.1 Bidirectional Transformation Languages 129
8.5.2 Logical Expression Approaches 130
8.6 Imperative Model Consistency Repair 130
8.6.1 General Approaches 131
8.6.2 Reactive Model Transformations with VIATRA 131

9 Conclusion and Future Work **133**
9.1 Conclusion 133
9.2 Future Work 134

Bibliography **135**

List of Figures

2.1 Basic concepts of model transformations, adapted from [21] 9
2.2 Execution modes of model transformations 11
2.3 Example for a system of projective views and synthetic views 17
2.4 Different roles and their interaction with languages and artifacts for consistency preservation in the Vitruvius framework, adapted from [54] . . 19

3.1 Simplified extract of the PCM metamodel 22
3.2 Simplified extract of the JaMoPP metamodel 24
3.3 Mapping of a PCM repository to Java packages 25
3.4 Mapping of a PCM component to a Java package and class 26
3.5 Mapping of a PCM interface to a Java interface 27
3.6 Mapping of a PCM composite data type to a Java class 28

4.1 Representation of two interfaces in Java code and a UML class diagram . 30
4.2 Representation of a PCM required role in the Java class implementation of the requiring component 31
4.3 Dependencies of the consistency overlap types for PCM required and provided roles and their Java implementation 35
4.4 Dependencies of minimal consistency overlap types for PCM required roles and their Java implementation 36
4.5 Achievable model consistency with different kinds of transformation specification 39
4.6 Change description types and the directions they describe 40
4.7 Extract of the change-relevant elements of EMOF 42
4.8 Extract of the change-relevant elements of the Ecore meta-metamodel . 43
4.9 Generic structure of a change-driven model transformation environment 45
4.10 Consistency-restoring transformations in a change-driven environment . 49
4.11 Structure of a change-driven consistency-restoring transformation 52

5.1 A metamodel for descriptions of atomic changes in Ecore 59
5.2 Completed effect grammar 77
5.3 Extract of the Java metamodel defining types and type references 88

6.1 Interfaces of the Vitruvius synchronization environment 95
6.2 Structure and dependencies in an exemplary runtime environment with one response and two repair routines 97

7.1 A minimal Ecore-based metamodel providing all kinds of elements that are changeable in model instances 111

List of Tables

4.1 Atomic change types in EMOF 42
4.2 Atomic change types in Ecore 44

5.1 Inference of concrete change types from generic changes and feature properties 64

6.1 Representation of response language elements in the implementing Java code 104

7.1 Operations for producing atomic changes in the minimal example metamodel 112
7.2 Consistency overlap types that were considered in the case study 114
7.3 Number of required and implemented responses for the different modification types of the consistency overlap types in Table 7.2 115
7.4 Numbers of used response language constructs in the case study 120
7.5 Numbers of responses, repair routines and repair routine calls in the case study 121

Listings

2.1 Type inference and operator overloading in Xbase 15

5.1 Basic language constructs of the response language 57
5.2 Demonstration of the keyword coloring in the examples 58
5.3 Language extension for triggers with change type specification and general preconditions 61
5.4 A trigger for reacting to the creation of a PCM repository 61
5.5 Language extension for triggers with change property restrictions 62
5.6 A trigger for reacting to the insertion of a PCM datatype 63
5.7 Completed trigger grammar 64
5.8 A trigger for the insertion of a composite data type into a repository . . 65
5.9 Language extension for retrieving an element 66
5.10 A matcher for retrieving the corresponding class of a component 67
5.11 Language extension for filtering retrieved elements 68
5.12 A matcher for retrieving an element with a user-defined filter function . 68
5.13 Language extension for restricting retrieved elements by correspondence tags 69
5.14 A matcher for retrieving an element using a tagging mechanism 69
5.15 Language extension for retrieving optional elements 70
5.16 A matcher for retrieving a potentially non-existent element 70
5.17 Language extension for requiring the non-existence of a correspondence 71
5.18 A matcher expecting the non-existence of a correspondence 71
5.19 Completed matcher grammar 72
5.20 Language extension for effects specified in a code block 73
5.21 Language extension for creating model elements 74
5.22 A response that creates a Java class after a PCM component creation . . 74
5.23 Language extension for creating and removing correspondences 75
5.24 A response for creating the contracts and data types package of a just created PCM repository with tagged correspondences between them . . 76
5.25 A response that deletes a Java class after removing the corresponding PCM component 77
5.26 Response language grammar with separated repair routines 79
5.27 Responses reusing repair routines for creating Java classes 80
5.28 A response iterating repair routine calls 81
5.29 Exemplary persistence specification interface 85
5.30 Exemplary user interaction interface 86
5.31 A response for a component removal with user interaction 87

5.32 Separate responses for primitive and class types in Java models 89

6.1 The interface of a ResponseElementState 98
6.2 The interface of a ResponsElementStatesHandler 98
6.3 The interface of a Response 99
6.4 The interface of a ResponseExecutor 99
6.5 The interface of a RepairRoutine 100
6.6 A repair routine for adding the implementation of a PCM required role to the Java model in the implemented response language 102

7.1 Example trigger implementation in the response language 118
7.2 Example trigger implementation in Xtend code 118
7.3 Retrieving the compilation unit corresponding to a PCM component in the response language 119
7.4 Retrieving the compilation unit corresponding to a PCM component in Xtend code 119

1 Introduction

Information about a software system is typically spread across several artifacts. In common software development processes, those artifacts can be the documentation of requirements, the design documentation and the implementation. An artifact contains information that represents a certain aspect or abstraction of the system under development. This information is also present within, or dependent from other artifacts.

The dependencies between different artifacts are often not formally documented [9]. Problems arising from such redundant or dependent information can be recognized in software development processes, in which a system is designed in UML diagrams and implemented in program code. The tracing information, which specifies the diagram elements to which a code fragment corresponds, is not recorded explicitly. It is just declared implicitly by equal naming. The absence of this tracing information can easily lead to inconsistencies because the modifications during an evolution of one artifact may not be propagated to the corresponding elements in other artifacts. A simple example is the renaming of a class in program code, which is not propagated to a related UML class diagram.

For certain cases of particular relevance, approaches for preserving consistency between artifacts have been developed. For example, consistency preservation between design documentation in form of UML diagrams and program code can be achieved with several round-trip engineering approaches. Examples are UML Lab, which is based on Fujaba [70], Borland Together [14] or the Enterprise Architect [87]. Such tools provide a consistency-preserving mechanism for two certain kinds of artifacts. Nevertheless, a software system usually consists of several artifacts, which all have to be kept consistent.

1.1 Describing Software Systems with Models

Models are used in several engineering disciplines for different tasks. A model describes an abstract representation of a real or an artificial object. Constructional engineers model components of planned buildings by describing only some of their properties, such as their dimensions and materials. These models can be used to compute or simulate the behavior of the components regarding different load situations to validate that they will sustain realistic load scenarios. Other roles within the construction process of the building potentially require models that represent other aspects of the same components. An architect may require a surface model of the components to ensure that they fit into each other and to discuss the building appearance with the building owner.

Model-driven software development brings these concepts to the domain of software engineering and becomes a common approach in various domains. Software models raise the abstraction of software to a higher level by omitting language-specific elements.

Different models of the same software system represent certain aspects of the system, which are relevant for different developer roles. For example, a software architect can work on an architecture model that represents an abstraction of the program code, which in turn software engineers works on.

While UML diagrams are commonly referred to as software models, requirements documentation and even program code can also be treated and described as models. Apart from these common models, in model-driven development environments several domain-specific models can be used for the representation of different aspects of the system. For example, business processes can be modeled using the Business Process Model and Notation (BPMN) [71], and software for the automotive domain is developed with models like AUTOSAR [85], ASCET [31] or AMALTHEA [86]. Finally, in model-driven software engineering, any artifact that is involved in the software development process is considered as a model.

The relationships between models are of central interest in model-driven development. Like described for UML diagrams and program code in the beginning, models contain interrelated information. Especially the finally delivered artifact, the program code or its compilation, respectively, combines the information of all models that describe the system and thus is related to most of the used models. To describe these interdependencies and to derive models from others, model transformations are used. A model transformation describes how one model can be transformed into another and thus implicitly defines the relation between these two models. The specification of model transformations is simplified through model transformation languages. They allow a specification of transformations that abstracts from technical requirements of transformations and generate executable code from such a specification.

1.2 Preserving Model Consistency

In general, the avoidance of inconsistencies due to interrelated information can be addressed in two ways. First, it can be attempted to avoid the existence of interrelated information. Second, interrelated information can be explicitly kept consistent.

The avoidance of the existence of interrelated information can be achieved by the definition of a single model that contains all system-relevant information without redundancies. Such a so called single underlying model was introduced by Atkinson, Stoll, and Bostan [6]. Nevertheless, in practice this approach is hardly applicable. Different domains use different models that would all have to be integrated and representable in such a single model.

The Vitruvius approach [56] addresses the problem by defining a virtually single underlying model that in turn consists of a set of models, which can only be modified through well-defined views. Internally, the redundant information of the models is kept consistent by an explicit consistency mechanism. Therefore, the developer has to define the relations between different models that have to be kept consistent by updating related models whenever one gets changed.

Such model relations and their consistency preservation can be expressed through model transformations. The Vitruvius approach proposes a transformation language family, which provides different transformation languages for preserving model consistency. One

of these is the mapping language. It allows the specification of relationships between models, which are automatically transformed into program code that ensures the satisfaction of these relations. Another language is the response language, which is intended to define imperative repair routines for relations between models that get executed whenever a model is changed.

1.3 The Response Language

Several model transformation languages already exist and can be used for preserving model consistency. Nevertheless, transformation languages can be used for different purposes, such as model analysis, model validation or code generation, rather than only for ensuring model consistency. Consequently, the specialization of a transformation language for preserving consistency can take advantage of the characteristics of this special domain. Apart from that, transformation languages mostly assume a completely automated execution of the specified transformations. However, model consistency preservation sometimes requires further information or even actions from the model user.

Most existing approaches for specialized consistency preservation languages have two characteristics. First, they rely on a formal definition of model consistency. This makes it easier to provide appropriate language statements and to reason that they are sufficient for ensuring consistency. Nevertheless, it complicates the specification of consistency constraints for a developer. Second, they provide a highly declarative transformation specification, which allows to describe the relations between two models from which certain consistency-restoring transformations for both directions are automatically derived. This restricts the possible influence of the developer on the consistency repair because he can only specify constraints for consistency but not how they are restored.

In this thesis, we develop the structure and concepts for a language, called the *response language*, which relaxes these characteristics of existing model consistency languages. First, we propose a consistency definition that is more comprehensible for developers who specify consistency relationships. Second, we define the language constructs in a way that they always allow the specification of imperative Turing-complete code for providing maximum expressiveness. At the same time, we tailor the language constructs to the needs of consistency-restoring transformations. Based on the insight that models can only get inconsistent if one of them gets changed, the developed language triggers transformations in reaction to changes.

In addition to the concepts, we develop the integration of the concepts into an exemplary language. We also provide a prototypical reference implementation of this language in the context of the Vitruvius project.

This thesis answers the following research questions:

Q1. Which changes can occur within models and how can they be characterized and categorized?

Q2. Which operation steps perform consistency-preserving, change-driven model transformations and what is their purpose?

Q3. How can a transformation language be designed which

(i) defines transformations that react to model changes,

(ii) provides language constructs for the steps identified by *Q2*,

(iii) and remains maximum expressive in terms of being Turing-complete?

1.4 Structure of the Thesis

This thesis is subdivided into an introduction of foundations and a running example in chapter 2 and 3, the presentation of our approach in chapter 4 and 5, its prototypical implementation and evaluation in chapter 6 and 7, and finally the comparison with related work and a conclusion in chapter 8 and 9. In particular, the individual chapters present the following:

Chapter 2 introduces the foundations of our approach and of its implementation. It gives an overview of different concepts of model-driven software development and its subtopic of view-based software development, as well as a practical realization of the concepts in the Eclipse Modeling Framework.

Chapter 3 defines our running example scenario, the consistency of software architecture descriptions and code models.

In *chapter 4*, model changes and model consistency are introduced. We first provide our definition of model consistency and model changes and explain the requirements to a change-driven model transformation environment. Afterwards, change-driven model consistency repair, the core concept of this thesis, and a structure for consistency-restoring transformations are discussed.

Chapter 5 introduces the response language. It first derives a language structure from the general structure of consistency-restoring transformations. Thereafter, the purpose of each part and necessary language constructs for these parts are argued. We develop an extension for the basic language structure, which improves the reusability of consistency-restoring transformations. After considering further properties and responsibilities of the language and its usage, possible language extensions are presented.

In *chapter 6*, the prototypical implementation of the response language structure and concepts in the context of the Vitruvius approach is outlined. After an introduction to some Vitruvius specifics, the runtime environment and structure of the final transformations is explained. Afterwards, some aspects of the language specification and code generation are discussed.

Chapter 7 explains the evaluation of our approach. First, the completeness of our approach in terms of fulfilling the requirements to its expressiveness, defined in research question *Q3*, is validated. Subsequently, the applicability is shown in a case study, which also compares the approach to manually written transformations. Finally, some considerations regarding the evolvability of transformations are provided.

Chapter 8 gives an overview of related work and exposes the differences of our approach compared to existing ones.

Finally, *chapter 9* closes the thesis with a conclusion, which summarizes the results of our work and gives an overview of possible future work.

2 Foundations

This chapter provides an overview of the foundations on which this thesis is based. Initially, the idea of model-driven software-development and its terminology is introduced, before model transformations, its properties and characteristics are discussed in more detail. After giving the introduction to a concrete modeling framework in the Eclipse environment and some important tools for it, the idea of view-based software development and different approaches in this context are explained. The chapter ends with an overview of change-driven development techniques in general and their relation to model-driven development.

2.1 Model-Driven Software Development

Model-driven software development (MDSD) is a generic term for techniques that automatically generate executable software from formal models [89, p. 11]. It is also referred to as model-driven engineering (MDE) or short model-driven development (MDD). The central idea is to raise the level of abstraction of software development from program code to models, which are automatically transformed into executable programs. MDSD aims to increase the per-developer productivity, as well as the quality and reusability of software components [89].

Models are commonly used in software development to describe certain aspects of a software system. While the final artifact of a software development process is usually the program code or its compilation, many other models are used to describe, for example, requirements or the architecture of a system. An example for commonly used models in the development process are UML diagrams [75]. By abstracting from the implementation, models reduce the complexity of the artifact to deal with because they only consider a certain aspect or extract of the system instead of the whole one.

More precisely, a model is a representation of objects and their relations restricted to the needs of a special use case. Several established model definitions have been proposed, whereof the most suitable in the context of software development is given by Stachowiak [88]. According to him, a model is characterized by at least three properties. A model is a *mapping* of some kind of original, it is a *reduction* of the original, as it does not provide each attribute of the original, and it has a *pragmatism*, as it is designed for a certain context and cannot necessarily be used in other contexts. Furthermore, Stachowiak distinguishes between descriptive and prescriptive models. While descriptive models represent objects how they actually are and work, prescriptive ones prescribe how objects are intended to be.

Because models provide different abstractions of a system, they usually contain redundant or dependent information. For example, the information of a UML class diagram is almost completely contained in the program code as well. Consequently, the evolution of a

model requires other models to updated consistently. The propagation of changes from one model to another that contains redundant or dependent information and back is referred to as round-trip engineering [46]. Without appropriate tools, the consistency has to be ensured by the developer, who needs to have the knowledge about all redundancies and dependencies between the different models within a development process of a software system.

In MDSD, *metamodels* are of special interest. They define how valid models have to be created [5] and are in turn models themselves. Models that are conform to different metamodels are referred to as *heterogeneous models*. In addition to models and metamodels, a central artifact in MDSD are *model transformations*. While models completely describe a certain aspect of a software system, transformations specify how these models can be transformed into each other. Consequently, transformations define the relations between models and can be used to achieve consistency of different models. Model transformations are introduced in detail in section 2.2.

2.1.1 Metamodels

A metamodel specifies how valid models, which are referred to as instances of the metamodel, have to look like. According to Stahl et al., a metamodel consists of four artifacts, which are the abstract syntax, the static and dynamic semantics, and concrete syntaxes [89, pp. 28]. Atkinson and Kühne describe the static semantics as well-formedness requirements and the dynamic semantics simply as semantics [5].

The most essential component of a metamodel is the *abstract syntax*, which represents the available model elements and relations between them. It defines the structure of documents that are written in that language and at the same time the structure in which such a document is internally represented, analyzed and persisted. A parser creates the representation of a document according to an abstract syntax, which is the basis for machine processing of the contents, such as code generation.

The *static semantics* specify well-formedness constraints that a valid model must fulfill [5]. Harel and Rumpe call these semantics *context conditions*, which are checkable conditions that reduce the set of valid metamodel instances in addition to the restrictions of the abstract syntax [42]. These constraints can, for example, be specified using the Object Constraint Language (OCL) [74].

One or more *concrete syntaxes* specify how a metamodel instance can be represented. Examples for different forms of concrete syntaxes are textual, graphical or tree-based representations. These syntaxes are essential for the human interaction with the language and for the persistence of models.

The last component of a metamodel is its *dynamic semantics*, which defines the meanings of the language elements, their relationships and representations. The semantics of a language can be defined in free text or in more formal ways. One example are transformational semantics [78], which allow the definition of semantics by specifying the transformation into another language that already has defined semantics. An example is the specification of a transformation from a developed language into Java code, which already has a defined semantics. In this thesis, we describe the dynamic semantics of a

metamodel in free text, whereas the implementation of our response language defines its dynamic semantics through the transformation into executable Java code.

We refer to the elements of a metamodel as *metaclasses*. If it is clear that elements of a metamodel instead of a model are meant, we also simply refer to them as *model elements*.

2.1.2 Meta-Levels

A concrete model is an instance of a metamodel, thus a set of objects and relations satisfying the abstract syntax and semantics. In fact, each model has a metamodel it conforms to, and because a metamodel has a metamodel itself, that model is called the meta-metamodel. The arising cascade of metamodels can be bound by a metamodel that is self-defining, which means that it is its own metamodel. These hierarchies can consists of a theoretically arbitrary number of so called *meta-levels* or *meta-layers*.

An example for a self-describing metamodel is the Meta Object Facility (MOF) [73], defined by the Object Management Group (OMG). The MOF is, for example, used for specifying the Unified Modeling Language (UML) [75]. The UML uses a four-layer hierarchy with the levels M3 to M0, in which the MOF is specified in the upper level, called M3. The next level, called M2, contains the UML specification that relies on the MOF as its metamodel. M1 contains instances of the UML specification, thus concrete UML diagrams. Finally, M0 covers the real objects, which are descried by the models on M1 level. The MOF has a subset called Essential MOF (EMOF), which are introduced in subsection 2.3.1.

In a general four-layer hierarchy, models are placed on M1, their metamodels are located on M2 and have a self-describing meta-metamodel on M3. Because the models that we use in this thesis rely on the MOF and on a four-layer hierarchy of models, we always refer to the self-describing metamodel of our model hierarchy as the meta-metamodel.

2.1.3 Domain-Specific Languages

A *domain-specific language (DSL)* is “a computer programming language of limited expressiveness focused on a particular domain” [35]. In contrast to a general-purpose language (GPL), which is designed to implement arbitrary programs in any domain, a DSL is designed for a special domain. Such a language provides constructs that provide a more compact or convenient way to implement scenarios that occur recurrently in that domain. However, it is possibly much more difficult or even impossible to implement scenarios that are not envisaged by the language.

Certainly, the given definition of a DSL does not provide a way of definitely classifying a language as DSL or GPL. There is even no common understanding in literature of what a DSL is. The main characteristic of a DSL is its purpose. According to Krahn [53], a language can be considered as a DSL if it is based on the semantic aspects of a special domain, if it allows to solve problems of the domain in a compact way, and if its expressiveness is constrained to the needs of the domain.

In the end, a DSL can be considered as a metamodel because it consists of the same components as a metamodel does. It comprises an abstract syntax, a static semantics, one or sometimes more concrete syntaxes and a defined semantics. In literature, there is no clear separation between those two terms. To distinguish them in this thesis, the term

metamodel is used if the focus is the structural representation of data and its behavior with little importance of the concrete syntax. It is called DSL if the focus is the specification of a new language for the convenient specification of domain-specific concepts, where the concrete syntax is very important due to user interaction. Thus, a DSL is tightly coupled to its concrete syntax, whereas a metamodel focuses on the concepts and semantics.

DSLs can be separated into *internal* and *external* DSLs [35]. Internal DSLs are embedded into another language and reuse concepts of that host language. They finally provide a "kind of API" [35, ch. 6]. Consequently, the implementation is generally easier in contrast to an external DSL, but the language is constrained to the constructs the host language provides. An example for an internal DSL are UML Profiles, a mechanism of the UML [75] that allows the definition and dynamic application of extensions to the UML language core. The benefit of an internal DSL, apart from the generally easier implementation, is the easier testing and debugging because existing concepts and tools of the host language can be reused. Additionally, complete language constructs of the host language can be reused. For example, if a DSL requires the availability of an imperative Turing-complete programming language, extending an existing programming language is potentially easier than the reimplementation of such a language.

An external DSL is a completely independent language that has to define its own abstract syntax and semantics and thus potentially requires higher effort for its implementation. External DSLs are usually more expressive and flexible than internal DSLs are. They benefit from their complete independence from any existing language and concept. This independence allows to fit them precisely to the requirements of the language and makes them independent of restrictions of a host language. Nevertheless, it cannot benefit from reusing concepts provided by a host language.

In this thesis, external as well as internal DSLs are used. While the developed language generally defines an external DSL, it also reuses an existing programming language and extends it with an internal DSL.

The development of DSLs is facilitated by language workbenches [35]. A language workbench provides a set of tools for developing especially external DSLs. These tools can, for example, cover the provision of parser generators for a specific grammar specification language or a generator for editors for the DSL with features such as code-completion and syntax highlighting. Some language workbenches even supply automated code generators that just need a specification of the relation between constructs of the DSL and constructs of the programming language they generate code for. The language workbench Xtext (see subsection 2.3.2) allows this kind of code generation for Java.

2.2 Model Transformations

Model transformations specify how models can be converted into other models. In general, a model transformation gets a set of models as its input and provides a set of models as its output. Nonetheless, we only consider transformations with a single input and output model. A formalization of model transformations is given in by Amelunxen and Schürr [1].

Transformations can be classified as model-to-model or model-to-code transformations. The former ones define conversions of a model into another one, which can be an instance

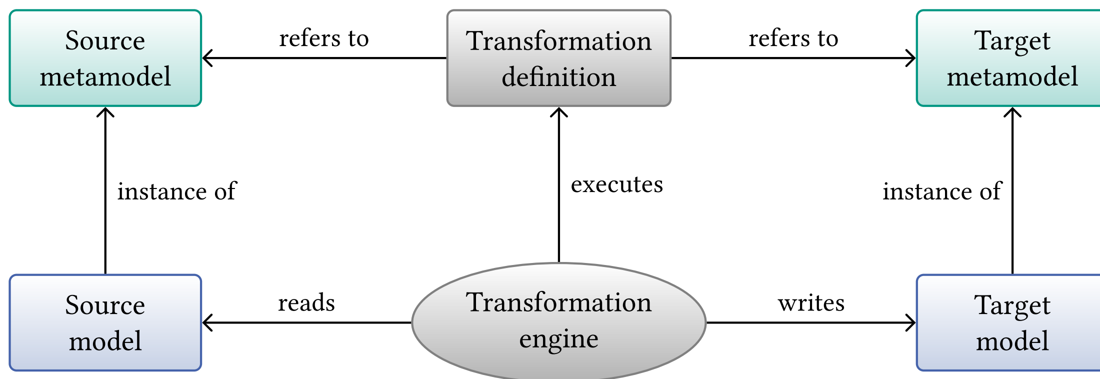


Figure 2.1: Basic concepts of model transformations, adapted from [21]

of a new or the same metamodel. The latter ones specify how a model can be transformed into program code, such as Java. Because code can also be treated as a model and is treated as such in this thesis, we do not distinguish between those types of transformations. We consider model transformations always as being model-to-model.

The basic concepts of a model transformation are visualized in Figure 2.1. A transformation definition refers to a source and a target metamodel and specifies how an instance of the source metamodel shall be transformed into an instance of the target metamodel. To execute a transformation, a transformation engine is required. It transforms the transformation definition into executable code and runs this code on a pair of models that are instances of the source and target metamodels of the transformation. In this basic concept, a clear specification of the source and target model is assumed, so that one model is only read, while the other is written to.

Because model transformations are of central significance in MDSD, several languages and tools for specifying them have been developed. Such a language usually specifies a DSL to write transformations in and provides a transformation engine that interprets the transformations written in the DSL and executes them. A popular example in the context of the MOF is the Query/View/Transformation (QVT) standard [72], which is also defined by the OMG. Based on it, several model transformation languages have been defined. Examples are QVT-Operational (QVT-O), an imperative language, and QVT-Relations (QVT-R), a descriptive, relational language. An example for a hybrid approach that allows the specification of transformation using imperative and declarative constructs is the Atlas Transformation Language (ATL) [52].

In a survey about transformation approaches, Czarnecki and Helsen provide an overview of the characteristics of model transformations [21]. Some of these characteristics, which are relevant for this thesis, are introduced in the following.

2.2.1 General Properties of Transformations

One property of model transformations is the source-target relationship. While some approaches assume the target model to be different from the source model, other approaches use the same model as the source and target of a transformation. If the source and target model of a transformation are the same, the transformation is characterized as *in-place*.

Another important property of a transformation is the directionality. Model transformations between pairs of models can either be defined *unidirectional* or *bidirectional*. A unidirectional transformation defines how in instance of one metamodel can be transformed into an instance of the same or another metamodel. If instances of both metamodels can be transformed into each other, transformations for both directions have to be defined in a unidirectional approach. To avoid the duplicate specification of mappings between model elements, bidirectional approaches provide a way to specify the relationship between elements of different models, from which transformations in both directions are derived.

2.2.2 Transformation Execution Modes

The simplest execution mode of model transformations is the *batch mode*. Batch transformations take an input model and completely derive a new target model by applying the transformation rules. The re-execution of a batch transformation delivers a completely new model, which overrides information that was potentially added to a previously generated model.

Incremental transformations reuse the model that was generated through a previous transformation and updates it according to the transformation definitions. Therefore, the transformation has to identify the modifications in the source model and add, remove or modify elements according to the transformation rules.

Incrementality can furthermore be subdivided into *source-incrementality* and *target-incrementality* [21]. The intuitive kind of incrementality is the target-incrementality, which concerns about updating the target model elements instead of generating a new target model whenever the source model changes. The source-incrementality of a transformation states how much of the source model has to be re-investigated by a transformation after a modification. Source-incrementality can be improved by more information about the modification. Specific information about the concrete change, such as the modified elements, reduces the extract of the source model to be considered by the transformation.

To update elements that were created by a previous transformation execution, they must be retrievable during a re-execution of the transformation. Therefore, some kind of trace information has to be provided, from which the elements that a previous transformation execution created can be retrieved. A common approach for providing such tracing information is the creation of links between source and target elements of a transformation rule and the assignment of these links to the transformation rule they were created in. Languages like QVT-O [72] create these trace links automatically. Nevertheless, other approaches, which even abstract from the transformation rules in which the elements were created, are possible and are used in this thesis.

Incrementality is not a property that a transformation can have or not, but it is a continuous property whose significance depends on the amount of modifications that are performed whenever the transformation is executed. Highly incremental transformations only consider a small set of elements after a model change and, therefore, only re-execute few transformations. The other extreme is the batch mode, which has no incrementality.

To achieve incrementality, the differences between the lastly transformed and the actual state of the source model have to be identified. This can be achieved by providing both

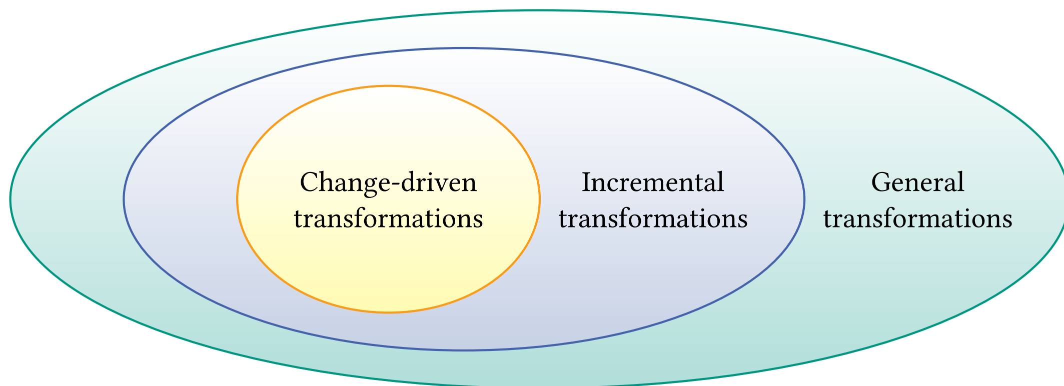


Figure 2.2: Execution modes of model transformations

states to the transformation and let it compare them to identify the modified parts. This kind of change detection is called *difference-based* or *state-based*. If modifications to the model are monitored and provided to the transformation in addition to the models, the change detection is called *delta-based*. Transformations that rely on a delta-based change detection are called *change-driven* [10] and inhibit a high degree of source-incrementality.

We distinguish between general transformations, incremental transformations and change-driven transformations. The relations between these execution modes are shown in Figure 2.2.

2.2.3 Transformation Specification Approaches

The most simple approach to write transformations is the direct model manipulation. In this approach, only the models are provided, and transformation rules, the transformation execution and tracing have to be implemented in imperative code manually. Transformation languages usually attempt to abstract from the different facilities of transformations and provide different kinds of transformation specifications. They can either be defined with *imperative* statements, with *declarative* rules or with a combination of both.

In imperative approaches, a transformation rule specifies the steps that have to be performed when it gets executed. For example, it specifies which new values or references have to be assigned and which elements are created or removed. Furthermore, the execution order of the statements is usually specified. Imperative transformation rules can be executed without much further effort.

Declarative approaches abstract from the specification of the control flow and from the specification of concrete operations that have to be performed. A declarative transformation rule does only define constraints that have to hold after its execution instead of specifying what has to be done during its execution. The transformation language generates executable code from these constraints, which performs model modifications to achieve the satisfaction of the specified constraints. This process is referred to as *operationalization*. Consequently, declarative approaches reach a higher degree of abstraction than imperative approaches usually do. Bidirectional transformations are usually realized through declarative approaches because an imperative specification always describes how to come from one model to the other without considering the way back. Nevertheless, in

contrast to Turing-complete, imperative statements, declarative rules have a limited expressiveness. For example, complex relationships between attributes cannot be expressed as their calculation cannot be derived for both directions.

Czarnecki and Helsen further distinguish the approaches into, among others, relational, graph-based and operational approaches [21]. Relational approaches, such as QVT-R [72], require the specification of relations between elements in the source and target metamodels, which are automatically ensured in model instances by the transformation engine. Graph-based approaches treat the models as graphs and define graph patterns, which are used to find relevant structures in the model graphs by matching a pattern and replacing it with another one. Operational approaches are closest to direct manipulation approaches as they still define operation that have to be performed on the models, but usually provide no imperative, Turing-complete language but a restricted query language, like in QVT-O [72].

In this thesis, we combine declarative statements to abstract from certain aspects of model transformations and imperative statements to provide a maximum, Turing-complete expressiveness.

2.3 The Eclipse Modeling Framework

Eclipse is a "community of tools, projects and collaborative working groups" [95]. It consists of several projects in the context of software development, which rely on the Eclipse framework. This framework provides a rudimentary core application that can be extended by plugins according to the Hollywood principle "Don't call us, we'll call you" [61]. A popular set of tools for the software development with Java is provided in the Eclipse Integrated Development Environment (IDE) for Java Developers [94].

The Eclipse Modeling Framework (EMF) is a project extending the Eclipse platform with modeling capabilities [90]. It provides concepts and tools for defining models based on a meta-metamodel called Ecore, which is further explained in subsection 2.3.1. The purpose of EMF is the unification and integration of models into the software development process as primary artifacts [90, p. 15].

EMF provides graphical tree-based editors for creating and modifying metamodels based on Ecore. Based on these metamodels, Eclipse plugins that provide unified graphical tree-based editors for the instances of these metamodels can be generated. Furthermore, EMF provides a code generation for Ecore based metamodels. This code generation transforms the metamodels into Java code, which represents the metaclasses as Java classes and the relations between metaclasses as object references. The generic model element interface, which all available and generated model elements implement, is the `EObject`. Certainly, the code generation is more elaborate and, for example, hides the model instantiation through a factory implementation for a better exchangeability of the metamodel realization, but those aspects are not relevant for this thesis.

Several tools have been developed for or based on EMF and especially the contained Ecore meta-metamodel. They cover different aspects of model-driven software development, such as the comparison and validation of models. Several model transformation languages are implemented for EMF, for example, QVT [72], ATL [52], and the more Java-

aligned Xtend, which is explained in subsection 2.3.3. The development of DSLs and textual editors for them can be conducted with Xtext, which is introduced in subsection 2.3.2.

2.3.1 Essential Meta-Object Facility and Ecore

The Essential Meta Object Facility (EMOF) is a self-describing meta-metamodel and is part of the MOF [73], defined by the OMG. It is contained within the Complete MOF (CMOF), which provides the full capability of the MOF standard. EMOF is "designed to match the capabilities of object oriented programming languages" [73, p. 3] and thus provides a reduced meta-metamodel in contrast to CMOF.

The EMOF meta-metamodel defines types, properties and operations. Types can in turn be data types or classes, properties belong to classes and reference other classes or data types, and operations belong to classes. Obviously, the EMOF elements are aligned with object-oriented programming languages, which is due to the intended usage of EMOF in that context.

Ecore is the meta-metamodel that ships with EMF. It is conform to EMOF, which means that Ecore-based models can be converted into EMOF-based models without information loss. One relevant difference between EMOF and Ecore is the distinction of properties into attributes and references in Ecore. In contrast to EMOF, Ecore has a concrete implementation in EMF and can be seen as a reference implementation of EMOF.

The MOF, and as a result also EMOF, use the XML Metadata Interchange (XMI) format [76] for persisting model data. XMI specifies a generic, XML-based format for persisting and interchanging models. EMF uses XMI for persisting Ecore-based models as well, which allows all EMF tools to rely on this persistence format.

2.3.2 Textual Domain-Specific Languages with Xtext

Xtext is an EMF-based framework that supports the implementation of textual DSLs [24] and is developed by the itemis AG. The specification of a DSL in Xtext consists of three parts. The first part is the specification of the abstract and concrete syntax of the language in a special editor. Secondly, Xtext provides different extension points, based on the abstract syntax, for specifying the static semantics of the language. The third part is the specification of a transformation of the abstract syntax into another model with defined semantics for determining the dynamic semantics of the language.

The Xtext framework provides an editor in which a concrete textual syntax for the language can be specified in an EBNF-like notation. From this specification, it automatically generates an EMF-based metamodel that represents the abstract syntax of the language, a parser that extracts the abstract syntax from a textual model representation according to the specified concrete syntax, and an editor with syntax highlighting, code completion, error checking and further features improving its usability. The editor also allows to specify some static semantics of the language, for example, through references to other elements of the language.

From the syntax specification of the language, an Ecore-based metamodel and several classes for defining the validation, scoping, linking and other semantics are generated. These classes extend predefined implementations for the different aspects that can be

reused. As an example, namespace-aware scoping implementations are provided and ease the access management to variables based on namespaces, like in the Java programming language.

To define the dynamic semantics of the specified language, Xtext provides the possibility to implement a generator that takes a model according to the specified language and transforms it into another model. The mechanism aims to transform the model into textual program code that can be executed.

To ease the code generation for languages that are similar to Java code, Xtext also provides a mechanism to convert the model into a Java code model instead of a textual code representation. The transformation is automatically performed by Xtext and only requires the specification of mappings between the model elements and Java language constructs. This also allows to reuse scoping and validation mechanisms of the Java language. Mapping a language construct to a method in the Java code model automatically provides access to the defined method parameters and to fields of the containing class, and requires the return type to fit to the specified method return type.

All developed artifacts are automatically integrated by Xtext to a complete language setup, which performs the validation and code generation for given models and also the parsing for textual representations. Furthermore, Xtext generates an editor for the language, which evaluates scoping and validation rules while editing the document and performs the code generation whenever the document is saved. The language developer can implement further extensions for this editor, such as an outline specification that represents the document structure.

DSLs that are defined with Xtext can be reused in other Xtext-based DSLs because Xtext provides a mechanism for extending and using already existing languages. One language that is shipped with Xtext and is predestined for being reused is *Xbase*. Xbase is an expression language, related to Java, and thus can be used within a DSL for allowing the definition of imperative code without having to implement a language with correct syntax, validation and scoping. In the following, the Xbase language is explained in more detail.

2.3.3 Reusable Programming Languages Xbase and Xtend

Xbase is an expression language that is shipped with Xtext [25]. It is tightly integrated with the Java type system and can be inherited by Xtext-based DSLs to provide Java-like expressions within a DSL. Xbase provides a parser for generating a model for textual notations in that language, validation specifications for the expression language and a compiler that generates Java code for given Xbase expressions. The editors that are generated for Xtext languages also inherit the scoping specification from the Xbase language definition, which provides a Java-like variable scoping within Xbase expressions.

In addition to the concepts of Java, Xbase also provides several extensions. A type inference mechanism allows to declare a variable as `var` to make it changeable, or as `val` if it shall be final, omitting the concrete variable type. Operator overloading allows to redefine operators for specified classes. A method can be declared as *extension*, which attaches the method to the type of the first parameter of the method. Finally, a dynamic dispatching mechanism allows to define overloaded methods that are selected by the

```
val aList = new ArrayList<String>();
aList += "A string";
aList += #["A second string", "Another string"];
```

Listing 2.1: Type inference and operator overloading in Xbase

dynamic, instead of the static types of their parameters. Dynamic dispatch provides a simple way of implementing the visitor design pattern [36], but is realized through type comparison instead of efficient callbacks as proposed by the pattern. Finally, Xbase also provides a short notation for initializing read-only collections.

Xbase and some of its extensions are used in the implementation of the language that is developed in this thesis and thus in the given examples. Therefore, Listing 2.1 shows an example code snippet written with Xbase. The type inference mechanism is used for the variable `aList`, which gets an `ArrayList` assigned and is automatically typed correctly. Instead of using a method call for adding an element to the list, like it has to be done in Java code, Xbase overloads the += operator for the `List` interface and maps it to the add method. In the last line, a read-only collection is initialized and automatically typed correctly, based on the two strings that are defined in it.

Xtend [25, p. 7] is an application of the Xbase language and is developed with the Xtext framework. It is an object-oriented, statically typed programming language for the Java Virtual Machine (JVM) but claims to provide a more concise notation by omitting redundant information that has to be written using Java. In additional to the more compact syntax, Xtend provides some new features, for example, multi-methods that allow the dynamic dispatch of method parameters, type inference and template expressions. Xtend classes are automatically compiled to Java classes and thus can be used in a plain Java project.

Xtend integrates the language Xpand [23], which was developed for model-to-code transformations. Xpand allows to define template expressions that consist of static and dynamic text, whereas the dynamic text parts can be specified using a Turing-complete expression set. With this mechanism, Xtend provides a way of easily specifying model-to-code transformations. Therefore, we use Xtend as the programming and model transformation language for the implementation of the response language in this thesis.

2.4 View-based Software Development

The term *view-based software development* describes an approach for developing software based on views. This approach is of special interest in model-driven environments, which describe a software system through one or more models. Instead of modifying these models directly, a view-based approach allows the modification of the models only through views, which show the aspects of the system that are of interest for a certain developer role.

The terminology for view-based development approaches in software engineering ranges from *view-based development* [6], over *view-based model-driven software development* [18] to *view-centric engineering* [55]. Essentially, all cover the idea of modifying the different software artifacts or models through well-defined views.

In contrast to conventional software development, which makes use of different models that are modified directly, view-based development provides some advantages [18]. Views can be tied to the needs and permissions of a certain developer role and consequently allow to omit details that are not needed by or should not be visible for a certain developer role. Furthermore, working with views can abstract from the concrete underlying artifacts, so that is does not matter for the developer how the concrete artifacts look like. Views can also hide redundancies in the models that can lead to inconsistencies when only one part of the redundant information is modified. A view can provide the information only once and synchronize it with all redundant occurrences in the models.

2.4.1 Projective and Synthetic Views

According to the ISO 42010 standard [50], two approaches for the construction of views exist. The first one is the *projective approach*, which derives views from existing models. Those views are defined by describing how and which elements of existing models have to be presented in a view. Therefore, a projective view can be created through a model transformation from the models on which it is defined to the view.

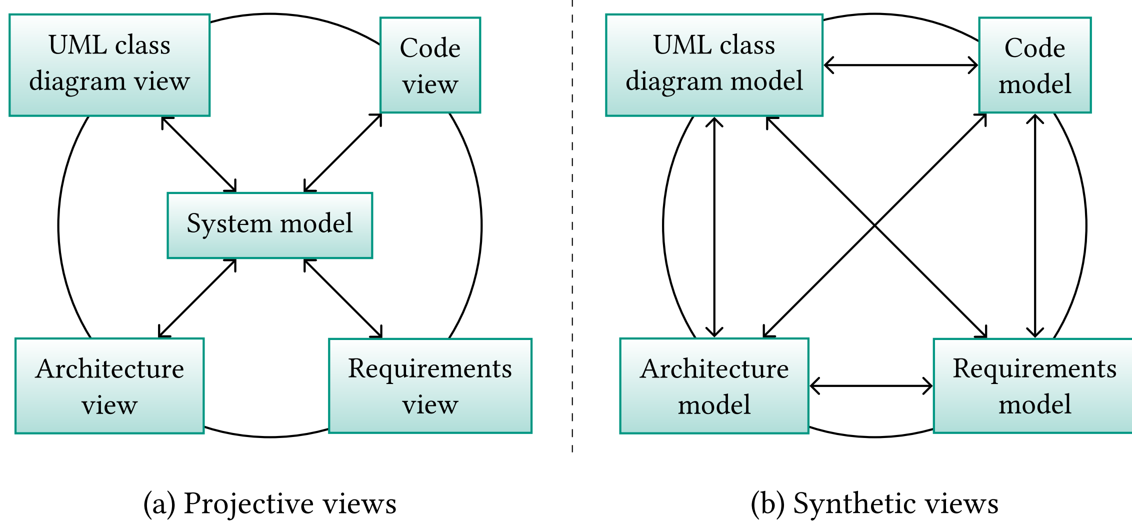
The second approach for view construction is the *synthetic approach*. In that approach, a developer defines views and relations between them, which together define the system model. For example, using existing models with their existing views, usually defined through editors, and defining transformations between them describes a synthetic approach for creating the system model from the single views.

An important aspect of views is the editability. Views can either be read-only and thus only provide information of the models, or editable, providing the possibility to modify the data. Synthetic approaches inherently require editable views as the views are also the models that must be somehow modified. Furthermore, views can provide different degrees of editability, depending on the kind of modifications they allow. Restricting the editability of views can be used to influence the modifications that are possible for a certain developer role [16].

The difference between synthetic and projective views in an exemplary system with four views is shown in Figure 2.3. On the left side, a system model contains all information about the software system, and different projective views, such as the code, UML class diagrams, the software architecture and requirements of the system, provide access to it. On the right side, different models for the different aspects define the complete system. In the projective approach, all views are only related to the system model and thus four relations have to be kept consistent. The synthetic approach potentially requires relations between all models, which implies six relations to be kept consistent in the example. In general, the number of relations that have to be kept consistent increases linearly in a projective approach and quadratic in a synthetic approach.

2.4.2 Orthographic Software Modeling

Atkinson, Stoll, and Bostan have developed a view-based development approach called *Orthographic Software Modeling (OSM)* [6]. The OSM approach relies on the idea of a so called *Single Underlying Model (SUM)*, which contains the whole information about a



(a) Projective views

(b) Synthetic views

Figure 2.3: Example for a system of projective views and synthetic views

software system. Access to the SUM is only provided through views that show certain aspects of the system.

The OSM realizes a projective approach for constructing views. Views are created on-demand whenever they are needed, but they are not persisted as the shown information is represented in the SUM. Each view must only be kept consistent with the SUM, but not with other views. While the minimal number of relations between the SUM and the views that have to be kept consistent leads to a low effort for ensuring consistency, the effort for defining a SUM is high. All aspects of a software system have to be represented in the SUM and must be provided by its metamodel, which can get quite complex.

The OSM approach defines two roles, the *developer* and the *methodologist*. While the developer uses views and manipulates the software system through them, the methodologist defines the views for the system.

2.4.3 The Vitruvius Approach

Based on the OSM approach by Atkinson, Stoll, and Bostan, another view-based development approach called Vitruvius was proposed by Kramer, Burger, and Langhammer [56]. Vitruvius reuses the idea of a SUM from which projective views are derived for providing access to the different aspects of the system.

A SUM can be hardly realized because of the complexity of a software system, which is hard to represent by a single model, and because the reuse of existing tools for the different aspects, such as the code editor, is difficult or even impossible. Therefore, Vitruvius proposes the idea of a Virtual SUM (VSUM). A VSUM is used like a SUM in the OSM approach but itself consists of different models that are kept consistent. This approach allows to reuse existing metamodels for the different purposes and tools which are available for them but also provides a consistent SUM rather than independent models.

Existing models and their views can be reused for modifying the different aspects of the software system. Furthermore, the language *ModelJoin* was designed to define

additional, so called flexible views on the heterogeneous models of which the VSUM consists [18, 17]. This allows to define views that represent information from different models within the VSUM. Particularly, information can be presented non-redundantly and can be automatically kept consistent with all their occurrences in different models automatically.

The SUM in the OSM approach is inherently consistent because the metamodel can be defined omitting redundancies and implicit dependencies that can lead to inconsistencies. Because Vitruvius reuses existing metamodels whose instances usually contain redundant or dependent information, the consistency of the models must be actively preserved. Therefore, Vitruvius follows a change-driven approach. The framework monitors the models contained in a VSUM for changes and uses these changes to trigger transformations that restore consistency between the changed and other models. Transformations have to be written for pairs of metamodels and have to provide a specific interface for being executed by the framework.

To better separate the specification of consistency constraints and consistency repair from the understanding of model transformations, Vitruvius provides the Mappings, Invariants and Responses (MIR) language family [54, 59, 58, 55]. The MIR language family is a change-driven approach for ensuring and repairing consistency constraints of models. Based on the constraints and operations defined with the languages, appropriate update routines are executed when the framework reports a modification of any model of the VSUM. This separates the knowledge about model transformations, which are contributed by transformation experts that develop the MIR languages, from the knowledge about consistency of the specific models, which can be provided as specifications in the MIR languages by domain experts.

The three languages of the MIR language family are referred to as the *mapping language*, the *invariant language* and the *response language*, or short *mappings*, *invariants* and *responses*. A single consistency specification in these language is called a *mapping*, an *invariant* or a *response*, respectively.

The mapping language of the MIR language family allows the definition of declarative mappings between elements of different models that constrain their relationship. The routines that restore these constrains after a change are automatically generated from the mappings. The invariant language allows the specification of invariants that must always hold. The developer can also enrich invariants with parameters that are bound to concrete values that are causal for the violation of the constraint. If an invariant is violated, it is the purpose of the response language to react to that in the future. It is supposed to allow the specification of imperative code routines that are executed whenever specified changes occur. In contrast to the mappings, which are supposed to provide a very compact and comprehensive specification of consistency relations, responses shall support the specification of Turing-complete logic to address possibly any consistency constraint. The design of the response language is the topic of this thesis.

Like in the OSM approach, two roles can be distinguished that interact with an instance of the Vitruvius framework, which are the methodologist and the developer. Their interaction with the framework and the different artifacts is shown in Figure 2.4. The methodologist specifies the consistency preservation through mappings, invariants and responses for specified metamodels. Afterwards, he runs the MIR code generator, which

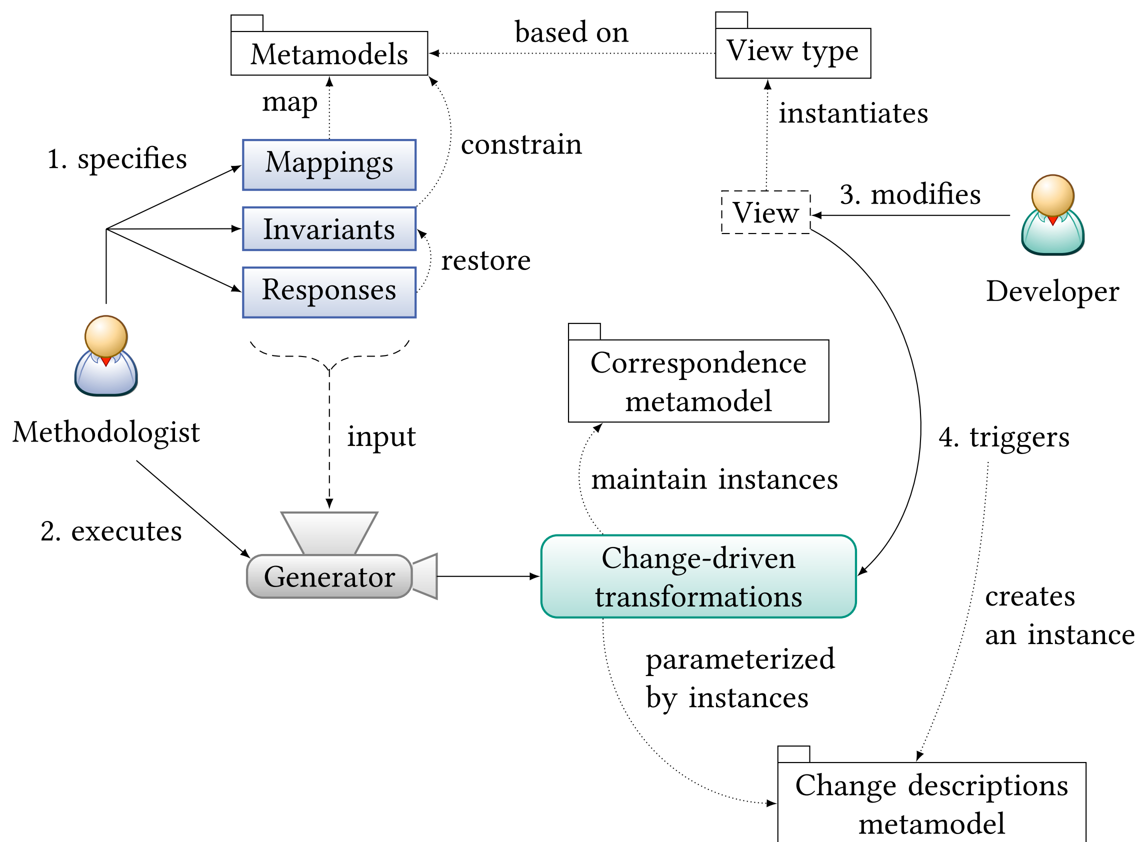


Figure 2.4: Different roles and their interaction with languages and artifacts for consistency preservation in the Vitruvius framework, adapted from [54]

produces executable change-driven transformations and registers them at the framework to react to appropriate model changes. The developer uses views to interact with concrete models in the VSUM of a Vitruvius instance. His interactions lead to changes that trigger the execution of the provided transformations. Whenever a change is performed, an instance of a change descriptions metamodel, which defines possible changes in the models, is created to parametrize the transformations. The transformations restore consistency constraints and update instances of a correspondence model, which describes the relations between elements of different models.

2.5 Change-Driven Development

Change-driven software development is a technique that focuses on the execution of program code whenever a specific change of data value occurs. Breu describes this principle in the context of changes in the development process [15]. Nevertheless, the statements can be transferred to the developed software.

In change-driven development, the occurrence of a data modification can trigger different kinds of reaction. In general, a particular piece of code is executed whenever a specific

change happens. This procedure is similar to the observer design pattern [36], which implements a notification mechanism in object-oriented software development. Change-driven development is more abstract than that because it does not specify a concrete realization of the mechanism and is not restricted to object-oriented software design.

Change-driven development can use two different evaluation mechanisms. *Push-based* systems use notification mechanisms that call other code whenever a change occurs. *Pull-based* systems check for changes at specified points of time and call the appropriate routines if necessary [7]. Change-driven development techniques usually use a push-based approach.

In object-oriented development, a similar approach called *event-driven programming* is popular [77]. It is a paradigm that focuses on events and handlers that are called whenever an event occurs. The actual difference to change-driven development is the application context. Event-based programming is often used for graphical user interfaces, where the interaction with the user interface produces events that lead to the execution of handlers. In that context, event-based systems are mostly pull-based. Except for that, there is no explicit distinction between change-driven and event-driven development. Finally, an event is potentially more generic than a change because it only requires a change of some kind of state, while a change is always also an event.

Reactive programming is another programming paradigm in this context [28]. Its purpose is similar to the event-driven approach but addresses some of its drawbacks. Reactive programming facilitates the declarative development of event-driven programs, rather than the classical imperative approach [7]. Only data dependencies have to be specified declaratively and the environment decides when and in which order changes have to be propagated. The advantage is that the developer must not explicitly ensure that update routines are called when a change occurs because the propagation is automatically generated from the data dependencies. Furthermore, event-driven programs often use callback mechanisms with limited possibilities of parameter passing, which leads to side-effects performed by the callback routines. These problems are tackled by the reactive programming approach.

Reactive programming may not be mixed up with the term *reactive system* of the Reactive Manifesto [13]. There, the term *reactive* focuses on the responsiveness of a system which the user perceives, rather than the control flow of the program execution, on which reactive programming focuses.

The presented concepts can also be applied to the context of model-driven development. Model modifications define changes that can be used to initiate arbitrary reactions. The reactions can reach from simple value changes to complete code routines and model transformations [10]. Finally, model changes can be used to trigger model transformations, similar to state changes triggering code routines in change-driven software development. Reactive programming focuses on the declarative description of data dependencies, thus knowledge from this topic can especially be transferred to declarative model transformations.

3 Running Example: Consistency of Software Architecture and Implementation

In this thesis, we develop the concepts for a change-driven transformation language for restoring model consistency. The concepts are illustrated in example routines for restoring consistency, which are written in an exemplary concrete syntax for the language.

The presented examples, as well as the final evaluation of the work, rely on a case study of consistency between software architecture descriptions and the object-oriented code model that describes their implementation. A software architecture and the implementing code contain elements and features that are represented in both models and have to be kept consistent whenever they are changed. Because the prototypical implementation of the developed language relies on the Eclipse Modeling Framework, the architecture descriptions and the code models are based on the Ecore meta-metamodel.

This chapter first introduces the Palladio Component Model (PCM), which is an architecture description language for software systems, and the Java Model Parser and Printer (JaMoPP), which provides a metamodel for the Java programming language. Afterwards, consistency relations between a PCM model and Java code are defined, which can be restored with change-driven transformation languages such as the one developed in this thesis.

3.1 A Software Architecture Description Metamodel

The Palladio Component Model (PCM) [8, 41] is a DSL for specifying component-based software architectures and their deployment on physical resources. It targets the prediction of certain quality aspects, especially the performance, of a software system before it gets implemented. For this purpose, different extensions are available, which allow the simulation of different design decisions to find a software architecture that fulfills the required quality constraints.

A PCM model of a software system can be used to automatically generate code stubs for the implementation of the system and, furthermore, can be used as a more abstract representation of the software system during its development. This allows to still analyze quality aspects of the system when design decision may need to be changed during the implementation. Consequently, the architecture model and the code have to be kept consistent.

Apart from the advanced capabilities of the PCM for simulating the system behavior, the core metamodel provides an architecture description language (ADL) that can be

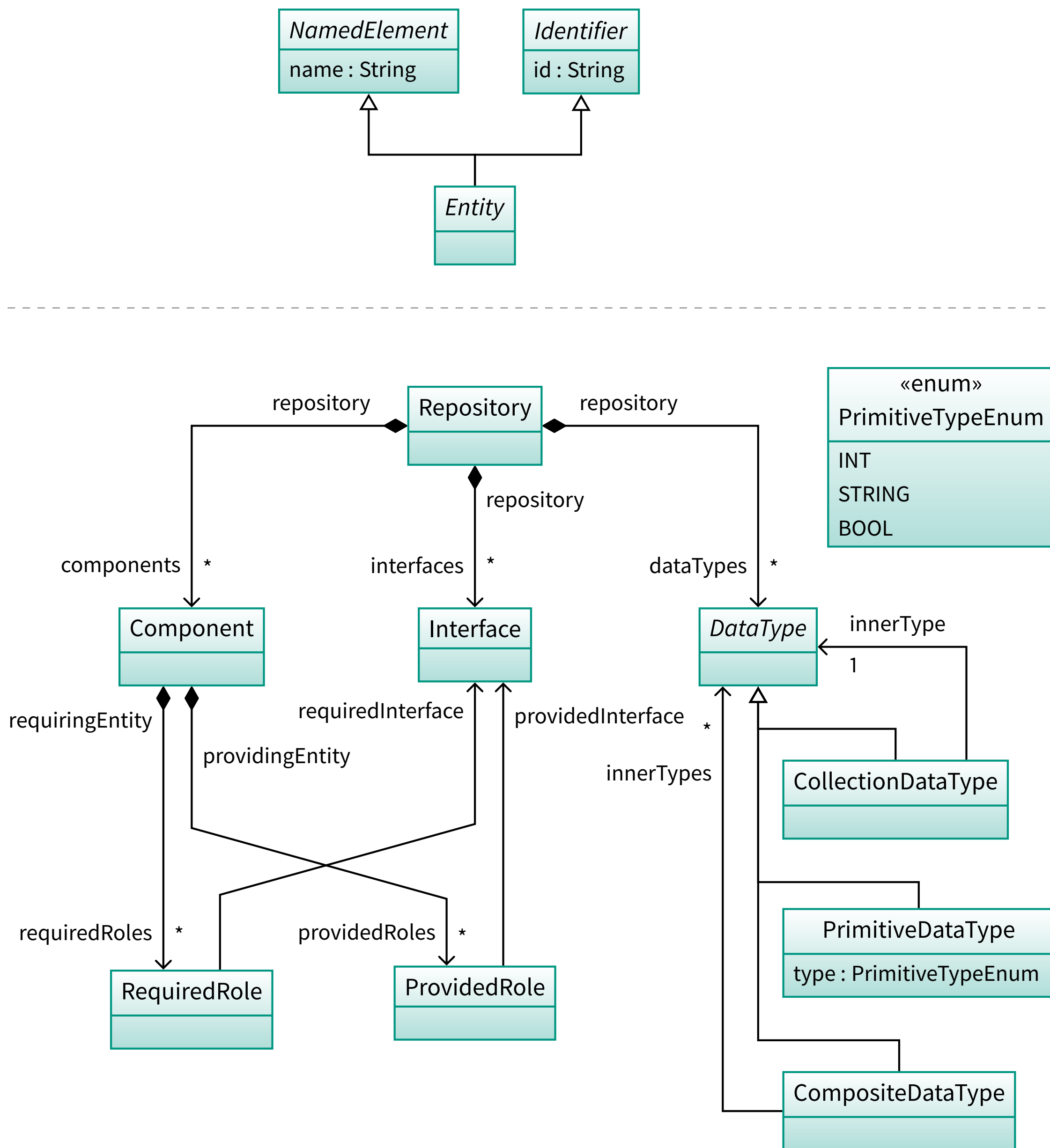


Figure 3.1: Simplified extract of the PCM metamodel

used to describe the software architecture on a more abstract level than code. The PCM relies on the Eclipse Modeling Framework and specifies its metamodel based on the Ecore meta-metamodel. The complete metamodel is discussed in [83]. For the evaluation of this thesis, we use a real subset of the PCM and map it to Java code. The examples in this these use a simplified version that is not completely conform to the PCM to ease its understanding. That metamodel is shown in Figure 3.1.

The diagram is split into two parts. The upper part shows the realization of the Entity element. This element is implemented by all elements in the lower part, except for the DataType and PrimitiveDataType. This implementation relation is not shown in the diagram for reasons of clarity. The implementing elements provide a name and an id attribute.

The important elements for the examples in this thesis are shown in the lower part of the graphics. A Repository defines the root element of a PCM repository model, which contains Components, Interfaces and DataTypes of which a software system can consist. A DataType is again subdivided into three types. A CollectionDataType represents a collection of elements of the type that it references, a PrimitiveDataType represents primitive values such as numbers and strings, and a CompositeDataType consists of other data types.

Interfaces in the complete PCM consist of signatures and define which functionality is provided by anything that implements this interface. Components can provide and require interfaces. They provide functionality that can be accessed through a provided interface, and they require entities that provide the required interfaces and implement the functionality which is expected from the signatures of the interfaces. These relations are designed using the RequiredRole and ProvidedRole metaclasses. A component can contain an arbitrary number of provided or required roles, which again reference the interfaces that are provided or required.

As mentioned before, the presented metamodel is a highly simplified version of the PCM metamodel. For example, the original one distinguishes between different kinds of components and different kinds of interfaces, supports more primitive types and encapsulates the inner types of a CompositeDataType into further objects that attach names to the inner types.

3.2 A Metamodel for the Java Programming Language

One of the most common object-oriented programming languages is Java. As the language that is developed in this thesis is implemented using the Java-based Eclipse Modeling Framework, we use Java as the example language for the object-oriented code model. Java is supplied in a standard and an enterprise edition, whereof we only consider the standard edition.

Object-oriented program code is traditionally written and represented in a textual syntax. Nevertheless, program code follows an abstract syntax and has defined semantics that are both specified in the language specification, as in [39] for the Java programming language. Consequently, program code can also be considered as the textual representation of a model that is conform to a metamodel which specifies the programming language. Parsers of compilers implicitly have to transform the textual representation of a program into an abstract representation, the abstract syntax tree, which represents the program code in an abstract syntax.

An Ecore-based metamodel for the Java programming language is supplied by the Java Model Parser and Printer (JaMoPP) [44, 43]. The provided metamodel is conform to the language specification of Java in version 5. Additionally, JaMoPP provides a parser, which takes a textual representation of Java code and returns the model representation according to its metamodel, and a printer, which returns the textual representation of a Java code model. Finally, JaMoPP allows us to treat a conventional textual Java code representation just as any other Ecore-based model.

Because the JaMoPP metamodel specifies the complete Java 5 language, the metamodel is quite large. For the examples in this thesis, we only need a small extract, which is shown

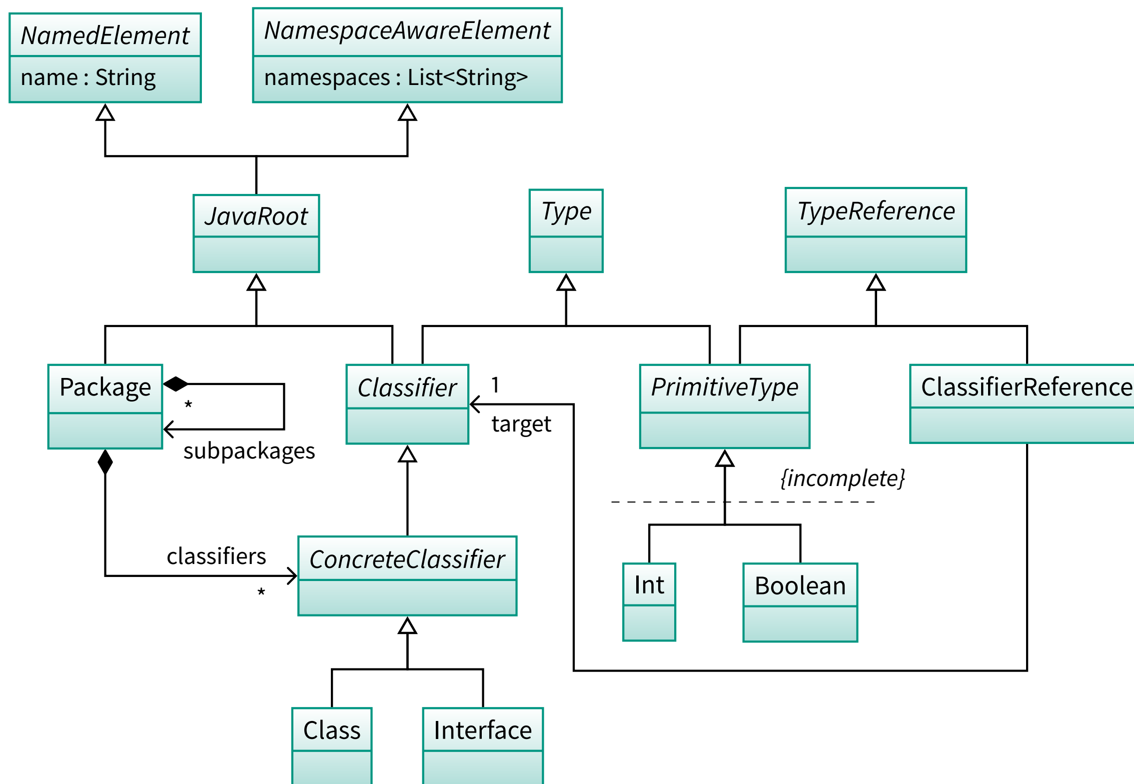


Figure 3.2: Simplified extract of the JaMoPP metamodel

in Figure 3.2. To simplify the examples, the extract is not fully conform to the JaMoPP metamodel. Different from our metamodel, a Package in JaMoPP contains compilation units instead of classifiers, which in turn contain classifiers. We ignore this technically necessary indirection and omit a CompilationUnit metaclass. Furthermore, the original metamodel does not specify a reference of a package to the packages contained in it, which we introduce with the subpackages reference to achieve transformations that are easier to understand.

The most important elements of the Java metamodel are the Package, Class and Interface metaclasses. A package contains classes and interfaces, summarized as ConcreteClassifers. They all represent JavaRoot elements, which have a name and know their namespaces. The namespaces attribute of a NamespaceAwareElement represents the hierarchy of package names which a classifier or package belongs to. In the JaMoPP metamodel, a package does not reference the packages it contains or the one it is contained in, as well as classifiers do not know the package in which they are contained. This information can only be extracted from the namespaces attribute, which makes code models difficult to use because information is spread across different models. It is not possible to navigate from a class to its package within a JaMoPP code model.

Additionally to classifiers, the metamodel provides the PrimitiveTypes known from the Java language specification, exemplarily represented by Int and Boolean. The remaining metaclasses, which are Type, TypeReference and ClassifierReference, are discussed later in a scenario in which they are relevant.

3.3 Relations between Software Architecture Descriptions and Code Models

As stated in the beginning of this chapter, a software architecture description and the implementing program code are artifacts of a software system that contain overlapping information. This information has to be kept consistent. A concrete consistency concept is first explained in section 4.1. Nevertheless, in this section we give an informal introduction to the relations between elements of an architecture description and the implementing Java code, which represent overlapping information and have to be kept consistent.

No natural mapping of architecture descriptions to program code of a software system exists. The mapping of elements from one of the models to the other has to be prescribed, which can be realized in different ways. For example, components can be either mapped to Java packages or to Eclipse plugins. Some of them are explained by Langhammer and Krogmann [60].

An extract of the initial mappings proposed by Langhammer and Krogmann, which is sufficient for the simple examples we provide, is introduced in the following. It is the basis for the example transformations in this thesis.

3.3.1 Repository Mapping

A repository of a PCM model is mapped to three Java packages. One package represents the repository itself and serves as the root package for anything in the PCM repository. Two further packages are intended to contain the realization of the interfaces and data types of the repository. A contracts package within the repository package contains the interfaces of the repository. They define the contracts between components that provide or require them. A data types package inside the repository package contains the data type realizations.

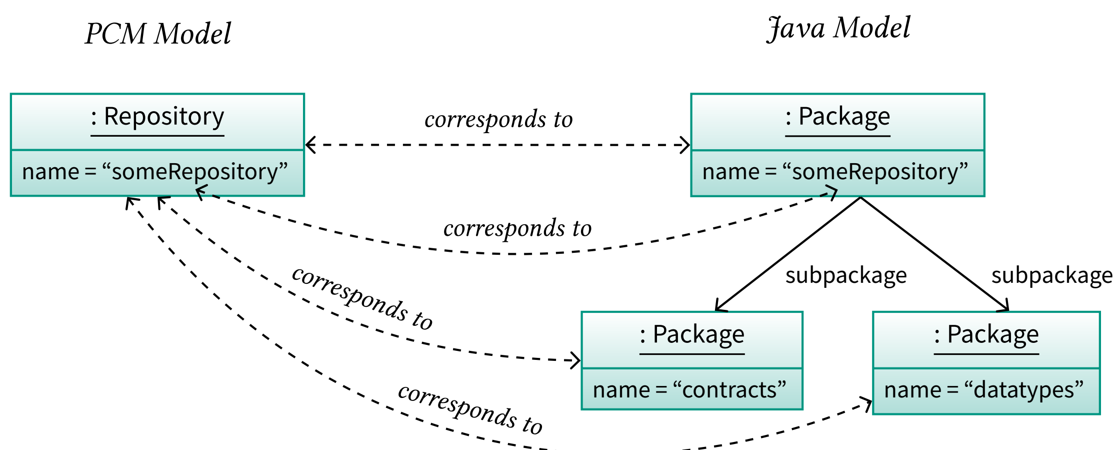


Figure 3.3: Mapping of a PCM repository to Java packages

The name of the repository package is the same as the name of the repository within the PCM model. The names of the contracts and data types packages are intended to be statically defined as contracts and datatypes. An example mapping for a repository is shown in Figure 3.3. The repository called someRepository is mapped to a package with the same name and the two mentioned sub-packages.

To achieve consistency between a repository and its representation in program code, the modification of any element that corresponds to another has to be monitored. If a repository is created, deleted or if its name is changed, appropriate modifications in the Java code are required. Vice versa, the deletion of any of the packages or the renaming of the repository package require an update of the PCM repository.

3.3.2 Component Mapping

Components that are added to a repository are mapped to a package and a facade class in the Java code. The component package is inserted into the package which the repository was mapped to, and the facade class is added to that component package. Both the package and the class have the same name as the component, except that the class name starts with an upper case letter and the package name starts with a lower case latter, according to the Java language constraints. From now on, the facade class of a component is simply be referred to as the component class.

The component mapping is exemplified in Figure 3.4. A component named someComponent is mapped to a package and a class with the same names. This mapping relies on the

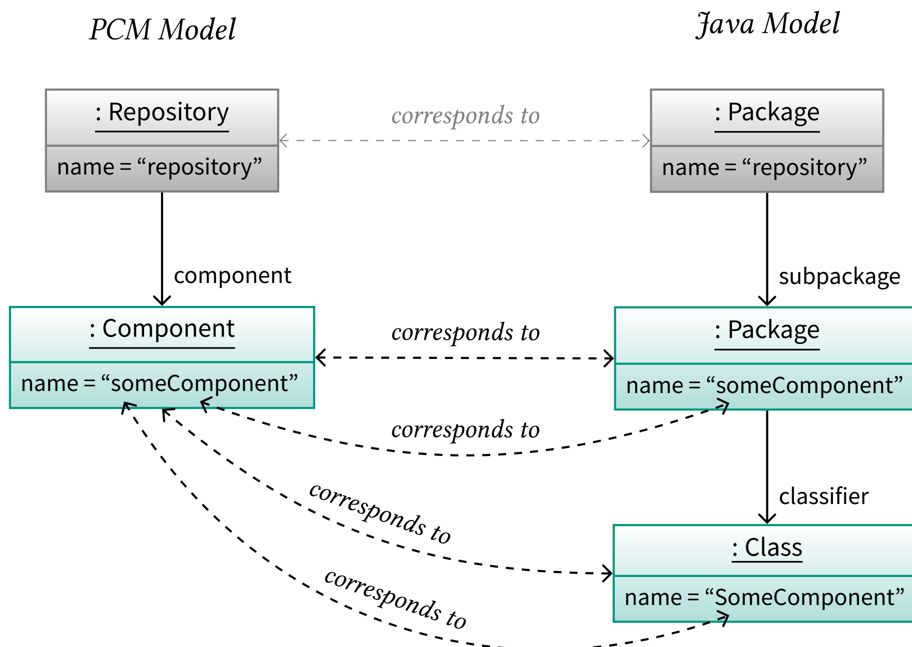


Figure 3.4: Mapping of a PCM component to a Java package and class

mapping of the repository that contains the component because the component package has to be inserted into the package which the repository is mapped to.

For preserving consistency between a component and its code representation, the modifications of any element having correspondences have to be investigated, as described for the repository mapping. Modifications in this mapping can require cascading updates. Changing the class name in the Java code requires an update of the component name in the PCM model, which in turn requires the adaption of the package name. Finally, the name of the class implicitly corresponds to the name of its package as well. Nevertheless, we only consider relations between the different models, which implicitly also define correspondences within a model, as this example shows.

3.3.3 Interface Mapping

The mapping of an interface within a PCM repository is straightforward. It is mapped to a Java interface with the same name, which is added to the contracts package of the repository.

This interface mapping is exemplified in Figure 3.5, where an interface someInterface within a PCM repository is mapped to a Java interface of the same name. It is placed in the contracts package of the containing repository, which requires the repository mapping to be existent.

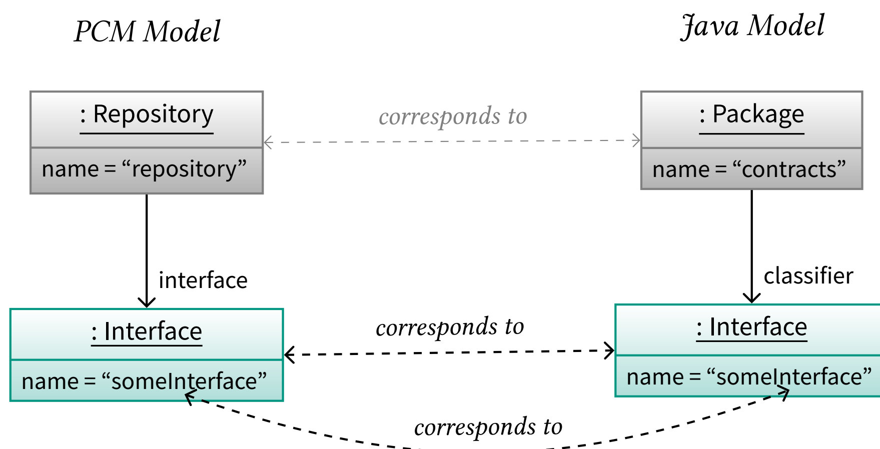


Figure 3.5: Mapping of a PCM interface to a Java interface

3.3.4 Data Type Mapping

The mapping of a data type in a PCM model depends on its concrete type. Data types are separated into collection data types, composite data types and primitive data types. The latter ones represent a special case, which is discussed later, because a concrete primitive type is represented in the Java metamodel instead of a Java model. Collection and composite data types are mapped to simple Java classes with the same name, which

are added to the data types package of the repository. Collection and composite data types also contain inner types, which actually have to be mapped, but are not considered here.

In Figure 3.6, the mapping of a composite data type called someDataType to a Java class is shown. Both have the same name, except that the class name has to start with an upper case letter. For inserting the class into the correct data types package, which the repository is mapped to, it has to be existing and must be retrieved through the repository mapping.

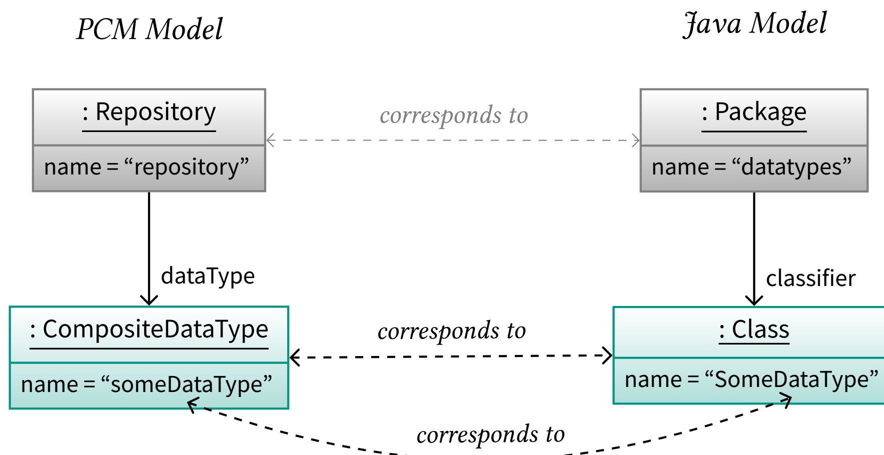


Figure 3.6: Mapping of a PCM composite data type to a Java class

4 Model Changes and Model Consistency

In this chapter, we introduce the basic terminology and concepts in the context of model changes and model consistency. After specifying the term model consistency, we explain model changes and change descriptions. Based on the specification of changes, the requirements for change-driven model transformation environments are derived. Finally, we join the different topics to the concept of change-driven model consistency repair.

Several approaches give a more formal definition of changes and consistency [46, 98]. In contrast, our definitions focus on the comprehensibility by a methodologist who has to specify the consistency between models, rather than on the theoretical provability of the applicability of a consistency-preserving mechanism.

4.1 Model Consistency

The goal of the language that is developed in this thesis is to restore consistency between different, especially heterogeneous models. To understand the rationale for design decisions in the proposed language, a common understanding of model consistency is necessary. In general, consistency can be required between sets of two or more models. We do only consider consistency between two models because it is possible to keep a set of models consistent by keeping all pairs of them consistent. Nevertheless, the statements can be extended to arbitrary large sets of models as well.

4.1.1 Consistency Overlaps

Two models can be considered inconsistent if they contain contradictory information. If, in contrast, models do not contain contradictory information, they are in a consistent state. Models can only contain contradictory information if the information in the different models is interdependent or even redundant.

In the following, we define a terminology for this information dependency and consistency between different models more precisely. To exemplify the terminology, we apply it to relations between a UML class diagram and Java code, which both are models that represent the same system. We consider the representation of two interfaces, *Component* and *DataType*, in both artifacts, as shown in Figure 4.1.

The interfaces in the UML diagram and in the Java code represent the same object of the software system in different ways. Therefore, we refer to them as *representations* of the same interface. Because they both abstract from the same object, they provide the same information redundantly, which consequently has to be consistent. This is why we define the term *consistency overlap* for this kind of information overlap between models.

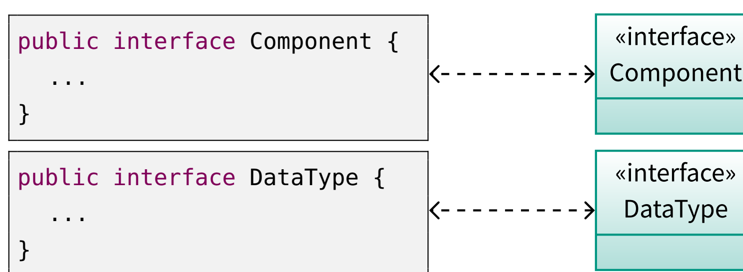


Figure 4.1: Representation of two interfaces in Java code and a UML class diagram

Definition 1 (Consistency overlap)
A consistency overlap is a relation between model elements that contain dependent information.

A consistency overlap describes a relationship between model elements that represent dependent information. The interfaces in the Java code and in the UML diagram represent the same objects and thus contain dependent information. For example, the names and methods of the interface representations depend on each other as they have to be the same. The example contains two consistency overlaps, one for the *Component* interface representations and one for those of the `DataType`. A consistency overlap can be described by the set of the elements that are in the relationship it represents, along with a specification of the dependency of their information. We also say that the elements *share a consistency overlap*.

The simplest form of a consistency overlap is redundancy. Two model elements that represent the same object are redundant and thus share a consistency overlap with dependent properties, like the interface representations in UML diagrams and Java code. In general, the information dependency of consistency overlaps can be more complicated. An example is the representation of a required role of a PCM model in Java code. A required role references a component and an interface and has the meaning that the component requires access to an implementation of that interface. The relation to a possible representation of the same information in Java code is shown in Figure 4.2. The component class has a field of the type of the required interface, a constructor parameter for an implementation of that interface and an assignment to the field. This relationship is not a simple redundancy any more, but describes more complex relations between a PCM model and the Java code.

Consistency overlaps are relationships between elements of concrete models. The dependency of the information provided by these elements can be described by constraints. For example, the relationship between the *Component* interface representations can be expressed by a constraint which specifies that both must have the same name. Nevertheless, this constraint is the same for the *DataType* interface and for all consistency overlaps that describe the relation between the representations of the same interface in Java code and UML class diagrams. Consequently, such a constraint can be specified on the metaclasses of the elements.

Definition 2 (Consistency constraint)
A consistency constraint is a constraint that describes a dependency of information between instances of two or more metaclasses.

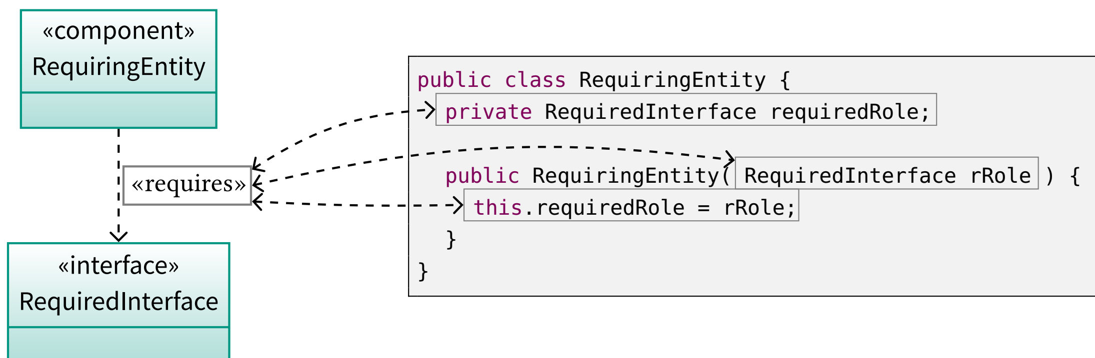


Figure 4.2: Representation of a PCM required role in the Java class implementation of the requiring component

A consistency constraint allows the description of dependencies between elements of a consistency overlap, but is not meant to hold for all instances of the metaclasses it is defined for. How consistency overlaps between models are identified is discussed in the next subsection. A consistency constraint for our interface representations overlap can, for example, define that an instance of the Java interface metaclass and the UML interface metaclass must have the same name. For two representations of the same interface sharing a consistency overlap, such as the two *Component* representations, their information dependency can be described with this constraint.

Consistency constraints describe constraints for certain types of actual relationships. The interface constraint describes all relationships of two representations of the same interface in Java code and UML diagrams. Consequently, we can describe the relations that this constraint specifies on the metamodel level as a relationship type, which we call a *consistency overlap type*.

Definition 3 (Consistency overlap type)
A consistency overlap type is a type of a possible relation between model elements described by two or more metaclasses and a set of consistency constraints on them.

A consistency overlap type specifies a type of relationship that can exist between model elements. It is described by the metaclasses of the elements of a certain consistency overlap and a set of consistency constraints that describes the dependency of the information of the elements. The consistency overlap type for our interfaces example would consist of the UML interface metaclass and the Java interface metaclass and the consistency constraints that describe the dependency of instances of the metaclasses that share such an a overlap, such as the equality of their names. While several interfaces can be represented in both a UML class diagram and Java code, only one consistency overlap type is needed to describe the information dependency in all these relationships. All consistency overlaps that can be described by a certain consistency overlap type are referred to as *instances of the consistency overlap type.*

Consistency overlaps are not independent of each other. The constraints of overlap types also specify the required existence of further consistency overlaps. The consistency constraints for the representation of an interface in a UML class diagram and Java code

require all methods of the interface to be represented by both of them. Because both interface representations are expected to provide representations of the same methods, the existence of consistency overlaps of all methods in both interface representations is expected. In turn, the different representations of one method specify representations of the same parameter list, which requires consistency overlaps between the parameter representations in the methods of the UML diagram and the Java code as well.

Given a consistency overlap, its consistency constraints specify which other dependent consistency overlaps are expected to exist. Consequently, a top-level consistency overlap can specify all further consistency overlaps that are expected to exist between two models. It does not consist of model elements but the models themselves and a top-level consistency constraints for their elements. This concept of top-level constraints can be exemplarily applied to the consistency overlap of a UML diagram and Java code representing the same system. The constraints can specify that for all interfaces and classes of the UML diagram a consistency overlap with a representation in Java code must exist.

With the definition of a consistency overlap type, a consistency overlap can be described by its elements and the type it belongs to. If the elements of a consistency overlap fulfill the constraints of its type, they do not contain contradictory information and thus can be considered consistent. In that case, we call a consistency overlap *satisfied.*

Definition 4 (Consistency overlap satisfaction)
A consistency overlap is satisfied if its elements fulfill the consistency constraints of its type.

Consequently, elements of two models are consistent if all their consistency overlaps are satisfied. With the specification of consistency overlaps, we can define consistency of models more precisely.

Definition 5 (Model consistency)
Two models are consistent if all the consistency overlaps their elements share are satisfied.

For a pair of models, checking the constraints of all their consistency overlaps identifies whether the models are consistent or not. This validation requires the recognition and precise specification of all consistency overlaps of the models, which makes out the difficulty of checking and achieving model consistency.

The consistency definition does not require that the models have to be different. Thus, model consistency can be defined for a single model, based on constraints its elements have to fulfill, as well. This is referred to as *intra-model consistency*, while consistency between different models is referred to as *inter-model consistency*. If not further specified, we talk about inter-model consistency in the following.

4.1.2 Identifying Consistency Overlaps

Checking and ensuring consistency between models requires the knowledge of all consistency overlaps these models share and of all their consistency overlap types. While the latter ones can, for example, be specified in OCL, the identification of all consistency overlaps in two models is not that straightforward. We already mentioned that consistency overlaps can be specified by the elements of which they consist and the type they belong to,

but not how this specification is performed. Different kinds of specifications are discussed in the following.

With the definition of consistency overlaps and their types, we can validate if elements sharing a certain consistency overlap fulfill its constraints and are consistent. Nevertheless, these definitions do not specify how to identify elements that actually share a consistency overlap. For example, the consistency overlap type of interfaces in UML diagrams and program code does not specify how to find concrete interfaces that share such a consistency overlap. We distinguish between two kinds of specifications for the identification of consistency overlaps in models, which are *implicit* and *explicit* ones.

Definition 6 (Implicit consistency overlap specification)
A consistency overlap specification is implicit if the identification of model elements that form it is performed using only the information available in the models.

Consistency overlaps often exist implicitly and thus the elements forming it can be extracted from the models without requiring further information. Implicitly existing consistency overlaps can be extracted from models by defining and checking constraints that sets of elements have to fulfill so that they are treated as sharing such a consistency overlap. For example, interfaces within Java code and a UML diagram can be supposed to share a consistency overlap if they have the same name. Such constraints should not be mixed up with the consistency constraints that describe the dependency of information in consistency overlaps. Especially in the UML and Java case, the constraints for identifying elements that share a consistency overlap are rather similar to the ones specifying their consistency. Nevertheless, in other domains it is possible that elements fulfill the constraints of a consistency overlap type although they do not share an overlap of that type.

An implicit specification of consistency overlaps can only be used if the models are in a consistency state. For example, if two interfaces sharing a consistency overlap currently have unequal names and are thus inconsistent, an implicit overlap specification would not identify them as sharing a consistency overlap. Furthermore, in some cases the extraction of the information that a set of elements shares a consistency overlap is not even unambiguously possible if the models are consistent. To circumvent the limitations of implicit consistency overlap specification, explicit ones can be used.

Definition 7 (Explicit consistency overlap specification)
A consistency overlap specification is explicit if it is not implicit.

Consistency overlaps can be made explicit by saving the sets of elements they consist of additionally to the models in a further model. This kind of specification allows to define consistency overlap types whose instances cannot be unambiguously extracted from the models. A Consistency overlap can also be specified explicitly if it could be specified implicitly because the information in the models is sufficient for identifying it. The drawback of an explicit specification is that it has to be stored and maintained externally.

Retrieving the elements of a consistency overlap with an explicit approach is simple because they can be directly accessed. For a given interface method within a UML class

diagram, the appropriate interface method in the implementing Java code can be retrieved by just examining the explicitly saved consistency overlaps. Using implicit specifications, this retrieval is far more complicated. First, the Java interface that shares a consistency overlap with the interface of the UML method must be retrieved. Afterwards, the correct method in the Java interface, which has the appropriate signature with equal name, return type and parameter types, must be found. Therefore, the return and parameter types must be also be matched correctly. Finally, the complete identification using an implicit approach is only possible if the models are consistent.

Both ways of identifying consistency overlaps can be combined. To allow the specification of overlaps whose elements cannot be extracted from model information and to ease the retrieval of related elements, parts of the elements of a consistency overlap can be saved explicitly. We therefore introduce the term *correspondence*.

Definition 8 (Correspondence)
A correspondence consists of two sets of elements of different models that share a consistency overlap.

A correspondence represents elements of a consistency overlap and separates them into two sets that represent the elements of the different models they belong to. We refer to two elements of a correspondence, one from each of these sets, as *corresponding*. A correspondence, which can even contain only a part of the elements of the overlap, can be used to describe and save the elements of a consistency overlap. Such correspondences provides a utility structure, which is the basis for specifying overlaps that can exist of several explicit correspondences and implicitly identified elements.

Many consistency overlaps consist of pairs of model elements instead of complex sets. Thus, the specification of correspondences can be reduced to pairs of elements in many cases, which are enriched with implicit specifications if necessary. For example, if one element contains an attribute that combines information of two other model elements, this consistency overlap can be described by two correspondences, one for each combined element with the combining one.

4.1.3 Dependency between Consistency Overlaps

We already discussed that consistency constraints specify which further consistency overlaps are induced by a certain one. For example, the consistency overlap of an interface representation in a UML diagram and Java code requires their methods to share consistency overlaps as well. This describes that a certain consistency overlap requires further consistency overlaps to exist for being satisfied. In turn, consistency overlaps can also depend on the existence of others to ever exist, which is simply the inverse relationship in many cases. For example, a consistency overlap between an interface method in a UML diagram and Java code requires their interfaces to share a consistency overlap.

This dependency relation is not always the simple inversion of the relationship defined by the consistency constraints. Considering the consistency overlap type of PCM required and provided roles and Java code, the consistency constraints of the overlaps between the providing or requiring component and the Java class implementation require that its roles in the PCM model have representations in the Java code that they share consistency

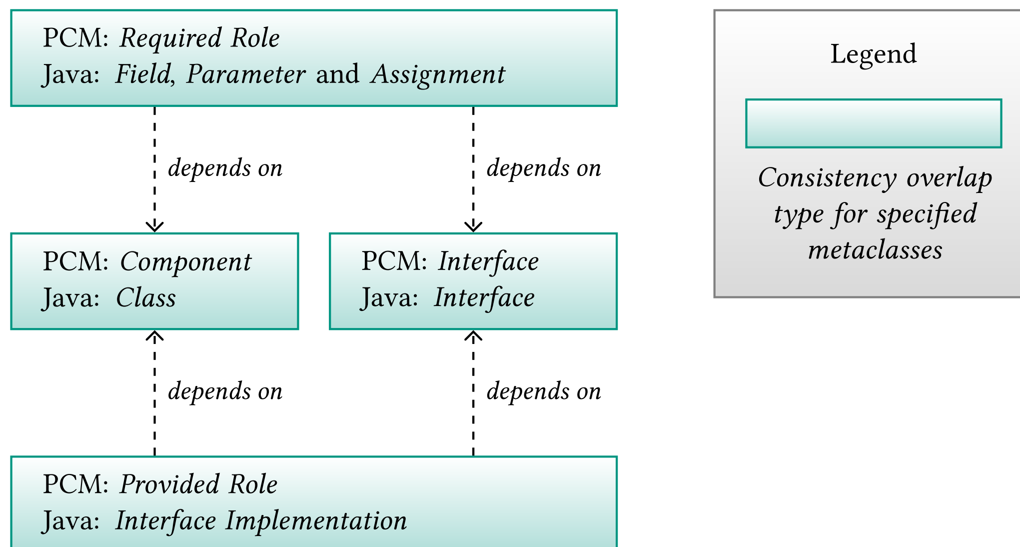


Figure 4.3: Dependencies of the consistency overlap types for PCM required and provided roles and their Java implementation

overlaps with. In turn, the consistency overlaps of the roles require at least two consistency overlaps to exist. The component of the role has to share a consistency overlap with the Java class that implements it. The PCM interface that is required or provided by the component must share a consistency overlap with the Java interface that represents it. These dependencies of the concerned consistency overlap types are shown in Figure 4.3.

As the example shows, consistency overlaps have a dependency hierarchy. This insight is important for the preservation of model consistency. Modifications of elements of consistency overlaps which other overlaps depend on can have cascading effects and require them to be updated as well. Furthermore, it is used for reasoning about the granularity of consistency overlaps in the following.

4.1.4 Granularity of Consistency Overlaps

A concrete specification of consistency overlap types for a specific pair of metamodels is not indisputable. Neither the definition of consistency overlaps or their types does restrict their granularity nor is it restricted naturally. Considering the required roles example in Figure 4.2, it is arguable if the required role shares one consistency overlap with each of the three Java elements, or one with all of them.

Two extremes for the granularity of consistency overlaps can be considered. In a most coarse-grained approach, two models only share one consistency overlap that potentially consists of all their elements and rather complex constraints. In a very fine-grained approach, potentially all consistency overlaps do only consist of two elements and types that define more likely easy constraints.

We attempt the specification of consistency overlaps that are as small as possible and as big as necessary. With this approach, a minimal number of model elements must be examined if a consistency overlap gets unsatisfied and a most simple repair logic must

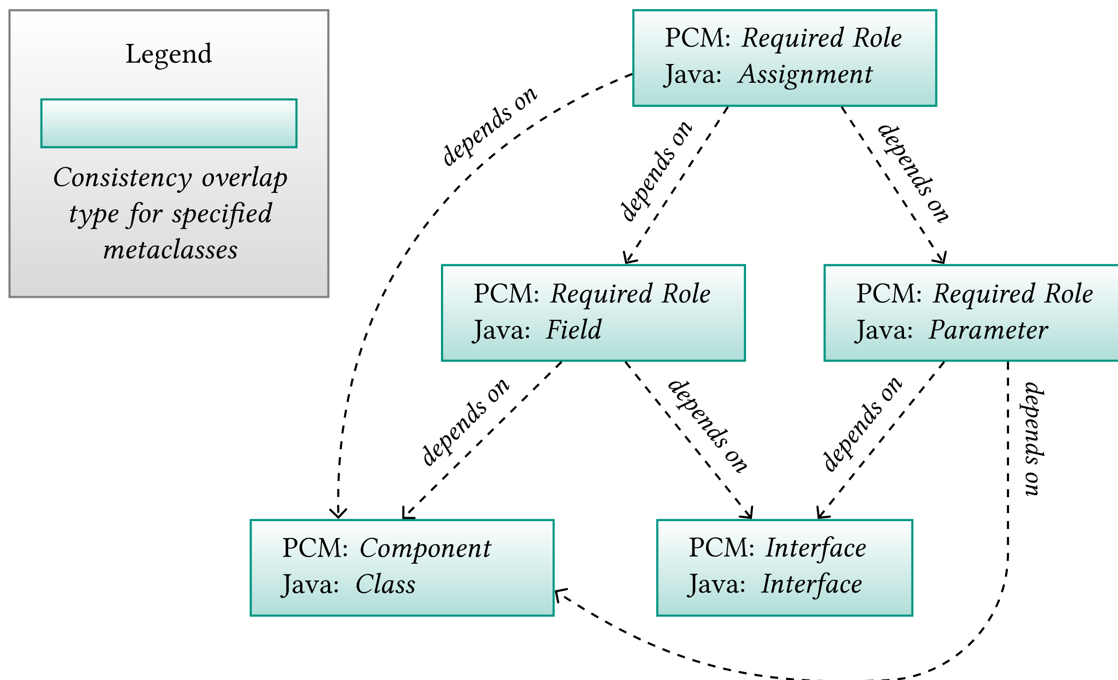


Figure 4.4: Dependencies of minimal consistency overlap types for PCM required roles and their Java implementation

be defined and executed to repair it. To determine if a consistency overlap is as small as possible, we introduce the term *minimal consistency overlap*.

Definition 9 (Minimal Consistency Overlap)
A consistency overlap is minimal if it cannot be split into two consistency overlaps that do not depend on each other.

According to this definition, many consistency overlaps are only minimal if they do just consists of two model elements. An example for a minimal consistency overlap that contains more than two elements is the combination of attribute values of two model elements in an attribute of another model element. The consistency constraint that defines this relation has to validate if the attributes are correctly mapped to the combined value, which is only possible if both combined attributes are known. If the consistency overlap was split into two, each consisting of the combining element and one of the combined ones, they would depend on each other.

The restriction to use only minimal consistency overlaps would be very strict and is potentially too fine-grained in practice. The consistency overlaps of the required roles with Java code elements shown in Figure 4.2 would be structured on type level as shown in Figure 4.4. Three consistency overlaps, each consisting of two model elements, would fulfill the definition of minimal consistency overlaps. The two consistency overlaps between the role and both the field and the parameter require the component to share an overlap with its implementing class. This overlap of the component is necessary because they are placed in the class and its constructor. These consistency overlaps also expect the required interface to share an overlap with the Java interface because it defines the type of the field

and the parameter. The assignment of the required role parameter to the field obviously depends on the field and parameter overlaps, as well as on the component overlap to place the assignment inside the constructor of the component class.

It is disputable if one consistency overlap for all three Java elements would be a more clear viewpoint in practice. In general, the discussion about the granularity of consistency overlaps is not important for the correctness of a consistency specification. It is just motivated by the fact that our consistency repair approach relies on the repair of consistency constraints and thus allows more simple specifications of consistency repair if the considered overlaps are rather small. Nevertheless, the proposed language or its constructs do not functionally rely on a certain granularity of consistency overlaps.

4.1.5 Prescriptive and Descriptive Consistency Overlaps

Consistency overlaps can exist for two different reasons. They can exist naturally, for example, because one model is an abstraction of the other one, or they are synthetic because the relations between elements are defined manually, which is the case for PCM and Java. Consequently, the specification of consistency overlaps can follow a descriptive or a prescriptive approach, analogously to the distinction of models as described in section 2.1 and natural language grammars [68].

In a descriptive approach, consistency overlaps are assumed to already exist. Considering UML class diagrams and Java code, the overlaps exist naturally as both models describe the same elements. This is always the case if models represent the same elements on different levels of abstraction. In these cases, the elements of both models are the same and so they inherently share a consistency overlap.

In a prescriptive approach, consistency overlap types are defined manually and have no natural reasoning. The relations between PCM models and Java code rely on a primarily prescriptive specification because different mappings of PCM elements to code fragments are possible, as described in section 3.3. Only by defining the consistency overlap types explicitly, they can be applied to concrete models.

The classification whether the specification of consistency overlaps follows a descriptive or a prescriptive approach is not strict. As the primary distinction, in a descriptive approach existing relations between different models are described by defining consistency overlap types, whereas in a prescriptive approach no relations exist yet but are prescribed by the specification of consistency overlaps types.

4.1.6 Correspondence Models

The concept of correspondences was introduced in the context of the explicit specification of consistency overlaps in subsection 4.1.2. Explicit specifications of consistency overlaps require the structure of correspondences to be precisely defined because they have to be persisted. The correspondences of models form a model themselves and are an instance of a correspondence metamodel.

Correspondence models can be seen as an approach for storing the tracing information of transformations, as introduced in subsection 2.2.2. A common metamodel for such tracing information is the trace model of QVT [72]. QVT defines *trace classes*, which have

properties referring to model elements that are related. *Trace instances* define concrete relations between two models. They are generated by a transformation defined in QVT and contain the elements that were mapped through the transformation rule. While a trace instance in QVT is tied to the transformation rule that created it, correspondences in our approach rely on consistency overlaps that can be potentially affected by different transformation rules.

In VITRUVIUS, an extensible correspondence metamodel is used. A correspondence consist of two sets of model elements and can be enriched with further attributes. The correspondences are contained within a correspondence model for each pair of metamodels. As this correspondence model fits our needs and as our approach contributes to the VITRUVIUS framework, the concepts of this thesis relies on that correspondence metamodel.

4.1.7 Preserving Model Consistency

According to the definitions of model consistency, consistency overlaps, and correspondences, the consistency of two models can be validated by checking the constraints of all consistency overlaps that the two models share. This validation requires that all consistency overlaps of the models are known and the constraints of their types are correctly specified.

Preserving consistency is a more complicated problem, which can be addressed in different ways. One solution is the inherent idea of the OSM approach, which was described in subsection 2.4.2, to have a SUM that contains all needed information. Using a SUM, no models must be kept consistent because there is only one model that ideally contains no redundancies and non-explicit dependencies. The SUM is always consistent and thus no consistency-preserving mechanisms are required in that scenario.

In general, there is no SUM and different models have to be kept consistent. This problem is addressed in the VITRUVIUS approach, which was introduced in subsection 2.4.3. Approaches for preserving consistency of models that share consistency overlaps rely on model transformations. The transformations between the models ensure that the constraints which all consistency overlaps of the models have to fulfill are restored and satisfied whenever they get violated. Because approaches that are based on transformations repair an inconsistent state, even though its usually just temporary, we refer to them as *consistency-restoring* or *consistency-repairing* rather than consistency-preserving.

One property of transformations is their degree of incrementality. Of special interest for restoring model consistency are incremental transformations in general and their subclass of change-driven transformations, as introduced in section 2.2. Because model transformations can be used for different purposes, we put the focus on change-driven transformations for restoring model consistency, which are further discussed in section 4.4.

Another property of transformations is the kind in which they are specified. Approaches are either declarative or imperative. While declarative approaches specify only the constraints that have to be fulfilled but not how, imperative approaches specify how constraints are ensured. Because declarative transformations have to be converted into imperative code that restores the specified constraints, they can just be as expressive as approaches that directly allow this imperative specification. Declarative approaches cannot achieve any kind of model consistency, as the repair of consistency cannot be automatically gener-

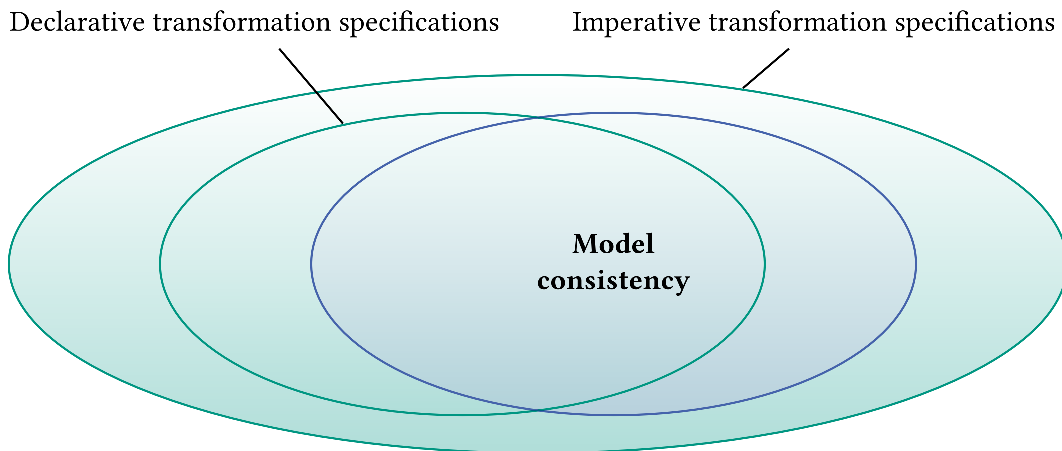


Figure 4.5: Achievable model consistency with different kinds of transformation specification

ated for any kind of declared constraint. An example for such constraints are aggregated attribute values. If a constraint specifies that an attribute value is derived from two others, it is unclear how to make them consistent if the aggregated one is modified. This problem is addressed by current research [57] and exemplifies the limited expressiveness of declarative approaches. The relationship between achievable consistency and the kind of transformation specification is visualized in Figure 4.5.

The MIR language family, described in subsection 2.4.3, provides different languages for restoring model consistency. These languages follow different paradigms and implement different characteristics of transformation languages. This thesis proposes an imperative, unidirectional language for restoring consistency that contributes to the MIR languages as the *response language*. The *mapping language* provides a declarative, bidirectional approach for preserving consistency and the *invariant language* allows the specification of constraints of consistency overlap types for checking consistency.

4.2 Model Changes

Models are typically no static artifacts, in fact they get changed when they are used. They can be changed due to different reasons, such as the fix of a bug, the implementation of new, or the adaption to changed requirements. In the software evolution context, these changes are called corrective, perfective and adaptive [49] and can be transferred to the evolution of models. A *model change* can be defined as follows.

Definition 10 (Model Change)
A model change is an action that performs modifications on model elements or properties of them.

The definition specifies that a model change performs modifications on model elements or properties of them. As we assume models based on EMOF, such changes can be the creation and deletion of model elements, modifications of their attributes and references

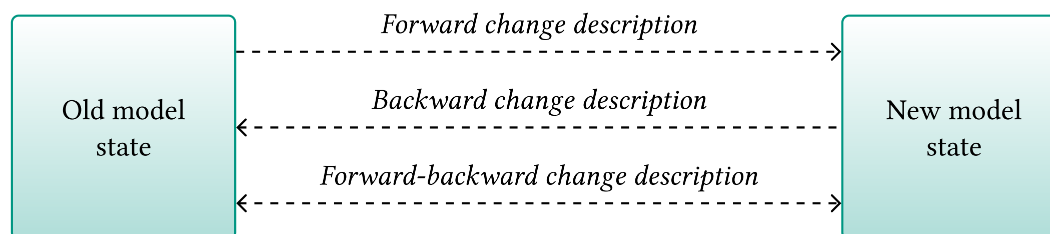


Figure 4.6: Change description types and the directions they describe

to other model elements, or sets of such modifications. Changes can also be *trivial*, which means that they modify the model in a way that its new state cannot be distinguished from its old state. An example for such a trivial change is the replacement of an attribute value with the same one.

4.2.1 Change Descriptions

A model change is abstractly defined as an action that modifies a model. While this is sufficient for identifying changes, they also need to be described. Such a description has to provide information about the differences between the old and the new model state to identify how a model was changed. Those descriptions can be provided in three different ways. We discuss them on an example change, the replacement of an attribute value of a certain model element with a new one.

A *forward change description* contains information for transforming the model state before the change into the state after the change. In the example, such a description has to provide at least the modified element, the modified attribute and the new attribute value. A *backward change description* contains information for transforming the model state after the change back to the state before the change. For the attribute replacement, such a description has to comprise the modified element and attribute, as well as the old attribute value. Finally, a *forward-backward change description* provides information to navigate between both model states and thus is a combination of the other kinds of change description. In the example, such a description has to contain the modified element and attribute, as well as both the old and the new value. The different change description types and the directions in which they can be applied are shown in Figure 4.6.

4.2.2 Atomic Model Changes

A general model change can affect an arbitrary large set of model elements and properties. The descriptions for such changes have to provide information about all the modified elements and properties. We distinguish between *atomic* and *composite* changes, whereof the first ones have to fulfill some criteria of atomicity.

Definition 11 (Atomic model change)
A model change is atomic if it can be unambiguously described by the old and new value of a single model element or property.

The smallest possible modification in a model is the change of a property value or the modification of an element reference in terms of creating or deleting the referenced element. Such changes cannot be subdivided into two changes, as they just affect a single value of a model. These kinds of modifications are covered by the definition of atomic changes, and thus atomic changes are the smallest possible modifications that can be performed in a model. Consequently, atomic changes represent the partition of possible model changes that cannot be divided into two changes, which justifies the name *atomic*.

The definition of atomic changes implicitly proposes a forward-backward description of them. The specification of a modified element reference or a property change with its old and new value is required by the definition and allows to navigate from the old to the new model state and vice versa. We therefore describe changes as forward-backward change in the following.

Atomic changes can be categorized by the kind of property that was modified. For example, changes of element properties can be subdivided into inserting and deleting changes if the property is multi-valued, whereas changes of single-valued properties are replacing changes. The used meta-metamodel inherently defines the types of atomic changes that are possible in models which are conform to that meta-metamodel. The meta-metamodel defines which elements a metamodel can contain and thus also which properties exist that can be modified in model instances. The extraction of possible atomic changes from a given meta-metamodel are discussed for EMOF in the following.

4.2.3 Atomic Model Changes in the Essential Meta-Object Facility

A meta-metamodel inherently defines possible types of atomic changes in model instances which are based on that meta-metamodel. A common meta-metamodel is EMOF, which was introduced in subsection 2.3.1. Figure 4.7 shows an extract of EMOF that contains the parts of the meta-metamodel that are relevant for potential changes in the model instances.

The basic elements of EMOF models are classes that contain properties. Properties have a type, which is a class or a data type, that in turn can be a primitive type or an enumeration, and they have multiplicities that define how many elements of their type they can reference. Properties can be part of an association that links two properties of two classes to indicate that they are opposites of each other. In such a case, the opposite reference as well as the isComposite property are derived. The isComposite property indicates if the referenced type is the container of the class that contains the property. Further elements of the meta-metamodel, such as packages that define namespaces for types, comments that annotate elements, and operations that can be defined on classes, are omitted because they are not relevant for changes in model instances.

Metamodels that are conform to EMOF define Class and their Property instances. Models that are conform to such a metamodel contain instances of these classes, which have references to other classes according to the concrete properties. This model structure results in two categories of possible changes. The first category of changes covers the creation and deletion of class instances, the second one consists of modification of properties.

The possible atomic changes within EMOF-based models are summarized in Table 4.1. The creation and deletion of class instances is of special interest for root elements of models. All other element creations and deletions come along with a property change

EMOF model element	Atomic change type
Class instance	Creation, deletion
Multi-valued property	Insertion, removal, permutation
Single-valued property	Replacement

Table 4.1: Atomic change types in EMOF

because the created or deleted element has to be contained in some other model element. Thus, they are inserted or removed from a property of another model element, which leads to the change of that property.

Property changes can be subdivided into inserting, removing and permuting changes, depending on whether the value of a property was added or removed from the list of properties or moved to another index within the list. Furthermore, properties have multiplicities, which allows us to categorize them into single-valued and multi-valued properties, depending on whether the *upper* multiplicity value is one or greater than one. Changes of a single-valued property can be simplified because it references a single element. This element can only be replaced with another, which leads to only one possible type of change, the replacement.

Although these basic atomic change types were extracted from the EMOF meta-metamodel, they can be transferred to any meta-metamodel that has a similar structure. If it consists of elements equal to classes and properties referencing types, the derived atomic change types are suitable for that meta-metamodel as well. A similar categorization of

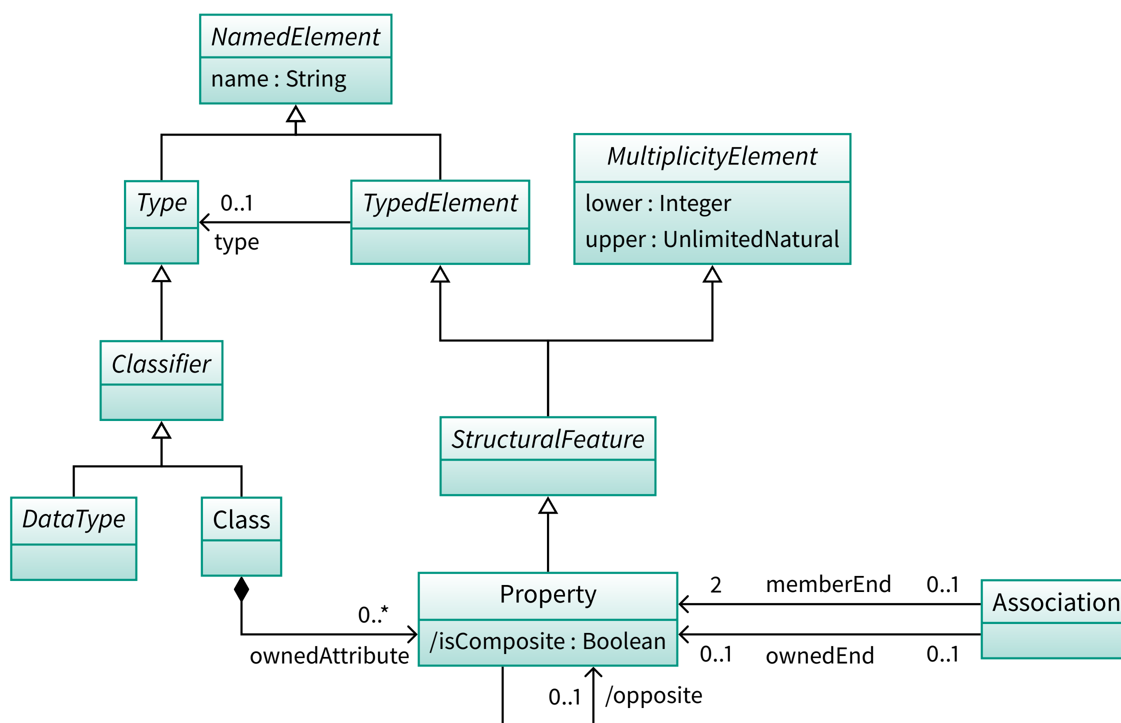


Figure 4.7: Extract of the change-relevant elements of EMOF

possible atomic changes, based on an abstract model definition, is provided by Hettel, Lawley, and Raymond [46]. Nevertheless, their categorization requires properties for data types, which they call attributes, to be single-valued and omits the direction and containment of references as they only consider abstract relations between types.

4.2.4 Atomic Model Changes in Ecore

An implementation of EMOF is the Ecore meta-metamodel of the Eclipse Modeling Framework, which was introduced in subsection 2.3.1. Because Ecore is used for the prototypical implementation of the language developed in this thesis, the differences to EMOF are explained in the following.

The Ecore meta-metamodel is an implementation of EMOF and as such its structure is conform to EMOF, which leads to similar atomic change types. An extract of the Ecore meta-metamodel with the elements that are relevant for atomic changes in model instances is shown in Figure 4.8.

Two essential differences distinguish EMOF and the Ecore meta-metamodel. First, the MultiplicityElement of EMOF is integrated into the ETypedElement in Ecore. Second, the Property element of EMOF is subdivided into EReference and EAttribute in Ecore. The important difference is that an EAttribute can only reference an EDataType, and an EReference can only reference an EClass, whereas a Property in EMOF can reference any type. Consequently, only an EReference element can define containment relationships, wherefore it has an attribute called containment. It has the semantics that the referenced elements

Figure 4.8: Extract of the change-relevant elements of the Ecore meta-metamodel

Ecore model element	Atomic change type
Class instance	Creation, deletion
Multi-valued attribute	Insertion, removal, permutation
Single-valued attribute	replacement
Multi-valued reference	Insertion, removal, permutation
Single-valued reference	Replacement

Table 4.2: Atomic change types in Ecore

are contained in the element that has the EReference. This semantics is the opposite of the derived isComposite attribute of a Property in EMOF, which describes that the referenced element is the container.

For the possible atomic changes, only one important difference arises. EMOF only provides a Property element, which allows modifications of single- and multi-valued properties in model instances. Ecore subdivides properties into EAttributes and EReferences, which allows to distinguish between changes of single- and multi-valued instances of both elements. This differentiation is summarized in Table 4.2. Nevertheless, modifications of both kinds of properties in Ecore models could also be summarized into more generic descriptions like in EMOF. Vice versa, changes in EMOF could be further specialized by investigating if the changed property references a DataType or a Class. This investigation would realize the implicit distinction that Ecore realizes through different meta-metaclasses.

Like EMOF, the Ecore meta-metamodel defines several further elements. However, they are not relevant for atomic changes that can be performed to Ecore models because their instantiation in the metamodel does not provide any changeable properties in the models. Examples are the type parameters of EClass instances, the super types of EClass instances and EPackages. Furthermore, EOperations with EParameters are omitted because they are also fixed on metamodel level and cannot be changed in concrete models.

4.2.5 Composite Model Changes

Atomic model changes form the partition of possible modifications in a model that do only affect a single model element reference or model element property. Thus, they are the smallest possible modifications that can be performed in a model, as stated in subsection 4.2.2. According to its definition, a model change can be any modification of model elements or their properties. Consequently, changes can be more complex than atomic changes are. We call these changes *composite*.

Definition 12 (Composite model change)
A model change is composite if it is not atomic.

Because models consist of model elements and properties, each model change concerns only elements and properties. As a result, any change can be described as a set of changed element references and changed properties together with at least their old or new values and therefore as a set of atomic changes. Finally, any model change that is more complex than an atomic change can be composed of atomic changes.

In general, composite model changes can be further distinguished by their cause. First, a composite change can be composed of atomic changes that were triggered by a single modification. For example, changing the container of an object can be expressed by the two atomic changes that describe the removal from the old container and the insertion into the new container. These two atomic changes represent a composite change that describes the movement of an element to a new container. Second, several atomic changes can be performed within a model independently, which together define a change as well. Such a series of atomic changes can also be described by a composite change.

More relevant than the cause of changes is their ordering. In general, the atomic changes of which a composite change consists have a specific order in which they were executed. Applying the changes to the model in a different order can lead to inconsistent intermediate states of the model. Therefore, the description of a composite change generally has to provide the ordering of the atomic changes of which it consists.

4.3 Change-Driven Model Transformation Environments

Change-driven model transformations are used in our approach to restore consistency between models. The application of such an approach requires an environment that recognizes changes in a model that shall be kept consistent with others and triggers appropriate transformations based on the recorded changes. The generic structure of such an environment is shown in Figure 4.9. We introduce the necessary components, monitors that recognize changes in models, transformation execution engines and the change-driven transformations, in the following.

4.3.1 Monitors for Model Changes

The monitoring of changes is an essential feature of a change-driven transformation environment. A monitor has to recognize changes in a model of interest and to convert them into change descriptions. As described in subsection 4.2.2 about atomic changes, a meta-metamodel inherently defines the possible changes in model instances. In consequence, a monitor that converts model modifications into change descriptions depends only on the used meta-metamodel.

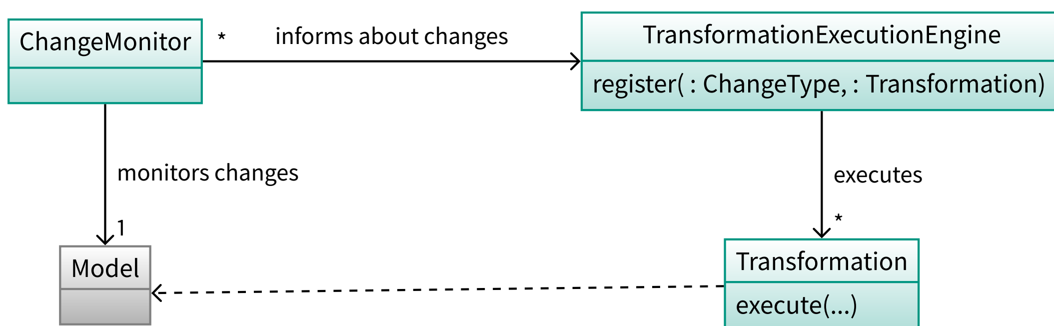


Figure 4.9: Generic structure of a change-driven model transformation environment

A change monitor needs to get informed about model modifications to generate the appropriate change descriptions. For example, Ecore models provide a notification mechanism that is based on the observer design pattern [36]. A change monitor can register itself for notifications at the Ecore-based model and react to notifications about modifications.

Whenever a modification is performed, it must be converted into a change description of a defined type. Depending on the way model modifications are recognized, this conversion differs. For example, the simple notification about any change would require the model to be compared with its old state to find differing model properties. In the case of Ecore, the notification mechanism provides sufficient information about the modified elements to generate a change description from the provided information. Depending on the size of the modification that is recognized, it can be necessary to subdivide the change into several atomic changes at this point.

The transformation environment used for the response language implementation relies on the Vitruvius framework, which was introduced in subsection 2.4.3. It provides a change monitor for Ecore-based models and change descriptions according to the change description metamodel defined by Vitruvius [58].

Model changes can be performed by different actors. Humans as well as the execution of a program can change a model, the former through defined editors and the latter by directly modifying elements. For monitoring changes, it is irrelevant who performs the changes within the models. It is just important that the change monitor is registered at the models and recognizes changes in any case a model can get potentially modified.

4.3.2 Transformation Execution Engines

A change monitor provides change descriptions for modifications that are performed in models which shall be kept consistent with others. The second part of a change-driven transformation environment covers the execution of defined transformations based on model changes. Its task is the execution of all transformations that react to a specified change type whenever a change of that type is recognized by the change monitor.

A transformation execution engine needs to define two interfaces. One interface must be implemented by transformations so that the transformation environment can accept and execute transformations that are conform to that interface. In Figure 4.9, this interface is represented by the Transformation. It must at least provide one method to execute the transformation with relevant parameters. Another interface must be implemented by the execution engine, which has to allow the registration of transformations that are conform to the first interface to be executed whenever a certain change occurs. In the presented generic structure, this interface is represented by the TransformationExecutionEngine. Such an interface must at least provide a method that accepts a transformation conform to the first interface and the specification of a change type that transformation reacts to.

The execution engine in our approach allows to register transformations for a pair of metamodels, so that a transformation is triggered by a change in an instance of one of the metamodels. A transformation checks its responsibility for the actual change and performs consistency-restoring modifications in an instance of the other metamodel.

The execution of transformations can be triggered in different ways. One possibility is to execute transformations whenever a change is recognized by the change monitor.

Another possibility is to call the execution explicitly at some point of time and execute the transformations for all recorded changes.

If at most one transformation is registered for a specific change, the execution environment can execute this transformation whenever such a change occurs. In general, several transformations can be registered for the same change. Although these transformations could be combined into one, a separation may be necessary for reusing a transformation that is executed in response to different changes. In this case, it can be necessary to specify an ordering of transformations because the execution of one transformation may require the completed execution of another one.

4.3.3 Change-Driven Model Transformations

Change-driven model transformations are small transformation rules that are executed in response to a change of a model and perform modifications in other models. Therefore, they investigate the change description of the actual change to decide if their operations shall be performed and utilize the correspondence model to retrieve elements in other models that they modify.

The creation and execution of change-driven transformations requires their registration at the transformation execution engine, which executes the transformations whenever an appropriate change is detected. Therefore, the transformations have to implement a well-defined interface that is provided by the environment, such that they can be called by it. In Figure 4.9, this interface is represented by the Transformation.

A change-driven transformation always represents a routine for restoring consistency between models. It modifies elements in reaction to a change, which inherently defines constraints that have to be fulfilled between certain model elements and ensures them. The specification to which modifications to react and the selection of elements from the correspondence model specifies the elements that belong to a consistency overlap that this transformations repairs. Therefore, change-driven transformations are a well-fitting kind of transformations for restoring model consistency.

4.4 Change-Driven Consistency-Restoring Transformations

Consistency between models can be achieved in different ways, which were introduced in subsection 4.1.7 about preserving model consistency. The approach in this thesis achieves consistency between two models through change-driven transformations between them. In the following, we reason the applicability of change-driven transformations for restoring consistency. Moreover, we derive responsibilities and a basic structure of change-driven transformations for restoring consistency and give an introduction to languages for specifying change-driven consistency-restoring transformations.

4.4.1 Atomic Changes as Triggers

Different types of model transformations can be used for restoring consistency between models. One possibility is to execute a batch transformation that transforms the modified

source model into a new and consistent target model. The drawback of this approach is that further information that was added to the target model additionally to the elements created by the transformation gets lost if the transformation is executed again. For example, generating new Java code from a UML class diagram every time it is changed results in losing at least the method implementations.

A second possibility is to use an incremental approach by checking models for consistency at some point of time and restoring consistency by transformation rules based on the potentially inconsistent states of the two models. This approach requires the transformations to be able to restore consistency of models from states in which inconsistencies can be arbitrarily complex. Along with the complexity of inconsistencies, the complexity of the needed transformations increases. Furthermore, the possibility of losing information still exists. For example, if a method in a UML class diagram is moved from one class to another, it is potentially not possible to identify that both methods are the same, especially if further properties like the name are changed. The consequence is still the loss of the method implementation in the Java code.

Another approach is to restore the consistency of models in the moment when they get inconsistent. Assuming two consistent models, they can only get inconsistent if one of them gets modified. The modification can be described as a model change and thus consistency can be restored by reacting to a change. We broke consistency of models down to the constraints satisfaction of the consistency overlaps that models share. Consequently, models can only get inconsistent due to a change that leads to the violation of a constraint of a consistency overlap. Only those consistency overlaps whose constraints depend on the model elements that were affected by the change can get unsatisfied. As a result, the required transformations do only need to consider the consistency overlaps in which the elements modified by a certain change are contained. Because any change either is atomic or can be subdivided into atomic changes, it is sufficient to define transformations that react to atomic model changes and repair inconsistencies that arise from such an atomic change. We focus on this kind of *change-driven transformations* for restoring model consistency.

The idea of *change-driven model consistency repair* based on atomic changes is visualized in Figure 4.10. We assume two models A and B to be initially consistent. If a potentially composite change Δ^A is performed in model A, it transforms the model into a new model A_n. The change can be subdivided into several atomic changes Δ_1^A to Δ_n^A, which transform the model into intermediate states A_1 to A_n. To keep model B consistent with A, the atomic changes Δ_1^A to Δ_n^A are transformed into changes Δ_1^B to Δ_n^B in model B, which keep the intermediate states A_1 to A_n consistent with B_1 to B_n. The changes in model B do not need to be atomic. Depending on the consistency overlaps that the elements modified by the atomic changes of A share with B, these changes can get arbitrarily complex. However, they are restricted to the restoration of constraints of consistency overlaps that the element modified in A shares with B.

This approach keeps the complexity of the required transformations for restoring model consistency low because they must only consider a small and well-defined set of consistency overlaps. The approach requires one transformation for each atomic change that can occur in a model A and may affect an existing consistency overlap into a change in model B.

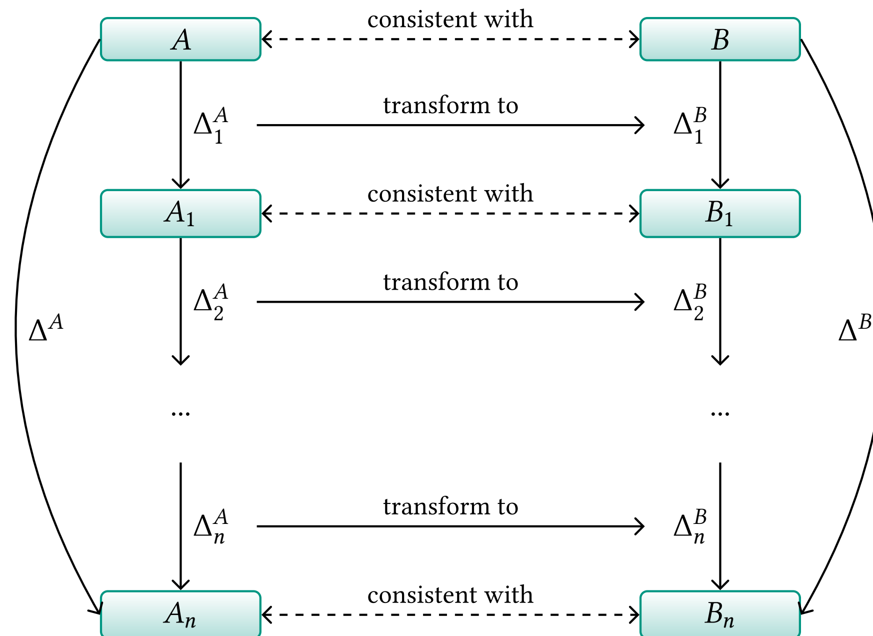


Figure 4.10: Consistency-restoring transformations in a change-driven environment

The presented concept assumes that it is possible to generate a change in model B for each possible atomic change in model A, so that both models are consistent after applying both changes. Therefore, two further assumptions have to be made. First, a consistent state of model B must exist for each consistent state of model A, so that both models are consistent with each other. Second, an atomic change in a consistent model A_i must transform it into a new consistent state A_{i+1}. If the model itself is not consistent after a change, it is generally not possible to make it consistent with another model. The second assumption is further discussed in the following.

4.4.2 Intra-Model Consistency after Atomic Changes

The previous subsection introduced the idea of using change-driven transformations for restoring model consistency. Based on atomic changes, these transformations restore the satisfaction of consistency overlaps between different models. Therefore, the consistency of the modified model itself has to be assumed after an atomic change.

Usually, a model is syntactically correct after an atomic change, at least if it is based on EMOF or Ecore. Nevertheless, a metamodel may define additional static semantics that may not be fulfilled after an atomic change. An editor can preserve intra-model consistency by allowing only modifications that lead to a new model state that fulfills the static semantics of the metamodel. Such an editor can, for example, only allow composite changes that lead to a consistent state. This is possible, although the atomic changes it consists of cannot be ordered in such a way that the intermediate model states are consistent.

To provide only change descriptions that contain information about modifications of the model that transform it into a new consistent state, the provided change descriptions would have to be metamodel-specific. Moreover, the potential changes that are necessary

to fulfill consistency constraints can get arbitrarily complex as the constraints can be that complex. In the end, metamodel-specific change descriptions would complicate the specification of a generic transformation language for change-driven transformations.

Consequently, consistency-restoring transformations have to deal with possibly inconsistent model states. They must either refuse their execution, so that inter-model consistency is only restored if the model is internally consistent again, or try to achieve inter-model consistency although the models are not consistent themselves. Either way, the transformations do only have to deal with the potential temporary violations of semantic constraints that the changed model has to fulfill, but usually not with violations of its syntax.

Models should only be editable in a way that they are consistent after a change. Such a change consists of atomic changes that may lead to inconsistent intermediate states of the changed model. Although consistency-restoring transformations are based on these atomic changes and consequently have to deal with inconsistent states of the changed model, they can assume that a processed sequence of atomic changes always leads to a consistent state in the end. If the methodologist knows about the possible modifications that finally achieve intra-model consistency, he can specify the inter-model consistency-restoring transformations in a way that these sequences of atomic changes are handled correctly.

4.4.3 Responsibilities of Transformations

The purpose of a change-driven consistency-restoring transformation is to restore consistency between models by repairing the consistency overlaps of these models after a specific change. As introduced in subsection 4.4.1, we consider change-driven transformations that specify a reaction to a certain atomic change. An atomic change affects a single model element and thus can only make consistency overlaps in which the changed element is contained unsatisfied. Consequently, a transformation that reacts to a certain atomic change must only restore these consistency constraints.

A consistency-restoring transformation does not depend on the concrete consistency overlaps but is the same for all instances of the same consistency overlap types. As a consequence, it does always define the repair for certain types of consistency overlaps. Furthermore, an individual transformation can be specified for each affected consistency overlap type. This is why we always consider a single transformation to be responsible for one certain consistency overlap type in the following.

Additionally, different changes can violate the constraints of the same consistency overlap type in different ways and thus also require different transformations to repair it. For example, the modifications of the name and the return type of a method in a UML diagram both affect the consistency overlap of that method but require different transformations to restore consistency. One transformation has to update the name of the method in Java code and one has to update its return type. Consequently, we consider a consistency-restoring transformation to be responsible for one kind of repair for one type of consistency overlap.

The actions for restoring consistency can be categorized into three different kinds, which are the update, the addition and the removal of consistency overlaps. The first kind

of action is the *consistency overlap update*, which only updates properties of elements of an existing consistency overlap to make it satisfied again.

A second kind of action is the *consistency overlap addition*, which initiates a new consistency overlap because the actual change makes a constraint of some existing consistency overlap expect such a new overlap. For example, if a method is added to an interface in a UML diagram, the constraints of the interface overlap with the Java code require the method to share an overlap with the Java code as well. As a result, the consistency repair has to create such a method in the Java code and create a correspondence to identify the newly created overlap.

The last kind of action concerns the *consistency overlap removal*, which removes a consistency overlap because of the actual change. If, for example, an interface in the Java code is deleted, the representation in a UML diagram must also be removed if a constraint requires each UML interface to share a consistency overlap with one in the Java code. Consequently, the consistency overlap between the interfaces and also the correspondences identifying it have to be removed after such a change.

The three discussed actions do only describe the general purposes of a consistency repair. How the transformations achieve the repair of consistency constraints in detail depends on individual needs. For example, deleting an interface in a UML diagram also removes the consistency overlap with the corresponding interface in the Java code. Nevertheless, it is up to the methodologist to define if the transformation deletes the Java interface as well, or if they only remove the consistency overlap. Both actions are sufficient for restoring consistency between the models.

From now on, we talk about the general repair of consistency overlaps, no matter which kinds of actions are required. If necessary, we distinguish between the different kinds of actions explicitly.

4.4.4 Structure of Transformations

As stated in the previous subsection, we assume a single change-driven consistency-restoring transformation to define one kind of repair for one type of consistency overlap. The transformation has to decide if the actual change affects an instance of the consistency overlap type it is responsible for and to perform modifications for restoring its constraints. This procedure can always be separated into three steps that are visualized in Figure 4.11. First, the transformation has to check if the actual change performed a modification that may affect a consistency overlap in a way that the transformation restores it. Second, it has to identify if the modified element actually shares a consistency overlap which the transformation can restore. Third, the transformation has to repair the consistency overlap.

The necessity of and difference between the three steps is clarified using the example relation between components within a PCM model and the Java classes implementing them. This consistency overlap type prescribes that each component must have a corresponding class with the same name, while a class may have a corresponding component or not. As an exemplary transformation, we assume the update of the name of the component whenever the name of the corresponding class changes.

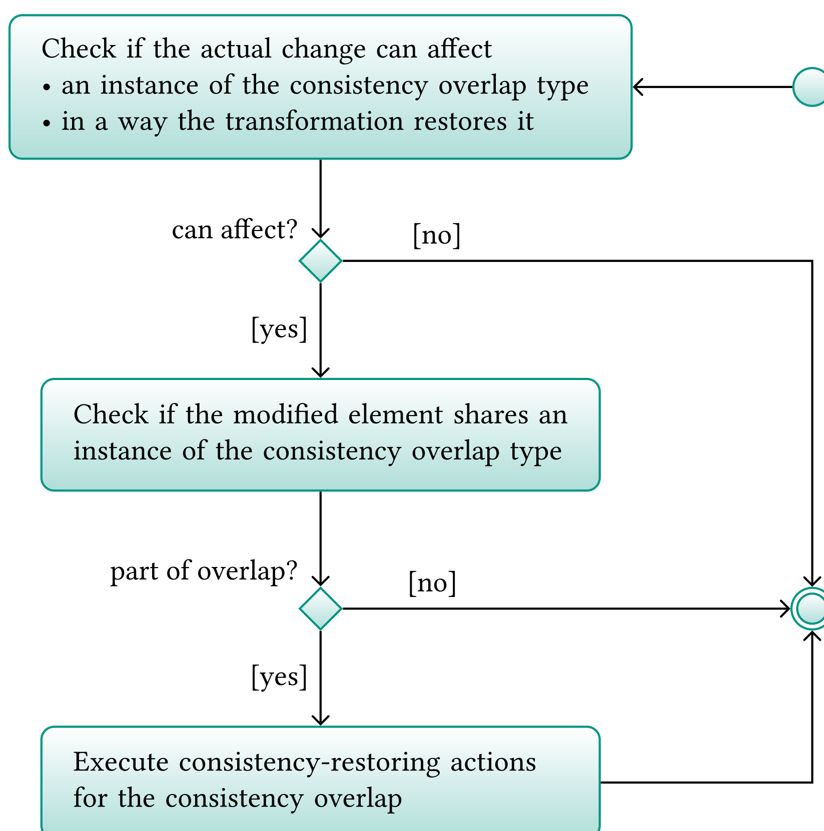


Figure 4.11: Structure of a change-driven consistency-restoring transformation

In a first step, the transformation has to decide if the actual change may describe a modification that leads to the violation of the constraints of an instance of the consistency overlap type for which the transformation is responsible. Because a transformation only describes one kind of repair for one type of consistency overlap, only modifications that can lead to a state which requires this kind of repair have to be further investigated. The transformation can decide this by considering only the information provided by the change, such as the type of the change, the type of the modified element and the actually modified property. Considering the exemplary Java to PCM transformation, in this first step the transformation has to identify if the actual change is the modification of the name of a Java class, which can be decided just based on the information provided by the change. Only if the change describes the replacement of the name attribute of a Java class, the transformation may be responsible for the repair of the consistency overlap.

If the transformation identified that the change performed a modification that may affect an instance of the consistency overlap type in a way that it can restore it in the first step, it continues its execution. In a second step, it has to check if the changed element actually is part of such a consistency overlap that is restored by the transformation. This check is necessary because an element must not necessarily be part of a consistency overlap just because it could be due to its type. Considering the exemplary Java to PCM transformation, not each Java class has a corresponding component. It can also be a simple class sharing no consistency overlap with a component. This information cannot be retrieved from

the change within the first step, but the correspondence model has to be investigated for getting that information.

The last step of a transformation comprises the modifications it performs to restore consistency. It is executed if the first two steps reveal that the modified element is actually part of a consistency overlap that was affected by a change in a way that this transformation can restore it. Necessary modifications may be the creation and removal of elements, the appropriate update of the correspondence model, and the modification of the elements of the consistency overlaps to achieve the satisfaction of the constraints of the consistency overlap. These actions were discussed in more detail in the previous subsection 4.4.3.

These three proposed steps must be always performed by a change-driven consistency-restoring transformation. The transformation cannot perform any modifications for restoring consistency without checking if the change may have affected an instance of the consistency overlap type that the transformation restores as nothing about the change is known and even the modified element is unknown. Without checking if the modified element shares a consistency overlap of the type that the transformation restores, it cannot perform any modifications. The other elements of the consistency overlap that would have to be modified cannot be addressed. Finally, without any consistency repair description, the transformation does nothing and is useless.

It is reasonable to clearly separate these three steps. The first step is necessary to get to know if the transformation is responsible after the actual change and to get the modified element at hand. Consequently, it has to be performed before the second step. The second and the third step have to be separated because the former one only retrieves information from the models while the latter performs modifications on them. Because the transformation can be aborted if the information retrieval reveals that there is no consistency overlap to repair, no modifications should be already performed to avoid the partial execution of the transformation. The clear separation of these steps can, for example, be assured within special languages for specifying such transformations. Those languages are discussed in the following.

4.4.5 Change-Driven Consistency Repair Languages

A change-driven model transformation can be defined by directly writing program code that repairs inconsistencies and that is conform to the interface that is required by the transformation environment as described in section 4.3. It can be registered at the execution engine for a specific change type and gets executed whenever a change of that type occurs. Further preconditions on the change, which restrict the cases in which modification in the models are performed, can be checked during the execution of the transformation and possibly abort it.

Certainly, the specification of model transformations can be simplified using transformation languages, which generate the final transformation code from more abstract specifications. Transformation languages are specialized DSLs that provide constructs abstracting from technical and recurring necessities. Such languages automatically integrate the defined transformations into the execution environment. A specialized language for change-driven consistency-restoring transformations can provide constructs for defining their triggers, based on the actual change, for providing access to the correspondence

model and for performing certain recurring actions for restoring consistency constraints. Providing such language constructs, the methodologist does not have to know about the internal realization of these concepts.

Additionally to the provision of appropriate language constructs, a consistency repair language can enforce a reasonable structure for transformations. A clear separation of necessary steps of a transformation can be achieved with the separation into three steps, according to subsection 4.4.4. This structure can be prescribed by providing three appropriate sections for a transformation specification in a DSL, which can be only used according to their purpose. Such a structure gives the methodologist a clearer understanding of the different actions that have to be performed in a transformation and of their dependencies. It also avoids unexpected effects of a transformation, for example, by performing modifying actions before completing all necessary checks and thus executing the repair logic partially.

5 The Response Language

In this thesis, we develop the concepts for a change-driven, unidirectional model transformation language for repairing model consistency. As the transformations define responses to the occurrence of specific model changes, the language is called the *response language* or short *responses*. This chapter introduces the basic idea and structure of the language, as well as the concepts of the three parts of which it consists. Finally, we discuss further cross-cutting aspects of the language, before giving an overview of possible extensions.

5.1 Introduction to the Response Language

The response language contributes to the MIR language family of the Vitruvius framework, which comprises two transformation languages. The mapping language aims to provide the simple and highly declarative, but potentially functionally restricted definition of consistency overlap types and the preservation of their constraints, whereas the response language focuses on expressiveness. Specifications in the mapping language are bidirectional and are translated into transformations for restoring consistency in both directions. The response language requires the specification of unidirectional rules, which provides a higher flexibility if consistency has to be restored differently in both directions.

The goal of the response language is to allow the convenient specification of model transformations that can restore any kind of model consistency constraint. Such transformations can be written manually in a Turing-complete programming language. Starting with this insight, the response language abstracts from technical aspects of model transformations and from recurring actions. A first simplification in contrast to manually written transformations is the integration of transformations into the transformation environment, which is automated by the language. Furthermore, responses enforce a three part structure of the consistency repair specification to provide a clear separation of concerns and to prevent unexpected behavior that can result from mixing up queries and modifications. Finally, the access to the correspondence model, as well as the creation and deletion of elements and correspondences are abstracted by the language. Nevertheless, the language design maintains the topmost goal of providing maximum expressiveness in the specification of transformations.

5.1.1 Language Structure

In subsection 4.4.4, the separation of a change-driven transformation into three steps was proposed. This separation is transferred to the basic structure of the response language, which consists of three major parts. The first part are the *triggers*, which specify in response to which change events consistency is restored. The second part are the *matchers*, which

identify if the modified element is actually part of a consistency overlap that is restored by the transformation. The last part addresses the *effects* of responses, thus it provides a way of specifying the actual transformation.

The trigger of a response specifies in which cases a transformation shall be executed, based on the information provided by the actual change. This information consists of the change type, the type of the modified element, the modified feature and the type of the feature value. Moreover, the user who modifies the model can be asked for decisions based on the information provided by the change. On the basis of that information, a trigger checks if the modified element may be part of a consistency overlap for which type the response specifies the consistency repair. This can, for example, be identified by checking if the metaclass of the modified element is contained in the consistency overlap type. Besides, it checks if the change violates the consistency overlap in a way that the response specifies the correct repair for. This check can, for example, be based on the modified feature and the change type. Finally, the trigger indicates whether the response shall be executed further or not.

The matcher of a response does also define a precondition for the execution of the transformation. While the trigger makes a decision only based on the change, the matcher investigates the actual model states. The trigger can only decide if the modified element may be part of a consistency overlap based on type information. In contrast, the matcher especially investigates the correspondence model to identify if the modified element is actually part of a consistency overlap which the response repairs. Because the matcher retrieves elements from the correspondence model for making its decision, we integrate the retrieval of the elements of the repaired consistency overlaps or overlaps it depends on from the correspondence into the matcher as well. To sum up, the matcher identifies if the changed element is part of a consistency overlap that the response repairs and provides the existing elements of that consistency overlap to the effect of the response.

After the trigger and matcher have checked the preconditions for the execution of the transformation, the effect of a response specifies the actual modifications. We rely on the assumption made in section 4.4 about change-driven consistency that consistency between two models can be preserved by transforming atomic changes in one model into changes within the other model. Consequently, the effects part must have access to the elements of the consistency overlap that it repairs, and which were retrieved by the matcher. To restore the satisfaction of the consistency overlap, it performs modifications in the correspondence model and in the model elements of the consistency overlap for restoring its constraints.

5.1.2 Basic Language Constructs

This chapter focuses on the introduction of the structure and concepts of the response language, which are explained with examples to clarify their purpose and potential realization. Therefore, an exemplary realization of the response language is developed along with the concepts. This exemplary realization is aligned with the final language implementation provided in chapter 6, but abstracts from technical restrictions and requirements that are relevant within that implementation.

According to subsection 2.1.3, a DSL consists of an abstract syntax, static and dynamic semantics and one specific concrete syntax. The abstract syntax of our language is described conceptually along with the rationale that leads to these concepts. The example language provides a concrete syntax, which is specified using a variant of the Extended Backus-Naur Form (EBNF) [48]. Repetitions are indicated by curly braces, and optional parts are enclosed in square brackets. Static and dynamic semantics of the language constructs are explained textually. The final language implementation defines its dynamic semantics by the operationalization of the transformations into Java code.

Some examples for the developed constructs given in the following sections specify complete responses. Those examples require a specification of the top-level structure of the language. Furthermore, the provided language relies on the existence of several language constructs that highly depend on the implementation of the language. They are only explained in natural language, which is not conform to EBNF. An example for such constructs is the referencing of metaclasses. The top-level language structure and assumed constructs are sketched within the grammar in Listing 5.1.

The *response* rule specifies the basic construct for defining a response. It has an identifier specifying its name and provides a trigger, a matcher, and an effect. The *id* element allows the specification of unique identifiers, for which we use simple strings without whitespaces in the examples. A *code-block* allows the specification of program code in a Turing-complete language. The static semantics have to specify to which elements the code has access and what it returns. In our examples, we use Xtend code, introduced in subsection 2.3.3, for the specification of code blocks. Code blocks that consist of multiple statements are enclosed in curly braces.

An *element-reference* defines the reference to an element, which can be realized by a code block returning an element or through the specification of an identifier. We use code blocks to define element references. The elements that can be referenced in a code block of a certain language construct have to be defined by its static semantics.

The *element-type* and *element-feature* rules specify the reference to a metaclass or a feature of a metaclass of any involved metamodel, respectively. The element type is assumed to be referenced by a metamodel identifier and the metaclass name. In the examples, the PCM metamodel is referenced by the identifier `pcm` and the Java metamodel

⟨*response*⟩ = 'response:' ⟨*response-name*⟩
⟨*trigger*⟩ ⟨*matcher*⟩ ⟨*effect*⟩
⟨*response-name*⟩ = ⟨*id*⟩
⟨*id*⟩ : unique identifier for elements
⟨*code-block*⟩ : block of code in a Turing-complete language
⟨*element-reference*⟩ : reference to a model element
⟨*element-type*⟩ : reference to a metaclass of an involved metamodel
⟨*element-feature*⟩ : reference to a feature of a metaclass of an involved metamodel

Listing 5.1: Basic language constructs of the response language

is referenced by the identifier `java`. As an example, the PCM component metaclass is referenced by `pcm.Component`. The referenceable elements have, again, to be restricted through the static semantics when using the construct within the language.

The examples in the following section either specify complete or only partial responses, in terms of syntactically missing parts of a response. Several examples provide specifications of parts of the PCM to Java relations introduced in chapter 3. Those examples may be semantically incomplete and omit parts of the relation that they implement.

Keywords of the exemplary concrete syntax of the developed language are colored red in the examples. Keywords of the Xtend language, which is used for the code blocks, are colored blue, and the properties of the change that invoked the response are colored green. In Listing 5.2, the coloring is demonstrated. The keyword `response` belongs to the concrete syntax of the response language, `newValue` is assumed to be a property of the change and `instanceof` is a keyword of the Xtend language.

```
response: ExampleResponse
code block example: newValue instanceof pcm.Repository
```

Listing 5.2: Demonstration of the keyword coloring in the examples

5.1.3 A Metamodel for Change Descriptions in Ecore

When considering changes within the trigger of a response, the type and properties of this change are of special interest. The possible changes in a model are restricted by the meta-metamodel as discussed in subsection 4.2.2. The example implementation in this thesis is based on the Ecore meta-metamodel, for which reasonable atomic changes were developed in subsection 4.2.4.

The distinguished atomic changes comprise the creation and deletion of class instances, the insertion, removal and permutation of multi-valued attributes and references, and the replacement of single-valued attributes and references. The final response language implementation uses a complete metamodel for atomic changes in Ecore provided by the Vitruvius framework. For our examples, we use a reduced metamodel as shown in Figure 5.1.

The provided change descriptions metamodel defines one metaclass for each of the possible change types in Ecore-based models, except for the permutation of multi-valued properties as they are rarely needed. Attribute and reference changes each have an abstract superclass, AttributeChange and ReferenceChange, which both contain only the feature that was modified by the change. They, in turn, have a superclass called FeatureChange, which provides the object that contains the modified feature. All change types inherit the generic AtomicChange.

The presented change descriptions metamodel is a highly simplified version of the Vitruvius metamodel for change descriptions. The latter one provides recurring elements, like the newValue and oldValue properties, by separate classes instead of specifying them redundantly in the subclasses. Nevertheless, the presented metamodel is sufficient for explaining the concepts of the developed language and providing examples.

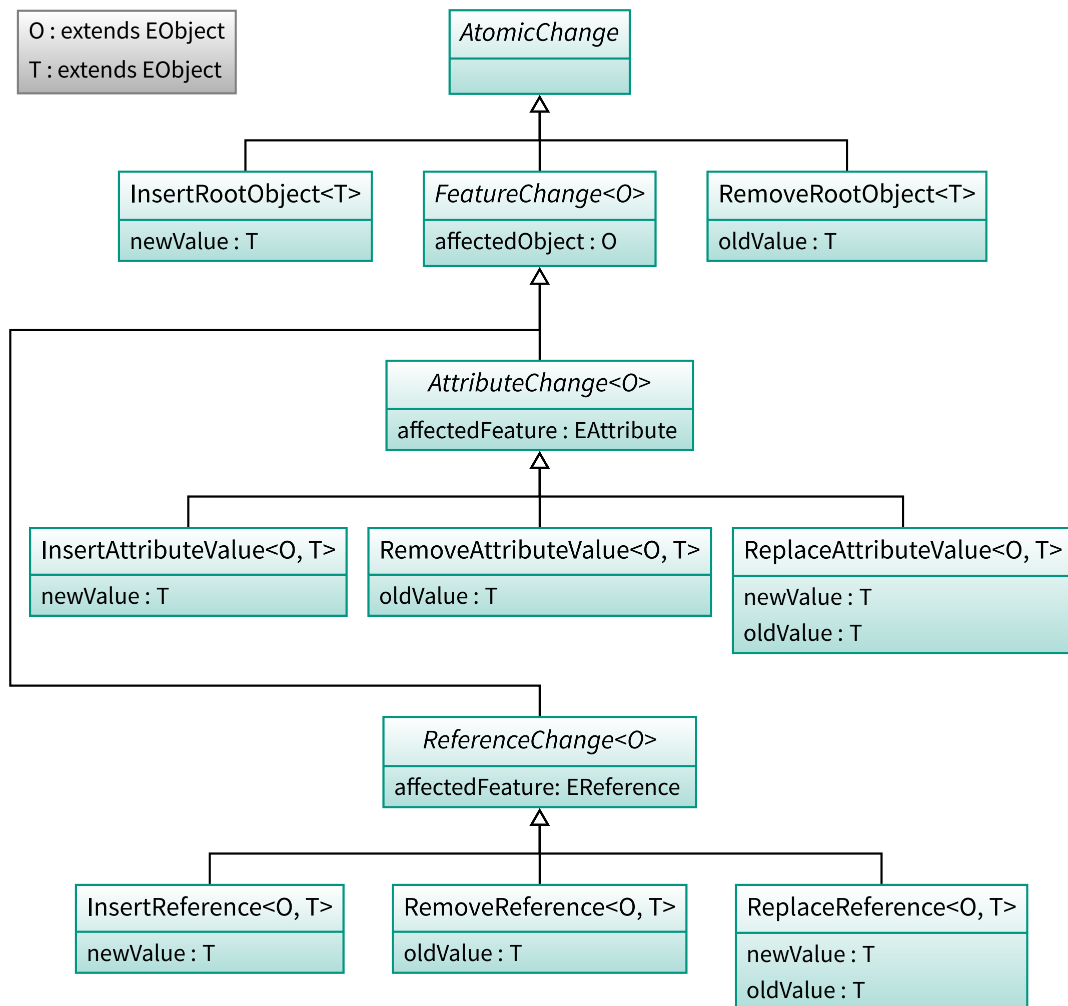


Figure 5.1: A metamodel for descriptions of atomic changes in Ecore

5.2 Triggers: Identifying Potential Inconsistencies

Triggers are the first of three parts of the response language. They represent the entry point for the execution of transformations by determining if the occurrence of a change may require the execution of the transformation just by considering the change, its type and its properties.

We motivate the specification of triggers with a continuous example from the PCM to Java relations. As introduced in section 3.3, a PCM repository shares a consistency overlap with three Java packages. Modifications of a repository element affect this consistency overlap, which is why triggers for such modifications have to be specified in consistency-restoring transformations. A repository represents a root element of a PCM model, so the InsertRootObject and RemoveRootObject changes of our change descriptions metamodel are relevant for the example.

As a response defines a specific kind of repair for a specific type of consistency overlap, its trigger has to check two kinds of preconditions. First, it has to identify if such an overlap may got affected by the change. This identification can be achieved by examining

the change properties, for example, by checking if the modified element can share such an overlap because the overlap type contains the metaclass of the element. In our example, the trigger must check if the modified element is of type Repository. Second, it has to determine if the consistency overlap was affected in such a way that the repair routine fits for its restoration. This is based on the type of the change and its properties. The insertion and the removal of a repository, for example, require different repair routines.

The change type, as one of the restrictions that a trigger has to specify, can only be one of the metaclasses of the used change descriptions metamodel. Therefore, we initially specify a trigger statement for the change type restriction. The preconditions on the change properties can be more complex, which is why we introduce a general precondition specification approach for the change properties at first in the following. Some change property restrictions are common and recurring, such as the type of the concrete modified element and potentially the changed feature. As a result, we develop compact statements for them afterwards. With further property restrictions, a more simplified specification of triggers is developed, which infers the concrete triggers to react to.

5.2.1 Specifying Change Type and General Preconditions

A response specifies a routine for restoring consistency of a specific consistency overlap type. Whenever a change is performed, consistency overlaps that contain the changed element or rely on a specific property of the element can get inconsistent. Consistency overlap types specify the metaclasses and the constraints for concrete consistency overlaps. Therefore, information that is provided by the change, such as the metaclass of the modified element, can be used to decide whether a consistency overlap of a specific type may got violated in consequence of the change. In our example, we consider modifications of a PCM repository, thus the metaclass of the modified element has to be Repository.

The change properties provide information about the actual modification but not about the model states. A change especially describes which type of element and potentially which feature of the element was modified. This identifies if consistency overlaps of the type that the response repairs can be affected. Dynamic model information, such as property values of the modified element or the correspondence model, are not considered in the trigger. For example, a Java package may share a consistency overlap with a PCM repository and thus require the execution of a specific response when it gets changed, but not all Java packages correspond to a repository. This information cannot be retrieved from the change, but from the correspondence model.

Furthermore, a response defines a specific kind of repair for a consistency overlap type. The change type gives information about the kind of modification and thus influences the required operations to restore the constraints of the affected consistency overlap. In our example, the insertion of a repository requires the three corresponding packages to be created and to be initialized with correct values. After a removal of a package, the corresponding packages have to be deleted. Both changes affect the same type of element, but the different change types require different repair specifications.

Consequently, each response requires the specification of the change type it reacts to. The possible types are defined by the used change descriptions metamodel. As a result,

the trigger must only provide the ability to specify one of those types for declaring the restriction of the response execution to the occurrence of changes of that type.

The restriction of the change type of a trigger defines the accessible change properties, which further preconditions can be based on. A completely tailored, declarative approach for restricting the change properties can potentially not be applied in certain cases. To avoid that non-applicability, we provide a Turing-complete, and as such maximum expressive construct for specifying preconditions on the change properties. Therefore, we allow the definition of a code block that returns a Boolean value. The return value indicates whether the response shall be further executed or not. Expressions in the code block have access to the change properties to define preconditions based on their values.

A concrete syntax for the realization of the proposed trigger constructs is shown in Listing 5.3. The element *change-type* specifies the type of change to react to and can be any change type that is specified in the change descriptions metamodel. The *precondition-block* allows to definition a general precondition. It has access to the actual change properties and returns a Boolean value. The usable change properties are restricted by those available in the specified change type. To identify the precondition block, it is preceded by the keyword *check*.

The operationalization of this trigger initially checks if a given change is of the specified type. Afterwards, it executes the defined statements of the precondition block, to which the correctly typed change is provided.

```
⟨trigger⟩             = 'trigger:' ⟨change-type⟩ ⟨precondition-block⟩
⟨precondition-block⟩  = 'check' ⟨code-block⟩
⟨change-type⟩         : change type from the used change descriptions metamodel
```

Listing 5.3: Language extension for triggers with change type specification and general preconditions

The realization of a trigger for the creation of a new PCM repository according to the given syntax is defined in Listing 5.4. It specifies the correct change type to react to and restricts the execution to cases in which the modified element of the change, provided by the newValue element in our change descriptions metamodel, is a repository.

```
trigger: InsertRootObject
  check newValue instanceof pcm.Repository
```

Listing 5.4: A trigger for reacting to the creation of a PCM repository

5.2.2 Restricting Change Properties

The proposed initial trigger specification allows to define all necessary types of restrictions to the change type and its properties. Nevertheless, the property restrictions have to be specified in an imperative code block although most types of preconditions can be provided in more compact, declarative statements.

In general, the execution of a response depends on the modified element, as it can be seen in the repository creation example in Listing 5.4, which restricts the type of the modified element to Repository. Each atomic change specifies the modification of exactly one element, as stated in the definition of atomic changes in subsection 4.2.2. In the case of the insertion or removal of a root element, it is the element itself. In the case of a property modification, the element that holds the property is modified. Therefore, a trigger can provide a construct that allows the specification of the type of the modified element.

Atomic changes are either root element insertions or removals, or modifications of features. Consequently, most changes are feature changes and as such provide information about the modified feature in addition to the modified element. In our exemplary change descriptions metamodel, the modified feature is supplied by the affectedFeature property of all FeatureChange instances. This information has to be restricted in the precondition of most transformations because depending on the modified feature, different consistency overlaps can be affected and require a different kind of repair. For example, the insertion of an interface into a PCM repository requires the creation of a Java interface, whereas the insertion of a component requires the creation of a new Java class. Both changes are described by an InsertReference change with the repository as the changed model element and are only distinguishable by the modified feature.

The restriction of the changed element and the modified feature are realized in the response language as shown in Listing 5.5. Additionally to the already added specification of the change type, the restriction is extended by the element type and feature. The specification of the modified feature is optional as it depends on whether the change is a feature change or not. This restriction cannot be expressed in the grammar but must be ensured by further well-formedness constraints. The precondition block is declared as an optional element because most preconditions can be expressed by the other constructs.

⟨*trigger*⟩	=	’trigger:’ ⟨*atomic-change*⟩ [⟨*precondition-block*⟩]
⟨*atomic-change*⟩	=	⟨*change-type*⟩ ’of’ ⟨*element-type*⟩ [’[’⟨*element-feature*⟩’]’]
⟨*precondition-block*⟩	=	’check’ ⟨*code-block*⟩
⟨*change-type*⟩	:	change type from the used change descriptions metamodel

Listing 5.5: Language extension for triggers with change property restrictions

The operationalization of the new language constructs compares the type of the change property that represents the modified element with the expected type. Therefore, it selects the correct change property in each case, which depends on the actual change type. Our change descriptions metamodel provides the changed model element as the newValue in the insertion, and as the oldValue in the removal of a root object. In a feature change, it is provided as the affectedObject. In the case of a feature change, the modified feature is compared to the expected one after comparing the element type.

An example trigger using the new constructs is provided in Listing 5.6. It shows how the restriction of a change to the insertion of a data type in a repository can be defined. The trigger is specified to react to changes of the type InsertReference with the changed model element Repository and the modified feature datatypes.

```
trigger: InsertReference of pcm.Repository[datatypes]
```

Listing 5.6: A trigger for reacting to the insertion of a PCM datatype

Another benefit of the proposed declarative statements is the possibility to let the language implementation perform further consistency checks. Some combinations of model elements or change types and features do not represent a meaningful trigger for a response because they can never occur. First, only the specification of a feature that is provided by the expected model element is reasonable. Second, only a feature that may be affected by a change of the specified type should be specifiable. For example, an attribute change of a reference property or vice versa can never occur.

Reacting to the creation and deletion of non-root elements has to be specified through an appropriate feature change of the container of the element. We do not provide a construct to specify the creation or deletion of an element of a specified type no matter where it is contained. Although such a construct may appear useful, it is unnecessary according to our definition of consistency. The existence of a consistency overlap is always induced by the constraints of another one, which are defined on the properties of the elements of that consistency overlap. Consequently, a modification of a property may induce the creation or removal of a consistency overlap, but not the simple creation or deletion of an element no matter where it is contained. For example, the consistency overlap of a PCM interface and its Java representation is induced by its insertion into the repository. The repository consistency overlap requires its interfaces to share a consistency overlap with interface representations in the Java code, but not by the creation of the interfaces themselves.

5.2.3 Change Type Generalization

The restriction of change types and properties as presented in the previous subsections relies on our concrete change descriptions metamodel. However, the specification of a change type together with the modified element and feature, like exemplarily shown in Listing 5.6, can be further simplified by generalizing from the given change descriptions.

Instead of allowing the specification of concrete change types of our change descriptions metamodel, we provide the specification of more generic change types. Two reasons justify this decision. First, fewer generic changes are needed because together with the specified element feature, different concrete changes can be inferred. Second, we abstract from the concrete realization and design decisions of the change descriptions metamodel. The transformation developer must only consider the more generic types and their properties. The generic changes inherently define a new change descriptions metamodel that is transformed into the underlying concrete one.

The possible reduction of the number of different change types arises from the duplication of change types for reference and attribute changes in our change descriptions metamodel. Their only difference is that the affected feature is once an `EAttribute` and once an `EReference`. Nevertheless, a trigger requires the specification of the feature that is expected to be modified, from which it is clear whether the modification of an attribute or a reference is expected. We combine the duplicate change types for attributes and references into one type and infer the concrete change from the type of the specified feature.

Generic change type	**Feature property**	**Inferred change type**
Insert in list	Is attribute	InsertAttributeValue
	Is reference	InsertReference
Remove from list	Is attribute	RemoveAttributeValue
	Is reference	RemoveReference
Replace value	Is attribute	ReplaceAttributeValue
	Is reference	ReplaceReference
Insert root	*None*	InsertRootObject
Remove root	*None*	RemoveRootObject

Table 5.1: Inference of concrete change types from generic changes and feature properties

That combination reduces the number of specifiable change types without reducing the expressiveness. Furthermore, the language must not explicitly prevent the specification of unfulfillable combinations of change type and specified feature any more, as they cannot be specified by design. The generic change types and their relation to concrete changes of our change descriptions metamodel are shown in Table 5.1. The reduced set of change descriptions is conform to the one proposed for EMOF in subsection 4.2.3.

The generic change types can be assigned to three categories with each having different demands on the specified features. The first category are insertions and deletions of root elements, which do not require the specification of a modified feature at all. The second category are modifications of single-valued features, which can be addressed by the *replace value* change type and require the specified feature to be single-valued. Finally, the third category comprises the insertion and removal of elements in a multi-valued feature and as such these changes require the given feature to be multi-valued.

An integration of these generic change types into the triggers of the response language is provided in Listing 5.7. In contrast to the previous realization of the change type definition, the new grammar defines the more generic change types within the language and does not refer to a change descriptions metamodel any more. The syntactical distinction between root element changes and feature changes allows to condition the specification of an affected feature on the grammar rule.

```
⟨trigger⟩              = ⟨atomic-change⟩ 'check' ⟨precondition-block⟩
⟨atomic-change⟩        = ⟨root-change⟩ | ⟨feature-change⟩
⟨root-change⟩          = ('insert root' | 'remove root') ⟨element-type⟩
⟨feature-change⟩       = ('insert in list' | 'remove from list' | 'replace value')
                         ⟨element-type⟩'['⟨element-feature⟩']'
⟨precondition-block⟩   = ⟨code-block⟩
```

Listing 5.7: Completed trigger grammar

An example trigger according to the updated grammar is defined in Listing 5.8. The trigger restricts the execution of the response to cases in which the change is the insertion of a composite data type into a PCM repository. As the datatypes feature of the Repository is multi-valued, the combination with the generic insert in list change type is valid. The operationalization infers an InsertReference because the feature is a reference.

```
trigger: insert in list pcm.Repository[datatypes]
  check newValue instanceof pcm.CompositeDataType
```

Listing 5.8: A trigger for the insertion of a composite data type into a repository

As the example shows, the restriction of the feature value type, in this case the data type, has to be performed in the general precondition block. To simplify this restriction, a language feature for restricting the expected feature value type could be added. However, many transformations do not consider the type of the feature value because in cases in which the type is relevant, the metamodel usually defines an individual reference for each type. The remaining cases can be realized by a restriction within the general precondition block.

5.3 Matchers: Retrieving Elements of Consistency Overlaps

The second of three parts of the response language are the matchers. The purpose of a matcher is to decide if the effect of a response shall be executed based on the current model states and correspondences.

Although triggers as well as matchers define preconditions for the execution of the effect of a response, they have distinct purposes. A trigger defines the preconditions for the execution of a response based on the change properties. In contrast, a matcher examines if the modified element is actually part of a consistency overlap that is restored by the response. Therefore, it investigates dynamic model information, such as property values of the modified element and the correspondence model. We identify if an element is part of a consistency overlap by the existence of certain correspondences. For example, checking if a Java class has a correspondence to a component identifies if the class shares a consistency overlap with a component.

The check if a certain element shares a consistency overlap requires to investigate the correspondence model for the existence of certain correspondences. After that check, the corresponding elements have to be retrieved for modifying the elements of the overlap to restore consistency. To avoid a duplicate specification of the correspondence retrieval for checking its existence and retrieving the element, we combine both in the matcher.

A matcher specifies how to retrieve the elements of consistency overlaps. If the retrieval fails because the expected correspondences do not exist, this indicates that the element does not share a consistency overlap of the expected type. The execution of a matcher ensures that the unsuccessful retrieval of an element leads to the abortion of the response execution because no consistency overlap that is repaired by the response exists.

The elements that are retrieved by a matcher are provided to the effect of the response, which operates on them. A separation of matchers and effects is necessary because they

have distinct responsibilities. The matcher of a response only accesses the correspondence model for retrieving elements, thus it is free of side-effects. If a matcher detects that no consistency overlap that can be repaired by the response is affected by the actual change, it aborts the response without having any modifications performed yet. Modifications to model elements are only performed by the effect.

In the following, the required statements for the matcher specification are developed. They simplify the access to the correspondence model and abstract from the mechanism to pass the retrieved elements to the effect. Additionally to the basic retrieval of elements from consistency overlaps, a check for the non-existence of corresponding elements is necessary for detecting that a consistency overlap does not exist yet. In certain cases, the existence of corresponding elements within a consistency overlap cannot be ensured and requires an appropriate handling.

5.3.1 Retrieving Corresponding Elements

Correspondences, according to our definition, can exist between sets of elements of arbitrary size. The correspondence model supports such kinds of correspondences. Nevertheless, correspondences between pairs of model elements are sufficient for describing consistency overlaps. Several one-to-one correspondences used within one consistency overlap implicitly define a more complex correspondence. This matches the distinction of explicit and implicit correspondence specifications described in subsection 4.1.2.

The restriction to one-to-one correspondences allows to provide simpler language constructs because it only requires the specification of one element to get a corresponding element for. Otherwise, it would be necessary to allow the specification of arbitrary large sets of elements and to specify conditions on the arbitrary large sets of the expected corresponding elements. As a consequence, the provided language constructs are limited to the handling of one-to-one correspondences.

The retrieval of elements from the correspondence model requires the specification of the element whose corresponding elements shall be received. Furthermore, a model element may have several corresponding elements. To distinguish the corresponding elements by their type, we allow the specification of a type by which the corresponding elements are automatically filtered. To make the retrieved element available in the effect of the response for investigating or modifying its features, it has to be named. An extension of our language syntax that realizes this retrieval action is presented in Listing 5.9.

The *element-reference* of the retrieve statement has to be specified through a side-effect free statement that allows to navigate the complete model starting from one of the elements provided by the change or starting from any already retrieved element. The latter ones are necessary because consistency overlaps can consists of elements that have

⟨*matcher*⟩	=	{ ⟨*retrieve-element*⟩ }
⟨*retrieve-element*⟩	=	’retrieve element:’ ⟨*element-type*⟩ ’as’ ⟨*id*⟩ ’corresponding to’ ⟨*element-reference*⟩

Listing 5.9: Language extension for retrieving an element

no correspondence to the modified element but to another element that corresponds to the changed one. We allow this specification in a code block that has access to the change properties and returns the model element.

The operationalization for such a statement first evaluates the specification of the element reference. It retrieves the corresponding elements of that element from the correspondence model and filters the retrieved elements by their type. The statements are executed in the order in which they were defined because a retrieve statement allows the access the previously retrieved elements.

An example that uses the retrieve statement is given in Listing 5.10. It shows a response that reacts to the renaming of a component in a PCM model and retrieves the Java class to update its name within the effect of the response.

```
trigger: replace value pcm.Component[name]
retrieve element: java.Class as javaClass corresponding to affectedObject
```

Listing 5.10: A matcher for retrieving the corresponding class of a component

Retrieve statements return corresponding elements of a specified type, which can potentially be an arbitrary number of elements. In our approach, we assume that the corresponding element is unique to provide a clearly identified element to the user instead of an arbitrary large set. The statement evaluation fails if the specified element has none or more than one corresponding element of the specified type. In some cases, this assumption does not hold, which requires the further restriction of the corresponding elements to find the unique searched element. These restrictions are discussed in the following.

5.3.2 Restricting Retrieved Elements by Filter Functions

Retrieving corresponding elements from the correspondence model can be simplified for the developer with the construct introduced in the previous subsection. It assumes that a unique corresponding element is retrieved by the statement, which is not always possible by filtering the corresponding elements only by their type. For example, a repository in a PCM model is mapped to three packages in the corresponding Java model, one for the package itself and one each for the data types and the contracts. Retrieving one of them is not possible with the proposed construct and requires further restrictions.

We allow the further restriction of corresponding elements by the specification of a code block that filters the searched elements from the set of corresponding elements. This restriction can be realized in at least two ways. One possibility is a code block that gets the set of corresponding elements as an input and returns the unique element of interest. Another approach is a code block that provides a filter, which gets only one of the corresponding elements and returns whether it is the searched element or not. That function gets executed for each element and may only return true for the searched one.

We choose the latter approach because it allows the direct specification of a constraint that has to hold for a given element and does not force developers to repeat code for iterating and modifying the set. This decision makes the specification shorter and simpler.

An extension for the already introduced retrieve statement is presented in Listing 5.11. It provides a code block for the restriction of the retrieved elements, which is introduced

```
⟨matcher⟩               = { ⟨retrieve-element⟩ }
⟨retrieve-element⟩      = 'retrieve element:' ⟨element-type⟩ 'as' ⟨id⟩
                          'corresponding to' ⟨element-reference⟩
                          [ ⟨retrieve-filter-block⟩ ]
⟨retrieve-filter-block⟩ = 'with' ⟨code-block⟩
```

Listing 5.11: Language extension for filtering retrieved elements

with the keyword *with*. The specification of such a filtering code block is optional because in many cases the restriction by the element type is sufficient.

The operationalization of the specification executes the code block for each corresponding element of the specified type. Thus, the actual element must be accessible within the code block, which we realize by making the element accessible by the name that is specified within the retrieve statement. Furthermore, we ensure that the code block returns a Boolean value.

With this filtering mechanism, the unambiguous retrieval of a specific Java package corresponding to a PCM repository is possible. Assuming that the data types and contracts package have predefined names, they can be identified in the retrieve statement by comparing the name of the actual package with the expected name. For example, the creation of a new interface in a PCM repository requires the creation of a new Java interface within the contracts package, which can be retrieved as shown in the example in Listing 5.12.

```
trigger: insert in list pcm.Repository[interfaces]
retrieve element: java.Package as contractsPackage corresponding to affectedObject
  with contractsPackage.name.equals("contracts")
```

Listing 5.12: A matcher for retrieving an element with a user-defined filter function

5.3.3 Restricting Retrieved Elements by Correspondence Tags

One approach for restricting the set of corresponding elements is the specification of a filter function. That approach allows the specification of arbitrary conditions based on the current model states, which are accessible through the provided change and the correspondence model. However, this restriction is not always sufficient.

For the retrieval of the correct corresponding Java package for a PCM repository, a solution using a filter function that checks the name of the actual package was presented previously. The mechanism assumes that at least two of the three corresponding packages can be uniquely identified by their names. If, in contrast, the contracts and data types packages have no predefined names but an arbitrary name that can be individually specified by the user, the mechanism is insufficient.

Like in the given example, it is not always possible to select the correct corresponding model element only by information provided by the models. In these cases, the identification of the correct element has to rely on the relation between the elements, which is persisted through a correspondence. For example, the correspondences of a

⟨*matcher*⟩	=	{ ⟨*retrieve-element*⟩ }
⟨*retrieve-element*⟩	=	'retrieve element:' ⟨*element-type*⟩ 'as' ⟨*id*⟩ 'corresponding to' ⟨*element-reference*⟩ [⟨*retrieve-filter*⟩] ['tagged with' ⟨*tag*⟩]
⟨*retrieve-filter*⟩	=	'with' ⟨*code-block*⟩

Listing 5.13: Language extension for restricting retrieved elements by correspondence tags

PCM repository to the different Java packages have to provide the information which role each corresponding package has. In consequence, a correspondence needs an identifying property.

While many kinds of properties can be added to a correspondence for identifying the relation, we propose a tagging mechanism that assigns an identifying tag to each correspondence. This mechanism is similar to the one used in QVT [72], in which the transformation rule is assigned to a trace on creation. Our approach relies on an extension of the correspondence metamodel that adds a tag property to the correspondence metaclass.

Tags have to be assigned to correspondences when they get created and have to be checked when corresponding elements are retrieved. Therefore, the creation of a correspondence must allow the attachment of a tag to it, which is an objective of the effects and is thus discussed in the next section. Furthermore, the provided retrieve statement has to be extended by the specification of tags that restrict the considered correspondences. A concrete syntax for such an extended statement is shown in Listing 5.13. The tagging construct is specified as optional because it is only required if the corresponding element cannot be unambiguously identified by model properties, which is not always the case.

The value specification of the tag can be realized in different ways. One possibility is the declaration of tags as external DSL constructs of the response language, which could be realized as enumeration values. A more simple variant is the specification of tags as simple strings, which is the approach that we choose. During the execution of such a retrieve statement, the retrieval of corresponding elements from the correspondence model is extended to only investigate the correspondences that have the expected tag value assigned.

The example for retrieving the contracts package when creating a new PCM interface, which was already realized with a filter function in Listing 5.12, is realized using the tagging mechanism in Listing 5.14 again. We assume that another response attached the specified tag to the contracts package correspondence when creating it in reaction to the creation of a PCM repository. When creating a new PCM interface, the contracts package is retrieved using its tag to create a corresponding Java interface within that package.

```
trigger: insert in list pcm.Repository[interfaces]
retrieve element: java.Package as contractsPackage corresponding to affectedObject
  tagged with "contracts"
```

Listing 5.14: A matcher for retrieving an element using a tagging mechanism

5.3.4 Retrieving Optional Elements

The developed mechanisms for handling correspondences, restricting the retrieved element set, and tagging correspondences target the retrieval of unambiguously identifiable correspondences. The presented approaches should be sufficient for restricting the set of elements that correspond to a given one to the unique expected model element.

The current construct assumes that there is always a set of corresponding elements that contains the one of interest. Actually, there can be situations in which none of the corresponding elements fulfills the restrictions that are made in the retrieve statement. Because the retrieve statement aborts the response execution if no expected corresponding element exists, it has to be extended to allow the handling of such cases. Expected correspondences can be missing due to different reasons. An important reason is the breaking of meta-levels, which is discussed in subsection 5.7.4.

We provide the possibility to declare a retrieve statement as *optional*. It specifies that the response execution is not aborted if no corresponding element can be retrieved and just delivers a null reference. The already existing characteristic of the retrieve statement is indicated by an additional *required* keyword, which clarifies that the successful element retrieval is required for the continuation of the response execution. An extension of the exemplary language is provided in Listing 5.15.

```
⟨matcher⟩               = { ⟨retrieve-element⟩ }
⟨retrieve-element⟩      = 'retrieve' ('optional' | 'required') 'element:'
                          ⟨element-type⟩ 'as' ⟨id⟩
                          'corresponding to' ⟨element-reference⟩
                          [ ⟨retrieve-filter-block⟩ ] [ 'tagged with' ⟨tag⟩ ]
⟨retrieve-filter-block⟩ = 'with' ⟨code-block⟩
```

Listing 5.15: Language extension for retrieving optional elements

A PCM data type shares a consistency overlap with a Java class if it is composite or a collection. It has no corresponding Java model element if it is primitive. The reason is a meta-level break, which is discussed in subsection 5.7.4. Consequently, the retrieval of the corresponding Java class of a general PCM data type can fail, which can be avoided using the *optional* keyword. An example that demonstrates this case is provided in Listing 5.16. The response reacts to the replacement of the inner type of a collection data type within a PCM model and is proposed to replace the old inner type in the Java implementation with

```
response: ChangedTypeOfCollectionDataType
trigger: replace value pcm.CollectionDataType[innerType]
retrieve optional element: java.Class as newInnerDataType
  corresponding to newValue
// create a type reference that refers to the inner data type
// or an appropriate type reference instance for the primitive PCM type
```

Listing 5.16: A matcher for retrieving a potentially non-existent element

the new one. In consequence, the inner type of the collection type must be retrieved by resolving the correspondence of the inner type. Because this type can also be primitive and then has no corresponding Java element, the retrieve statement is declared as optional.

5.3.5 Expecting the Non-Existence of Corresponding Elements

The last subsections dealt with the retrieval of elements from the correspondence model to collect the elements of consistency overlaps. However, the creation of a new consistency overlap can also depend on the non-existence of another one.

When creating a Java class within a package, it is possibly intended to be the implementation of a PCM component. Thus, no other class in the package may be the implementation of a component yet. If another class already implements a component, the package has a correspondence to a PCM component, which disallows the initiation of a new consistency overlap of the created class with a component. Consequently, the continuation of a response execution for the Java class creation depends on the non-existence of a correspondence between the package of the newly created class and a component.

Non-existence checks for correspondences are achieved by reusing the already defined retrieve statement with changed semantics. Instead of expecting the execution of the retrieval to return exactly one corresponding element, it is assumed to return none. The only difference to a retrieve statement is the omission of an identifier specification, as the statement does not return an element that must be referenceable. Listing 5.17 shows the realization in our example language, omitting the retrieve statement to stay short.

⟨*matcher*⟩	=	{ ⟨*retrieve-element*⟩ | ⟨*require-non-existence*⟩ }
⟨*require-non-existence*⟩	=	’require non-existent element:’ ⟨*element-type*⟩ ’corresponding to’ ⟨*element-reference*⟩ [⟨*retrieve-filter-block*⟩] [’tagged with’ ⟨*tag*⟩]
⟨*retrieve-filter-block*⟩	=	’with’ ⟨*code-block*⟩

Listing 5.17: Language extension for requiring the non-existence of a correspondence

The example scenario that motivated the new language construct is realized in Listing 5.18. The matcher of the response checks that the package of the newly created class has no corresponding PCM component.

```
trigger: insert in list java.Package[classifiers]
  check newValue instanceof java.Class
required non-existent element: pcm.Component corresponding to affectedObject
```

Listing 5.18: A matcher expecting the non-existence of a correspondence

5.3.6 Specifying Further Conditions

The introduced constructs of a matcher investigate the correspondence model to decide whether the response effects shall be executed or not. These conditions are essential as

they check the necessary existence or non-existence of consistency overlaps. Nevertheless, further conditions that are based on the model state, and thus belong to the matcher, may need to be specified.

An example for such a condition was implicitly introduced in the previous subsection. If a user inserts a Java class into a package, it may be intended to be a component. This intention has to be requested from the user. Although that request can be technically realized within the code block of the trigger, it ought to be implemented within the matcher. If the package already contains a class that implements a component, the developer should not be asked if the created class realizes a component, as this is not possible. Consequently, the user request should follow the successful non-existence check in the matcher.

Like in the general precondition block of the trigger, the checks in the matcher can be implemented with code blocks, which can check arbitrary preconditions that depend on the model state. Although such a block would be theoretically sufficient as the last statement of a matcher, we allow the specification of multiple code blocks anywhere in the matcher. This realization avoids the unnecessary continuation of the execution of further matcher statements if the evaluation of a user-defined check aborts it.

The completed matcher grammar of the example language is shown in Listing 5.19. In addition to the retrieve and non-existence requiring statements, check blocks can be defined within the matcher. Such a block has to return a Boolean value and has access to the change properties, as well as the already retrieved elements. Furthermore, it has to be free of side-effects as the whole matcher has to be side-effect free.

```
⟨matcher⟩                 = { ⟨retrieve-element⟩ | ⟨require-non-existence⟩ |
                              ⟨matcher-check-block⟩ }
⟨retrieve-element⟩        = ’retrieve’ (’optional’ | ’required’) ’element:’
                              ⟨element-type⟩ ’as’ ⟨id⟩
                              ’corresponding to’ ⟨element-reference⟩
                              [ ⟨retrieve-filter-block⟩ ] [ ’tagged with’ ⟨tag⟩ ]
⟨require-non-existence⟩   = ’require non-existent element:’ ⟨element-type⟩
                              ’corresponding to’ ⟨element-reference⟩
                              [ ⟨retrieve-filter-block⟩ ] [ ’tagged with’ ⟨tag⟩ ]
⟨retrieve-filter-block⟩   = ’with’ ⟨code-block⟩
⟨matcher-check-block⟩     = ’check’ ⟨code-block⟩
```

Listing 5.19: Completed matcher grammar

5.4 Effects: Restoring Consistency

The third and last part of the response language are the effects. While triggers and matchers restrict the cases in which a response is executed, the effects specify how consistency overlaps are repaired. Therefore, an effect modifies the elements of a consistency overlap and the correspondence model to restore the satisfaction of the consistency overlap constraints.

An effect repairs a consistency overlap whose elements are provided by the previously executed matcher. It is the only part of a response that performs side-effects by modifying the involved models and the correspondence model. Therefore, an effect has to be executed completely or not at all. Preconditions for the execution of an effect have to be checked within the trigger or the matcher of the response. This separation of concerns avoids inconsistencies by design because partially executed transformations are impossible.

The modifications that an effect performs affect the elements provided by the matcher, the correspondence model, and potentially newly created elements. The effects could be realized through an imperative code block that has access to the mentioned artifacts. We develop the effects starting with this insight.

From that starting point, further language constructs, especially for simplifying modifications of the correspondence model, are derived. Because the correspondence model has a simple and well-known structure, it is possible to embed potential modifications to it in easily understood language constructs. The constraints that a consistency overlap has to fulfill can be far more complicated, thus they are hard to simplify by dedicated language constructs and are therefore suggested to be repaired within imperative code.

5.4.1 Specifying General Effects

An effect defines modifications of model elements with the purpose of restoring the satisfaction of a consistency overlap they share. These model elements consist of those retrieved by the matcher, those provided by the properties of the change and potentially newly created elements.

The least restricted way of defining modifications within a model is possible in a Turing-complete language. To provide a language with maximum expressiveness for restoring model consistency to the developer, the response language provides an code block to specify the effects of a response. We call this block the *execution block*. It is introduced by the keyword *execute*, as the language grammar extension in Listing 5.20 shows.

```
⟨effect⟩             = ⟨execution-block⟩
⟨execution-block⟩    = ’execute:’ ⟨code-block⟩
```

Listing 5.20: Language extension for effects specified in a code block

Within the execution block, the properties of the actual change, the elements retrieved by the matcher, and the correspondence model are accessible to perform reasonable changes. We make the properties of the actual change accessible by their names as specified in the change descriptions metamodel introduced in subsection 5.1.3. The model elements retrieved in the matcher are accessible by their names defined in the retrieve statements.

The execution block is specified and executed at the end of a response, after any further language constructs that are introduced in the following. The execution block has no return value that further language constructs can use. Consequently, the expressiveness of the language would not increase if further statements could be specified after the execution block or if more than one execution block could be defined.

According to subsection 4.4.3, three kinds of actions can be performed depending on the actual change. First, an existing consistency overlap can be restored. Second, an existing consistency overlap can be removed by removing its elements. Third, a new consistency overlap can be initialized by creating new elements, adding correspondences between them and making them consistent. It is up to the developer to define by which concrete actions the satisfaction of a consistency overlap is restored.

For any modification of a consistency overlap, its own elements and possibly elements of overlaps it depends on must be accessible. In our case, the existing elements are already retrieved by the matcher and provided to the effect. Additionally, new elements may need to be created, especially due to the induction of a new consistency overlap. While the modifications of model elements that are required for restoring consistency deeply depend on the actual constraints of a consistency overlap, the creation and deletion of elements as well as the handling of correspondences is similar in all responses. This is why those operations are addressed by language constructs in the following.

5.4.2 Creating Model Elements

A common action in consistency repair is the creation of new model elements. Especially if a model change requires the initiation of a new consistency overlap, new model elements have to be created and inserted into the models. For example, the insertion of a PCM component into a repository requires a corresponding Java class to be created because the consistency constraint of the consistency overlap of the repository requires each component to share such a consistency overlap.

A create statement must only specify the type of the element to create along with an identifier to make it accessible. Only concrete metaclasses are allowed to be specified, as only non-abstract metaclasses can be instantiated. The realization of that statement in the response language is shown in Listing 5.21.

⟨*effect*⟩	=	{ ⟨*create-element*⟩ } ⟨*execution-block*⟩
⟨*create-element*⟩	=	'create element:' ⟨*element-type*⟩ 'as' ⟨*id*⟩
⟨*execution-block*⟩	=	'execute:' ⟨*code-block*⟩

Listing 5.21: Language extension for creating model elements

An example that makes use of the statement is presented in Listing 5.22. The response in the example reacts to the creation of a component in the repository of a PCM model. The create statement specifies the creation of a Java class with which the component has to share a consistency overlap.

```
trigger: insert in list pcm.Repository[components]
create element: java.Class as javaClass
// Initialize class and add correspondence to component
```

Listing 5.22: A response that creates a Java class after a PCM component creation

5.4.3 Adding and Removing Correspondences

Accessing the correspondence model for creating new correspondences and removing existing ones is an essential task of an effect. Especially if a new consistency overlap is introduced due to the insertion of a new model element or if a consistency overlap shall be removed because a model element is deleted, the correspondence model must be updated to represent this modification. As already proposed for the retrieval of elements within a matcher, the handling of correspondences in the effects is realized through special language constructs. They abstract from the access to the correspondence model and reduce the required knowledge about the technical realization, the structure and the accessibility of the correspondence model.

As already motivated for the retrieval of corresponding elements in subsection 5.3.1, we do only support one-to-one correspondences. Therefore, only language constructs for creating and removing correspondences between two elements are provided.

The addition and removal of a correspondence requires only the declaration of the elements between which a correspondence shall be established or removed. In subsection 5.3.3, tags for correspondences were introduced for identifying corresponding elements not only by their properties but also by the context in which they were created. A tag has to be attached to a correspondence when it gets created, which is why they can be specified in the statement for the correspondence addition. The specification of a tag is optional because a tag is only needed in few cases in which the model state information is not sufficient for unambiguously retrieving the expected element. A concrete syntax for the correspondence handling is shown in Listing 5.23.

The referenceable elements in a correspondence addition or removal can be restricted to the ones created in the effect and those provided by the change properties or the matcher. The restriction is valid because the effects do only modify the consistency overlap that consists of those elements. To provide maximum flexibility, we allow the specification of a code block that returns an element reference and has access to the mentioned elements.

Elements created in the effects potentially have to be added to new correspondences, but the element creation does not rely on any modifications of the correspondence model. As

```
⟨effect⟩                 = { ⟨create-element⟩ }
                           { ⟨add-correspondence⟩ | ⟨remove-correspondence⟩ }
                           ⟨execution-block⟩
⟨create-element⟩         = 'create element:' ⟨element-type⟩ 'as' ⟨id⟩
⟨add-correspondence⟩     = 'add correspondence between'
                           ⟨element-reference⟩ 'and' ⟨element-reference⟩
                           [ 'tag with' ⟨tag⟩ ]
⟨remove-correspondence⟩  = 'remove correspondence between'
                           ⟨element-reference⟩ 'and' ⟨element-reference⟩
⟨execution-block⟩        = 'execute:' ⟨code-block⟩
```

Listing 5.23: Language extension for creating and removing correspondences

a result, the statements are ordered accordingly. The operationalization of these constructs either creates a correspondence between the specified elements and then adds it to the correspondence model, or it removes the correspondence between the specified elements from the correspondence model, if existing.

In Listing 5.24, an example using the construct for adding correspondences with the tagging mechanism is provided. It specifies the creation of the contracts and data types packages in the Java model corresponding to a newly created PCM repository, as well as the addition of the required correspondences. To unambiguously differentiate these two packages later, they are tagged with unique names. The example realizes the creation of the tagged correspondences that were assumed in the example for retrieving elements with tagged correspondences in subsection 5.3.3.

```
trigger: create root pcm.Repository
create element: java.Package as contractsPackage
create element: java.Package as datatypesPackage
add correspondence between contractsPackage and newRoot tag with "contracts"
add correspondence between datatypesPackage and newRoot tag with "datatypes"
```

Listing 5.24: A response for creating the contracts and data types package of a just created PCM repository with tagged correspondences between them

5.4.4 Deleting Model Elements

The effect of a response defines the repair logic for specific consistency overlaps. Although this repair usually means that a new consistency overlap is instantiated or an existing overlap is repaired, it can also mean to remove a consistency overlap. Such a removal is especially necessary if an element of the overlap is removed and the repair logic defines the removal of the consistency overlap as the appropriate repair action.

When removing a consistency overlap and deleting elements contained in it, the deletion also requires the removal of all its correspondences to other elements. In contrast, removing a consistency overlap, and thus its correspondences, does not necessarily require the deletion of all the associated elements. In consequence, a defined delete statement for model elements should lead to the removal of its correspondences as well. The explicit removal of correspondences is consequently only necessary if correspondences shall be removed without the deletion of at least one of its elements.

Based on this insight, we define a statement for specifying the deletion of an element. A realization of the complete grammar for the effects is presented in Figure 5.2. The delete statement does only require the specification of the element to be deleted, which can be any element that is referenceable because it is accessible through a change property or because it was retrieved by the matcher.

Additionally to the deletion of the specified element, the operationalization of the delete statement also removes all correspondences of the element from the correspondence model. Depending on the concrete implementation of the correspondence model, this removal may be performed automatically if one of the corresponding elements is removed.

⟨*effect*⟩	= { ⟨*create-element*⟩ | ⟨*delete-element*⟩ } { ⟨*add-correspondence*⟩ | ⟨*remove-correspondence*⟩ } ⟨*execution-block*⟩
⟨*create-element*⟩	= 'create element:' ⟨*element-type*⟩ 'as' ⟨*id*⟩
⟨*delete-element*⟩	= 'delete element:' ⟨*element-reference*⟩
⟨*add-correspondence*⟩	= 'add correspondence between' ⟨*element-reference*⟩ 'and' ⟨*element-reference*⟩ ['tag with' ⟨*tag*⟩]
⟨*remove-correspondence*⟩	= 'remove correspondence between' ⟨*element-reference*⟩ 'and' ⟨*element-reference*⟩
⟨*execution-block*⟩	= 'execute:' ⟨*code-block*⟩

Figure 5.2: Completed effect grammar

A response written in the proposed concrete syntax, which exemplifies the deletion of an element, is given in Listing 5.25. In the example, the Java class corresponding to a PCM component is deleted in response to the removal of the component within the PCM model. This deletion implicitly removes the correspondences between the Java class and the component as well. The example is identical with the one given in subsection 5.4.3, except that the Java class is also deleted and the correspondence is now removed implicitly.

```
trigger: remove from list pcm.Repository[components]
retrieve element: java.Class as javaClass corresponding to oldValue
delete element: javaClass
```

Listing 5.25: A response that deletes a Java class after removing the corresponding PCM component

5.5 Separating Triggers from Repair Routines

The basic structure of the response language, as proposed in subsection 5.1.2, separates a response into three parts with different purposes. The requirements to, and possible realizations of the different parts were discussed in the previous sections. In this section, the further development of the top-level language structure is discussed.

With the basic language structure, the specification of independent change-driven transformations is possible by writing single responses. In terms of reusability, which are discussed in subsection 5.5.2, and other conceptual restrictions, which are discussed in subsection 5.5.3, the tight coupling of all three parts of a response is not useful.

To resolve the tight coupling of triggers, matchers and effects, the following subsection deals with the separation of the parts of a response. Afterwards, the consequences for reusability and further actual restrictions of the language are presented in examples.

5.5.1 Specifying Repair Routines with Matchers and Effects

In general, all three parts of a response could be reused independently. They have well-defined requirements to the data that they can access and get as their input, and to the data that they provide. A trigger expects a change and the correspondence model. Properties of the change and the correspondence model are passed to the matcher, which additionally provides several elements of certain consistency overlaps. The effect again expects the change properties and the correspondence model, as well as certain elements of the consistency overlaps that it repairs and that are retrieved by an appropriate matcher. Consequently, all parts could be specified independently with each trigger defining which concrete matcher it calls and each matcher specifying the effect it calls.

A matcher and an effect are coupled more tightly than a trigger and a matcher. Matcher and effect both define operations on a concrete instance of a consistency overlap type. They rely on the change delivered by a trigger, which only decides whether a consistency overlap can be affected due to type information of the change. The matcher and the effect can also rely on other inputs than the change to identify a consistency overlap and still define a way of repairing it. Although a certain model change requires a certain way of repairing the affected consistency overlap, the same repair can be applied in different scenarios. For example, the renaming of an interface or a data type in a PCM model both require their corresponding Java class names to be updated. This update can be expressed in the same matcher and an effect although the trigger has to be defined differently.

A matcher and an effect define a *repair routine* for a specific consistency overlap. The trigger does only decide if a repair routine is called due to a change. Therefore, we separate the repair routine, defined by a matcher and effect, from the trigger of a response. This separation also allows to make a repair routine independent from the actual change because only the change properties that describe the affected elements have to be passed to the repair routine. The input of arbitrary elements to a repair routine allows to further modularize them and reuse repair routines within others.

An extension for the response language that supports the proposed separation is provided in Listing 5.26. The definitions of matcher and trigger are moved into a separate *repair-routine*. We allow an implicit specification of the repair routine right within the response to still allow the compact specification of a complete response. The *implicit-repair-routine* expects the elements that were previously supplied to a matcher, which are the correspondence model and the change properties. In contrast, the *explicit-repair-routine* defines its input explicitly in the *routine-input*, while the correspondence model is obviously still required. A dedicated root element *responses-document* defines a new top-level element that illustrates the possibility of specifying responses and explicit repair routine that can be called from responses.

An explicit repair routine should accept further element types than only elements of instances of the considered metamodels. We also allow to pass primitive values and strings to make a repair routine more generic and increase its reusability. We advise against allowing the specification of element collections as inputs because each consistency overlap type consists of a static number of metaclasses, thus a repair routine that restores such an overlap relies on a static number of elements and not arbitrary large collections of them. The handling of element collections is further discussed in subsection 5.5.3.

⟨*responses-document*⟩	=	{ ⟨*response*⟩ | ⟨*explicit-repair-routine*⟩ }
⟨*response*⟩	=	’response:’ ⟨*response-name*⟩ ⟨*trigger*⟩ ⟨*implicit-repair-routine*⟩
⟨*implicit-repair-routine*⟩	=	⟨*repair-routine*⟩
⟨*explicit-repair-routine*⟩	=	’repair routine:’ ⟨*repair-routine-name*⟩ ⟨*routine-input*⟩ ⟨*repair-routine*⟩
⟨*routine-input*⟩	=	’input:’ ⟨*element-type*⟩ ’as’ ⟨*id*⟩ { ’,’ ⟨*element-type*⟩ ’as’ ⟨*id*⟩ }
⟨*repair-routine*⟩	=	⟨*matcher*⟩ ⟨*effect*⟩
⟨*response-name*⟩	=	⟨*id*⟩
⟨*repair-routine-name*⟩	=	⟨*id*⟩

Listing 5.26: Response language grammar with separated repair routines

The specified repair routines must also be invoked. The external repair routines can be made accessible through further language constructs within the effects of our external DSL. The drawback of this approach is that a repair routine cannot be called by another one at a specific point of its effect execution, but only before or after the execution block. Therefore, we allow to call explicit repair routines through internal DSL constructs in the execution block. We make them accessible through methods that have parameter lists according to the expected input. Consequently, a repair routine can be externalized from a response by specifying its name and input, and leaving the matcher and effect of the response empty, except for the call of the repair routine within the execution block.

5.5.2 Reusing Repair Routines

One important motivation and use case for the separation of responses and repair routines is reusability. A consistency overlap can get inconsistent due to different modifications, but it can be restored by the same routine that retrieves the right elements and performs appropriate modifications. Especially the externalization of parts of a repair provides good potential for reuse. It allows to compose a response of several repair routines.

An example for the useful externalization of a repair routine is the creation of Java classes. Different elements in a PCM model share consistency overlaps with Java classes. Whenever such an element is created, a Java class has to be instantiated. Instead of specifying this initialization in each case, it can be separated into an explicit repair routine.

An example for reusing a repair routine is given in Listing 5.27. The creation of a Java class is externalized into an explicit repair routine. The routine expects the package, in which the new class shall be placed, and a named element, to which the class corresponds and which provides its name. This routine is used by responses that react to the creation of a component and a data type for creating the corresponding classes. The routine is invoked by a method that is available in the execution block, as described in the previous subsection. In our example, these methods have the name of the repair routine prefixed with *call*. An implementation of the language has to ensure that the generated code has access to such methods. The mechanism used in our implementation is discussed in chapter 6.

```
response: CreatedComponent
trigger: insert in list pcm.Repository[components]
retrieve required element: java.Package as repositoryPackage
  corresponding to affectedObject tagged with "rootPackage"
create element: java.Package as componentPackage
add correspondence between newValue and componentPackage
execute: {
  val component = newValue;
  componentPackage.name = component.name;
  componentPackage.namespaces += repositoryPackage.name;
  repositoryPackage.subpackages += componentPackage;
  callCreateClass(interface, componentPackage);
}

response: CreatedCompositeDataType
trigger: insert in list pcm.Repository[dataTypes]
  check newValue instanceof pcm.CompositeDataType
retrieve required element: java.Package as datatypesPackage
  corresponding to affectedObject tagged with "datatypes"
execute: {
  val compositeDataType = newValue;
  callCreateClass(compositeDataType, datatypesPackage);
}

repair routine: CreateClass
input: pcm.NamedElement as sourceElement, java.Package as containingPackage
create element: java.Class as javaClass
add correspondence between javaClass and sourceElement
execute: {
  javaClass.name = sourceElement.name;
  javaClass.namespaces += containingPackage.namespaces;
  javaClass.namespaces += containingPackage.name;
  containingPackage.classifiers += javaClass;
}
```

Listing 5.27: Responses reusing repair routines for creating Java classes

5.5.3 Iterating Repair Routine Calls

Another problem that can be solved with the separation of repair routines is the repair of dynamic numbers of consistency overlaps. If a repair routine is tightly coupled to a complete response, the matcher and effect part do only provide language constructs for dealing with a static number of elements. Each create or retrieve statement is assumed to provide at most one element. The number of elements to operate on is thus bounded by the number of statements.

Dealing with a dynamic number of elements could be realized in two ways. The operations on the correspondence model can be performed manually within the execution block

or further language constructs can be provided to handle that requirement. Nevertheless, the number of elements of which a consistency overlap consists is usually static. A problem arises if a modification requires the repair of an arbitrary number of consistency overlaps. Although each of them consist of a static number of elements, all together consist of an arbitrary number of elements. Because each consistency overlap consists of a static number of elements, the repair of each of them can be realized in a repair routine.

With the separation of responses and repair routines, the repair for a single consistency overlap, consisting of a static number of elements, can be defined in one repair routine. If a modification potentially affects an arbitrary number of consistency overlaps, a response can call the repair routine for each of the affected consistency overlaps.

An example for such a scenario is the renaming of the repository in a PCM model. A repository is mapped to a Java package that contains several other packages and classes. Among others, it contains one package and one class for each component in the repository. Each package and class in our Java metamodel contains its package hierarchy as strings in the `namespaces` property. Renaming the repository and thus its Java package requires the namespaces of all packages and classes that implement components to be updated. The example is realized in Listing 5.28. It is reduced to the repair of the consistency overlaps of the components and omits several other required modifications, such as updating the contracts and data types implementations accordingly.

Cases like in the example usually arise from bad metamodel design. The namespace of a class or a package is defined by the packages it is contained in and thus should not be saved within a property redundantly. Nevertheless, we have to deal with such metamodel designs in practice and thus need to support such setups in our language.

```
response: RenamedRepository
trigger: replace value pcm.Repository[name]
retrieve required element: java.Package as rootPackage
  corresponding to affectedObject tagged with "rootPackage"
execute: {
  rootPackage.name = newValue;
  for (component : repository.components) {
    callRenameComponentPackageAndClass(component);
  }
}

repair routine: UpdateComponentNamespaces
input: pcm.Component as component
retrieve required element: java.Package as componentPckg corresponding to component
retrieve required element: java.Class as componentClass corresponding to component
execute: {
  componentPckg.namespaces.clear();
  componentPckg.namespaces += component.repository.name;
  componentClass.namespaces = #[component.repository.name, component.name];
}
```

Listing 5.28: A response iterating repair routine calls

5.6 Language Properties and Responsibilities

The last sections introduced the structure and constructs of the response language. In this section, we discuss some further properties of the language concerning the target model of a transformation, transitivity of change propagation and the visibility of model states. We also explain responsibilities for model persistence that it has to fulfill.

5.6.1 Target Models of Responses

Incremental transformations store the relations between elements of the transformed models within trace models. These traces usually exist between a clear source and target model. Consequently, a transformation has a clear source model that was changed and a clear target model on which the transformation operates.

The response language relies on the Vitruvius framework, which defines one correspondence model for each pair of metamodels. These correspondence models do not consider the models of the corresponding elements. The elements can be contained in different models that are conform to the same metamodel. This is why the target of a transformation is not that clear in this context as it is in classical source-to-target transformations.

A response operates on individual model elements, rather than on a dedicated target model. Corresponding elements can be addressed no matter in which models they are contained. Thus, elements of the consistency overlaps that are accessed in a response can be spread across different models. For preserving consistency of PCM and Java models, the restriction to a single model would even be insufficient because each class and package in the Java model is represented as a single model. As a result, only a single class or package could be addressed in a transformation if only one target model was allowed.

5.6.2 Transitivity of Consistency Repair Routines

Consistency-restoring transformations can use and provide different kinds of transitivity. On the one hand, transitivity of transformations can be required for repairing dependent consistency overlaps between two models. On the other hand, a transformation performs modifications that can again trigger new transformations for restoring consistency with further models.

The first kind of transitivity was implicitly discussed in subsection 5.5.2 about separating triggers and repair routines in the response language. The restoration of different consistency overlaps can be defined in individual repair routines. In case of the dependency of one consistency overlap to another, the repair routine of the dependent one can be called from the one it depends on. Calling a repair routine from another one can be considered as an implicit trigger that is not provided by the transformation environment but by the repair of another consistency overlap. In the case of element deletions, this kind of transitivity is often implemented implicitly, without further repair routines. For example, deleting a component in a PCM model triggers the deletion of the corresponding class in the Java code and implicitly removes all dependent elements and thus consistency overlaps, such as required or provided roles.

The effect of a response performs modifications within models. Consequently, these modifications can again trigger consistency-restoring transformations, even to preserve consistency with further models. This second kind of transitive transformation execution is currently not considered within our transformation environment provided by the Vitruvius framework. Although reacting to modifications that are performed by the repair routines is conceptually and technically simple, it leads to some problems, such as the cyclic triggering of transformations. Nevertheless, this mechanism would ease the specification of consistency-restoring transformations between three or more metamodels, as they would not have to be defined between each metamodel pair. Several transformations could be omitted because of the transitive execution of repair routines.

5.6.3 Visibility of Model States

A transformation environment can execute change-driven transformations at different points of time. In the best case, transformations are executed right after each atomic change because then the actual model state is the one right after the change for which the transformations are executed.

If the execution environment does not necessarily execute the transformations after each atomic change, the model state that is visible when executing the transformation is not obvious. The visible state can either be the state right after the change that the transformation reacts to or the actual state after all the recorded changes. Both variants have benefits and drawbacks, which are explained in the following.

Accessing the model state that was actual right after the currently processed change allows to retrieve reasonable information from the whole model because it contains the same information as if just the atomic change was performed. However, this model cannot be modified because it is not in the actual state. Performing modification to the old state of the changed model requires a merge with the model state after the further changes, which is not possible in general.

If the model state after the already performed changes is accessible in the transformation, generally no information can be retrieved from this model state. Arbitrary modifications may have been performed to the model after the currently processed change. Imagine that two model elements a_1 and a_2 in model A share a consistency overlap with an element b in model B. If a_1 is modified and a_2 is deleted afterwards, the transformation that repairs the consistency overlap by modifying b in response to the modification of a_1 fails. The element a_2, which is part of the consistency overlap and is potentially required for restoring consistency, is not contained in the visible model state any more. If arbitrary modifications can be performed in the models before executing consistency-restoring transformations, the transformations cannot rely on any model information.

To avoid the drawbacks of both approaches, the transformation environment would consequently have to ensure that the appropriate transformations are executed right after an atomic change occurs. Nevertheless, the model might not be in a consistent state after each atomic change and can possibly not be saved, which could lead to the eventual abortion of this and further changes in the source model. Because the changes would have been transferred to other models already, a rollback of the transformations would be required.

As long as changes do only require modifications in another model to restore consistency, as described in section 4.4, providing access to the model state right after the change is the best alternative. In this case, consistency constraints have to be defined in a way that the changed model must not be further modified for restoring consistency. In addition, the transformations must be able to deal with possibly inconsistent states of the changed model, as some model constraints may not be fulfilled after each atomic change. Triggering transformations due to changes of the satisfaction state of constraints can omit this problem. Such triggers abstract from changes to more complex constraints defining the preconditions for a transformation execution. Those triggers are further discussed in subsection 5.8.2.

5.6.4 Persistence of Logical and Physical Models

Up to now, we have only considered logical models. Logical models can be distinguished from physical models that describe their persistence. This distinction is different from the data modeling domain, in which the logical representation of data in object-oriented entities is separated from the persistence within a relational database [91]. In that separation, logical and physical models are instances of different metamodels. We distinguish logical and physical models by the granularity in which they are persisted, which means that still both conform to the same metamodel.

Logical models are the models that we want to consider when working with them in a development environment and also deal with when restoring consistency through transformations. Physical models describe the persistence of models and must not correspond to the logical model description. For example, Java programs are persisted across different folders and files based on the package and class structure of the program. Consequently, each class represents an individual physical model. Nevertheless, the complete Java program defines a coherent model that describes the implementation of the software system and thus represents one logical model.

The distinction between logical and physical models must generally not be considered when working with the models. However, models must eventually be persisted and reloaded and therefore need a physical representation. While most logical models are persisted completely, especially code models are persisted according to their established structure of package folders and class files. As the separation of logical models into physical models does primarily depend on the metamodel, the required persistence rules should be specified generally for a metamodel rather than in each transformation.

The persistence specification for a metamodel must be able to decide whether a certain model element is the root element of a model or not. Primarily, this decision can be based on the type of the element. For example, Java package elements are always persisted as root elements. Nevertheless, further information may have to be examined. As an example, a Java class is generally persisted as a physical model, but not if it is nested within another class. The persistence specification can also combine the decision if a certain element shall be persisted with the actual persisting operation. Therefore, it has to determine the persistence path. This path usually depends on the current project and consists of a default path and a name that is extracted from the model, such as the name of the root element. Some of the information may have to be requested from the user. An interface for such a persistence specification can be defined as shown in Listing 5.29.

```
public interface PersistenceSpecification {
  public boolean persistIfNecessary(ModelElement element, Project project);
}
```

Listing 5.29: Exemplary persistence specification interface

In a transformation, a newly created model has to be persisted. Therefore, a response calls the persistence specification of the metamodel for each created or modified element to persist it if necessary. This operation is performed automatically and must not be specified by the methodologist, as the retrieve and especially the create statements specify which elements are affected by the response and potentially have to be persisted.

5.7 Aspects of Restoring Consistency with Responses

This section summarizes some aspects that have to be considered when specifying responses for restoring model consistency. After shortly discussing the execution order of responses, we present some considerations for the treatment of opposite references. Finally, we introduce the necessity and realization of interaction with the model user during the response execution and discuss the problem of meta-level breaks.

5.7.1 Response Execution Order

Responses are executed whenever the occurrence of an atomic change is processed. Consequently, the order of change occurrences defines the order of response executions. Nevertheless, different responses can be written for the same change event.

The transformation environment can either consider a specific execution order for the responses to one change or execute them in a random order. For a predetermined ordering, the responses would usually have to specify their order explicitly. This could, for example, be achieved by defining relations between responses that define if a certain response requires the execution of another. However, we assume that it is infeasible to both define a reasonable comparison metrics for responses and comprehend this mechanism as a methodologist.

We do not consider an explicit mechanism for ordering responses, as they can be ordered implicitly using the available language constructs. Considering a change that has to be processed by two different repair routines, one response can be defined that reacts to this change and executes both explicitly defined repair routines. The ordering of their execution is implicitly specified by the ordering of repair routine calls within the response.

5.7.2 Modifications of Bidirectional References

In the response language, we currently only consider atomic changes, which are sufficient for reacting to any model modification that may affect consistency overlaps. Some atomic changes do always occur together because semantic constraints of the model require them

to occur together for achieving a consistent model state. An interesting example of such modifications are changes of bidirectional references.

In EMOF- and Ecore-based models, references can be declared as opposite of another, which means that two model elements share a bidirectional association. The creation or removal of such an association leads to the modification of both references. Consequently, the transformation environment should generate two change descriptions for this modification. Writing responses for the modification of bidirectional references requires the methodologist to be aware of the duplicate change that describes the same modification within a consistency overlap. Responses may not be written for both changes but only for one of them to avoid a duplicated execution of the repair logic.

Depending on the transformation environment, the change monitor could also just provide one of the changes, as usually only the reaction to one change is required. If it is clearly specified which of the two possible changes occurs, responses can be specified for that event. Nevertheless, the described change could also depend on the modification that was performed by the model user. Depending on which of the two references he modifies, different change descriptions can get generated. This uncertainty requires to define responses for both changes.

5.7.3 Interaction with the Model User

In the previous sections, several examples for consistency repair routines between PCM models and Java code models were discussed. All these examples were specified fully automated, depending on static decisions. However, sometimes repair routines can or even must require decisions from the model user that specify how a consistency overlap has to be repaired.

Interactions with the user can be necessary in different situations. One example is the selection of the Java collection type to map a newly created collection data type of a PCM model to. Even though any collection type provided by Java could be statically used, the user possibly wants to select from a set of possible collection types. As another example, the decision whether the corresponding Java class shall be deleted after removing a component can be made by the user. Although it is possible to avoid user interaction in these scenarios, for example, by prescribing that the corresponding class is always deleted, consistency repair cannot always be automated completely.

The examples show that user interaction can especially be the selection from a list of possible options, which also covers yes-no questions. Therefore, responses have access to a well-defined interface for performing user interactions. A simple exemplary interface is provided in Listing 5.30. It defines only one method for the selection from a list of options

```
public interface UserInteracting {
  public <T> T askUser(String message, Iterable<T> options);
}
```

Listing 5.30: Exemplary user interaction interface

```
response: RemovedComponent
trigger: remove from list pcm.Repository[components]
retrieve required element: java.Package as javaPackage corresponding to oldValue
retrieve required element: java.Class as javaClass corresponding to oldValue
remove correspondence between javaPackage and oldValue
remove correspondence between javaClass and oldValue
execute: {
  val answer = userInteracting.askUser("Component " + oldValue.name +
    " was deleted.  Delete corresponding class and package?", #["yes", "no"]);
  if (answer == "yes") {
    callDeleteClassAndPackage(javaPackage, javaClass);
  }
}

repair routine: DeleteClassAndPackage
input: java.Package as javaPackage, java.Class as javaClass
delete element: javaPackage
delete element: javaClass
```

Listing 5.31: A response for a component removal with user interaction

that can be of any type. A reasonable string representation that can be presented to the user must be derivable from the specified options.

Assuming the accessibility of such an interface via `userInteracting` within the effect of responses, the decision if the component class should be deleted together with the deletion of the component can be passed to the user as shown in Listing 5.31. An extra repair routine that just deletes the elements has to be specified when performing the user interaction within the execution block. As a result, it could be thought of further language extensions for a better integration of the user interaction.

Generally, the user interaction should be possible at any point of time in the execution of a response. A decision based on the change information can be performed within the precondition block of the trigger. A decision based on the correspondence model can be performed in a check block of the matcher. All other decisions, for example, the selection of the type of a created element, can be performed within the execution block. Nevertheless, a user interaction should be performed as late as possible. This avoids that a user interaction gets obsolete because its result is used in a statement that is not executed due to another decision that lead to the abortion of the response in between.

5.7.4 Meta-Level Breaks

Models are defined on different meta-levels. As introduced in subsection 2.1.2, a model is always an instance of a model that is defined on the next higher meta-level, its metamodel. Thus, model elements are instances of metaclasses defined in their metamodel and usually only reference elements on the same meta-level but not from their metamodel. Nevertheless, this boundary between meta-level is sometimes broken.

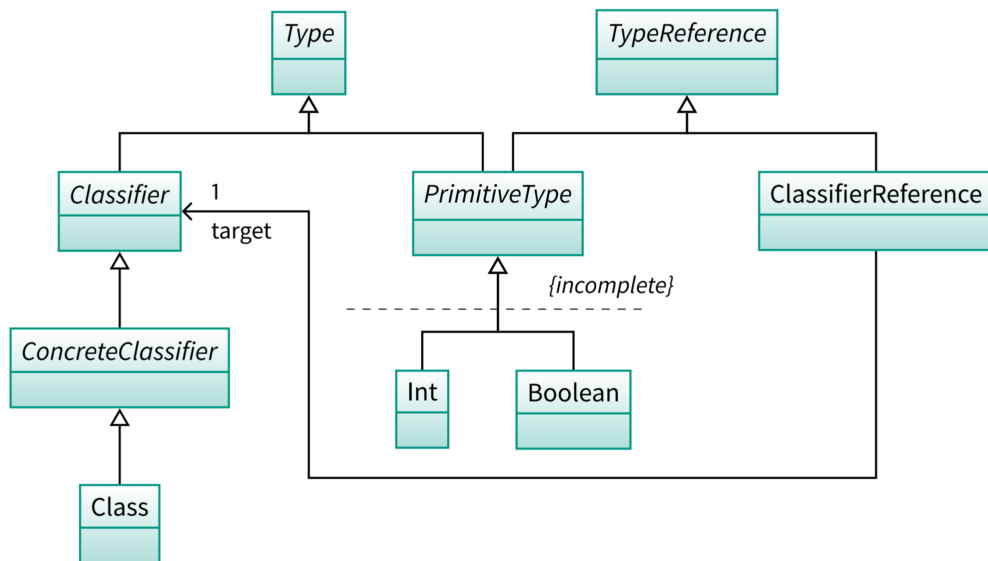


Figure 5.3: Extract of the Java metamodel defining types and type references

An example for broken meta-levels can be found in the Java metamodel. An extract showing the relevant classes of the metamodel is presented in Figure 5.3. The metamodel provides different types and references to these types. Ordinary classes are instances of the Class metaclass. A reference to a class, for example, in order to declare the type of a variable, is represented by a ClassifierReference instance that has a reference to the class. In contrast, concrete primitive types, which are references as well and are examplarily represented by Int and Boolean, are already defined in the metamodel. Consequently, references to concrete classes are defined on the model level, while references to concrete primitive types are defined on the metamodel level.

In a PCM model, the repository contains data types that can be primitive, composite or collections. While the last two are mapped to Java classes, the primitive types have no correspondences in the Java model. They correspond to the primitives types defined in the Java metamodel, thus on another meta-level. If a data type of the PCM repository is used, for example, as the return type in a method signature, a reference to the data type has to be established in the Java model. If the used data type is a collection or a composite type, its corresponding Java class has to be retrieved and a ClassifierReference to this element has to be created. Otherwise, the data type is primitive and has no corresponding Java class, which requires the creation of a PrimitiveType that is a reference and a type at once.

The existence of a correspondence in a case of broken meta-levels depends on whether the corresponding concept is defined in the model or in the metamodel, as the latter cannot be referenced by a correspondence. In such cases, different transformations have to be written for both cases because otherwise the element retrieval fails if no corresponding element exists. This necessary differentiation leads to the duplication of transformations, which do only differ in the way one element is addressed.

Listing 5.32 shows an example for responses that deal with the mentioned break of meta-levels in the Java metamodel. The responses react to the modification of an inner type of a CollectionDataType. Different responses are defined for the cases in which the

```
response: ChangedNonPrimitiveTypeOfCollectionDataType
trigger: replace value pcm.CollectionDataType[innerType]
  check !(newValue instanceof pcm.PrimitiveDataType)
retrieve required element: java.Class as newInnerDataType
  corresponding to newValue
// create a type reference that refers to the inner data type

response: ChangedPrimitiveTypeOfCollectionDataType
trigger: replace value pcm.CollectionDataType[innerType]
  check newValue instanceof pcm.PrimitiveDataType
// instantiate a type reference for the corresponding primitive Java type
```

Listing 5.32: Separate responses for primitive and class types in Java models

inner type is primitive or not. In the former case, an appropriate primitive type in the Java model has to be instantiated, whereas in the latter case, the corresponding class has to be retrieved and referenced.

A separation of transformations has to be applied each time a data type reference must be established in a PCM to Java transformation. This duplication of transformation specifications can be avoided by using the *optional* keyword introduced in subsection 5.3.4. The retrieval is declared as optional and returns a null reference if the data type is primitive. This reference has to be handled correctly in the effect.

Finally, the approach using the *optional* keyword does not solve the problem within the retrieve statement completely. Further actions can also depend on whether a corresponding element exists or not. For example, a distinction between classes and primitive types is still necessary in the effect, as both require different reference types to instantiated. Such problems have to be handled within the effect part in any case.

5.8 Possible Extensions

This section closes the introduction of the response language with an overview of extensions for increasing its usability and ease of applicability. After introducing further types of triggers and an explicit specification of consistency overlap types, we shortly discuss the necessity of extending the user interaction mechanisms. Finally, the validation of code blocks and the provision of further language constructs in the effects are discussed.

5.8.1 Composite Change Triggers

The response language allows the execution of transformations triggered by atomic changes. An obvious extension of the available triggers are composite changes, which describe compositions of atomic changes and were introduced in subsection 4.2.5.

The idea of using atomic changes as the triggering events for model consistency repairing transformations relies on the assumption that any model change can be subdivided into a sequence of atomic changes that transform the original model into consistent intermediate states. Suppose that this assumption is wrong and the intermediate states are not

necessarily consistent, the complete application of the original change does still transform the model into a consistent state. Consequently, that change would be a reasonable trigger for a transformation that restores consistency.

Any change that is performed in a model is either atomic or composite. Composite changes can be subdivided into atomic changes, which allows to define them as a sequence of atomic changes. If the order of changes is irrelevant, a composite change can also be defined by a set of atomic changes. Nevertheless, the order can be important. For example, if a well-formedness constraint allows an element to have at most one container, the composite change that describes the move of an element from one container to another has to be specified by an atomic remove before an atomic add. Ordering them the other way round, the constraint would be violated meanwhile.

For using composite changes within responses, an adaption of the change descriptions metamodel is not necessary, as the composite changes can be described as a sequence of atomic changes. As a result, composite changes can be specified as triggers by multiplying the trigger specification within a response. This multiplication allows to define a sequence of atomic changes, which are required to occur in the specified order. Furthermore, the transformation environment must be able to collect change events and execute a response when a sequence of occurred changes matches its trigger.

5.8.2 Constraint Satisfaction Change Triggers

In addition to changes, further events that trigger the execution of a response are imaginable. Because the context is still a change-driven environment, those other types of events must be triggered by changes instead.

One example for such an event type is the change of the satisfaction of constraints. A trigger based on such an event would specify the execution of a response if a specific constraint got satisfied because of the last change or if it got unsatisfied through the change. Constraints can be defined on elements of a single model, referred to as intra-model constraints, or for elements of different models, referred to as inter-model constraints.

Considering an element that has to fulfill some complex constraint for being transformed, the satisfaction of this constraint can be achieved through different kinds of changes. A transformation must be written for each of these changes that can fulfill the constraint to transform the element when possible. If a transformation could react to the satisfaction of the constraint, just one transformation would be sufficient to transform the element.

Providing a response trigger for constraint satisfaction changes requires an extension of both the response language and the framework. The language must allow to reference a constraint, which can be specified in some existing constraint language like OCL or in the MIR invariant language. It must also allow to specify whether to react to its satisfaction or to its violation. Furthermore, the context element of the constraint must be provided to the response, like the modified element provided by a change event. Without having any reference object, affected consistency overlaps cannot be identified.

The framework has to provide some mechanism for detecting if the satisfaction state of a constraint changed. This detection can, for example, be achieved by storing the satisfaction state of all constraints for all model elements and update this state after each change by checking them again. Because the reevaluation of all constraints after each change can be

costly, the effort could be reduced by checking only those constraints that may be affected by a change because they reference elements of the type of the modified element.

When reacting to the violation of a constraint, the violating element can also be of interest. An approach proposed by Fiss, Kramer, and Langhammer extracts the violating element from a given constraint [32]. This approach allows the specification of a variable for a constraint, which gets the value of the violating element assigned.

The proposed mechanism is of special interest for intra-model constraints as they describe when a model element reaches a state that can be transformed into corresponding elements in another model. Nevertheless, constraints can also be used for describing inter-model preconditions, as such constraints implicitly specify consistency overlaps. While intra-model constraint can be especially used for identifying reasonable situations in which a new consistency overlap can be established, the violation of inter-model constraints can be used as an indicator for the corruption of consistency overlaps.

5.8.3 Explicit Consistency Overlap Type Specification

A response specifies the repair for a certain consistency overlap type that is affected in a specific way. Because a consistency overlap can be affected by the modification of each element property that is used in the constraints of its overlap type, a response must be written for each possible modification of these properties.

If the metaclasses of a consistency overlap type and their properties that are used in the constraints of that type are clear, all changes for which responses have to be written can be derived. Consequently, an explicit specification of the metaclasses and relevant properties of an overlap type can be used to derive necessary responses and assign them to the consistency overlap type. This approach reduces the risk for the methodologist to miss a possible modification of elements of a consistency overlap. Furthermore, it reduces the specification effort as several responses are derived from one specification.

A language extension can provide constructs for specifying this consistency overlap types and connect them with responses. The type specification can even be further extended by simple relations between the elements that are automatically translated into repair actions in the responses, similar to the mechanisms in declarative transformation languages. That way, we can provide a compact specification of simple consistency relations and still allow to define the repair completely imperative.

5.8.4 Extensions for User Interaction

The developed language constructs support the possibility to provide user interaction in any code block, using a user interaction interface as proposed in subsection 5.7.3. Nevertheless, the user interaction would also be useful in some of the other defined language constructs. Furthermore, user interaction is currently limited to the user who performed the change that triggered the response execution.

A further useful point for user interaction is the specification of the element type in statements for the creation of elements. The proposed construct assumes the specification of a static element type. Using the user interaction interface within this statement would enable the user to select from different types.

Currently, we assume responses to be executed in the moment when the change occurs and that, if necessary, the user who performed the change is asked for decisions. However, decisions within the responses potentially concern models for which another user than the one who performed the modification is responsible and should be asked. Such a mechanism requires further research about the point of time for the execution of a response. However, that topic concerns the transformation environment that executes the responses, rather than the responses themselves.

5.8.5 Imperative Code Validation

The response language uses imperative code blocks within several constructs to provide a maximum expressive language. This approach ensures a high expressiveness, but it allows the methodologist to write potentially undesired code within certain constructs.

The triggers and matchers are assumed to be free of side-effects. Nevertheless, they provide code blocks in several constructs. Within a trigger, a general precondition block can be defined, and a matcher allows the specification of check blocks. Moreover, we explained that code blocks can also be used for specifying element references within the retrieve statements of a matcher. All these code blocks should be free of side-effects to allow a safe abortion of the response execution if any precondition validates to false, without having performed partial model modifications yet.

As we do not want to ensure side-effect freeness by restricting the expressiveness of these statements, another approach is to validate the imperative code for the property of side-effect freeness. Several approaches try to provide analyses of complete programming languages for side-effects, such as for the Java programming language [84]. Whereas integrating such an approach may be possible, the provision of a side-effect free API for commonly required expressions, such as a query API like OCL, would represent an easier approach that is sufficient in most cases. If only the provided API is used in a code block, it can be proven as side-effect free. Otherwise, no statement about side-effect freeness could be made and the responsibility would stay at the methodologist.

5.8.6 Language Constructs for Element Modifications

The execution block of a response effect specifies operations for restoring consistency between models. This operation comprise modifications that lead to a satisfaction of constraints between elements of different models, which mostly require the assignment of new values to model element properties.

Language constructs for such assignments could be provided additionally to the imperative execution block, like in QVT-O. Using OCL, the calculation of a value to be assigned could be determined and assigned to a property of any referenceable model element.

The specification of the effects with language specific statements instead of imperative code has several advantages. Unwanted side-effects that may arise from statements in imperative code can be avoided as an assignment definitely only performs this operation. Furthermore, modifications of the changed model can be avoided, if necessary, by not allowing assignments to elements of that model.

6 Response Language Implementation

In the last chapters, we defined model consistency and their restoration based on atomic model changes and developed the change-driven response language for restoring consistency constraints. Additionally to the language design, we developed a prototypical implementation of the language in the context of the Vitruvius project for this thesis.

In this chapter, we first introduce some relevant aspects of the Vitruvius framework. Afterwards, the structure and the final implementation of responses and repair routines, as well as their integration into the execution environment are explained. Finally, the language specification and some of its aspects, as well as the code generation, which produces the explained runtime structure, are discussed.

6.1 The Vitruvius Framework

The prototypical implementation of the response language contributes to the MIR language family of the Vitruvius approach. These languages provide different approaches for defining change-driven consistency-restoring transformations. The response language is responsible for providing a maximum expressive language that is capable of defining the repair of preferably any consistency constraint, especially of those which are not expressible in the mapping language.

The current implementation of the Vitruvius approach relies on the Eclipse plugin mechanism, introduced in section 2.3. Different plugins implement different aspects of the Vitruvius framework. The central artifact is the implementation of the Virtual Single Underlying Model, which is first introduced in the following.

The response language relies on specific kinds of correspondence and change descriptions metamodels. Concrete realizations of them are provided by the Vitruvius framework and are also discussed in the following. The Vitruvius framework specifies a model synchronization mechanism, which is responsible for executing consistency-restoring transformations and thus serves as the transformation environment for responses. This section closes with the explanation of that mechanism.

6.1.1 Virtual Single Underlying Model

The essential component of the Vitruvius framework is the Virtual Single Underlying Model (VSUM). Instances of this VSUM provide all necessary information about supported metamodels, the currently managed models and correspondences between them.

To implement a software system using Vitruvius, the developer has to instantiate a VSUM by specifying the used metamodels. He must also specify the pairs of metamodels that have instances sharing consistency overlaps because they have to be managed in

correspondence models, which the VSUM instantiates. A VSUM instance manages and provides access to instances of the metamodels that it is responsible for. Such models can be loaded into the VSUM through a well-defined interface. Furthermore, the correspondence models for the metamodel pairs are managed by the VSUM and can be retrieved from it.

The MIR languages do especially rely on the correspondence models provided by the VSUM because they are required to retrieve the elements of consistency overlaps. This is why further details about the VSUM realization are omitted.

6.1.2 Correspondences, TUIDs and Changes

The response language relies on specific kinds of correspondence and change descriptions metamodels, which are conform to the ones that Vitruvius provides. Therefore, we can use the available metamodels without adapting ideas of the response language constructs.

Within correspondences, elements have to be uniquely identified. This identification cannot be achieved using in-memory references. Correspondences are eventually persisted and when reloading them from their persistence, the elements must be uniquely identified across different models. Therefore, the identification must rely on the properties of models and model elements. In Vitruvius, an element is identified by a so called temporary unique identifier (TUID). TUIDs combine information of model elements in a way that they can be unambiguously identified and retrieved across different models. The calculation of the TUID for a model element depends on its metamodel. For example, if the metamodel prescribes a unique identifier for each model element, it can be used to uniquely identify an element inside the model. Otherwise, the container hierarchy of an element can be utilized to identify an element uniquely. To make the element uniquely identifiable across different models, this identifier has to be combined with a unique identifier of the model.

To persist a Vitruvius correspondence model, the TUIDs of the model elements must be persisted, and therefore they have to be computable. This requires an element to be contained within a model and to have all its values that are used for the TUID to be set. Otherwise, no model-spanning unique identifier can be calculated for the element. Thus, consistency-preserving transformations that affect correspondences and their elements have to ensure that the TUIDs can be computed, so that the resolution of elements within the correspondences is possible.

In our approach, we rely on the correspondence model provided by Vitruvius. It consists of single `Correspondence` elements, that in turn consist of two sets of TUIDs that identify the corresponding model elements. Correspondences can also be specialized for the usage context. For example, the mapping language uses special `MappingCorrespondences`, whereas we use `ResponseCorrespondences` in the response language. The latter ones define an additional string property called `tag`, which is used for the tagging mechanism.

A concrete correspondence model for a pair of metamodels is represented by a `CorrespondenceModel` instance. The correspondence model has a generic type parameter for the correspondence type it handles. The VSUM manages correspondence models typed with the general `Correspondence` type. To work only with the type of interest, a correspondence model provides an operation for retrieving a view on the correspondence model that is typed with the desired kind of correspondence. The generated response code retrieves and operates on a view for `ResponseCorrespondences`.

Vitruvius also provides a change descriptions metamodel, which is conform to the one that we used for the response language development, introduced in subsection 5.1.3. In contrast to the metamodel in this thesis, the one of Vitruvius provides a more fine-grained metaclass hierarchy, which reuses concepts instead of redefining properties in different change types like we did. It also provides some further change properties. The most generic change type in this metamodel, which all other change types extend, is the EChange.

6.1.3 Model Synchronization

The second artifact of the Vitruvius framework that is relevant for the response language is the synchronization mechanism. It is responsible for executing transformations that keep the models of a VSUM consistent. Therefore, it realizes a change-driven transformation environment as described in section 4.3.

A Change2CommandTransformingProviding provides access to all available transformations. It consists of Change2CommandTransformings for each pair of metamodels and provides an operation to retrieve this Change2CommandTransforming for a specified pair of metamodels, as shown in Figure 6.1. A metamodel, as well as a model instance, is identified through a so called Virtual Unique Resource Identifier (VURI) in Vitruvius.

A Change2CommandTransforming provides several operations, of which the most important is the transformChanges2Commands method. This method expects a Blackboard element, which especially contains the current and previous changes, as well as the correspondence model for the currently processed metamodel pair. The method takes the current changes from the blackboard, transforms them into Commands, which define the transformations to be executed, and puts them on the blackboard. In our case, this operation generates commands that call the appropriate responses for the actual changes.

The synchronization mechanism of Vitruvius relies on change descriptions that are recorded by change monitors. The monitors are registered for change notifications at the model of the VSUM and convert them into instances of the used change descriptions metamodel.

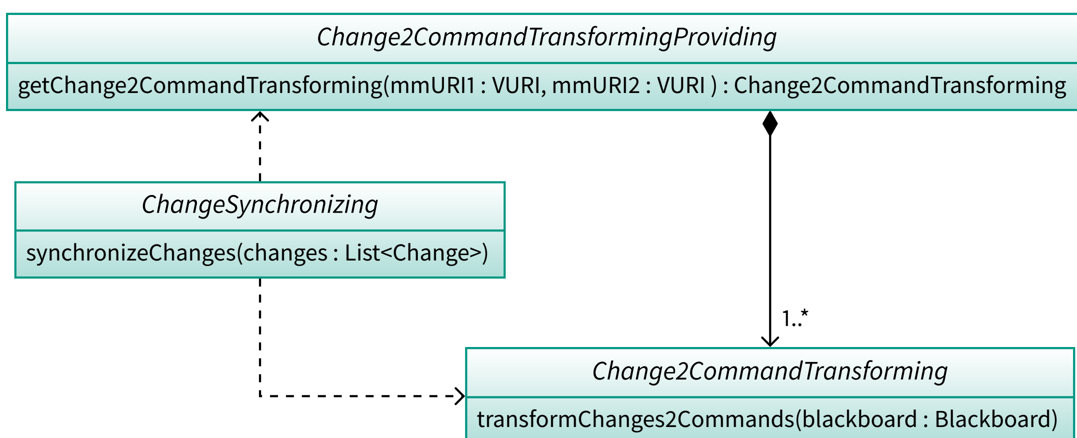


Figure 6.1: Interfaces of the Vitruvius synchronization environment

At specific points of time, currently when saving a model, the transformation environment processes the changes and calls transformations that react to them. This is the responsibility of a ChangeSynchronizing implementation, which is shown in Figure 6.1. It defines only one method, which is called by the framework and whose purpose is to call appropriate transformations for the changes delivered by the change monitors.

An instance of a ChangeSynchronizing implementation has access to the current VSUM and to the Change2CommandTransformingProviding. For each correspondence model of the changed model's metamodel in the VSUM, it retrieves the Change2CommandTransforming for the appropriate metamodel pair from the Change2CommandTransformingProviding. Afterwards, it generates a Blackboard with the actual changes and the correspondence model and calls the transformChanges2Comands method of the Change2CommandTransforming with it. Finally, it executes the commands that were added to the Blackboard.

6.2 Runtime Environment and Transformation Structure

In this section, we introduce the structure of change-driven transformations that we generate from a response specification. We also explain the additional structures required for the integration of single responses into Change2CommandTransformings, which we refer to as the *runtime environment*. Responses are transformed into such an environment.

We first introduce the general structure of a runtime environment and explain the purposes of special runtime data structures. Thereafter, we introduce the different elements of which the runtime environment consists and which are realizations of the response language concepts.

6.2.1 Structure of the Runtime Environment

As introduced in subsection 6.1.3, Vitruvius provides a transformation environment that relies on so called Change2CommandTransforming elements. They manage the transformations for a certain metamodel pair and transform given model changes into commands that execute the appropriate transformations.

The runtime environment for a specific set of transformations between two metamodels has to define a Change2CommandTransforming as the entry point to the transformations. The elements and the structure of a runtime environment are explained on the exemplary environment shown in Figure 6.2.

A single transformation that is generated from a response specification is defined by an implementation of the Response interface, which provides a method for executing the transformation. Such a concrete transformation is represented by the ConcreteResponse class in our example. We separate repair routines from responses, as motivated in section 5.5. A repair routine is represented by an implementation of the RepairRoutine interface, which provides a method for its execution. The example contains two concrete repair routines, which are the ConcreteExplicitRepairRoutine and the ConcreteImplicitRepairRoutine. A response implicitly defines a repair routine, which it executes after checking the trigger. In the example, the ConcreteImplicitRepairRoutine implements the implicit repair routine of the ConcreteResponse.

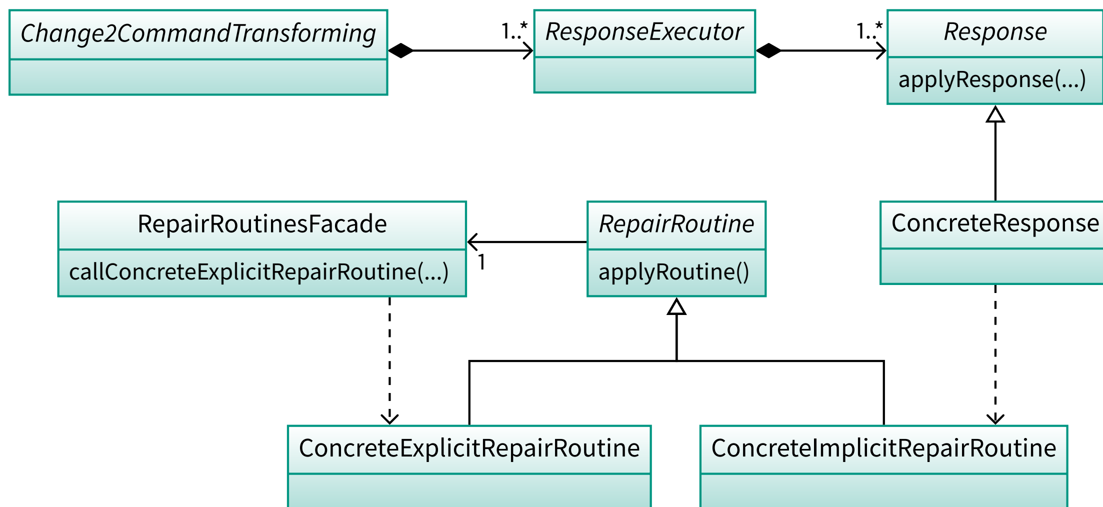


Figure 6.2: Structure and dependencies in an exemplary runtime environment with one response and two repair routines

In contrast to an implicit repair routine, which is defined by each response and is only called by it, an explicit repair routine is supposed to be called by others. A facade class, in the example the RepairRoutineFacade, defines one method for each of the specified explicit repair routines. Such a method expects the model elements required by the related repair routine as parameters and delegates the call to a new instance of the repair routine by calling its applyRoutine method. In the example, the facade class only defines one method for the ConcreteExplicitRepairRoutine because the other concrete repair routine is implicitly defined by the response and thus cannot be called by others.

For each set of transformations, later defined in one document of responses, the environment contains one package for the responses and one for the repair routines. The repair routines package also provides a facade class for all of its repair routines. The responses package contains a ResponseExecutor, which provides the access point to all responses. The executor is responsible for executing appropriate responses for an atomic change. In turn, a Change2CommandTransforming, the entry point of the transformation environment, can consist of different ResponseExecutors, which it calls for all processed changes. The rationale for the separation of these two elements is explained later.

6.2.2 Runtime Data Structures

In addition to responses and repair routines, we use two further data structures in the runtime environment, the ResponseExecutionState and the ResponseElementState. The ResponseExecutionState encapsulates data for the execution of one response that is passed between the response and the repair routines. It consists of the correspondence model, an implementation of a user interaction interface as explained in subsection 5.7.3, and a so called TransformationResult, which is required for defining the persistence of models.

The response language provides several constructs that define different modifications of model elements and their correspondences. Instead of generating code for these operations each time they are required, we encapsulate the functionality in ResponseElementStates.

```
interface ResponseElementState {
  void preprocess();
  void postprocess();
  void addCorrespondingElement(EObject newCorrespondingElement, String tag);
  void removeCorrespondingElement(EObject oldCorrespondingElement);
  void delete();
}
```

Listing 6.1: The interface of a ResponseElementState

Such a state is responsible for managing a single model element during the execution of a transformation. It is initialized with a model element and provides the operations for influencing its state according to the response language constructs, as shown in the interface specification in Listing 6.1.

A ResponseElementState provides methods for specifying the correspondence addition or removal to another element and for marking the element to be deleted. The correspondence addition or removal produces the same result, no matter on which of the two elements it is performed. To perform these operations correctly, they are not executed immediately, but by calling the preprocess and postprocess methods. The preprocessing has to be performed before, and the post-processing must be executed after calling the execution block. For example, correspondences have to be added after the operations of the execution blocks because the correspondences require the element TUIDs to be computable, which is usually not possible before setting the appropriate property values in the execution block.

All element states of a transformation are handled by a ResponseElementStatesHandler. Its interface is presented in Listing 6.2. It provides methods for initializing the element states for created or retrieved elements. The distinction between created and retrieved elements is currently necessary because the modification of retrieved elements can influence their TUIDs, which is automatically updated by the element state. The handler provides methods for the same operations as an element state and delegates their calls to the appropriate elements. Finally, pre- and post-processing methods are provided that execute those operations on all states.

The element states are responsible for TUID updates, which are automatically performed in the post-processing. Because of that update, the element states of all retrieved elements have to be initialized explicitly, even if none of their state changing operations are called.

```
interface ResponseElementStatesHandler {
  void addCorrespondenceBetween(EObject firstElem, EObject secondElem, String tag);
  void removeCorrespondenceBetween(EObject firstElem, EObject secondElem);
  void deleteObject(EObject element);
  void initializeCreateElementState(EObject element);
  void initializeRetrieveElementState(EObject element);
  void preprocessElementStates();
  void postprocessElementStates();
}
```

Listing 6.2: The interface of a ResponsElementStatesHandler

6.2.3 Responses

A response defines a single transformation. It implements the trigger part of a response and delegates the execution to its implicit repair routine. A concrete response is realized as an implementation of the Response interface, which is shown in Listing 6.3.

```
interface Response {
  TransformationResult applyResponse(EChange change,
      CorrespondenceModel<Correspondence> correspondenceModel);
  boolean checkPrecondition(EChange change);
}
```

Listing 6.3: The interface of a Response

The response interface does only provide two methods, which a concrete response has to implement. The checkPrecondition method realizes the trigger of a response and identifies if the response is responsible for the given change. Its implementation first checks the change type. After a positive check, the change is casted to the correct type and its properties are validated. Finally, further preconditions are checked. They are defined in a precondition block in the response language, which is realized through a dedicated method in the response implementation. The applyResponse method can be called if the checkPrecondition method validates the satisfaction of all preconditions. It instantiates the appropriate implicit repair routine and calls its applyRoutine method.

Based on this interface, we provide an AbstractResponse implementation. It realizes the initialization of the ResponseExecutionState in the applyResponse method and delegates the execution to an executeResponse method, which a concrete response has to implement.

6.2.4 Response Executors

A ResponseExecutor defines the connector between a single transformation and the access by the transformation environment through a Change2CommandTransforming. It is supposed to provide appropriate access to a set of responses. In our case, such an executor is generated for each responses document, while a Change2CommandTransforming combines all executors for a pair of metamodels. This structure eases the separation of responses for one metamodel pair into different documents and even allows the usage of responses that are generated from specifications in the mapping language because only one executor per responses source must be integrated instead of all single responses.

The ResponseExecutor interface, which concrete executors have to implement, provides only one method, as shown in Listing 6.4. The generateCommandsForEvent method accepts a change and the correspondence model. Based on the change, a concrete executor checks

```
interface ResponseExecutor {
  List<Command> generateCommandsForEvent(EChange change,
    CorrespondenceModel<Correspondence> correspondenceModel);
}
```

Listing 6.4: The interface of a ResponseExecutor

the triggers of all the responses managed by it and creates commands that execute the responses with matching triggers.

Because all concrete executors only differ in the responses that they manage, we additionally provide an AbstractResponseExecutor, which implements the management of responses and the required execution logic. It requires the implementation of a setup method that just has to call an also provided addResponse method for all the responses it is supposed to manage.

6.2.5 Repair Routines

Repair routines define the transformation logic of responses. A concrete repair routine has to implement the RepairRoutine interface, shown in Listing 6.5. It defines only an applyRoutine method that executes the transformation. Concrete routines expect access to certain model elements and the correspondence model, which we provide to them in the particular constructor.

```
public interface RepairRoutine {
  public void applyRoutine();
}
```

Listing 6.5: The interface of a RepairRoutine

Additionally to the generic interface, we provide an AbstractRepairRoutine realization, which implements further recurring functionality. First, this abstract realization implements the ResponseElementStatesHandler interface, which was introduced in subsection 6.2.2, and thus provides certain methods for modifying model elements. It expects a ResponseExecutionState in the constructor to have access to the correspondence model and the user interaction implementation.

The abstract implementation also provides a method for getting a corresponding element from the correspondence model. It expects the source element, the element type, the tag and a filter method, according to the elements of the element retrieval construct in the response language. Furthermore, it implements the applyRoutine method with error handling that avoids exceptions during runtime and delegates the execution to an executeRoutine method.

A concrete repair routine extending the AbstractRepairRoutine must only implement the executeRoutine method. The response language constructs are realized in code by calling the appropriate ResponseElementStatesHandler methods and providing a method that implements the execution block. This execution block method is called after pre- and before post-processing the element states in the executeRoutine method.

6.2.6 Repair Routine Facades

The last artifact of the response runtime environment are repair routine facades. During the introduction of the runtime environment structure in subsection 6.2.1, we already stated that such a facade provides one method for each explicit repair routine. Just like the response executors, we provide one of these facades per source of responses respectively repair routines.

Providing such a facade for each set of repair routines, a routine does not have to be correctly instantiated and called everywhere it gets called. This logic is encapsulated in the method provided by the facade class, which reduces the effort for executing the repair routine to the call of this method. We use a naming for these methods that consists of the prefix *call* followed by the name of the repair routine. The parameter list corresponds to the expected inputs of the repair routine.

Another benefit of such a facade class arises from its usability as an internal DSL. In section 5.5, we stated that providing the access to repair routines through an internal DSL within the execution block of a response has some advantages over their realization within the external response constructs. We proposed the accessibility through methods, whose names correspond to the repair routine names with parameter lists according to the routine inputs. These methods are provided by such a facade class and can be accessed through it. Finally, Xbase provides an `extension` modifier for fields of classes, which makes the methods of the type of the field accessible as if they were methods of the class that contains the field. This can be used to make the plain method call available in the execution block, which is further explained in the context of the code generation in subsection 6.3.3.

6.3 Response Language Specification and Code Generation

The last section introduced the structure and the elements of a runtime environment that implements certain responses and repair routines and makes them available to the transformation environment of VITRUVIUS. In this section, we explain the implementation of the response language and the code generation, which generates an environment according to the one introduced in the last section. We also discuss further aspects of the language, like necessary scoping, validation and debugging.

For the language implementation, we use the Xtext framework, introduced in subsection 2.3.2. The necessary code blocks are reused from the Xbase language, which ships with Xtext. Some constructs, which are reused across the different languages of the MIR language family, are generalized in a so called *MIR base language*, which is shortly introduced in the following.

6.3.1 The MIR Base Language

The MIR base language provides a common basis language for the response, mapping and invariant languages. It defines constructs that are required by all languages and restricts them with necessary well-formedness constraints.

One essential mechanism, which the MIR base language provides, concerns the import of metamodels. All languages define constraints or transformations between metamodels and thus need to allow the specification of references to metamodels and their metaclasses. The MIR base language provides an import construct, which allows to select from all available metamodels and makes the imported one available in the language document.

The languages need to reference metaclasses of the imported metamodels as well as their features. Therefore, constructs that realize the *element-type* and *element-feature* of the responses grammar, specified in subsection 5.1.2, are provided by the MIR base language.

```
routine: AddRequiredRoleImplementation
input: pcm.RequiredRole as requiredRole
match:
  retrieve required element: java.Interface as requiredInterface
    corresponding to requiredRole.requiredInterface
  retrieve required element: java.Class as javaClass
    corresponding to requiredRole.requiringEntity
effect:
  create element: java.ClassifierImport as requiredInterfaceImport
  create element: java.Field as requiredInterfaceField
  add correspondence: requiredInterfaceImport, requiredRole
  add correspondence: requiredInterfaceField, requiredRole
  execute: [...]
```

Listing 6.6: A repair routine for adding the implementation of a PCM required role to the Java model in the implemented response language

6.3.2 Response Language Grammar

The Xtext grammar that specifies the response language is based on the language structure and constructs developed in chapter 5. Some deviations from this specification are required due to technical limitations and are discussed in the following.

Responses and repair routines describe transformations between instances of two specific metamodels. A response defines the source metamodel of the transformation implicitly in its trigger. However, it must not necessarily contain any retrieve or create statements, which identify the target metamodel of the response. To identify the target model, the repair routines that are called by the response would have to be analyzed for the metamodels they affect. Therefore, we decided to require the explicit definition of the source and target metamodel of responses within one responses document and register the defined responses for this metamodel pair.

We also offer the ability to define responses that are executed after any change, defined with a trigger on *any change*. The response executor triggers such a response whenever any change occurs by registering the response for the occurrence of a generic `EChange`. Consequently, most of the logic of this response will be defined in the execution block as there are no change properties that can be used for specifying matcher or effect constructs.

Finally, we have to alter the syntax of some constructs to ensure that the documents can be parsed by the parser generated by Xtext. The matcher and the effect of a response have to be explicitly introduced with the keywords *match* and *effect*. Furthermore, the element references in the addition and removal of correspondences cannot be separated with an *and*, but have to be separated with a comma. An example repair routine that is defined according to the syntax of the responses implementation is shown in Listing 6.6.

6.3.3 Code Generation

For the specification of a language by its concrete syntax, Xtext generates a metamodel that represents an appropriate abstract syntax of that language. Each parser rule of the

language is mapped to a metaclass of that metamodel. This can be influenced by manually defining the element that is created when parsing a rule, so that different rules are parsed to the same elements.

The automatically generated parser delivers instances of this metamodel from a responses document that is conform to the specified syntax. Based on such a model, we specify the generation of executable code.

Xtext provides two ways of defining a code generator. The first possibility is to specify the generation of a textual code representation. This specification requires high effort for ensuring that the generated code is syntactically and semantically correct as it cannot be automatically validated. A second option is to map the elements of the response model to elements of a Java code model. Therefore, Xtext offers a Java metamodel, which can be addressed by implementing a `JvmModelInferrer`. In such an implementation, we have to define which Java elements have to be created for the elements of a response model.

The declaration of such a mapping also applies scoping and validation rules of the Java constructs to the response language constructs that are mapped to them. This mapping is of special interest for the used code blocks, which is further explained in subsection 6.3.5.

When a responses document is open in the generated editor, this editor automatically triggers and updates the mapping to Java constructs, which initially only exist in-memory. The continuous update of these mappings ensures correct scoping and validation of the mapped elements whenever the document is modified. The final generation of Java code according to the defined mappings is performed when the responses document is saved.

A summary of our mappings from the elements of the response language metamodel to Java code elements is given in Table 6.1. We also map the repair routines to a facade class with appropriate methods and the responses to an executor that consists of all defined responses.

6.3.4 Scoping for Response Constructs

All language constructs that reference elements of a certain type require the definition of a scoping mechanism that defines the selectable elements. Because most of our language constructs are realized through code blocks, only two constructs remain for which the scoping has to be defined. These constructs are the specifications of element types and of features of them. Although both constructs are already defined in the MIR base language, their scoping has to be adapted to the context in which they are used.

The specification of element features is used within the trigger for a feature change. It has to be restricted to the features of the specified element type and to those that are conform to the specified change type. We syntactically separate triggers into root changes as well as single-valued and multi-valued feature changes by representing them as different metaclasses. Consequently, the selectable features have to be restricted to those with an upper bound of one if the metaclass of the trigger is a single-valued feature change and to those with an upper bound of more than one if the trigger is an instance of a multi-valued feature change.

For the specification of an element type, all metaclasses of the imported metamodels are valid in general. Only statements for the creation of elements must forbid the specification of abstract types as they cannot be instantiated.

Response language element	Elements in implementing Java code
Response	Java class with name of the response in responses package extending `AbstractResponse`, complete logic in `executeResponse` method
Trigger change type	Type comparison of the actual change with the expected one and response abortion on fail
Trigger element (and feature)	Call of a `checkChangeProperties` method that compares the property values with the expected values and response abortion on fail
Trigger precondition block	Method with change properties as parameters and return type `boolean`, call of that method and response abortion on fail
Repair routine	Java class with name of the repair routine in repair routines package extending `AbstractRepairRoutine`, complete logic in `executeRoutine` method
Matcher required element	Call of `getCorrespondingElement` with appropriate parameters, assignment to a variable with the specified name and routine abortion if the element does not exist, call of `initializeRetrieveElementState`
Matcher optional element	Call of `getCorrespondingElement` with appropriate parameters and assignment to a variable with the specified name, call of `initializeRetrieveElementState`
Matcher non-existent element	Call of `getCorrespondingElement` with appropriate parameters and routine abortion if element exists
Effect element creation	Creation of an element of the specified `EClass`, assignment to a variable with the specified name, call of `initializeCreateElementState`
Effect element deletion	Call of `delete` method for the specified element
Effect correspondence addition	Call of `addCorrespondence` method with the specified elements as parameters
Effect correspondence removal	Call of `removeCorrespondence` method with the specified elements as parameters
Effect execution block	Method in the repair routine class that expects all retrieved and created elements as parameters, call of the `preprocessElementStates` method, the execution block method and the `postprocessElementStates` method

Table 6.1: Representation of response language elements in the implementing Java code

6.3.5 Scoping and Validation of Code Blocks

As stated in subsection 6.3.3 about code generation, we map the code blocks in the response language to Java methods. This applies the scoping and validation rules for Java methods and the contained expressions to the code blocks and their contents within the responses.

The methods to which the code blocks are mapped are added to the class to which the containing repair routine is mapped. They get appropriate and unique method names assigned. They expect the elements that are required in a certain code block as parameters and define an appropriate return type. For example, code blocks for specifying element references are mapped to methods that have a return value of type `EObject`. This makes the specified parameters accessible within the response language code blocks, even in terms of code completion and type checking.

This mapping mechanism allows the convenient implementation of an internal DSL for repair routine calls, as described in subsection 6.2.5. The matcher and effect are mapped to a class implementing the `RepairRoutine` interface and the execution block is mapped to a method in that class. The class gets an `extension` field that references the repair routines facade, which makes the methods of the facade available in the execution block as if they were defined in the class.

6.3.6 Identifier Validity

The code that is generated for a specification of responses uses several identifiers, which must be unambiguous. Different from the development with an IDE that automatically checks this uniqueness, the specification of code generation rules is not capable of identifying potentially ambiguous identifiers in the generated code.

The retrieved and created elements in a response have to be referenced in the generated code. They are made available in the code by the identifiers that are defined in the responses. The language ensures that they are unique within one response. All further identifiers in the generated code must be different from them. We achieve this by starting the names of generated variables with an underscore and disallowing identifiers that start with an underscore in the response language, whenever generated variables and variables defined in language constructs are mixed up.

Furthermore, the names of all generated methods in one class must be different. This is simple for methods of constructs that have a unique name, such as those for the retrieval of elements as they define an identifier for the element. However, some constructs that are mapped to methods do not provide a unique identifier. An example for such a construct is the correspondence addition, which defines code blocks for retrieving the corresponding elements and for defining the tag. These blocks must all be mapped to methods with different names. To ensure that their names are unique, we number these methods.

6.3.7 Runtime Environment Generation

As described in the code generation subsection, the Java code for a responses document is generated when it gets saved. This step produces an implementation of responses, repair routines, a repair routines facade class and an executor for the responses.

This mechanism is insufficient due to two reasons. First, transformation descriptions in the mapping language are also transformed into responses that in turn have to be transformed into Java code. Because these responses are not defined in a responses document but just as an in-memory model, the code generation has to be performed for them manually. Second, the entry point for the VITRUVIUS transformation environment is a Change2CommandTransforming for each metamodel pair. Because the transformations for a pair of metamodels can come from several responses documents and even specifications in the mapping language, it cannot be generated from a single responses document.

We provide an environment generator, which runs the code generation for all given sources of responses. It generates the Change2CommandTransformings, which reference all ResponseExecutors of the different sources. This generator can be triggered externally and accepts responses documents and response models, which are, for example, produced by mappings. The MIR language environment provides a mechanism that collects all relevant artifacts from an Eclipse project and runs the generator with these artifacts. It can be run from the context menu of an Eclipse project.

6.3.8 Constructs for Debugging Responses

Sometimes, a response does not produce the expected results because its specification contains errors. To debug the transformations for finding the errors, debugging concepts for Java code can be reused because the finally executed transformations are defined in Java code. Nevertheless, the developer does not know how the generated code works and how it is structured, thus debugging that code will be difficult for him.

To support debugging, the responses and repair routines generate a log about their execution using the Apache log4j mechanism[1]. The logger access is defined in a Loggable class, which the response and repair routine classes extend.

The information which repair routines get called from which responses due to which changes is of certain interest for a developer. To provide information about that call hierarchy of repair routines, we define a CallHierarchyHaving class. It has a reference to its caller and provides a getCalledByString method, which returns a string that represents the call hierarchy by recursively calling this method on the calling object. The response as well as the repair routine classes extend this class and log their call hierarchy when executing a repair routine.

6.4 Possible Extensions

The prototypical implementation of the response language provides several possibilities for extensions and improvements. The Xtext framework offers several possibilities to improve the usability of the language, for example, by providing an automated formatting. Furthermore, some possible extensions of the response language concept were presented in section 5.8. They also represent possible extensions for the implementation as they have to be implemented after developing the concept.

[1] https://logging.apache.org/log4j/2.x/

In the following, we give a short overview of possible further extension for the language. We introduce a possibility to reduce the cases in which responses that will never be executed are specified. Afterwards, we explain an important extension of the language implementation regarding the modularization of responses specifications.

6.4.1 Restriction of Referenceable Metaclasses

Responses can define constraints that are unsatisfiable and so they will never be executed. This problem cannot be completely avoided because several constraints depend on the concrete consistency overlap types of metamodels. For example, it is impossible to identify if a corresponding element can exist, thus if a retrieve statement can ever be executed successfully. However, especially the trigger specification can be further restricted to reduce the cases in which responses that are never executed are defined.

As explained in subsection 6.3.2, a responses document requires the explicit specification of the source and target metamodel for which the responses are registered. The language does currently not restrict the metaclasses that can be defined in the trigger of a response. It is possible to define a trigger for the modification of instances of a metaclass that belongs to another metamodel than the one the response gets registered for. This can be repaired by restricting the element types that can be referenced in a trigger to the ones that are contained in the specified source metamodel.

A similar mechanism can be used for all specifications of element types. The provided correspondence model does only contain elements of the defined source and target metamodels, thus the specification of metaclasses from other metamodels is useless.

6.4.2 Modularization of Responses Specification

An important possible extension for the response language implementation is the modularization of response specifications. Currently, responses for a pair of metamodels can be split across different documents and the environment generator produces an entry point for the transformation environment that correctly executes all appropriate responses for a given change. Nevertheless, these documents are completely independent from each other.

It is not possible to reference repair routines from other responses documents at the moment. Thus, the responses for a pair of metamodels cannot be separated into meaningful fragments to define reused repair routines in only one responses document that is referenced from the others, for example, for the creation of a Java class.

A mechanism that allows to reference one responses document from another would allow to modularize the responses for a metamodel pair and to define reused repair routines in a central document. The code generation would have to embed the repair routines facade from all imported responses documents into the classes of the repair routines of the actual document. In this case, the `extension` mechanism used for the repair routines facade cannot be reused because different documents can define repair routines with equal names, which would lead to ambiguities. Consequently, the facade class should be made available by an explicit field whose name must be assigned when importing it in the responses document. With this mechanism, access to the repair routines of the imported documents can be provided within the execution blocks of the current document.

7 Evaluation

In this chapter, we present an evaluation of our approach, consisting of the designed response language and its implementation. We evaluate the *functionality*, the *applicability* and the *benefits* of the language.

The functionality evaluation considers two aspects. First, the language must be capable of reacting to all possible changes, which is successfully evaluated with unit tests. Second, it must be possible to specify arbitrary repair logic, which is inherently possible through the possibility of specifying Turing-complete code.

For evaluating the applicability, we implemented a case study for the consistency between architecture description models and code. We reused integration tests from an existing implementation of that consistency repair. Except for one, all tests were executed successfully with our implementation and demonstrate the applicability of our language.

The benefits evaluation shows the conciseness of specifications in the response language compared with their direct implementation in code. Moreover, it reveals the relevance of the provided language constructs based on an analysis of their usage in the applicability case study.

After these evaluation steps, we finally discuss the impact of the evolution of required artifacts, which are the transformation environment, the metamodels and the consistency overlap types defining their consistency. Because a support for these scenarios was not an objective of the language, we do not evaluate this aspect, but only give a short overview of potential benefits of the language for such evolution scenarios.

In the following section, we frequently compare our language or constructs of it with an implementation in general-purpose programming languages. We refer to such transformation specifications as *directly written transformations*.

7.1 Functionality

The central design principle of the change-driven consistency repair language designed in this thesis was its expressiveness. We aimed to develop a language that is capable of specifying the repair of any model consistency constraint. The design of the complete language and the provided constructs was based on that prerequisite.

Providing the possibility to define any change-driven repair of consistency constraints induces two requirements. It must be possible to react to any possible model change that can lead to an inconsistency between models, and it must be possible to define arbitrary repair logic that is capable of restoring consistency. These two requirements were the basis for the development of the triggers and the repair routines of responses. In this section, we recapitulate the argumentations for the fulfillment of these requirements through the provided language constructs.

7.1.1 Reaction to Possible Changes

The first functional requirement to achieve the aimed expressiveness of the designed language is the possibility to react to each change that can be performed in a model. The language concept must consider this requirement and provide appropriate constructs to fulfill it. Furthermore, the prototypical implementation has to be realized correctly, so that it fulfills the conceptual ideas and proves their correctness.

The reaction to each possible model change is simple if the transformation environment observes each modification of a model and executes the transformations consequently. Nevertheless, it is not sufficient to get the information that something changed, but information about what changed is also necessary to execute appropriate repair logic. The modification that was performed and potentially lead to an inconsistency between the changed and another model must describe the complete difference of the model state before and after the change. This can be provided in a forward-backward change description.

As stated in subsection 4.2.2, the possible changes that can be performed in a model are implicitly defined by the meta-metamodel on which it is based. They can be described by a predetermined set of atomic changes, which only affect one model element or property, and can be composed to composite changes. The underlying types of atomic changes can be described in a change descriptions metamodel. This metamodel provides a metaclass for each possible atomic change as a forward-backward change description.

For the Ecore meta-metamodel, we developed the types of possible atomic changes in subsection 4.2.4 and used them to define the exemplary change descriptions metamodel for the response language in subsection 5.1.3. The response language provides the triggers to define the reaction to these changes. It expects the specification of generic change types that are based on that metamodel and that provide the same expressiveness as stated in subsection 5.2.3. The response language implementation is based on the Vitruvius transformation environment and the change descriptions metamodel provided by it, which is conform to our change descriptions metamodel. The implementation assumes that the transformation environment observes all changes and provides them as appropriate change descriptions.

For showing the possibility to react to each possible change with our language, we defined a minimal Ecore-based metamodel, which contains all kinds of model elements and properties that can be modified in the model instances. That metamodel is shown in Figure 7.1. For this metamodel, we wrote responses for all possible changes that transfer the modifications to another instance of the same metamodel and log the changes that were processed. Based on these responses, we defined unit tests that perform all possible modifications to the model elements and their properties of an example model. The tests expect the processing of the appropriate changes to be logged and the other model, which the responses keep consistent, to be equal to the modified one. The operations that are performed on the instance of the example metamodel for producing the different kinds of changes are summarized in Table 7.1.

We implemented responses and tests for the insertion and removal of root objects, the replacement of single-valued attributes and references, and the insertion and removal of multi-valued attributes and references. Regarding reference changes, we further distinguished between containment and non-containment references. These reference types

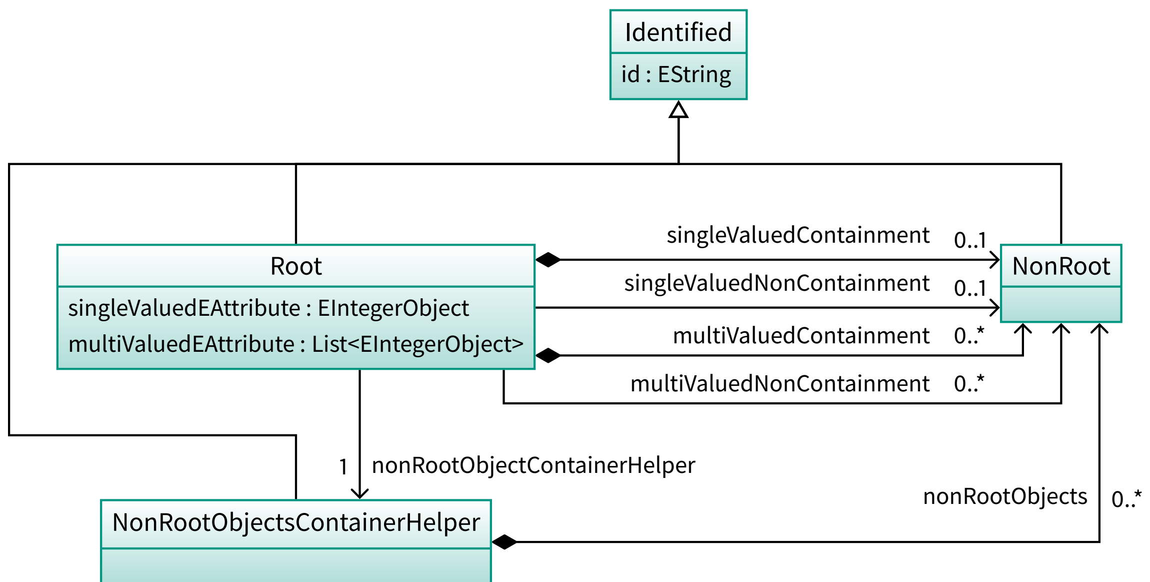


Figure 7.1: A minimal Ecore-based metamodel providing all kinds of elements that are changeable in model instances

are conceptually not different because both kinds of references provide the same types of changes. However, the modification of a containment reference usually arises together with the creation or deletion of the contained object, whereas the modification of a non-containment reference requires the object to be contained somewhere else. To ensure that these cases are handled correctly, we distinguish the cases in the tests.

Ecore supports two further kinds of property modifications, which are the explicit set or unset of them. A property can be declared as *unsettable* and in that case provides a further *unset* state, which is indicated by a special attribute. If the property is explicitly unset, this attribute is set to true. This operation represents a possible model change that can currently not be reacted to in our language, but is provided by the Vitruvius change descriptions metamodel. The same applies to the permutation of elements in multi-valued properties. Nevertheless, both kinds of changes are performed seldom and do usually not require to be transformed. Because both types of changes are already detected by the Vitruvius framework, they can easily be integrated into the response language by adding them to the trigger specification.

The missing support for certain kinds of changes reveals a general difficulty. Our tests revealed that we support the reaction to all kinds of changes that we identified. Although we claim to consider and detect all kinds of modifications in Ecore-based models and supply complete information about each change, we did not prove this claim. It is possible that further, rarely performed changes are possible in Ecore-based models, which our change descriptions do not cover.

With the presented tests, we evaluated two aspects of the response language. On the one hand, we validated that the implemented mapping of generic change types in the responses to concrete change types of the Vitruvius change descriptions metamodel is correct. On the other hand, the tests revealed that the approach executes the correct responses if the triggers are defined correctly.

Change type	Change producing operations
Root element	
Insertion	Create a new Root element as a new model
Removal	Create a new Root element as a new model and remove it
Attribute feature	
Single-valued replacement	Overwrite Root.singleValuedEAttribute with a new value
Multi-valued insertion	Add a new value to Root.multiValuedEAttribute
Multi-valued removal	Add a new value to Root.multiValuedEAttribute and remove it
Containment reference feature	
Single-valued replacement	Overwrite Root.singleValuedContainment with a new NonRoot object
Multi-valued insertion	Add a new NonRoot object to Root.multiValuedContainment
Multi-valued removal	Add a new NonRoot object to Root.multiValuedContainment and remove it
Non-containment reference feature	
Single-valued replacement	Overwrite Root.singleValuedNonContainment with another NonRoot object from Root.nonRootObjectContainerHelper.nonRootObjects
Multi-valued insertion	Add another NonRoot object from Root.nonRootObjectContainerHelper.nonRootObjects to Root.multiValuedNonContainment
Multi-valued removal	Add another NonRoot object from Root.nonRootObjectContainerHelper.nonRootObjects to Root.multiValuedNonContainment and remove it

Table 7.1: Operations for producing atomic changes in the minimal example metamodel

7.1.2 Definition of Arbitrary Repair Logic

The second functional requirement to achieve the aimed expressiveness of the designed language is the possibility to define arbitrary repair logic in response to a change.

In general, such arbitrary repair logic can be defined with a Turing-complete programming language. We provide code blocks that allow the specification of Turing-complete code in all parts of the language. The additional constructs that we provide in the matcher and the effect of a response just provide an abstraction of actions that can also be specified in the execution block and thus do not influence the expressiveness of the language.

Especially the execution block of a response provides the possibility to define arbitrary repair logic according to our functional language requirements. Nevertheless, we also wanted to provide access to the correspondence model through language constructs and specify a three-part structure of the response specification. These aspects does not affect the functionality but the applicability of the language, which is discussed in the following.

7.2 Applicability

In the last section, we have shown that our approach is capable of fulfilling functional requirements regarding the aimed expressiveness of a consistency repair language. In this section, we examine the applicability of the developed language in a case study. Therefore, we implemented responses for the consistency of architecture description models and code that implements them, which was already the basis for the examples in this thesis.

The realized consistency constraints for keeping a PCM model consistent with Java code are explained in the following section. Afterwards, we summarize limitations regarding the applicability of our approach that the case study revealed. Finally, we discuss threats to validity of the case study.

7.2.1 Consistency of Architecture Description and Code

In our case study, we implemented the repair of the majority of the consistency overlap types between architecture description models and code models as proposed by Langhammer and Krogmann [60] in the response language. Our architecture descriptions are instances of the PCM metamodel and the code is defined in the Java programming language, represented as instances of the metamodel offered by JaMoPP. Both metamodels and essential relations between them were already introduced in chapter 3.

The PCM elements whose consistency overlap types with Java code were implemented in our case study are summarized in Table 7.2. The table shows the containment hierarchy of the PCM elements as a tree structure in the left column and explains the mapping to Java elements in the right one. These consistency overlap types have to be instantiated whenever one of the PCM elements is created and inserted into its container. A consistency overlap has to be removed whenever a PCM element is deleted. Furthermore, the consistency overlaps rely on some property values of the PCM elements, which requires the update of the corresponding Java elements to restore consistency whenever one of these properties gets changed. These properties are implicitly specified in the described mappings of the PCM elements to Java elements and are summarized beneath each consistency overlap type description.

Basic PCM elements are repositories, components, interfaces and data types. These elements and their mappings to Java code were already described in detail in chapter 3. Further elements, which were already discussed in section 4.1, are provided and required roles, which describe the interfaces that a component provides and the ones to which it requires access. In Java code, a provided role is represented by the implementation of the provided interface through the providing component class. A required role is represented by a field with the required interface type in the requiring component, a constructor parameter that expects an object providing this interface, and an assignment of the parameter to the field within the constructor.

Further PCM elements that were not yet discussed are signatures and their parameters. They can be defined in PCM interfaces and are directly mapped to methods with appropriate parameters in the interfaces they belong to. Service-effect specifications (SEFFs) describe the implementation of an interface signature by a component and are thus represented by the appropriate method implementation in the component class.

PCM element	Java elements and feature changes affecting the overlap
Repository	Java package with repository name Two sub-packages with names “contracts” and “datatypes” Affecting feature modifications: Name
├ Component	Java package with component name inside main repository package Java class with component name inside the package Affecting feature modifications: Name
├ Provided Role	Interface Implementation of provided interface by providing component class Affecting feature modifications: Interface, Component
├ Required Role	Field of required interface type with role name inside requiring component Parameter of required interface type in each constructor of requiring component class Assignment of parameter to field in each constructor Affecting feature modifications: Name, Interface, Component
├ SEFF	Class Method according to described service signature inside providing component class Modifications: Described Service (Signature)
└ Resource Demanding Internal Behavior	Class Method with behavior name and void return type inside providing component class Affecting feature modifications: Name
├ Interface	Java interface with interface name in “contracts” package Affecting feature modifications: Name
└ Signature	Interface method with signature name and return type inside providing interface Affecting feature modifications: Name, Return Type
└ Parameter	Method parameter with parameter type and name inside expecting signature Affecting feature modifications: Name, Type
├ Collection Data Type	Java class with data type name in “datatypes” package Extension of class with user defined collection type with type parameter according to inner type of collection data type Affecting feature modifications: Name
└ Composite Data Type	Java class with data type name in “datatypes” package Affecting feature modifications: Name
└ Inner Declaration	Field of inner data type with declaration name inside containing composite data type class Getter and setter methods for the field in containing composite data type class Affecting feature modifications: Name, Type

Table 7.2: Consistency overlap types that were considered in the case study

Modification type	Required responses	Implemented responses
Element creation	12	12
Element deletion	12	12 (1 inaccurate)
Property modification	18	18 (3 inaccurate)

Table 7.3: Number of required and implemented responses for the different modification types of the consistency overlap types in Table 7.2

Inner declarations represent the inner types of composite data types. The original PCM contains them as an indirection to assign a name to each inner type, whereas we introduced the inner types as a direct containment in the reduced PCM model in section 3.1.

In our case study, we implemented responses for keeping Java code consistent with the mentioned elements of PCM models. The responses define the consistency repair in cases of the insertion and removal of the specified elements, as well as the modification of the specified properties that influence the consistency overlaps to which the elements belong. For these repairs, 42 responses are necessary, which consist of twelve responses for the creation of PCM elements, 12 responses for the deletion of them and 18 responses to the modification of relevant properties. We implemented all these required responses but with one inaccuracy in the responses for the deletion and the modification of properties of a required role, which expect the component class to define only one constructor. This assumption is usually sufficient but cannot be assumed and is further discussed in subsection 7.2.2. The numbers of required and implemented responses for the different modification types are summarized in Table 7.3.

We used integration tests that perform all described creations, deletions and property modifications in exemplary PCM models and check the corresponding Java code for containing the expected representations and fulfilling required constraints after the modifications. These tests were not specifically written for the responses, but were reused from previously implemented Java transformations that repair consistency between PCM and Java models. 52 of the 53 specified tests validate the correct realization of consistency repair for the specified consistency overlap types. Only the renaming of an inner declaration produced an unexpected result. Nevertheless, this error was not caused by the responses specification but revealed a problem of the Vitruvius framework regarding the update of TUIDs that makes the correspondences of an inner declaration unresolvable if it is renamed. This shortcoming of the framework is currently getting fixed in the Vitruvius project. The tests do not reflect the limitation of our responses for required roles because the tests do not consider the edge case that we do not support.

Most of the provided language constructs concern the interaction with the correspondence model. All implemented responses completely rely on these provided language constructs and do not have to specify manual manipulations of the correspondence model in the execution blocks. Consequently, at least in this case study, the provided language constructs are sufficient for the necessary interactions with the correspondence model.

The implementation of the consistency-restoring responses for PCM and Java shows that our language can be applied in a realistic use case. The generalizability and significance of this result is discussed in the final threats to validity subsection.

7.2.2 Limitations

The evaluation of the response language in the introduced PCM to Java case study revealed one limitation regarding the capabilities of the provided retrieve construct of the response language. The limitation concerns the retrieval of dynamically sized consistency overlaps, whose necessity was not assumed by our language. Nevertheless, it does not affect the expressiveness of the language as a whole, which was already discussed in section 7.1 about functionality.

During the development of the response language, we assumed that a consistency overlap always consists of a static number of elements. The case study revealed that this assumption is not always fulfilled. The consistency constraints for PCM and Java propose the mapping of a required role to a field in the requiring component class and a parameter in each constructor whose value is assigned to the field. Because there can be an arbitrary number of constructors, there can also be an arbitrary number of parameters and assignments to which the required role corresponds. A repair routine for the creation of a required role can call another one for each constructor that instantiates a correspondence between the required role and the constructor. However, it is not possible to retrieve these correspondences correctly with the provided language constructs. The mechanisms for filtering corresponding elements by a function or a tag are not sufficient as we do not need to retrieve a specific element for which we know the condition but want to process all corresponding elements. Consequently, the correspondences between the role and the parameters would have to be managed manually at the moment, or explicit correspondences have to be completely omitted and all parameters must be extracted manually from the constructors each time they are needed. In the implementation of the case study, we simply assumed the existence of only one constructor, which circumvents the problem but makes the responses fail if there is more than one constructor.

It is disputable if the presented mapping of a required role to a constructor parameter is reasonable. Instead of the proposed mapping, the required role could also be delivered to the component through a simple setter method, which is simpler to keep consistent. Nevertheless, a similar mapping could be necessary in practice, thus we have to support its realization.

In the end, the matcher of the response language can be extended to support such a scenario without much effort. The existing language constructs must not be replaced but just expanded by the support for handling dynamic numbers of elements.

7.2.3 Threats to Validity

Concluding the applicability evaluation of the response language, we discuss the significance of the results of our case study. The PCM to Java case study does only represent one example for consistency relations between models. We have shown that the response language can be applied to restore the consistency of Java code with a PCM model whenever a consistency constraint of the considered types gets modified. Nevertheless, the consistency overlap types always define a mapping from a single PCM element to one or more Java code elements. Thus, only the modifications of a single element in the PCM model have to be considered for the consistency of each single consistency overlap type.

Keeping the PCM model consistent with the Java code in the opposite direction, thus whenever modifications are performed to the code, requires the reaction to modifications of different elements for one consistency overlap type.

Although it makes no difference for the responses if one or more elements in the changed model belong to a consistency overlap, for example, the retrieval of elements gets more complicated because not all necessary elements are corresponding to the changed element or at least an element in the changed model. In completely different domains than architecture descriptions and code, further unexpected requirements can arise, which we did not consider when designing the response language. Therefore, the response language has to be evaluated with further case studies in the future, to evaluate whether it can be applied in other scenarios as well.

Furthermore, the case study evaluated the applicability of the language in terms of the capabilities and the correct realization of the language constructs and not in terms of usability. We applied the language to the scenario on our own, but if the language shall be used by domain experts for specifying the consistency repair between different metamodels, it is important to evaluate that the language can be applied by such a user and that it supports him in his task. Therefore, an empirical study with test groups that implement the repair of a given set of consistency overlap types, once with our approach and once using directly written transformations could be performed. The average number of errors and the average required implementation time can be traced and compared.

7.3 Benefits

In the previous sections, we evaluated the functionality and applicability of our approach. Although those aspects are important, the approach must provide a benefit in contrast to other approaches so that it makes sense to apply it. Therefore, we discuss some of the benefits of the response language in the following.

First, we consider the conciseness of the provided language constructs for specifying triggers and handling correspondences and compare them with their implementation in code. Afterwards, we evaluate the relevance of the language constructs by analyzing their usage in the context of the PCM to Java case study. Finally, we discuss the relevance of the reuse mechanism for repair routines and also analyze its usage in the context of the case study.

7.3.1 Trigger Specifications

The response language provides a highly declarative construct for specifying the trigger for the execution of a response. It usually just requires the declaration of the expected type of the change and of the modified element, and in case of a feature change the considered feature. The complete logic for realizing that check is derived from the specification.

A trigger that specifies the reaction to the creation of a component within a repository is realized in the response language as shown in Listing 7.1. The different checks would have to be declared explicitly in an Xtend implementation as shown in Listing 7.2. The Xtend implementation requires higher effort to specify the condition for each repair routine and

```
trigger: insert in list pcm.Repository[components]
```

Listing 7.1: Example trigger implementation in the response language

```
if (change instanceof InsertRootEObject) {
   val typedChange = change as InsertRootEObject;
   if (change.affectedEObject instanceof pcm.Repository &&
      change.affectedFeature.name == "components") {
      // Repair routine specification
   }
}
```

Listing 7.2: Example trigger implementation in Xtend code

also increases the possibility to make mistakes. The correct change properties have to be checked for their type and the right feature name has to be compared. The response language performs automatic consistency checks and, for example, only allows to define features that are provided by the specified element type and that can be affected by the specified change. In contrast, the Java implementation does not provide any validation for the correctness of the trigger.

As the example shows, the specification of triggers in the response language is usually more concise and provides a better validation than a Java implementation. Moreover, the abstraction from concrete change types of the change descriptions metamodel requires less knowledge about the possible changes and their representations by model elements.

7.3.2 Correspondence Handling

The correspondence model is a central artifact for the consistency repair, and the access to it is abstracted by several constructs in the response language. The matcher provides constructs for defining different retrieve operations on the correspondence model, whereas the effects define constructs for modifying it.

In the best case, the methodologist does not have to care about the implementation of the correspondence model at all, but just has to know about the provided language constructs. Furthermore, the access to the correspondence model with language constructs is usually more concise and abstracts from several necessities. This benefit is shown by an exemplary comparison of the retrieval of a Java compilation unit that corresponds to a PCM component. In Listing 7.3, this retrieval is defined in the provided response language construct, which just requires the specification of the expected type, the source element and a name for making the retrieved element accessible. An implementation in Xtend code is shown in Listing 7.4. It has to perform the required actions manually and must first retrieve all corresponding elements of the component, filter them for compilation unit instances and assure that there is a corresponding element.

Apart from the conciseness, the example shows that a Java implementation has to consider further requirements, such as the existence of a corresponding element, on its own. Furthermore, is has to perform several operations that are automatically and correctly realized through the response language construct.

```
retrieve required element: java.CompilationUnit as compilationUnit
  corresponding to component
```

Listing 7.3: Retrieving the compilation unit corresponding to a PCM component in the response language

```
val correspondingEObjects = correspondenceModel.getCorrespondingEObjects(component)
val compilationUnits = affectedEObjects.filter(typeof(CompilationUnit))
if (!compilationUnits.isEmpty) {
   val CompilationUnit compilationUnit = compilationUnits.get(0)
   // Repair routine for compilation unit
}
```

Listing 7.4: Retrieving the compilation unit corresponding to a PCM component in Xtend code

The response language constructs provide a more concise and less error-prone way of specifying interactions with the correspondence model. However, the expressiveness has some limitations, which were already discussed in subsection 7.2.2 and which require the methodologist to know about the methods of the correspondence model and their semantics to retrieve the appropriate elements. Even if the constructs are not sufficient in each case, the case study indicates that such cases may be rare.

7.3.3 Relevance of Language Constructs

The language constructs of the response language primarily address the access to the correspondence model. The benefits of these constructs regarding conciseness and error prevention were discussed in the previous subsection. Nevertheless, to profit from the benefits of the constructs, they must first of all be relevant for the specification of many responses.

We analyzed the usage of the offered language constructs in the PCM to Java case study and summarize the results in Table 7.4. The consistency repair consists of 42 responses, which use 23 additional external repair routines. The retrieve construct is used 59 times and thus on average more than once per response, which meets the expectations as all responses operate on the elements of consistency overlaps. The retrieve statement for optional elements is required for PCM data types in the case study and is utilized five times. The language construct for requiring the absence of a corresponding element is only used once and in that case primarily to test its functionality. The case study does not require the usage of this feature, but the implementation of consistency repair for modifications in the Java code would require it, as discussed in subsection 5.3.5.

The remaining language constructs are provided in the effects. The creation of elements and addition of correspondences is used half a time per response on average. In the case study, the creation of elements and the addition of a correspondence were always performed together. Although there are definitely cases in which one construct is used without the other, it could be reasonable to provide a language construct that combines both operations if further case studies show that they are used in only few cases independently from each

Language construct	Count
Response	42
External repair routine	23
Retrieve required element	59
Retrieve optional element	5
Require element absence	1
Create element	21
Delete element	13
Add correspondence	21
Remove correspondence	0

Table 7.4: Numbers of used response language constructs in the case study

other. The removal of a correspondence is only necessary if none of the corresponding elements is deleted because otherwise it is removed automatically. That is the reason why this language feature is not used in our case study.

The set of elements that is considered in a response consists of the retrieved and created ones. Our case study uses 85 creating and retrieving constructs, which are used by a set of 42 responses. Consequently, a response in the case study deals with two elements of another model on average, which emphasizes the relevance of the provided language constructs.

7.3.4 Reuse of Repair Routines

The response language allows the specification of external repair routines, which can be called by responses and other repair routines. The essential purpose of them is to provide a reuse mechanism.

We analyzed the usage of repair routines in our PCM to Java case study to see how they were used to implement the consistency repair and summarize the results in Table 7.5. The complete repair specification of 42 responses uses 23 external repair routines. These routines are called 62 times, which means that each response or repair routine calls almost one external repair routine on average. Each repair routine is called 2.7 times on average, which shows that they are suitable for reusing repair specifications.

Furthermore, we distinguished the repair routine calls by their reason. Apart from reuse, there are two further cases in which the specification of external repair routines is necessary. The iteration of calls for handling dynamic counts of consistency overlaps was already discussed in subsection 5.5.3 and is necessary two times. First, we need it for the removal of all constructor parameters of a require role, and second, it is required for updating the method names of SEFFs whenever the name of the implemented interface signature changes in all basic components that provide the interface.

Another case in which external repair routines are necessary is the chaining of repair routines. If one repair routine requires an element that must first be created by another, reused repair routine, the element requiring repair routine must be externalized because it cannot retrieve the element in its matcher as it is not yet created. For example, the

Language construct	Count
Response	42
External repair routine	23
for reuse	19
for chaining	2
for iterating	2
Repair routine call	62
for reuse	58
for chaining	2
for iterating	2

Table 7.5: Numbers of responses, repair routines and repair routine calls in the case study

response for the creation of a collection data type reuses the repair routine for creating the corresponding Java class. Because a collection class has to be added as its super class, the newly created Java class must be retrieved, which is only possible in another repair routine. This chaining is necessary twice in the case study.

Consequently, 19 of 23 repair routines are specified for reasons of reuse and they are called 58 times. Thus, each of these repair routines is called almost 3 times, which confirms the necessity of providing such a reuse mechanism in the response language.

7.4 Evolution Scenarios

The main goal of the response language was the provision of a consistency repair language with certain requirements to its expressiveness. We also support the specification of responses with language constructs that abstract from the correspondence model handling. We did not consider the impact on certain evolution scenarios during the language design. Nonetheless, we shortly discuss the consequences of the language design for the evolution of different artifacts in the following. The discussion only provides an overview of qualitative evolution impacts to the responses and gives an overview of potential benefits in contrast to directly written transformations.

We consider evolution scenarios of the artifacts on which the specification of responses is based. These artifacts are the transformation environment, the metamodels for which the consistency repair is specified, and the consistency overlap types considered in the responses. If one of these artifacts changes, the adaption of the transformations is required.

7.4.1 Transformation Environment Evolution

The transformation environment can evolve in different ways. First, the essential execution mechanism can change, for example, due to a modification of the interface that the transformations have to implement. Second, the change descriptions metamodel can evolve, and third, the correspondence metamodel can get changed.

Because the response language abstracts from the integration of responses into the execution environment, the first evolution scenario only requires the environment generator of the language to be updated and to be re-executed for existing response specifications. Especially if consistency-restoring transformations for several metamodels in different projects are defined, only having to re-execute the generator once is a benefit. For directly written transformations, the implementation of the access point for the transformation environment, in the VITRUVIUS implementation the Change2CommandTransforming, must be updated manually in each case.

Modifications of the change descriptions metamodel can only influence the representation of certain changes but not the information they contain, as this is defined by the meta-metamodel. As the response language abstracts from concrete types of the change descriptions metamodel, only the change inference mechanism has to be adapted but not the implementation of any response. Even the modification of a change property name does not require a modification of already written responses because the property is passed to the code blocks as a parameter that can keep its old names. In directly written transformations, each modification of the change descriptions metamodel requires the adaption of all references to the elements of the metamodel that were changed.

Modifications of the correspondence model are also completely transparent to a methodologist because the language constructs abstract from the correspondence model implementation. Furthermore, most of the interaction logic with the correspondence model is statically specified in the runtime environment of the response language and is not generated for each response. Consequently, most changes in the correspondence model will not even require a re-execution of the code generation for the responses.

Even if an implementation of directly written transformations provides a good abstraction from these artifacts, only this concrete implementation benefits from the adaption to an evolved artifact. Using the response language, only its code generation must be updated and all users of the response language only need to re-execute the code generation, if at all, and no further evolution effort is necessary.

7.4.2 Metamodel Evolution

The evolution of a metamodel for which consistency-restoring transformations are specified can require their adaption. Most metamodel changes are backwards compatible to keep existing instances usable. Nevertheless, if backwards compatibility is not required, metamodels can evolve in ways that affect transformations specified for them.

The structure of a metamodel depends on its context and is not generic the correspondence and change descriptions metamodels. Therefore, it is difficult to provide specialized language constructs that abstract from certain aspects of concrete metamodels, which is why the response language does not provide any related constructs. Even the abstraction of usual model operation, such as the assignment of an attribute or reference, does not provide any abstraction from a concrete metamodel.

As long as responses and directly written transformations reference metamodel elements and properties using typed objects, which are checked by the language compiler, the developer gets informed about errors due to a changed metamodel. However, assuming a trigger implementation as shown in Listing 7.1 and Listing 7.2, the response language

benefits from the validation mechanism that checks if the specified feature belongs to the expected element type. If this feature is changed, the response language validation reports the error statically to the developer. The Java implementation will fail during its execution, potentially even silently so that just unexpected results are produced because the feature comparison fails every time.

A realistic metamodel change could, for example, be the improvement of the package hierarchy representation in the Java metamodel of JaMoPP by explicit containment references instead of the implicit namespaces list. Another possible change is the realization of PCM provided roles as lists of provided interfaces within a component instead of a dedicated metaclass. Both modifications require the responses of all consistency overlap types in which these elements are contained to be updated. The benefit of the response language in these scenarios is its conciseness, which may reduce the number of references to a modified element. At least, the number of references will not exceed the one in directly written transformations. Consequently, a fewer number of references must be adapted.

7.4.3 Consistency Overlap Type Evolution

The last evolution scenario is the modification of consistency overlap types on which responses are based. Such a modification can have two reasons. Either an error in the specification of consistency constraints is found, or, if using a synthetic specification, the prescription of the consistency overlap types changes.

As consistency overlap types and their repair implementation can be rather different, it is hard to categorize possible modifications of them and discuss their impact on response specifications. Nonetheless, we can at least make a statement about the number of affected responses. If a single consistency overlap type is changed, only the responses that are responsible for it possibly have to be updated. These are at most as many responses as features of metaclasses are used in the consistency constraints because only the modifications of those features require responses that update such an overlap.

An example for a consistency overlap type evolution in the context of our case study would be the mapping of PCM components to a central components package instead of one package for each component. This modification requires the adaption of the consistency overlap types of both the component and the repository because the component must not have a corresponding package any more but the repository must be mapped to an additional package. The responses for the insertion and removal of a component into or from a repository, as well as for the creation, renaming and deletion of the repository have to be updated. Directly written transformations must also implement reactions to all these changes, thus at least as many code fragments have to be updated.

One benefit of responses regarding the evolution of a consistency overlap type is that the language design is based on the concept of these types and provides constructs for accessing elements of a consistency overlap. Consequently, a modification of a consistency overlap type can potentially be easier transferred to responses than to directly written transformations that do not have this explicit structure. However, this is just a hypothesis, which has to be verified in a controlled study. Different test groups have to conduct such evolution scenarios in responses as well as directly written transformations and are compared regarding the required time and the performed errors.

8 Related Work

In this chapter, we give an overview of work that is related to our approach and discuss the differences between them. After classifying our approach in the context of general change-driven development, we consider different topics of model consistency and its repair, as this thesis contributes to them.

We first discuss different definitions of consistency and relate them to our definition. Then, we look at some general topics of consistency repair, often also referred to as *model synchronization*, like its formalization, the relevance of changes and transformation frameworks. After shortly discussing the relation to approaches for the problem-specific consistency repair, we consider declarative approaches for restoring model consistency. Finally, we compare our language to other approaches for imperative consistency repair, which is the topic to which our approach is most closely related.

8.1 Change-Driven Development and Reactive Programming

We provide an approach for consistency repair that is based on atomic changes. The concepts are similar to those in *event-based* or *event-driven* software development [77]. Events lead to the execution of some program logic as a reaction to that event. Event-driven programs are often implemented manually using common imperative object-oriented language constructs and possibly fitting design patterns, such as the observer pattern [36]. The mechanisms for notifications about events, such as the observer pattern, are the concepts upon which a transformation environment relies.

In contrast to change-driven development, reactive programming focuses on language constructs to define data dependencies that are automatically updated [7]. The mechanism of calling the reactions to events is not implemented manually but is provided by the environment, which is what change-driven model consistency also aims for. The reactive programming idea was initially described by Elliott [26] and was a concept of a virtual reality modeling language. Later on, he used this idea for a reactive animation framework [28] and recently in the context of general functional reactive programming [27]. Many reactive programming frameworks have been developed, especially for functional languages. Popular implementations for Haskell are Fran [26], NewFran [27] and Yampa [47]. Reactive programming constructs for Scala have been introduced in Scala.React [66].

The relation of reactive programming to model consistency can be mainly seen in declarative transformation languages. They describe relations between metamodels from which mechanisms to preserve consistency of these relations are derived. Consequently, they also describe data dependencies like reactive programs do, rather than control flow dependencies in imperative approaches.

8.2 Consistency Specification and Validation

To reason about consistency repair, a definition or common understanding of consistency is necessary. We presented our view on and definitions for model consistency in section 4.1. The specification of consistency is highly related to its validation, as the validation relies on a consistency specification and a specification usually implies a way how to validate it. For example, our definition of consistency is based on consistency overlaps and consistency constraints, which implies the validation of consistency by checking those constraints.

Several approaches describe consistency in a similar way. One of them is the correspondence specification mechanism of Linington [63]. He provides a consistency specification that comprises correspondence rules and correspondence links. The correspondence rules consist of constraints, thus they are similar to our consistency constraints. The correspondence links can be seen as partial consistency overlaps, as they describe two sets of model elements and reference one consistency rule, whereas we allow multiple constraints to be defined on them. In contrast to our specification, this one does not explicitly provide a concept for the description of consistency overlap types on the metamodel level, which is the basic concept for our consistency repair approach.

Diskin, Xiong, and Czarnecki provide a way of checking consistency between heterogeneous models [22]. They informally use the term *overlap* similarly to our definition and also consider the explicit specification of *inter-metamodel constraints*, which represents the concept that we call *consistency constraints*. Nevertheless, they also omit the explicit specification of the type level of overlaps. Their consistency validation merges models into views for certain aspects of the system, which allows to reduce the consistency check to constraints on these views instead of complete models.

Another approach for the validation of model consistency is based on so called design rules [82]. These rules define conditions that are validated on a context, which can be a model element or a metaclass. That concept of design rules is similar to that of our consistency constraints but can be defined on both the metamodel and the model level. We only consider constraints that have to hold level as they are sufficient for our consistency-restoring transformations defined on them.

The discussed approaches, as well as ours, rely on some kind of correspondence model, either explicitly provided or implicitly extracted from the models, which identifies if certain elements represent overlapping information. The correspondence models consist of elements and dependency rules represented by some kind of constraints. A different specification is provided in an approach for ensuring multi-model consistency [93], which utilizes the entity-relationship model [19] for specifying correspondences between model elements. The paper proposes to use QVT-R for specifying the rules that define actions for restoring consistency after such a relationship was affected by a change.

8.3 General Topics of Model Consistency Repair

The repair of model consistency always relies on model changes. Only a model modification can introduce an inconsistency if the models were consistent before. Consequently, all discussed approaches have to deal with changes. Most approaches are *difference-based*,

calculating the change from the differences of an old and new model state. In contrast, our approach relies on an *edit-based* change detection, which gets the information about model changes directly. The approaches using an edit-based change detection are what we consider as *change-driven*.

A formalization for model consistency repair and changes is given by Hettel, Lawley, and Raymond [46]. They define round-trip engineering, which also concerns the preservation of consistency between two models. Furthermore, they give a formal definition of so called “change translation functions”, which represents our view on change-driven model consistency proposed in subsection 4.4.1, relying on the transformations of a change in one model into changes in the other model. While we only consider atomic changes in the source model so far, they define the mechanism on composite changes. However, their definitions of changes and change translation rely on a specific model and metamodel definition, which is not fully conform to EMOF. Moreover, they only formalize consistency repair but do not provide concepts for a realization of it.

The granularity of changes to react to is also discussed by Wimmer, Moreno, and Vallecillo [97]. They propose the usage of coarse-grained changes as they better fit to the developer’s expectation of considering high-level evolution rather than low-level atomic changes. We also want to support these scenarios by the integration of composite changes and the changes of constraint satisfaction in future work. Nevertheless, an abstraction from atomic changes as they propose is beyond our concept of consistency overlaps and thus contrasts the concepts of our approach. Furthermore, their approach relies on a difference-based detection of atomic changes, which is not capable of reconstructing each change correctly, rather than the edit-based mechanism that we use.

Consistency repair requires a transformation framework like Vitruvius. Another framework for the consistent evolution of models is CoWolf [37]. The approach builds upon the specification of transformations between pairs of metamodels and rules for the detection of changes in the models. These rules have to be specified in the Henshin language [4], which relies on triple graph grammars, discussed in subsection 8.5.1. It is integrated into the Eclipse environment and automatically executes the appropriate consistency-restoring transformations after changes. However, the approach focuses on the development of the transformation environment and does rarely consider how the consistency repair mechanisms are specified using Henshin rules.

8.4 Problem-Specific Model Consistency Repair

Several approaches have been proposed for the preservation of consistency in specific domains, thus between instances of predefined metamodels. Especially for the consistency between design models in UML and the implementation of a software system, for example, written in Java, round-trip engineering approaches have been researched for a long time. These approaches propagate changes between both models. Examples are the commercial tools Borland Together [14] and Enterprise Architect [87], as well as the research project Fujaba [70]. Nevertheless, these approaches are tied to the consistency between UML diagrams and code, whereas our approach allows to define consistency repair between arbitrary metamodels.

The DUALLY project [67] considers the consistency of architecture descriptions. The developers claim that a variety of architecture description languages exist for different purposes of which often several are used within one software project. Consequently, such descriptions have to be transformed into each other. The approach provides a core metamodel that defines common concepts of architecture description languages to which relations of all different languages can be specified. From these relations, transformations between the languages are generated. Because only relations to the core metamodel have to be specified, the required effort for the integration of new languages is reduced in contrast to general-purpose approaches for preserving consistency. They usually require the definition of consistency between pairs of models as there is no common metamodel to which all concepts can be mapped. The approach is also extended to support incremental updates [29], rather than re-executing the transformations after modifications in batch mode, which leads to the loss of information.

The realization of such problem-specific approaches is rather different than general-purpose approaches for restoring model consistency, such as the one we introduced in this thesis. While we have to ensure problem-independence by allowing the specification of any kind of consistency and its repair, the problem-specific approaches can rely on specifics of the domains.

8.5 Declarative Model Consistency Repair

In this section, we discuss declarative approaches for model consistency repair. They rely on the specification of consistency rules that describe constraints for consistent models, from which the repair of consistency is automatically derived.

As we discussed in subsection 4.4.3, the repair of specified consistency constraints provides degrees of freedom, which can even require the determination by the user who introduced the inconsistency. Because declarative approaches automatically derive the repair of consistency, it is not possible to influence the possible way of repair to use. Even approaches that let the user select from different models that fulfill the consistency specification after a change do not allow to specify the way of consistency repair within the consistency specification a-priori. They require this decision from the user every time. This automatic derivation of consistency repair is already the essential difference to our approach and imperative approaches in general, as they explicitly define the repair actions. However, most research about model consistency repair focuses on declarative approaches.

A simple declarative approach for consistency repair is presented in an early paper about incremental model transformations [51]. It distinguishes create, update and delete operations depending on whether a target element *does exist* and whether it *should exist* due to a change. Nevertheless, this approach is limited as it does not allow to specify consistency overlaps explicitly but relies on implicit overlap specifications that describe the retrieval of its elements using model information. Moreover, it does only support simple redundancy mappings.

The most important category of declarative approaches for consistency repair are bidirectional transformation languages. Some other approaches are based on a consistency specification with logical expressions. Both categories are discussed in the following.

8.5.1 Bidirectional Transformation Languages

Bidirectional transformation languages allow to specify relations between instances of metamodels, from which transformations that keep these relations consistent in both directions are derived. As stated by Stevens [92], the purpose of bidirectional transformations is to repair consistency between models.

A common approach for bidirectional transformations are Triple Graph Grammars (TGGs) [62]. They allow the specification of graph transformation rules based on three graphs. Two graphs represent the patterns in the source and target graphs and a correspondence graph links the elements of these two graphs. Because models can be considered as graphs, TGGs can be used to define model transformations [38, 2]. For model consistency preservation, TGGs initially had some drawbacks, especially the lack of support for delete operations and the limitation to equivalence relations between attributes. Although delete operations are supported by recent TGG tools, the limitation of attribute relations is just addressed in current research [3, 57]. TGGs were even extended to handle concurrent modifications of different models that have to be kept consistent [45], which our approach is not capable of.

QVT-R [72] is the bidirectional transformation language of the QVT standard. It allows the specification of relations between model elements, which can be seen as the specification of the consistency overlap types of two metamodels. QVT-R allows to execute the transformation in a checkonly mode, which validates if the rules are fulfilled by the models and thus are consistent, or in an enforce mode, which restores the consistency relations specified in the rules. Current implementations of QVT-R are not mature as they do not support the complete standard. One implementation is Echo and is discussed later.

Tratt presents a change propagating language for keeping models consistent [96]. As the change propagation is derived from defined relations instead of requiring their explicit specification, it is similar to QVT-R. The approach requires the specification of relations between elements and thus does also not provide a possibility to influence the derived repair mechanism. Tratt works out several aspects of change propagation that we considered as well. Especially what we defined as implicit and explicit consistency overlap specifications is considered as a mechanism for relating source and target model elements. Nevertheless, it omits a concept like consistency overlaps.

The discussed approaches depend on the specification of relations between model elements. They derive a fixed repair logic that predetermines how the repair of a certain inconsistency is performed. As already mentioned, this lack of influenceability is the main drawback in contrast to our approach. The following approaches relax this restriction as they allow the user to select from different possible repairs.

Echo is a model repair tool that relies on the definition of consistency constraints in OCL and QVT-R [65]. It transform metamodels and the consistency specification into a formal specification in the Alloy language [64]. This language uses a SAT solver to find instances of the target metamodel after a change in an instance of the source metamodel that are closest, depending on some metrics, to the actual one. It is also capable of proposing several possible consistent target models to let the user select from.

Reder proposes an approach for repairing consistency based on the specification of design rules [81, 80]. From these design rules, repair trees, which are trees of actions

that describe how such a violated rule can be repaired, are derived [82]. False and non-minimal repairs are removed from this tree to let the developer select from the remaining, reasonable alternatives. Although the realization is rather different, the idea is similar to the one of Echo, as both approaches try to derive minimal repair operations for consistency rules and let the developer select from the most appropriate ones.

In contrast to the other approaches, the last two provide alternatives of restoring a consistent state rather than predetermining the repair logic. In these approaches, the user has to select a possible repair each time a change is performed, even if the expected repair of a certain kind of modification is always the same. Our approach allows to define the appropriate repair for a certain case within the transformation. As a consequence, it reduces the cases in which the user has to be asked for a decision but still keeps the possibility to involve him, if necessary, rather than completely automating the repair logic.

8.5.2 Logical Expression Approaches

Some approaches for consistency repair use logical languages to specify consistency and derive its repair. One approach is the change propagation by answer set programming (ASP). A model that was generated by a transformation can be changed so that the new state can never be the result of a transformation. ASP allows to approximate a source model that can be transformed in a model that is closest to the changed one [20]. Considering models and metamodels as graphs of nodes, edges and properties, they can be represented as rules in a logical program that performs that approximation.

Eramo et al. present a similar approach that focuses on the propagation of changes using ASP [30]. As stated in that work, the approaches using ASP have limited expressiveness, for example, because they only support scalar data types and weak abstraction. Especially the limited expressiveness makes them hardly applicable in the context of model consistency repair. Furthermore, the specification of dependencies in logical programs requires a certain understanding of logical expressions, whereas our approach allows the specification of constraints and their repair in imperative code, which developers are usually familiar with.

An early approach by Nentwich, Emmerich, and Finkelstein considered the consistency of documents through repair actions [69]. They define the consistency rules of documents in a language based on first-order logic, from which they claim to derive a complete set of necessary repair actions for changes that violate the rules. They require a repair administrator who removes the generated actions that would fulfill the constraints, but are not reasonable. The idea of revising the transformations derived from a declarative specification is interesting, as this idea could be adapted by a further integration of the mapping and response language of the MIR language family.

8.6 Imperative Model Consistency Repair

Most approaches for model consistency preservation focus on declarative specifications, from which the consistency repair mechanism is derived. Some other approaches follow an imperative approach, which requires the repair mechanism to be explicitly specified. Whereas the imperative approach usually requires higher effort, at least because both

directions of transformations have to be defined separately, it provides higher flexibility as the way in which consistency is restored in certain situations can be defined explicitly. This possibility is useful, as the satisfaction of a single consistency specification can be achieved by different kinds of repair implementation, as stated in subsection 4.4.3.

8.6.1 General Approaches

Xiong et al. proposed an approach that automatically deduces a backward transformation from a forward transformation defined in the Atlas Transformation Language (ATL) [98]. Thus, it generates both directions of the consistency repair for a consistency overlap type from the specification of the transformation in one direction. The approach is based on a formal specification of the synchronization problem consisting of four properties that rely on the lens laws [33, 34]. The authors specify the generation of reverse transformations for certain model operations, more precisely the replacement, the deletion and the insertion of model elements. The modifications that have to be performed by the transformation in the target model must also rely on these supported modifications types. Furthermore, the inversion of attribute value calculations and the parametrization of a transformation with user decisions is not covered by the approach. These characteristics make it less expressive than our approach regarding the specifiable relations between models. Nevertheless, it is capable of handling concurrent modifications of both models, which we do currently not support.

An explicitly change-driven approach for restoring model consistency are the so called *event-driven grammars* [40]. They allow to specify the possible modifying actions in a model and define events in response to them as TGG rules. This approach even allows to handle delete operations with TGGs. Although the change-driven concepts and the triggered execution of events is comparable to our approach, they define metamodel-specific change events in the TGG rules rather than the metamodel-independent changes we support. Moreover, we allow to define the changes to react to and the repair routines to execute imperatively. In contrast, their approach allows to define the triggering of a repair imperatively through the change it reacts to, but the repair itself is derived from a declarative TGG rule.

8.6.2 Reactive Model Transformations with VIATRA

Bergmann et al. have developed the VIATRA framework [12]. It relies on a so called *Event-driven Virtual Machine (EVM)*, which can be used to execute *reactive programs*. In their understanding, reactive programs execute code in reaction to an event, which is why such a reactive program consist of two parts specifying the event and the reaction. This concept is especially applied to the context of model transformations, in which transformations in reaction to events can be specified. The events are expressed by modifications of graph structures, which can be expressed in their EMF-IncQuery language [11]. This language allows to define queries on models, which are defined as graph patterns and are matched with the model graph. A reactive program can define the operations to perform when a certain graph pattern was created, updated or deleted. Those operations are specified in imperative code that accesses the elements of the matched graph pattern.

The current VIATRA project arose from VIATRA2 [10, 79]. Just like in the current version, a trigger had to be defined by the appearance, update or disappearance of a graph pattern. In contrast to the current VIATRA framework, the reaction had to be specified as a declarative graph rule rather than through imperative operation. Such a rule specified conditions that had to hold after a specified event occurred.

The VIATRA approach is similar to ours, as it defines an explicitly change-driven approach for restoring model consistency. The graph patterns describe the consistency overlap types of the metamodels. It is even possible to define more complex events than the atomic changes, which we currently support, as more complex graph patterns can be specified for triggering transformations. This is what we want to support with the constraint satisfaction change triggers proposed in subsection 5.8.2.

In contrast to our approach, the EMF-IncQuery language, which is used to define the graph patterns, is less expressive as it is not Turing-complete. For example, cycles of references cannot be detected with the query language, which is possible with our approach due to the possibility to specify Turing-complete triggers and matchers. Correspondences have to be managed in an explicit trace model that must be explicitly addressed by graph patterns and from which is not abstracted such as in the response language. As the transformation execution after an event relies on the EVM, the methodologist has to understand parts of this additional layer of abstraction for debugging. Our responses are transformed into plain Java code with only few API calls, which does not require further knowledge by the methodologist. Finally, the concept of graph pattern matching is a completely different and highly declarative concept compared to the rather imperative response language. Although the latter one may be more verbose, it is potentially easier to learn as developers are usually familiar with the specification of imperative code.

9 Conclusion and Future Work

In this last chapter, we conclude the thesis with a summary of its contributions and an overview of possibilities for future work.

9.1 Conclusion

This thesis presented a change-driven transformation language for the repair of model consistency, which relies on a definition of consistency that focuses on the comprehensibility by users of such a language. We introduced an implementation of these concepts based on the Eclipse Modeling Framework and presented an evaluation of the applicability and the benefits of our approach. Finally, we compared our approach with related ones.

In the main part of the thesis, we first introduced our definitions of model consistency and model changes. We proposed a definition for model consistency that breaks the specification of consistency between two models down to the definition of smaller relations between certain sets of elements, which we call consistency overlaps. Based on atomic model changes, which can induce model inconsistencies, we defined our view on consistency repair, which lead to the concept of change-driven consistency repair. Finally, we derived an execution structure, consisting of three steps, that change-driven consistency repair always possesses.

After introducing those theoretical foundations, we designed a change-driven language for consistency repair, called the *response language*. It relies on the proposed consistency definition and separates the repair description into three parts, according to the identified steps of change-driven consistency repair. Different language constructs for restricting the considered changes and managing the correspondence model, which provides information about related elements, were developed. We ensured that it is still possible to define maximum expressive repair actions in terms of being Turing-complete. Afterwards, we discussed further properties and responsibilities of the language and considerations for writing responses. We finally provided an overview of possible extensions to the approach.

The development of the language concepts was followed by an overview of a prototypical implementation of the response language. We proposed a structure for the runtime environment of the final consistency-restoring code, which is generated from a specification in the response language. The realization of the language specification with the Xtext framework and important aspects such as the reuse of a Turing-complete programming language in certain language constructs were discussed.

In the evaluation, we demonstrated the applicability of the language in a realistic use case, the consistency between architecture models and object-oriented code. We figured out the relevance of our language constructs and that the approach provides a more concise specification of consistency repair than without a specialized language.

Nevertheless, further case studies have to be performed to verify the general applicability of the approach in future work. In a final discussion of related work, we positioned our approach in the field of research about model consistency repair.

To sum up, we revealed that it is possible to identify a general structure of change-driven consistency repair specifications based on a certain definition of consistency. We developed a language that is tailored to the repair of model consistency and provides constructs according to the identified structure while still providing maximum expressiveness. Because we successfully applied our language to a realistic consistency scenario, we are confident that the approach can be reasonably applied to further scenarios by other users.

9.2 Future Work

In future work, three important topics can be further investigated: the language can be extended by further concepts, aspects of transitivity of consistency repair can be researched and applied to the response language, and further evaluation can be performed to verify the applicability and usability of our approach.

Several possible extensions for the language concepts were discussed in the thesis. New types of triggers can extend the available preconditions for a response execution to composite changes and even more complex conditions on modifications of the model state. An important extension is the further integration of user interaction to increase the influence of the user to the way in which consistency is restored. In the evaluation, we revealed a limitation of our language regarding consistency overlaps that consist of a dynamic number of elements, which has to be addressed by a language extension.

Currently, several responses have to be written for the repair of the same type of consistency overlap, as it can be affected in different ways. The metaclasses of a consistency overlap type and the properties that are relevant for the constraints can be made explicit. From that specification, the changes that can affect such an overlap can be derived, just as in approaches for declarative consistency repair. Consequently, necessary responses for a certain overlap type could be automatically derived from such an explicit overlap specification to reduce the specification effort and error-proneness.

A second research area of interest is the preservation of consistency between more than two models. Rather than specifying consistency between all pairs of models, it is preferable to define consistency repair transitively. Such transitivity reduces the effort for specifying consistency but also induces new difficulties, such as the cyclic propagation of changes. Research results about defining transitive consistency repair mechanisms can be transferred to the response language and potentially also require an adaption of its constructs.

The applicability of our approach was initially evaluated in a single case study. In future work, the approach should be applied to further domains to verify the general applicability of the language. Furthermore, we did not analyze the usability of our approach. Although we assume that our concepts support the specification of consistency repair by providing reasonable abstractions from recurring and underlying technical aspects, this expectation has to be verified in controlled studies.

Bibliography

[1] Carsten Amelunxen and Andy Schürr. “Formalising model transformation rules for UML/MOF 2”. In: *IET Software* 2.3 (2008), pp. 204–222.

[2] Anthony Anjorin. “Synchronization of Models on Different Abstraction Levels using Triple Graph Grammars”. PhD thesis. Technische Universität Darmstadt, Oct. 2014.

[3] Anthony Anjorin, Gergely Varró, and Andy Schürr. “Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques”. In: *Proceedings of the First International Workshop on Bidirectional Transformations (Bx 2012)*. Vol. 49. Electronic Communications of the EASST. European Assoc. of Software Science and Technology, 2012.

[4] Thorsten Arendt et al. “Henshin: Advanced Concepts and Tools for In-place EMF Model Transformations”. In: *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I (MODELS 2010)*. Vol. 6394. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 121–135.

[5] Colin Atkinson and Thomas Kühne. “Model-Driven Development: A Metamodeling Foundation”. In: *IEEE Software* 20.5 (Sept. 2003), pp. 36–41.

[6] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. “Orthographic Software Modeling: A Practical Approach to View-Based Development”. In: *Proceedings of the 3rd and 4th International Conferences on Evaluation of Novel Approaches to Software Engineering (ENASE 2008/2009)*. Vol. 69. Communications in Computer and Information Science. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 206–219.

[7] Engineer Bainomugisha et al. “A Survey on Reactive Programming”. In: *ACM Computing Surveys* 45.4 (2013), 52:1–52:34.

[8] Steffen Becker, Heiko Koziolek, and Ralf Reussner. “The Palladio component model for model-driven performance prediction”. In: *JSS* 82 (2009), pp. 3–22.

[9] Lars Bendix and Pär Emanuelsson. “Requirements for Practical Model Merge - An Industrial Perspective”. In: *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)*. Vol. 5795. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 167–180.

[10] Gábor Bergmann et al. “Change-driven model transformations”. In: *Software & Systems Modeling* 11.3 (2012), pp. 431–461.

[11] Gábor Bergmann et al. "Incremental Evaluation of Model Queries over EMF Models". In: *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I (MODELS 2010)*. Ed. by DorinaC. Petriu, Nicolas Rouquette, and Øystein Haugen. Vol. 6394. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 76–90.

[12] Gábor Bergmann et al. "Viatra 3: A Reactive Model Transformation Platform". In: *Proceedings of the 8th International Conference on Theory and Practice of Model Transformations (ICMT 2015)*. Vol. 9152. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 101–110.

[13] Jonas Bonér et al. *The Reactive Manifesto (Version 2.0)*. 2014. URL: `http://www.reactivemanifesto.org/pdf/the-reactive-manifesto-2.0.pdf`.

[14] Borland Software Corporation. *Borland Together UML 2.1 Guide Version 2008 R3*. 2005. URL: `http://techpubs.borland.com/together/2008R3/EN/TogetherUML21.pdf`.

[15] Ruth Breu. "Ten Principles for Living Models - A Manifesto of Change-Driven Software Engineering". In: *Proceedings of the 4th International Conference on Complex, Intelligent and Software Intensive Systems (CISIS 2010)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–8.

[16] Erik Burger. "Flexible Views for View-based Model-driven Development". In: *Proceedings of the 18th International Doctoral Symposium on Components and Architecture (WCOP 2013)*. New York, NY, USA: ACM, 2013, pp. 25–30.

[17] Erik Burger. "Flexible Views for View-based Model-driven Development". PhD thesis. Karlsruhe, Germany: Karlsruhe Insititute of Technology (KIT), 2014.

[18] Erik Burger et al. "View-Based Model-Driven Software Development with ModelJoin". In: *Software & Systems Modeling* 15.2 (2016), pp. 473–496.

[19] Peter Pin-Shan Chen. "The Entity-Relationship Model–Toward a Unified View of Data". In: *ACM Transactions on Database Systems* 1 (1976), pp. 9–36.

[20] Antonio Cicchetti, Davide Di Ruscio, and Romina Eramo. "Towards Propagation of Changes by Model Approximations". In: *Proceedings of the 10th IEEE on International Enterprise Distributed Object Computing Conference Workshops (EDOCW 2006)*. Washington, DC, USA: IEEE Computer Society, 2006, p. 24.

[21] Krzysztof Czarnecki and Simon Helsen. "Feature-based Survey of Model Transformation Approaches". In: *IBM Systems Journal* 45.3 (2006), pp. 621–645.

[22] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. "Specifying Overlaps of Heterogeneous Models for Global Consistency Checking". In: *Proceedings of the First International Workshop on Model-Driven Interoperability (MDI 2010)*. New York, NY, USA: ACM, 2010, pp. 42–51.

[23] S Efftinge and Clemens Kadura. *OpenArchitectureWare 4.1 Xpand Language Reference*. 2006.

[24] Sven Efftinge and Markus Völter. "oAW xText: A framework for textual DSLs". In: *Proceedings of Workshop on Modeling Symposium at Eclipse Summit* (2006).

[25] Sven Efftinge et al. “Xbase: Implementing Domain-specific Languages for Java”. In: *ACM SIGPLAN Notices* 48.3 (2013), pp. 112–121.

[26] Conal Elliott. *A Brief Introduction to ActiveVRML*. Tech. rep. MSR-TR-96-05. Microsoft Research, 1996.

[27] Conal Elliott. “Push-Pull Functional Reactive Programming”. In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell 2009)*. New York, NY, USA: ACM, 2009, pp. 25–36.

[28] Conal Elliott and Paul Hudak. “Functional Reactive Animation”. In: *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP 1997)*. New York, NY, USA: ACM, 1997, pp. 263–273.

[29] Romina Eramo et al. “A model-driven approach to automate the propagation of changes among Architecture Description Languages”. In: *Software & Systems Modeling* 11.1 (2012), pp. 29–53.

[30] Romina Eramo et al. “Change Management in Multi-Viewpoint System Using ASP”. In: *Proceedings of the 12th Enterprise Distributed Object Computing Conference Workshops (EDOCW 2008)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 433–440.

[31] ETAS. *ETAS - ASCET Software-Produkte*. URL: `http://www.etas.com/de/products/ascet%7B%5C_%7Dsoftware%7B%5C_%7Dproducts.php` (visited on 12/08/2015).

[32] Sebastian Fiss, Max E Kramer, and Michael Langhammer. “Automatically Binding Variables of Invariants to Violating Elements in an OCL-Aligned XBase-Language”. In: *Proceedings of Modellierung 2016*. Vol. P-254. Lecture Notes in Informatics (LNI). Bonn, Germany: Gesellschaft für Informatik e.V. (GI), 2016, pp. 189–204.

[33] J Nathan Foster et al. “Combinators for Bi-directional Tree Transformations: A Linguistic Approach to the View Update Problem”. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*. Vol. 40. 1. New York, NY, USA: ACM, 2005, pp. 233–246.

[34] J Nathan Foster et al. “Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem”. In: *ACM Transactions on Programming Languages and Systems* 29.3 (2007).

[35] Martin Fowler. *Domain Specific Languages*. 1st. Addison-Wesley Professional, 2010.

[36] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[37] Sinem Getir et al. “CoWolf – A Generic Framework for Multi-view Co-evolution and Evaluation of Models”. In: *Proceedings of the 8th International Conference on Theory and Practice of Model Transformations (ICMT 2015)*. Vol. 9152. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015. Chap. CoWolf - A, pp. 34–40.

[38] Holger Giese and Robert Wagner. “From model transformation to incremental bidirectional model synchronization”. In: *Software & Systems Modeling* 8.1 (2009), pp. 21–43.

[39] James Gosling et al. *The Java Language Specification, Java SE 8 Edition*. 2015. URL: `https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`.

[40] Esther Guerra and Juan de Lara. "Event-Driven Grammars: Towards the Integration of Meta-modelling and Graph Transformation". In: *Proceedings of the 2nd International Conference on Graph Transformations (ICGT 2004)*. Vol. 3256. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2004, pp. 54–69.

[41] Jens Happe et al. "Getting the Data". In: *Modeling and Simulating Software Architectures - The Palladio Approach*. Ed. by Ralf H Reussner et al. Cambridge, MA: MIT Press, 2016.

[42] David Harel and Bernhard Rumpe. "Meaningful Modeling: What's the Semantics of "Semantics"?" In: *Computer* 37.10 (2004), pp. 64–72.

[43] Florian Heidenreich et al. "Closing the Gap between Modelling and Java". In: *Proceedings of the 2nd International Conference on Software Language Engineering (SLE 2009)*. Vol. 5969. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 374–383.

[44] Florian Heidenreich et al. *Jamopp: The Java Model Parser and Printer*. Tech. rep. 2009.

[45] Frank Hermann et al. "Concurrent Model Synchronization with Conflict Resolution Based on Triple Graph Grammars". In: *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012)*. Vol. 7212. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 178–193.

[46] Thomas Hettel, Michael Lawley, and Kerry Raymond. "Model Synchronisation: Definitions for Round-Trip Engineering". In: *Proceedings of the First International Conference on Theory and Practice of Model Transformations (ICMT 2008)*. Vol. 5063. Lecture Notes in Computer Science. Berlin, Heidelberg: Sprinter-Verlag, 2008, pp. 31–45.

[47] Paul Hudak et al. "Arrows, Robots, and Functional Reactive Programming". In: *Summer School on Advanced Functional Programming 2002, Oxford University*. Vol. 2638. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 159–187.

[48] International Organization for Standardization. *ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF*. 1996.

[49] ISO/IEC. *ISO/IEC 14764 IEEE Std 14764-2006 Software Engineering - Software Life Cycle Processes - Maintenance*. 2006.

[50] ISO/IEC/IEEE. *ISO/IEC/IEEE 42010:2011 - Systems and software engineering - Architecture description*. 2011.

[51] Sven Johann and Alexander Egyed. "Instant and Incremental Transformation of Models". In: *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 362–365.

[52] Frédéric Jouault et al. "ATL: a QVT-like Transformation Language". In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications.* OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 719–720.

[53] Holger Krahn. "Monticore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering". PhD thesis. RWTH Aachen University, 2010.

[54] Max E. Kramer. "A Generative Approach to Change-Driven Consistency in Multi-View Modeling". In: *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA 2015).* New York, NY, USA: ACM, 2015, pp. 129–134.

[55] Max E. Kramer. "Synchronizing Heterogeneous Models in a View-Centric Engineering Approach". In: *Software Engineering 2014 – Fachtagung des GI-Fachbereichs Softwaretechnik.* Ed. by Wilhelm Hasselbring and Nils Christian Ehmke. Vol. 227. Lecture Notes in Informatics (LNI). Gesellschaft für Informatik e.V. (GI), 2014, pp. 233–236.

[56] Max E. Kramer, Erik Burger, and Michael Langhammer. "View-centric Engineering with Synchronized Heterogeneous Models". In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO 2013).* New York, NY, USA: ACM, 2013, 5:1–5:6.

[57] Max E. Kramer and Kirill Rakhman. "Automated Inversion of Attribute Mappings in Bidirectional Model Transformations". In: *Proceedings of the 5th International Workshop on Bidirectional Transformations (Bx 2016).* Ed. by Anthony Anjorin and Jeremy Gibbons. Vol. 1571. CEUR Workshop Proceedings. CEUR-WS.org, 2016, pp. 61–76.

[58] Max E. Kramer et al. "Change-Driven Consistency for Component Code, Architectural Models, and Contracts". In: *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2015).* New York, NY, USA: ACM, 2015, pp. 21–26.

[59] Max E. Kramer et al. *Realizing Change-Driven Consistency for Component Code, Architectural Models, and Contracts in Vitruvius.* Tech. rep. Karlsruhe: Karlsruhe Institute of Technology, Department of Informatics, 2015.

[60] Michael Langhammer and Klaus Krogmann. "A Co-evolution Approach for Source Code and Component-based Architecture Models". In: *17. Workshop Software-Reengineering und -Evolution.* Vol. 4. 2015.

[61] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* 3rd. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[62] Erhan Leblebici et al. "A Comparison of Incremental Triple Graph Grammar Tools". In: *Proceedings of the 13th InternationalWorkshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2014).* Vol. 67. Electronic Communications of the EASST. EASST, 2014.

[63] Peter F. Linington. “Black Cats and Coloured Birds - What Do Viewpoint Correspondences Do?” In: *Proceedings of the 2007 Eleventh International IEEE EDOC Conference Workshop (EDOCW 2007)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 239–246.

[64] Nuno Macedo and Alcino Cunha. “Implementing QVT-R Bidirectional Model Transformations Using Alloy”. In: *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013)*. Vol. 7793. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 297–311.

[65] Nuno Macedo, Tiago Guimarães, and Alcino Cunha. “Model Repair and Transformation with Echo”. In: *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)*. IEEE Computer Society, 2013, pp. 694–697.

[66] Ingo Maier and Martin Odersky. *Deprecating the Observer Pattern with Scala.React*. Tech. rep. 2012.

[67] Ivano Malavolta et al. “Providing Architectural Languages and Tools Interoperability Through Model Transformation Technologies”. In: *IEEE Transactions on Software Engineering* 36.1 (2010), pp. 119–140.

[68] Tom McArthur and Roshan McArthur. *Concise Oxford Companion to the English Language*. Oxford Paperback Reference. Oxford University Press, 1998.

[69] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. “Consistency Management with Repair Actions”. In: *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 455–464.

[70] Ulrich Nickel, Jörg Niere, and Albert Zündorf. “The FUJABA Environment”. In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*. New York, NY, USA: ACM, 2000, pp. 742–745.

[71] Object Management Group (OMG). *Business Process Model And Notation (BPMN) Version 2.0*. 2011. URL: `http://www.omg.org/spec/BPMN/2.0/`.

[72] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Query/View/-Transformation (QVT), Version 1.2*. 2015. URL: `http://www.omg.org/spec/QVT/1.2/`.

[73] Object Management Group (OMG). *Meta Object Facility (MOF), Version 2.5*. 2015. URL: `http://www.omg.org/spec/MOF/`.

[74] Object Management Group (OMG). *Object Constraint Language (OCL), Version 2.4*. 2014. URL: `http://www.omg.org/spec/OCL/2.4/`.

[75] Object Management Group (OMG). *Unifed Modeling Language (UML), Version 2.5*. 2015. URL: `http://www.omg.org/spec/UML/2.5/`.

[76] Object Management Group (OMG). *XML Metadata Interchange (XMI) Specification, Version 2.5.1*. 2015. URL: `http://www.omg.org/spec/XMI/2.5.1/`.

[77] David Parsons. “Event-Driven Programming”. In: *Foundational Java*. London, UK: Springer-Verlag, 2012, pp. 417–464.

[78] Peter Pepper. “A Study on Transformational Semantics”. In: *International Summer School on Program Construction.* Ed. by FriedrichL. Bauer et al. Vol. 69. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1979, pp. 322–405.

[79] István Ráth, Gergely Varró, and Dániel Varró. “Change-Driven Model Transformations”. In: *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009).* Ed. by Andy Schürr and Bran Selic. Vol. 5795. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 342–356.

[80] Alexander Reder. “Automated Consistency Management Framework for the Model Based Software Development”. PhD thesis. Johannes Kepler University (JKU), Linz, Austria, 2013.

[81] Alexander Reder. “Inconsistency Management Framework for Model-based Development”. In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011).* New York, NY, USA: ACM, 2011, pp. 1098–1101.

[82] Alexander Reder and Alexander Egyed. “Incremental Consistency Checking for Complex Design Rules and Larger Model Changes”. In: *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS 2012).* Vol. 7590. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 202–218.

[83] Ralf Reussner et al. *The Palladio Component Model.* Tech. rep. 14. Karlsruhe Institute of Technologie, 2011.

[84] Alexandru Salcianu and Martin Rinard. “Purity and Side Effect Analysis for Java Programs.” In: *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2005).* Vol. 3385. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 199–215.

[85] Oliver Scheid. *Autosar Compendium - Part 1: Application & Rte.* AUTOSAR - Compendium Series. CreateSpace Independent Publishing Platform, 2015.

[86] S. Schmidhuber et al. “Overview of the itea2-project amalthea”. In: *Proceedings of the 2nd Applied Research Conference.* SHAKER Verlag, 2012, pp. 60–62.

[87] Geoffrey Sparks. *Enterprise Architect User Guide.* Sparx Systems, 2014. URL: `http://www.sparxsystems.com.au/bin/EAUserGuide.pdf`.

[88] Herbert Stachowiak. *Allgemeine Modelltheorie.* Springer-Verlag, 1973.

[89] Thomas Stahl et al. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management.* 2. Heidelberg: dpunkt-Verlag, 2007.

[90] David Steinberg et al. *EMF - Eclipse Modeling Framework.* Ed. by David Steinberg. 2nd. Addison-Wesley Professional, 2009.

[91] Ryan K. Stephens and Ronald R. Plew. *Database Design.* Pearson Education, 2000.

[92] Perdita Stevens. “A Landscape of Bidirectional Model Transformations”. In: *Generative and Transformational Techniques in Software Engineering II.* Berlin, Heidelberg: Springer-Verlag, 2008, pp. 408–424.

[93] Claudia Szabo and Yufei Chen. “A Model-Driven Approach for Ensuring Change Traceability and Multi-Model Consistency”. In: *Proceedings of the 22nd Australasian Software Engineering Conference (ASWEC 2013)*. IEEE, 2013, pp. 127–136.

[94] The Eclipse Foundation. *Eclipse desktop & web IDEs*. 2016. URL: `https://eclipse.org/ide/` (visited on 05/09/2016).

[95] The Eclipse Foundation. *Eclipse - The Eclipse Foundation open source community website*. 2016. URL: `http://www.eclipse.org/` (visited on 05/09/2016).

[96] Laurence Tratt. “A change propagating model transformation language”. In: *Journal of Object Technology* 7.3 (2008), pp. 107–126.

[97] Manuel Wimmer, Nathalie Moreno, and Antonio Vallecillo. “Viewpoint Co-evolution through Coarse-Grained Changes and Coupled Transformations”. In: *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns (TOOLS 2012)*. Vol. 7304. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 336–352.

[98] Yingfei Xiong et al. “Towards Automatic Model Synchronization from Model Transformations”. In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. New York, NY, USA: ACM, 2007, pp. 164–173.