A Declarative Language for Preserving Consistency of Multiple Models

Bachelor's Thesis of

Joshua Gleitze

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:	M. Sc. Heiko Klare
Second advisor:	Dr.-Ing. Erik Burger

20th June 2017 – 19th October 2017

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis entirely by myself. I have not used sources or means without declaring them in the text.

Karlsruhe, 19th October 2017

. . . .
(Joshua Gleitze)

Abstract

Using multiple models to describe a software system, often poses the challenge to keep them consistent automatically. While there is much research on preserving consistency of two models, fewer works address the specifics of keeping more than two models consistent. This thesis proposes a new programming language, allowing to create transformations for consistency preservation of more than two models. The language uses an intermediate metamodel, such that any transformation is first executed from an existing model into an intermediate model, and then into other models.

We start by looking at different possibilities of how consistency of multiple models can be preserved using only binary transformations. Subsequently, we show advantages of introducing an intermediate metamodel to the consistency preservation process. To support consistency preservation with intermediate metamodels, we thereupon introduce the Commonalities Language. It allows developers to declare metaclasses of the intermediate metamodel together with their attributes and references. The mappings from the intermediate model to other models and back are given directly at the mapped intermediate metaclasses, attributes, and references. To avoid duplication of logic, bidirectional expressions can be used for the mappings. The language is declarative to make understanding the transformations easy.

We have developed a prototypical implementation of it for the Vitruvius framework, which can be used in Eclipse for EMF models. The implementation serves as a proof of concept but is not yet sufficiently mature to be used in practice. The idea of using intermediate metamodels to achieve scalable and modular consistency preservation for multiple models was already applied successfully to realistic scenarios in other works. To the best of our knowledge, no existing approach allows defining an intermediate metamodel together with the transformations for it in the same language.

Zusammenfassung

Der Einsatz mehrerer Modelle zur Beschreibung eines Softwaresystems birgt oftmals die Herausforderung, diese konsistent zu halten. Während es viel Forschung zur Konsistenzhaltung zweier Modelle gibt, untersuchen nur wenige Arbeiten die Spezifika der Konsistenzhaltung mehrerer Modelle. In dieser Bachelorarbeit wird eine neue Programmiersprache vorgestellt, die es erlaubt, Transformationen zu erstellen, die mehr als zwei Modelle konsistent halten. Die Sprache verwendet ein Zwischen-Metamodell, sodass alle Transformationen zuerst von einem existierenden Modell in das Zwischenmodell und dann erst in die anderen Modelle ausgeführt werden.

Zunächst betrachten wir verschiedene Möglichkeiten, wie Modelle mit ausschließlich binären Transformationen konsistent gehalten werden können. Im Weiteren demonstrieren wir Vorteile davon, ein Zwischen-Metamodell in den Konsistenzhaltungsprozess einzuführen. Im nächsten Schritt präsentieren wir die Gemeinsamkeiten-Sprache als eine Möglichkeit der Konsistenzhaltung mittels Zwischen-Metamodellen. Sie ermöglicht Entwicklern, Metaklassen des Zwischen-Metamodells gemeinsam mit deren Attributen und Referenzen zu deklarieren. Die Abbildungen vom Zwischenmodell in die Modelle, die konsistent gehalten werden sollen, und zurück, können direkt in den Zwischen-Metaklassen, -Attributen und -Referenzen festgelegt werden. Um Logik nicht zu duplizieren, können bidirektionale Ausdrücke für die Abbildungen verwendet werden. Die Sprache ist deklarativ und soll auf diese Weise eine hohe Nachvollziehbarkeit der Transformationen ermöglichen.

Wir haben ein prototypische Implementierung der Sprache für Vitruvius erstellt, die in Eclipse für EMF-Modelle verwendet werden kann. Die Implementierung kann als Machbarkeitsnachweis gesehen werden, eignet sich allerdings noch nicht für den Praxiseinsatz. Die Idee, Zwischen-Metamodelle für die skalierbare und modulare Konsistenzhaltung mehrerer Modelle einzusetzen, wurde in anderen Arbeiten in realistische Szenarien bereits erfolgreich umgesetzt. Soweit uns bekannt ist, existiert noch kein Ansatz, der es erlaubt, ein Zwischen-Metamodell und die Transformationen für dieses in der selben Sprache zu definieren.

Contents

1 Introduction **1**

2 Foundations **3**
2.1 Model-Driven Software Development 3
2.2 Models and Metamodels 4
2.3 Domain-Specific Languages 5
2.4 Model Consistency 6
2.5 Metamodelling Technology 7
2.6 View-Based Modelling 10
2.7 Consistency Preservation in Vitruvius 11

3 Running Example **13**

4 Transformation Layout **15**
4.1 Multiary Consistency Preservation Using Only Binary Transformations . 16
4.2 Multiary Consistency Preservation Using an Intermediate Model 22

5 The Commonalities Language **29**
5.1 Design Goals 29
5.2 Overview 31
5.3 Common Constructs 32
5.4 Expressions 32
5.5 Commonalities 37
5.6 Participations 38
5.7 Attributes 43
5.8 References 45
5.9 Design Decisions 46
5.10 Feature Overview 48

6 Implementation **51**
6.1 Platform Requirements 51
6.2 Platform-Specific Language Features 52
6.3 Expressions 54
6.4 Compilation 55

7 Evaluation **57**
7.1 Validity 57
7.2 Implementation Functionality 57
7.3 Threats to Validity 58

8 Related Work **59**
8.1 Transformation Languages 59
8.2 Orthographic Software Modelling 61
8.3 Preserving Consistency through a Central Model 61
8.4 Theoretical Work 62

9 Future Work **63**
9.1 Implementation 63
9.2 Evaluation in Practical Use Cases 63
9.3 Language Evolution 64
9.4 Theoretical Evaluation 64

10 Conclusion **65**

Glossary **67**

Bibliography **69**

Demonstrative Material

3.1 The metaclasses of the running example 14

4.1 A fully connected graph of bidirectional transformations 17
4.2 Bidirectional transformations forming a tree 19
4.3 Unidirectional transformations forming a circle 21
4.4 Preserving consistency using an intermediate metamodel 23
4.5 Example of an intermediate metamodel 24
4.6 Example a transformation execution with an intermediate model 25

5.1 A simple Commonality File with annotated abstract syntax 31
5.2 Syntax to reference metaclasses or properties 33
5.3 The different cases of the reference syntax 33
5.4 Abstract Syntax of a Commonality File, a Commonality and a Concept . . . 37
5.5 Three Participation Classes for a Java package 40
5.6 How to reference Participation Classes 40
5.7 Example for an enforceable condition on a Participation 42
5.8 Participation Classes related with the in-operator 42
5.9 The different types of attribute mapping specifications 43
5.10 Checked attribute mappings remove duplicated information 45
5.11 Reference for components in a repository 46
5.12 Overview of the Commonalities Language's features 49

6.1 The virtual resource metaclass added to every Vitruvius Domain 53

1 Introduction

When developing software systems, *modelling* the developed software helps to tackle the process' inherent complexity. By creating models of the targeted system, it is possible to analyse it or execute simulations for it, even before the system has been fully realised [Sel03]. Because developing software covers many activities and stakeholders, there are many different types of models, often tailored to fit the needs of a specific domain. This allows experts of a domain to focus on their concerns and to use models that have the abstraction and concepts required for their activities, without needing a "one size fits all"-solution [VS06, p. 15]. In effect, the same software system might be developed using multiple, different models.

Meanwhile, using multiple models means having multiple representations of the software. Because even though the models are specific to a domain, they still describe the same system. Therefore, some information will be contained in multiple models. We say that the models overlap with each other. If different domain experts edit them simultaneously, the models might contradict each other afterwards. This raises the need to actively keep them *consistent*. Consistency preservation—the process of keeping models consistent—identifies spots where models overlap with each other and transforms changes to such spots into other models [Kra17, p. 38 f.]. It ensures that changes to one model are represented in the other models and that the models hence do not contradict each other.

There are many works on model transformations that could be used for consistency preservation [MJC17]. However, the approaches are often only concerned with preserving consistency between two models [MJC17; Ste17]. As others have already shown [NER01; Ste17], and as we will also explore in chapter 4, preserving consistency of more than two models using only binary model transformations has shortcomings. It may make it difficult to add new models to the system or require an inadequate amount of development effort. Having explicit mechanisms to support more than two models might lead to a more modular and scalable approach. This thesis contributes to that goal by covering the following questions:

Q1 Given a set of multiple models, which artefacts (like transformations) should be created to preserve the models' consistency, such that the approach

- allows adding models to or removing models from the set,
- and scales well when adding models to the set?

Q2 How can a domain-specific language for specifying model transformations be designed, such that it

- allows realising multi-model consistency preservation with the approach determined for Q1,
- and supports developers in understanding and verifying transformation rules?

We will propose a new programming language for consistency preservation of more than two models. It introduces an intermediate model to the consistency preservation process, which models the relevant semantic overlap of models. The language allows developers to declare such intermediate models together with transformations from existing models to the intermediate model and vice versa.

The thesis will be structured as follows: We begin by giving background and introducing relevant terms in chapter 2. We will introduce the paradigm of Model-Driven Software Development, describe basic concepts of models and their consistency and introduce relevant technology. Chapter 3 will introduce a running example, to which we will refer throughout the thesis.

In chapter 4, we compare different possibilities of how consistency of multiple models could be preserved using only binary model transformation. We will then describe our approach, using an intermediate metamodel, and compare it to the presented possibilities. Chapter 5 will introduce the Commonalities Language, the programming language we have developed to support the approach. We will explain the goals that we had when designing the language and explain its features in detail. We have also developed a prototypical implementation of the language; chapter 6 will present features of the Commonalities Language that have been designed to solve problems that are specific to the framework we implemented the language in and also show notable properties of our prototype.

We evaluate the results of this thesis in chapter 7. Chapter 8 will give an overview of other publications related to multi-model consistency preservation. Where it is appropriate, we will compare our results with these works. Finally, chapter 9 gives an outlook on where and how our results could be improved before chapter 10 concludes the thesis.

2 Foundations

2.1 Model-Driven Software Development

Software engineering suffers from a gap between the concepts of the problem and the implementation domain. Even highly evolved programming languages like Java offer comparably little abstraction and require developers to think in different terms than those of the problem they are trying to solve [FR07]. *Model-Driven Software Development* is a paradigm trying to make this gap smaller by allowing developers to describe software in the problem domain. It can be seen as the next step in an ever continuing attempt to raise the abstraction level in programming [AK03].

In Model-Driven Software Development, models are at the centre of the development process and specify the software system. They can be general-purpose models for aspects like software architecture, but also be tailored for the specific domain the software is being built for. Either way, they do not merely document the software but are made a part of it, for example by automatically transforming them into executable code [FR07]. Thereby, models get equal to programming languages. In fact, a programming language can be regarded as just another model that is used to describe and form the software. The software development process becomes similar to the practice in other engineering disciplines: Analogous to how a mechanical engineer can feed a model from its Computer Aided Design software directly into a computer-controlled mill to create a physical workpiece, software engineers are enabled to create executable software by modelling it. The comparison goes further: using their model of the system software engineers can execute analyses, predict properties of the system and evaluate its design before it was built—just like mechanical engineers can [VS06, p. 6].

Using Model-Driven Software Development promises several advantages. First, it can increase development productivity. By working on a higher level of abstraction, developers can ignore details and focus on the software's "core". Studies suggest that this productivity boost occurs in practice, but only if adequate tooling and code generators are available [Kap+09; MCM12; WHR14]. Second, Model-Driven Software Development is a method to ensure that requirements and specifications from different stakeholders are met. As programming still forces developers to solve problems on a detailed and technical level, they can lose sight of the "bigger picture". Without an integrated view of the software, they are then forced to implement suboptimal solutions: The system's architecture might be violated, or a feature may fail to meet its requirements precisely [Sch06]. If the system's architecture, its requirements or other domain-specific properties are explicitly modelled using Model-Driven Software Development, they are thereby enforced. Moreover, because they can be formulated in a domain-specific way, it is often easier to reason about their correctness.

2.2 Models and Metamodels

What *is* a model? Models are used throughout the sciences to make complex situations manageable. Stachowiak [Sta73] thus created a "general model theory", describing the fundamental characteristics of any model: representation, reduction and pragmatics. According to it, a model first and foremost represents an original. There is no restriction of what that original can be. In particular, it can be another model. Secondly, a model captures attributes of its original, usually selecting only a subset of them (reduction). Which of the original's attributes should be represented is dictated by the model's pragmatics. That means that the reduction should be carried out in the way that serves the model's purpose best. Particularly the target audience and the time and context of use should be taken into account when selecting attributes [Sta73, pp. 131 f.].

Being an integral part of it, models used in Model-Driven Software Development have further typical characteristics; the most obvious being that they are represented in a computer—they could not be integrated into a software development process otherwise. Selic [Sel03] presents further traits models should possess to be "useful and effective" in Model-Driven Software Development. These criteria are concerned with the pragmatics applied to the model's reduction, to put it in Stachowiak's [Sta73] terms. Similar to Stachowiak, Selic discusses that leaving out irrelevant information from the model is often the only option to make it possible to cope with "ever-more sophisticated functionality [of] software systems". Models should only contain the essence of what is required from their viewpoint and abstract from everything else. However, abstraction alone is not enough; the model should secondly be understandable. That is the conveyed information should be intuitively perceivable by the target audience. Source code, Selic argues, is not very expressive because it requires the reader to invest a significant amount of intellectual effort to translate the syntactic constructs into meaning. Models offer a possibility of a more direct access to the modelled concepts, which should be exploited. Somewhat antipodal to this are the next characteristics, accuracy and predictiveness. A model's contained information should be sufficiently accurate to make it useful. In particular, it should be possible to perform predictions about interesting but unknown properties of the original based on the model. Finally, models should be inexpensive. They should be significantly cheaper to create and analyse than constructing the modelled system [Sel03].

Whenever operating on a model—for instance when generating executable code from them, transforming them into other models or running analyses on them—tools need to know the permissible elements and values they may encounter in it. The set of rules defining what can occur in a model can be a model of its own, called the model's *metamodel*. Consequently, a metamodel also has a metamodel, which could be called the initial model's meta-metamodel. This chain needs to stop somewhere, of course: not every metamodel can have a new metamodel. It is understood that at some point, metamodels reach a level of "metaness" that makes them self-explanatory.

In this thesis, we will only look at metamodels that are specified in an object-oriented way. This means that it declares a set of *metaclasses* that have arbitrarily typed attributes and cardinalities and can also reference other metaclasses. Models conforming to such a metamodel consist of objects that are instances of one of the metaclasses and have the attributes declared in their metaclass. To say that a model object o is an instance of a metaclass M, we will briefly write $o : M$.

2.3 Domain-Specific Languages

Programming languages can be roughly categorised by two extremes: general-purpose languages and domain-specific ones. The former describes languages that can be used for almost any problem. *Domain-specific languages*, on the other hand, target only use cases that occur in a specific domain. They deliberately sacrifice claims of generality to fit the requirements of their usage context better.

The C programming language, for instance, is a general-purpose language. It was designed with no restriction on what it will be used for in mind [Ker88, p. xi]. It can be, and has been, used to create all kinds of computer programs. Latex, in opposition to C, is a domain-specific language that was created with one particular goal in mind: typesetting documents. To do so, Latex offers several commands and syntactic features tailored for typical tasks when writing documents. While documents could be created with C, too, it would be significantly more difficult and require a multitude of code lines compared to Latex. For example, Latex allows writing the document's text directly into the file, without any further markup. Text and commands can be mixed freely. Arguments to commands can, but do not always have to, be enclosed by special characters. Strings in C, on the other hand, must always be contained in double quotes. Arguments to functions must be given in double quotes, and the return value of functions must be combined with strings text by the + operator. To put it in a nutshell, Latex offers functionality that is useful for creating documents and makes this task easier. C may often require more effort to solve a task than a domain-specific language, but can in return be used in any context.

Domain-specific languages have successfully been applied to various areas of information technology. HTML to create web pages, CSS to style them, Make to build software, Perl-style regular expressions for pattern matching or SQL to query databases are just a few of the numerous popular examples [MHS05; FP10, p. xxi]. They are especially important in the context of Model-Driven Software Development: A model can be seen as being written in a domain-specific language. That language's abstract syntax is it the model's metamodel, the language's concrete syntax is the model's—graphical or textual—representation. From that point of view, abstract syntax trees can be seen as a meta-metamodel. These observations are true for any language, not just for domain-specific ones. However, as Model-Driven Software Development focuses on using domain-specific models, domain-specific languages are of particular interest. Because both concepts are so closely related, metamodels are often referred to as (domain-specific) "modelling languages".

A fundamental distinction can be made regarding the implementation of domain-specific languages: there are *internal* and *external* domain-specific languages. The former type does not specify an entirely new syntax, but re-uses an existing language, the host language. Code written in an internal domain-specific language is still valid code in the host language. It is merely a specific way to use the host language. External domain-specific languages, on the other hand, use their own parser, type system, etcetera [FP10, p. 28]. There are good arguments for using either type of domain-specific language. For example, internal domain-specific are often cheaper to realise, and existing tools can be used for them. External domain-specific languages, on the other hand, can choose syntactic constructs that make the most sense and are not limited by the syntax of the host language [FP10, pp. 105 ff.]. The best decision between the two types will often depend on the use case [FP10, p. 29].

2.4 Model Consistency

When multiple models describe the same original, it is possible that the same piece of information is contained in multiple models. In conformance with Burger et al.'s [Bur14] nomenclature, this situation will be called *"semantic overlap"* in this thesis. If a model shares semantic overlap with other models, those models will be called the model's *overlapping* models. Semantic overlap does not imply that a piece of information is contained as a syntactic copy in the concerned models. It can be represented in very different ways, even just implicitly.

Problems arise when a model which shares semantic overlap with other models is edited. The model might then become out of synchronisation with the other models; in other words, the models are expected to share the same piece of information, but, in fact, contradict each other. If this happens, the models are called *inconsistent*. If no model contradicts another—either because no models share semantic overlap or because all models having semantic overlap are in unison—a set of models is called *consistent*. Consistency is not a feature that could be defined in the metamodel, as it spans over multiple models (that will often be from different metamodels). It is an external property. In addition, there is no inherent or unique definition of when a set of models is consistent, although there may be an intuitive one. Consistency is relative to a *consistency specification* [Kra17, p. 38].

2.4.1 Handling Inconsistency

Can inconsistency be avoided? As there can be no inconsistency if there is no semantic overlap, a possible solution might be to remove any semantic overlap from all models. Any piece of information would then be contained in at most one model, and other models could just point at that instance, which would remove any semantic overlap. Atkinson et al. realised this in their approach by only using one central model, see section 2.6.1. It might however not always be practical (see section 2.6.2, for instance).

2.4.2 Consistency Preservation

If neither eliminating any semantic overlap nor sufficiently restricting model editing is an option, consistency has to be actively enforced [Kra17, pp. 5 f.]. This means propagating changes from an edited model to its overlapping models. If done manually, the task is time-consuming, prone to error and requires knowledge about all overlapping models. Experts may not be capable of consistently making changes to models because it would require modifying models of a domain which they do not have sufficient knowledge of. All of this motivates the need for automatic enforcement of consistency [Kra17, p. 6].

This process is not bound only to act when consistency has been violated but may also modify already consistent models to avoid inconsistency in the future. Because of that Kramer [Kra17] calls it *consistency preservation*. The term does not militate against that inconsistency might be tolerated in some cases.

There are different ways how consistency can be preserved in practice. It is for example possible to exhaustively search the space of all possible models for a combination that fulfils all consistency rules [Kra17, p. 39]. A more direct approach, which was also chosen for this thesis, is to specify appropriate *model transformations* that are executed after a model was edited.

How can model transformations look like? Fundamentally, a model transformation is an algorithm taking models as its input and returning a model as its output. It is specified at metamodel level and executed on model instances. Often, the models conform to different metamodels, and the output model will be a modified version of a pre-existing one. Because of that, it is common for transformations to also have the pre-existing model as part of their input. It allows them to react to what is already in the model, and to not change more than necessary and avoid unpleasant surprises [Che+15; Che+17].

Most commonly, a transformation will act on two models and will be called *binary*. This is, however, not necessary; a transformation could also transform multiple models, in which case it is called *n-ary* if it acts on *n* models, or *multiary* if the value of *n* is irrelevant [Ste17]. In the binary case, a fundamental distinction can be made between *unidirectional* and *bidirectional* transformations. Unidirectional transformations only support transforming the source models into the target model, but not the other way around. Compilation of high-level languages is a typical example for them [Ste07]. In Model-Driven Software Development it is, however, more likely that both sides of a transformation can be edited. This calls for bidirectional transformations; i.e. transformations in which both participating models can be the source of transformation. Bidirectionality does not imply bijectivity. Requiring bidirectional transformations to be bijective is too restrictive for many applications [Ste07].

Realising consistency preservation through transformations also establishes a consistency specification; in the sense that every state of the models reachable through the transformations is consistent and any other state is not. When working with model transformation languages, we assume that this will be the usual case: Instead of creating any form of explicit consistency specification for the models, developers create transformations that express their implicit understanding of how consistent models should look like.

2.5 Metamodelling Technology

2.5.1 Eclipse Modelling Framework

The Eclipse Modelling Framework (EMF) offers an infrastructure for working with metamodels. It provides means to create tooling for Model-Driven Software Development and lowers the barrier to develop domain-specific model assets. Metamodels can be created in a graphical editor. Based on such a metamodel definition, EMF creates Java classes representing the metamodel. These Java classes can be used like regular classes in Java code, but also carry additional metadata from the metamodel. They also feature a notification mechanism, making it possible for Java code to be notified of any change to the model objects. EMF can furthermore generate a graphical editor allowing to edit model instances of the metamodel. On the whole, EMF lays the foundations for adopting Model-Driven Software Development: The generated graphical editors offer an easy way to create models and the generated Java code can be used to write model transformations and code generators for them or use them directly in an application.

Metamodels for EMF are defined using *Ecore*, EMF's meta-metamodel. It provides metaclasses to define metamodels using concepts known from object orientation, like packages, which contain other packages and classes, which again can have attributes, references and operations. Ecore is aware of the Java programming language: It defines the meta-metaclass "EDataType", which describes an existing Java class in a metamodel. Such data types can be used for attribute types or the type of the parameters or the return type of an operation. This mechanism allows integrating existing Java classes into metamodels, which eases the usage of Ecore models in Java applications.

Another relevant feature of EMF is built-in support for model serialisation. Any Ecore model can be serialised without any further code needed. The default serialisation uses XMI as exchange format. However, EMF is not restricted to XMI but can be extended to use any serialisation format. One interesting application is to define a programming language as a serialisation format of an EMF metamodel. Instances of the metamodel can then be serialised to code in the language, and code written in the language can be deserialised to an instance of the metamodel. This approach realises the close relation of languages and metamodels laid out in section 2.3. It is used by Xtext and Jamopp, see sections 2.5.5 and 2.5.4.

Model objects must eventually be persisted. In EMF, every model object is therefore contained in an EMF Resource, which is an abstraction from a file. An EMF Resource contains a tree of objects. Only the root node has to be placed in a Resource explicitly, all other model elements in the tree will be included when serialising an EMF Resource. When creating a metamodel in Ecore, references to other metaclasses can be marked as being a *containment reference*. These references are considered edges of the model tree in an EMF Resource. Every model object may participate in at most one containment reference. In effect, every model object is either in one Resource—because it is the root of a containment tree or in exactly one containment reference—or in no Resource.

2.5.2 Unified Modelling Language

The Unified Modelling Language (UML) offers a standardised [Obj15] and widely-used [Pet13] metamodel for various aspects of software development, together with a graphical syntax for it. The metamodel covers areas like architecture description, object-oriented design and business processes. For graphical representation, UML knows fifteen diagram types, seven for describing structural and eight for describing behavioural circumstances [Obj15, p. 683]. The best-known diagram type is probably the class diagram, which is also used in this thesis. UML has been described as the "lingua franca" of software engineering (as cited by Petre [Pet13]). It is known by most practitioners, although practical use is often informal and limited to specific parts of the metamodel [Pet13]. The UML specification also proposes a format for data exchange of UML-conform models [Obj15, pp. 685 ff.].

2.5.3 Palladio Component Model

Modelling software has more to offer than merely being a better abstraction for the implementation. It also allows simulating a software system before it has been implemented. This is, for example, possible by using the Palladio Component Model (PCM), proposed by Becker et al. [BKR09]. It is a metamodel for component-based software architecture, which also captures software's abstract behaviour and context of use. This allows the so-called Palladio-Bench to predict non-functional properties of the software. Architects can, therefore, carry out experiments on a PCM model, predict its behaviour and thereby compare different architectures or deployments [BKR09].

2.5.4 Java Model Parser and Printer

As already mentioned in section 2.1, it is desirable to treat source code as just another model describing the software system in Model-Driven Software Development. The Java Model Parser and Printer (Jamopp) (own spelling: "JaMoPP") defines an Ecore-metamodel of the Java programming language, together with an appropriate deserialiser (parser) and serialiser (printer) for EMF [Hei+09]. Using Jamopp, tools developed for Ecore-metamodels can be used for Java code, too, without the need for any special handling. It is, for example, possible to transform a PCM model to Java source code and back using solely the mechanisms provided by EMF, as Langhammer showed [Lan17]. Jamopp only defines the Java language at version five and does not support handling files written in more recent versions of Java [Dev16].

2.5.5 Xtext

Defining a new programming language includes several steps: after devolving the language's grammar, a lexer is required to split input files into tokens, and a parser needs to be created to process the tokens and create an abstract syntax tree from them. To simplify writing in the language, it is often desired to create an editor in an integrated development environment, which supports syntax highlighting, code completion, annotating errors in the source code and so on.

Xtext, a framework for developing external domain-specific languages, automates a lot of this tasks [EV06]. Language developers start creating a new language by defining its concrete and abstract syntax together in one grammar specification. Based on the specification, Xtext generates various artefacts for the language, including a parser, an abstract syntax tree and editors for the integrated development environments Eclipse and Intellij. These editors already include syntax highlighting for keywords, error annotation, autocompletion for referenced elements, and more. All of these artefacts are designed in a way that makes extending or overriding them easily possible. All classes use dependency injection, allowing developers to provide their own implementation for any aspect of a generated parser, compiler and editor [ES17]. Because of that, the framework reduces the effort to create a domain-specific language significantly, without restricting developers' possibilities.

The abstract syntax tree generated by Xtext is an Ecore metamodel [ES17]. This is another example of the close relation of domain-specific programming languages and metamodels, which we described in 2.3. It also means that Xtext can be used to create a textual concrete syntax for Ecore metamodels and that tools from Model-Driven Software Development can be used to process the metamodel, for example to create code from it. This thesis uses Xtext to implement the Commonalities Language and exploits many of its advanced features, like importing existing grammars.

2.6 View-Based Modelling

The software development process includes a variety of stakeholders, all having their specific concerns and thus specific views on the software. View-based software development attempts to represent this observation in the tools used for software development. While there are different understandings of what a view should exactly be [ATM15], it is sufficient for this thesis to imagine it as an "object which encapsulates partial knowledge about the system and domain" [Fin+92]. The concept relates well to domain-specific models in Model-Driven Software Development, as a model can represent a certain view on its original [Bur14, p. 31]. A distinction can be made for how a view model relates to other models. A *synthetic* view is first defined and then integrated with the other models by defining appropriate transformations. A *projective* view, on the other hand, is derived from existing models through some extraction procedure [ISO11]. This allows the view to automatically transform modifications back to the source models based on the information from the extraction procedure.

2.6.1 Orthographic Software Modelling

Atkinson et al. [ASB10] presented Orthographic Software Modelling, an approach using only projective views. It is inspired by the orthographic projections used in technical drawing. At its core, it uses a single underlying model (SUM) containing all information about the system. This model is kept free of semantic overlap [ASB10; ATM15]. Views are created on demand like spokes around a hub, transforming the information edited in them back into the SUM. The approach thus does not need consistency preservation.

2.6.2 Vitruvius

The language presented in this thesis was implemented in Vitruvius, a framework for view-based modelling presented by Kramer et al. [KBL13]. Vitruvius is inspired by Orthographic Software Modelling, but uses a virtual single underlying model (VSUM) that is formed by multiple models. A SUM has to be defined up front and must not contain semantic overlap, which makes extending it difficult. Vitruvius' VSUM, on the other hand, combines metamodels in a modular manner, meaning that metamodels can be added and removed from it. This has the advantage that existing, well-tried metamodels and their tools and editors can be used. In consequence, the models making up the VSUM can share semantic overlap and must actively be kept consistent.

Like Orthographic Software Modelling, Vitruvius permits changes to the models only through projective views. These can, for example, be existing editors for the models in the VSUM. Views can also be defined using a domain-specific language, Modeljoin, and can combine information from multiple models of the VSUM [Bur+14; Bur14]. In effect, Vitruvius has a hybrid structure: It uses projective views for model editing, but active consistency preservation for the VSUM.

Vitruvius is implemented as an extension to the integrated development environment Eclipse. It uses EMF as its metamodelling infrastructure. While the approach was developed with software engineering in mind, it can be applied to various fields that use computer-aided modelling. It has, for example, been used to build a unified model of the so-called smart grid to provide better electricity outage management [BMK16]. To support Model-Driven Software Development, Langhammer presented an approach to co-evolve Java code and an architectural model in PCM using Vitruvius. Recent works used the framework to define consistency preservation rules between PCM and UML [Kla17] as well as between UML and Java [Che17].

2.7 Consistency Preservation in Vitruvius

As outlined above, the Vitruvius framework needs to actively preserve consistency for the models contained in its VSUM. The language presented in this thesis is meant to support this task in Vitruvius and builds upon another consistency preservation language already developed for it. This last section of this chapter will, therefore, give an overview of how consistency preservation is realised in Vitruvius and introduce said language.

Consistency preservation in Vitruvius is change-driven. The framework monitors the views presented to the user and records changes made to them. These changes are then applied to the correct models in the VSUM. Based on each changed element's type and the nature of the corresponding change, the framework selects transformations that declare to handle this combination. These transformations then execute consistency preservation logic reacting to the changes [KBL13; Kra17, p. 20].

2.7.1 Correspondences

Consistency preservation rules are specified at metamodel level but executed to keep model instances consistent. To document that two model elements are consistent with each other, Vitruvius allows transformations to store *correspondences*. A correspondence relates a set of model elements to another set of model elements and thereby witnesses their consistency. This information can be queried by transformations to act in conformance with the system's state. Correspondences are more than a cache, they witness decisions made earlier in the consistency preservation process, that might not be reproducible from the current model state. We assume, for example, that consistency is to be preserved for two metaclasses A and B, such that for every instance of A there is an instance of B, and vice versa. Additionally, both metaclasses have an attribute, and its value should be the same for every pair of corresponding instances of A and B. After an instance a of A is created, the transformations create a corresponding instance b of B. At a later point, the attribute of a is modified and needs to be updated in b. The transformations now need to know which instance of B corresponds to a. This information can only be derived from current state if A and B both declare a set of attributes that uniquely identify their instances and a transformation between those attribute sets is available. This will not be the case in general, so the correspondence information must be stored explicitly in the correspondence model instead. Correspondences can additionally carry string tags to differentiate the context they were created for [Kra17, p. 102].

2.7.2 The Reactions Language

Vitruvius contains a domain-specific language, the Reactions Language, which can be used to create transformation for consistency preservation. The Reactions Language follows the paradigm of reactive programming. Programming includes three steps: First, the developer sets up triggers, that describe changes to models which should be reacted to. Second, the developer retrieves model elements. Third, the developer specifies a transformation that should be executed in reaction to the trigger's changes. The transformation uses the model objects retrieved in the second step [Kra17, pp. 107, 115]. In all steps, the language offers suitable abstractions to relieve the programmer from dealing with technical concerns, like how to retrieve and store model elements [Kra17, p. 119]. Kramer showed that the Reactions Language is complete in terms of the events it can react to and that the executed routines are Turing complete. In other words, the Reactions Language can react to every possible change to a model and execute every computable reaction to such changes [Kra17, pp. 211 f.].

3 Running Example

Throughout the thesis, we will illustrate findings with examples. One scenario such examples will particularly often stem from is consistency preservation for the notion of a component in component-based software. We will assume a software project that uses UML to create its architecture, PCM to model its quality of service, and Java for the implementation. For the sake of simplicity, the consistency preservation will only be concerned with a very basic version of component-based architecture: The software is made out of components which have a name and can contain other components. Every component is contained in a repository, which collects components. A repository also has a name and no further properties (see figure 3.1).

The project wants to preserve consistency of their different models of the software. To do so, they decide to apply the conventions proposed by Langhammer [Lan17], Chen [LK15; Che17], and Klatte [Kla17]. PCM and UML already contain metaclasses for components. Those should be kept consistent with each other. To represent components in source code, a Java package together with a public, final class is created for every component [Lan17, pp. 68-70]. The project wants to use UML also for modelling their object-oriented classes and therefore wants to apply the equivalent transformation from a component to UML packages and classes. Naturally, consistency must also be preserved between the representations in UML and Java, which can be done using the intuitive equivalents of a package and a class in both metamodels [Che17, pp. 26 f.].

A repository of components is represented using the repository metaclass in PCM. In UML, which does not have an explicit class for repositories, it is mapped a metaclass called "Model" [Kla17, p. 9]. For Java, the conventions established by Langhammer are used again: Every repository is represented by three Java packages: One main package, containing the packages created for components, a package for contracts, that will contain interfaces and a package for data types [Lan17, p. 68]. The packages have the name of the repository, with ".main", ".contracts" or ".datatypes" appended to it. Once again, the analogous conversion rules are applied to UML.

To keep the example simple and manageable, we ignore some specifics of the used metamodels. For example, every Java class must be contained in a compilation unit and PCM differentiates between composite components (which can have subcomponents but no behaviour of their own) and basic components (which cannot have subcomponents but can have behaviour of their own) [Reu+11, p. 127].

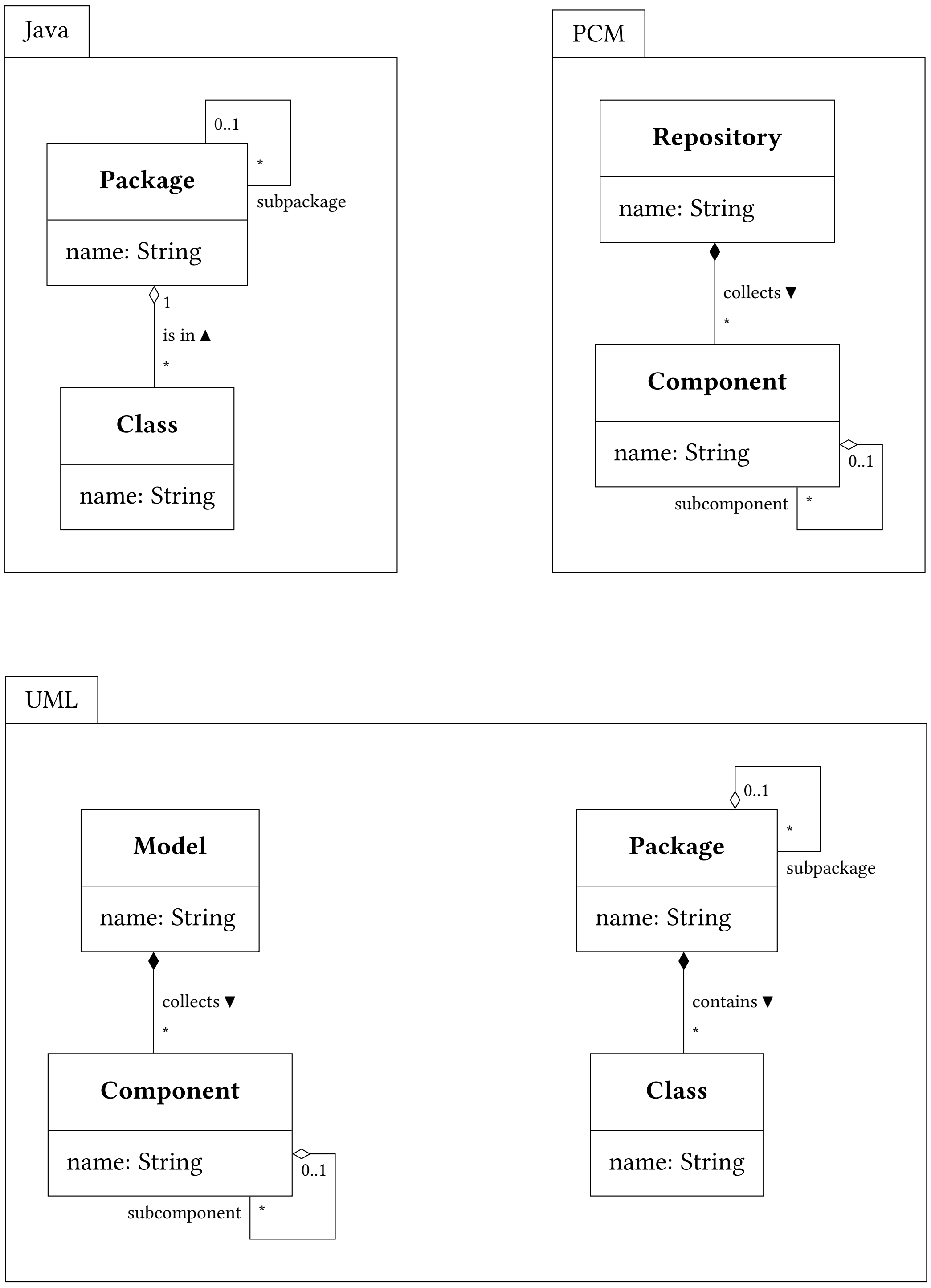


Figure 3.1: The metaclasses of the running example.

4 Transformation Layout

This thesis presents a language to specify consistency preservation rules for multiple models. It is concerned with the case that one consistency specification applies to more than two models. Usually, these models will conform to different metamodels. In this chapter, we will explain the fundamental approach that was chosen to specify multi-model consistency using binary model transformations. We will look at different transformation layouts; that is, which transformations should be created to preserve consistency for a given set of models. We will first explore solutions that only use binary transformations and then explain what this thesis' chosen solution is and why it was selected. When discussing constellations of models and transformations, we will look at the graph that is intuitively formed by them: The involved models form the graph's nodes, and there is an edge between two nodes if there is a transformation between the models. The edges' directions indicate the transformations' directions. We will call this graph the *transformation graph*.

We will assume a set of models, of which any pair shares semantic overlap. This simplification is without loss of generality: If a solution can preserve consistency for an arbitrary set of models with pairwise semantic overlap, it can preserve consistency for any set of models if it is applied to any subset of models sharing pairwise semantic overlap. In practice, the models will likely conform to different metamodels, but we never use this assumption. Transformations are, however, specified at metamodel level (see 2.4.2).

We will additionally assume that only one model is edited before transformations are executed. Because of that, executing the transformations will always start from only one model, and all other models may be modified. This assumption is not realistic, however, executing a transformation graph starting from multiple nodes leads to several principle problems that shall not be discussed in this thesis. Stevens [Ste17] discusses this situation from a theoretical point of view and analyses under which circumstances a transformation graph can successfully be executed if more than one model may not be modified freely. Dam et al. [Dam+16] present a practical approach showing how edited models can be merged while respecting a consistency specification. The process is automated to a high degree and asks the user to decide on conflicts when an automated decision cannot be made. Because Dam et al. pose no requirements on how the consistency specification must look like, their approach could be used with the language developed in this thesis. Regarding only one changed model is thus not an insuperable restriction.

The following criteria will be used to assess different solutions:

Rule Confirmability Developing consistency rules is a non-trivial task. The solution should, therefore, support developers in confirming that the set of transformations is correct; that is, it actually preserves consistency as defined by consistency specification for any allowed edit.

Generality The solution should be feasible for any set of models. It should not require the metamodels or models to have a specific structure. We assume, though, that the models can be kept consistent using only binary transformations. This might not always be the case in reality [Ste17], but that is discussed separately in section 8.4.

Model Modularity It should be possible to modify the set of models of which consistency is preserved. To differentiate the criterion from the others, we only set two requirements at this point: First, adding a new metamodel to the system should only require to add transformations that have the added metamodel at one end. Second, removing a model should be possible without having to change the transformations.

Development Effort Developing the necessary transformations for a set of models should require the least effort possible. At this chapter's level of abstraction, at which no concrete consistency preservation language is considered, this criterion is only concerned with how many transformations need to be specified.

4.1 Multiary Consistency Preservation Using Only Binary Transformations

Given that there are already languages to preserve consistency from one model to another, it suggests itself also to use these tools for the multiary case. Multi-model consistency shall then be achieved by creating binary transformations for two metamodels multiple times. If enough transformations are provided, any change can be propagated to all other models, possibly passing through different models on the way. However, as will be shown in the following, this approach introduces complications in practice. We will present three solutions that only use binary transformations and discuss their advantages and disadvantages.

4.1.1 Fully Connected With Bidirectional Transformations

A simple approach to preserving consistency of multiple models is to specify bidirectional transformations for all pairs of metamodels. The transformation graph hence is fully connected (figure 4.1). If one model is changed, the changes can be transformed directly to any other model.

Rule Confirmability If they must be defined between every pair of models, transformations can become incompatible to each other. It is possible that any single bidirectional transformation helps to preserve consistency when looked at in isolation, while the combination of all transformations still fails to preserve consistency. This leads to the unfortunate situation that the consistency preservation transformations are inconsistent on their own. As a consequence, developers can never concentrate on one transformation in isolation but are forced to always consider the system as a whole. This makes it challenging to both develop correct transformations and confirm the correctness of a given system of transformations.

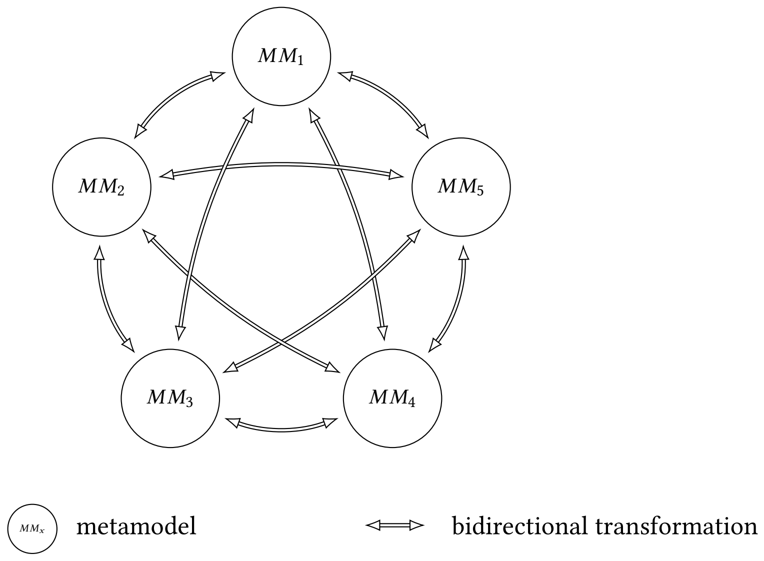


Figure 4.1: A fully connected transformation graph, kept consistent by bidirectional transformations.

To illustrate, we look at a set of consistency transformations for the running example. The specifications are concerned with keeping the name of a component consistent in all models. The process is straightforward, with one restriction: The permissible values for names differ between the models. Neither PCM nor UML restricts the values for names [Obj15, p. 47-50][Reu+11, p. 99], but Java does. In particular, Java class names must not contain symbols like spaces or hyphens [Gos+15, pp. 20 f.]. Our example's project wants to use such symbols, especially spaces and hyphens, in the models to make them more readable. Thus, the names need to be modified when being transformed into the Java model. For the sake of simplicity, we ignore the packages created for components in this example. The following bidirectional transformations are used to preserve consistency:

- For every UML component, there is a corresponding PCM component with the same name. For every PCM component, there is a corresponding UML component with the same name.
- For every UML component, there is a corresponding Java class. The class is public and final and has the UML component's name, but with all impermissible characters removed, characters after spaces in uppercase, and "Impl" appended to it. For every public, final Java class whose name ends in "Impl" there is a corresponding UML component. The component has the name that is obtained by removing the suffix "Impl" from the Java class' name and by adding a space before every uppercase letter but the first.
- PCM components are transformed to Java classes and back in the same manner.
- For every UML component, there is a corresponding UML class. The class is public and final and has the UML component's name, with "Implementation" appended to

it. For every public, final UML class whose name ends in "Implementation", there is a corresponding UML component. The component has the name that is obtained by removing the suffix "Implementation" from the UML class' name.

- PCM components are transformed to UML classes and back in the same manner.
- For every public, final UML class that ends in "Implementation", there is a corresponding, public, final Java class with the same name, but with all impermissible characters removed, characters after spaces in uppercase, and the suffix "Implementation" changed to "Impl". For every public, final Java class whose name ends in "Impl", there is a corresponding, public, final UML class. The UML class' name is obtained by changing the suffix "Impl" of the Java class' name to "Implementation" and by adding a space before every uppercase letter but the first.

These transformation rules correctly transform the models and preserve consistency according to the consistency specification implied by them. However, as described above, the transformations are closely interrelated and can not be verified in isolation. Imagine, for example, that the developers used the same function to transform names of UML classes to names of Java classes as they do to transform names of UML and PCM components to names of Java classes. When looked at in isolation, the transformation would be meaningful and correct. But because the transformations to a UML class already add the suffix "Implementation", a Java class created for a UML class would have the suffix "ImplementationImpl". Such Java classes would then be *inconsistent* to the UML and PCM components, because transforming the name between the components and the class would not obtain the same name. In our small example, the mistake would, of course, be easy to discover and fix. However, in larger projects, with multiple developers working on the transformations, the duplication of similar logic means that no transformation can be changed without also looking at the others. The system is difficult to verify and maintain.

Generality The solution can be applied to any set of models. As consistency can be preserved for the models per assumption, the models can be kept consistent using a fully connected transformation graph, too. Because if there is a transformation graph using only binary transformations that can keep the models consistent, this graph either already is fully connected, or it requires multiple transformation steps after changes for at least one pair of models. In the latter case, a new transformation can be added that realises the same transformation that occurred by the multi-step transformation.

Model Modularity The approach is modular by the definition given for this criterion: Removing models never requires modification because there still exists a transformation for any pair of models. Adding a new metamodel to preserve consistency of requires creating as many transformations as there were models in the set before.

Development Effort This solution has the highest development effort possible, as it requires writing the maximal number of bidirectional transformations possible for n models:

$$\sum_{i=1}^{n-1} i = \frac{n^2 - n}{2}$$

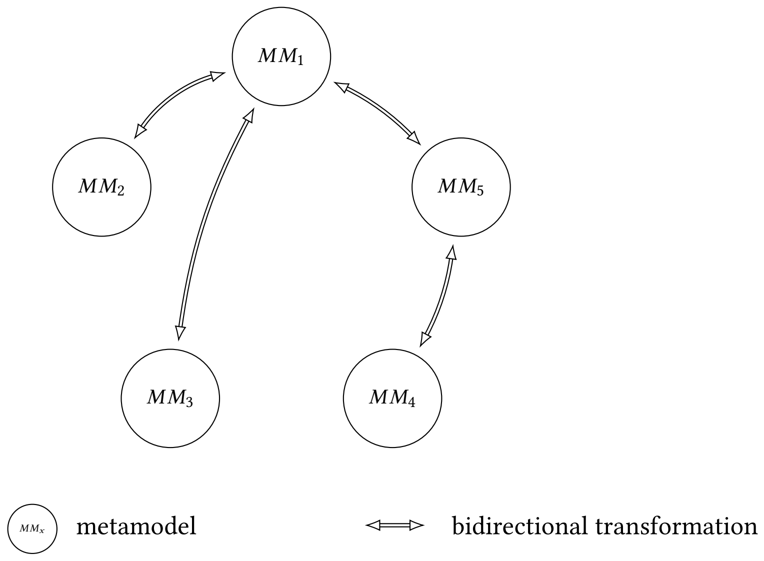


Figure 4.2: Bidirectional transformations, forming a transformation graph that is a tree.

The effort is quadratic in the number of participating metamodels. Our example above already showed that many transformations were required although the example was simple.

4.1.2 Spanning Tree with Bidirectional Transformations

The main disadvantage of the previous solution stems from the number of transformations that need to be defined. To fix this, we might stick to using bidirectional transformations, as they are well understood, but reduce the number of them. For n metamodels, we obviously need at least $n - 1$ transformations, as at least one model would never be kept consistent otherwise. The solution might thus be to define $n - 1$ bidirectional transformations which then form a spanning tree in the transformation graph.

As there is no direct transformation for every pair of models, transformations must now be executed transitively. After a model was changed, it is recursively transformed using all applicable transformations, but only if the target model has not already been changed during the current transformation execution. This realises a breadth-first search on the transformation graph, which must ultimately reach every node because it is an undirected tree.

Rule Confirmability This approach is easier to verify than the last one. There is a unique path the transformation will take for any given edited model. There are no circles in the transformation paths, which avoids that the pathological dependencies between transformations of the fully connected solution. This means that developers trying to verify a set of transformations can concentrate on whether one binary transformation correctly preserves consistency for its two metamodels. If this is the case for all transformations, the whole system will correctly preserve consistency. It can, however, be difficult to answer how editing one model will impact the others, as a change might pass through several transformations after one change. This factor is influenced by the graph's layout. If the graph consists of only leaves except for one root node, the longest transformation path will always include only two steps.

Generality When applying this solution to a set of models, there will always be a unique path of transformations that will be taken to transform one model into another. To make the transformations work, it is required that no information is lost along that path. In other words, the non-leaf models in the transformation graph must be able to store all information that is transformed between any pair of models. This cannot be guaranteed in general. If the information that is transformed between the models can be divided into different parts, such that for every part there is at least one model for which no other model contains more information about that part, then this model could be made the only root of the "sub-transformation tree" that only preserves consistency for this part of information. All other models could be made the leaves and consistency could be preserved for this part of information. However, there will usually not be one model that contains all the transformed information, so a subdivision of the models into different "information parts"—if possible at all—would lead to multiple parts. For every part, the transformation graph would look different. So for every part, the transformations would need to be specified for different model pairs. To summarise, the solution can be applied to a set of models with a good-natured structure. Other sets of models could maybe be kept consistent using this solution from a technical point of view, but the need to subdivide the models' information into different parts and create a different transformation graph for each part would make it impractical.

Model Modularity Adding a metamodel only requires writing another bidirectional transformation. However, if any metamodel that is not a leaf in the graph formed by the metamodels and transformations, is removed, the graph falls into two components (as it was a tree before). Certain models will not be updated if certain other models are edited, even though they share semantic overlap. New transformations have to be created to preserve consistency of the system. The solution is thus not modular, as it prescribes a set of metamodels that have to be in use if consistency is to be preserved.

Development Effort Assuming that there is one transformation graph that can be used to preserve consistency of the models, the solution requires developers to create one bidirectional transformation for any metamodel added to the system. This is a reasonable amount of required effort.

4.1.3 Circle With Unidirectional Transformations

The least amount of transformations in one direction are needed if only using unidirectional transformations that form a circle containing all models present in the system (figure 4.3). That way, the number of transformations equals the number of models to preserve consistency for. Any lower number of transformations would mean that changes to at least one model cannot be transformed into at least one other model. Like the last solution, this one requires executing the transformations transitively. In this case, the only

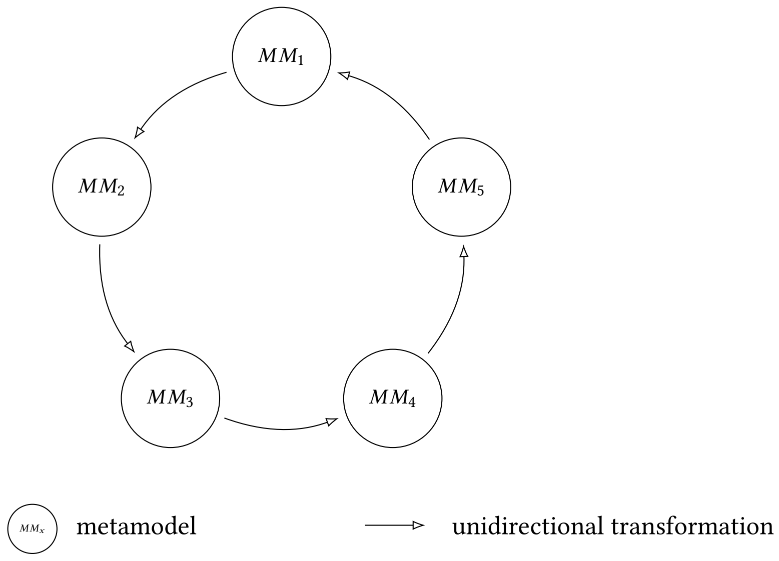


Figure 4.3: Binary, unidirectional transformations preserving consistency in a circle.

applicable transformation is executed in every step until it would change the model that was initially edited.

Rule Confirmability Like the last one, this solution has the advantage that only one transformation needs to be considered at a time when confirming that the system correctly preserves consistency. Verification is furthermore made a little easier by the fact that only unidirectional transformations are used. What makes confirmability worse, however, is the fact that with this solution, the longest transformation path will always have $n - 1$ transformation steps for n participating metamodels. Understanding how a change to a model will affect the model furthest away from it requires accounting for $n - 1$ transformations. This becomes impractical when there are more than just a few metamodels involved.

Generality This solution can only be applied to special constellations of metamodels; namely those were no necessary information is lost in *any* transformation. If a consistency specification requires one transformation to have information that is present in the changed model, but was lost it a previous transformation, consistency cannot be preserved with this solution. For a simple example, we assume three metamodels $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, all having only one metaclass, called A, B or C, respectively. A and B have a number attribute using a floating point number, C has an integer attribute. The consistency specification requires that the number attribute of corresponding instances of A and B must be equal, while corresponding instances of A and C or B and C must have number attributes that are equal when rounded to the next integer. These metamodels cannot be kept consistent using this approach. Because regardless of how the graph is built, there will be one metamodel of $\mathcal{A}$ or $\mathcal{B}$ that can reach the other only through $\mathcal{C}$ in the transformation graph. Because no other facilities than unidirectional, binary transformations are available, the decimal places of the respective model element's number attribute's value will be lost when being transformed into the corresponding instance of C.

Model Modularity Adding a metamodel to the system is easily possible. Removing a model, on the other hand, will always break the transformation cycle, meaning that new transformations have to be created to preserve consistency. This solution thus is the least modular one.

Development Effort We assume that a set of models can be kept consistent without losing information. In that case, this solution needs the least amount of effort, because the least transformations need to be defined.

4.1.4 Other Solutions

There are, of course, other transformation graphs that could be used to preserve consistency for multiple models using only binary transformations. We looked at the most minimal and most maximal possible solutions (regarding the number of transformations), as well as one solution "in between". We would like to argue that any other solution will have properties along the lines of those presented in this section. As a general rule of thumb, using more transformations per metamodel makes a solution more modular, while using fewer improves development effort. Confirmability profits from using fewer transformations, although understanding how the transformations affect the models gets more complicated with fewer transformations. Overall, it seems like restricting ourselves to using nothing more than binary transformations does not lead to satisfactory results.

4.2 Multiary Consistency Preservation Using an Intermediate Model

Inconsistency can only occur if models share semantic overlap (see section 2.4). In usual transformation algorithms, overlap is never made explicit but implicitly assumed. It could be derived by looking at what model elements and attributes are being modified. Imagine, for example, that a set of transformations makes sure that the name attribute of corresponding instances of different metaclasses always has the same value. We could then rightfully conclude that these metaclasses share semantic overlap, at least for the name attribute.

As semantic overlap is the cause of inconsistency, it might be worthwhile to find an explicit representation of it before transforming it between models. The solution proposed by this thesis is to introduce a model of the semantic overlap that model instances of metamodels can have. This model is called the *intermediate metamodel.* It is a metamodel of the semantic overlap that a set of model instances of metamodels have. In our example above, the intermediate metamodel would contain at least a metaclass with a name attribute, to represent the semantic overlap we have found.

When preserving consistency, transformations first transform changes into an intermediate model, which is an instance of the intermediate metamodel of the participating

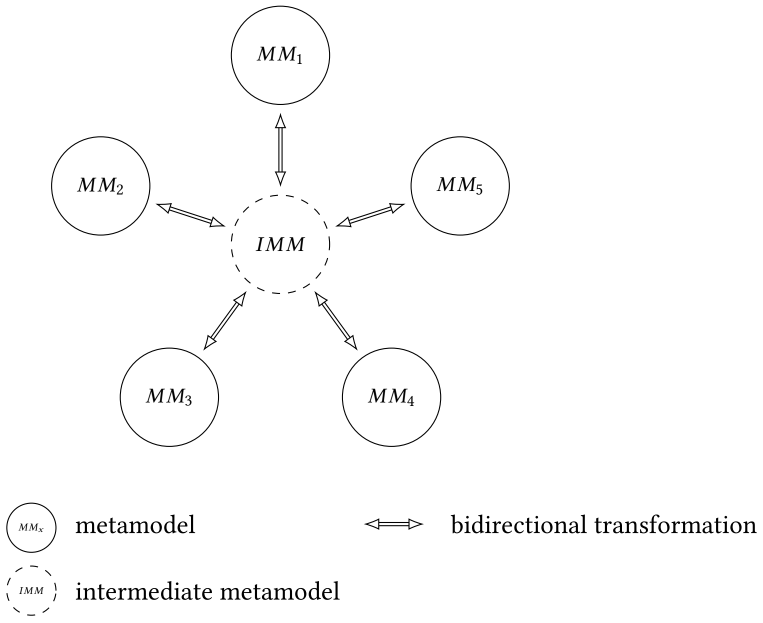


Figure 4.4: Preserving consistency using an intermediate metamodel and bidirectional transformations.

metamodels. The intermediate model is then transformed into the other models. There are no direct transformations between the models that are kept consistent. The transformation graph of this solution is thus a tree that has all models for which consistency should be preserved as leaves and the intermediate models as non-leaf nodes (figure 4.4). All nodes are connected by bidirectional transformations. In a system with multiple instances of different semantic overlap, there will be multiple intermediate models. Maybe contrary to intuition, we do not demand that the intermediate metamodel is free of semantic overlap. Instead, tolerating semantic overlap in it might sometimes make it easier to maintain and allow a better separation of concerns. Because developers have complete freedom in designing an intermediate metamodel—which they have not for the existing metamodels they want preserve consistency of—consistency can be preserved for intermediate models using tree-structured transformations, as discussed in section 4.1.2. Model modularity is not an issue for intermediate models, as they only contain information coming from other models. Users thus never need to interact with the intermediate models, and there are no reasons why they would want to remove an intermediate model from the system. Nevertheless, intermediate metamodels should be designed in a way that makes preserving consistency easier, which includes keeping the amount of semantic overlap to a minimum.

To illustrate, we apply the solution to our running example. Here, our semantic overlap is a common notion of a repository, corresponding to a PCM repository, a UML model, three UML packages, and three Java packages, as well as a common notion of a component, corresponding to a PCM component, a UML component, a UML class in a UML package and a Java class in a Java package. We might thus create a metaclass called "Repository" and a metaclass called "Component" in the intermediate metamodel (see figure 4.5).

We would then create the bidirectional transformations between the intermediate metamodel and the existing metamodels. These would realise the relationships as explained

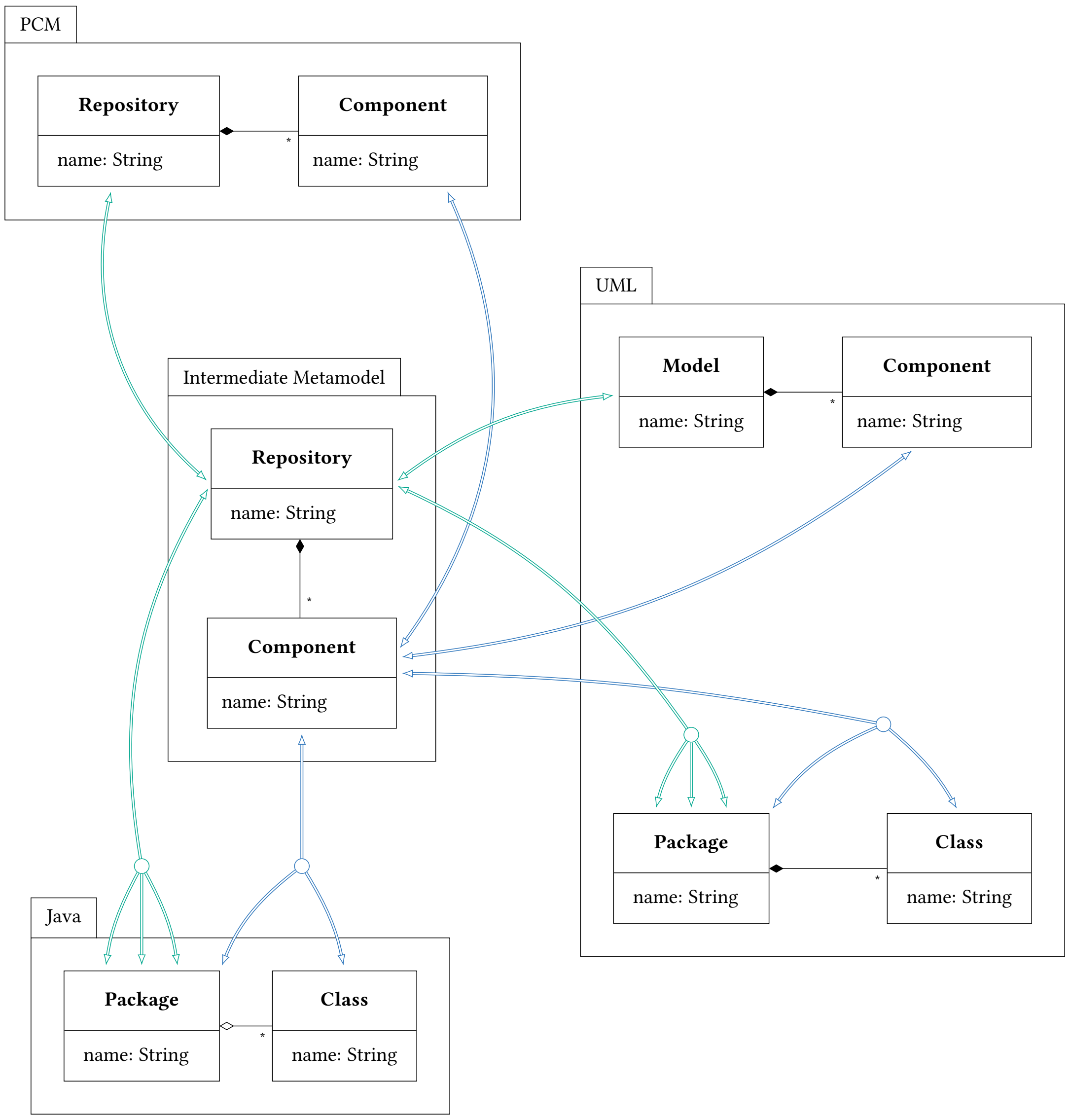


Figure 4.5: The metaclasses for consistency preservation with an intermediate metamodel in our running example. The coloured arrows suggest the bidirectional transformations that would be created. The different colours are only used for better readability and have no further meaning.

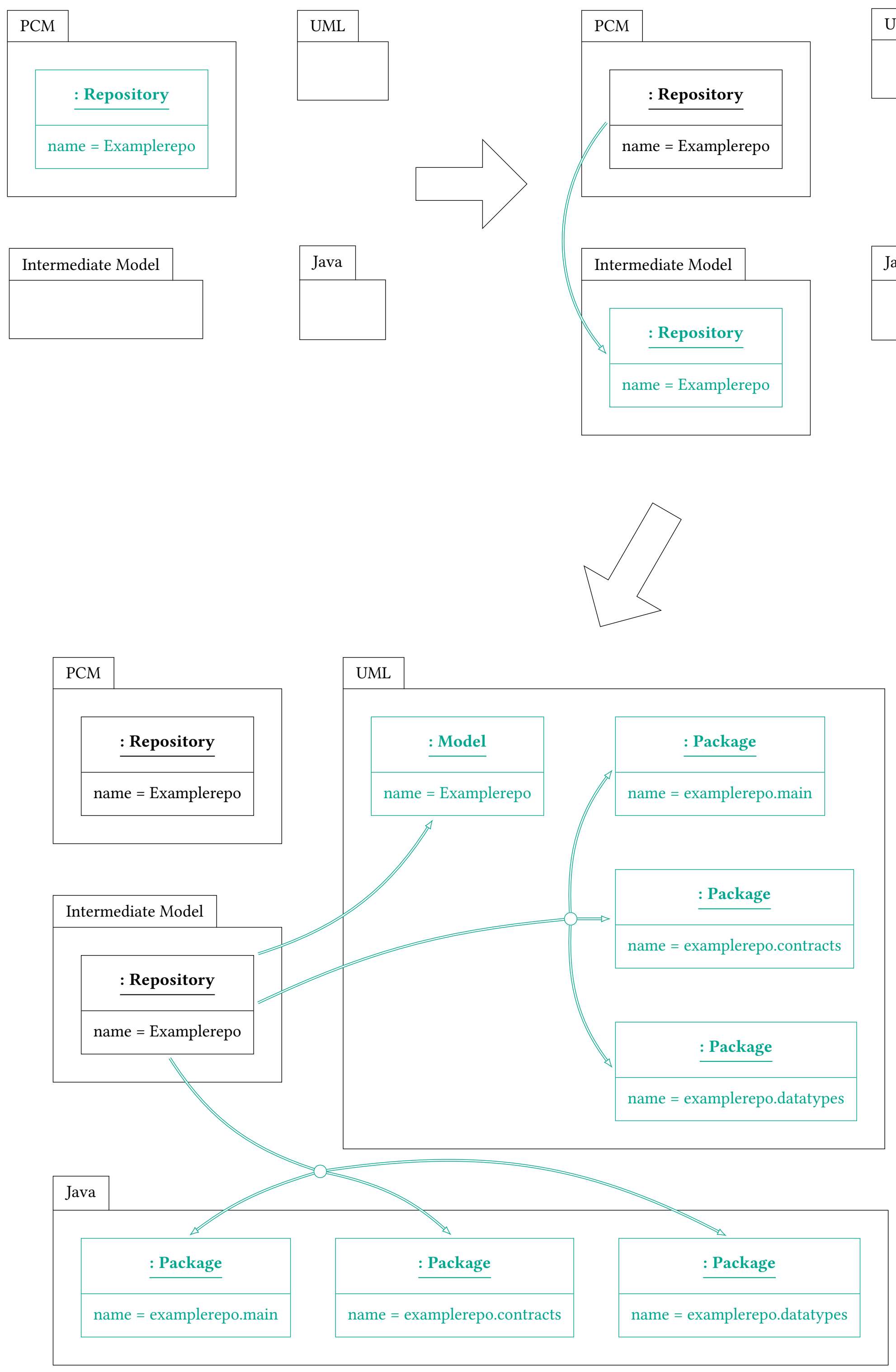


Figure 4.6: An example of how the transformations in our running example would act after a PCM repository was created. New model objects are coloured green in each step. The arrows suggest the transformations that were executed in the respective step.

in chapter 3. For example, a PCM Repository would be transformed into a corresponding repository in the intermediate model with the same name. A repository in the intermediate model would be transformed into three packages in the Java model, having the repository's name with ".main", ".contracts" or ".dataypes" appended to it. All transformations are bidirectional. Figure 4.6 shows how the transformations in our example would act if no model objects existed and a PCM repository would be created. We ignore the fact that components can contain subcomponents for this example.

How does consistency preservation using an intermediate metamodel compare to the other solutions? We again use the criteria developed at the beginning of this chapter:

Rule Confirmability The situation for this property is similar to the solution using only bidirectional transformations that form a tree. When trying to understand how changes impact other models, developers have to take at least two steps, as all information passes through the intermediate models. While there may also be transformations between intermediate models, we think that they will be rare and if they occur, they will only include few steps.

Generality The solution can be applied to any set of models. A set of models can be kept consistent using a fully connected transformation graph with only bidirectional transformations (see 4.1.1). Each of those transformations can be made to pass through the intermediate models by adding all information needed by to the intermediate models. This already leads to the desired tree structure, with the existing metamodels as leaves. However, the intermediate models would then share much semantic overlap and require a considerable amount of consistency preservation rules for themselves. Yet, all information contained in the intermediate models stems from the existing models, and there are transformations to keep the latter consistent per assumption. We can thus remove duplicated information from the intermediate models and append the transformations that were required to keep the intermediate models consistent to the transformations that transform the existing models into the intermediate models. Ultimately, we obtain a set of intermediate models with no or little semantic overlap, together with appropriate transformations, that can be used for consistency preservation of the existing models.

Model Modularity Using intermediate models is a modular solution. Removing models will never require further work. When adding models, either one bidirectional transformation or two bidirectional transformations and a new intermediate model have to be created per set of overlapping models that is not a subset of another set of overlapping models.

Development Effort Given that defining the intermediate model is sufficiently easy, the development effort for this solution is comparable to the last two solutions. It requires to create *n* bidirectional transformations to preserve consistency of *n* models.

Intermediate models have been applied successfully to preserve multi-model consistency in the past. Di Ruscio et al. even describe star-arranged models that are being kept consistent through a central notation as "recently getting consensus in different application domains" [Di +12]. Nevertheless, there are also differences between existing applications of the solution and this thesis' approach, which will be laid out in section 8.3.

Using an intermediate metamodel for multi-model consistency preservation is a solution that combines the generality and modularity of the fully connected solution with the reduced development effort and improved confirmability of solutions with less transformations. Compared to the other solutions, developers have the additional effort of creating the intermediate metamodel. However, the metamodel does not introduce new information. As already discussed in section 2.4.2, model transformations already contain an implicit notion of the semantic overlap. The intermediate metamodel makes this implicit knowledge explicit. Nevertheless, the difficulty and effort of creating the intermediate metamodel seem to be crucial for the effort required for the whole solution. In the next chapter, we will present a domain-specific language for creating consistency preservation rules, which combines the specification of the model transformations with the specification of the intermediate metamodel. It aims to make consistency preservation using an intermediate model a straightforward process.

5 The Commonalities Language

The main contribution of this thesis is the introduction of a new, domain-specific language, the *Commonalities Language*. We designed it to support the consistency preservation process with intermediate metamodels. Its primary goal is to make consistency preservation rules defined in it intuitive and easily understandable.

The Commonalities Language, as presented in this chapter, is not bound to any particular technology. It could be used in any context in which consistency preservation for models is desired; the only conceptual requirements are that the models need to conform to object-oriented metamodels and that a facility to store correspondences of model objects must be available (see section 2.7.1). The language is designed to create transformations that can be used to update models as they change over time. It assumes that at most one model was edited before consistency preservation is executed. The implications of this assumption have already been discussed at the beginning of chapter 4. We call the model instance that was edited and must thus not be changed while its changes are propagated to the other models the *authoritative instance* [Ste17].

This chapter will unfold as follows: First, we will explain what we wanted to achieve when designing the language. We will then provide a brief glance at how the language looks like and what its central constructs are. Subsequently, we will introduce necessary terms and constructs and then describe the language's features in detail. The chapter concludes with an overview of the language's features. Not all designed features of the Commonalities Language could be realised in the prototypical implementation developed for this thesis. Details about the implementation follow in chapter 6, but such not-yet-implemented features will already be marked with a star (*) in this chapter.

5.1 Design Goals

5.1.1 A Declarative Language

The Commonalities Language is designed to be declarative and problem-oriented. Developers specify only *how* consistency looks like, not *what* needs to be done to preserve it. "Declarative" is a vague term and there is no hard line to be drawn between "declarative" and "imperative". Instead, most languages will find themselves on a spectrum between those extremes. Nevertheless, an often-cited definition from the internet paraphrases our goal: With declarative programming, we mean

> "the act of programming in languages that conform to the mental model of the developer rather than the operational model of the machine"

(as cited in [McG16]). The mental model of consistency preservation we want to build is that models have something in common, which the developer wants to describe. Programming with the Commonalities Language is describing the commonalities of models—hence the language's name.

On a more formal level, the Commonalities Language lets developers define an intermediate metamodel and how existing metamodels map to it. Developers do not need to define when transformations need to be executed. Neither must they concern themselves with details of the transformation realisation, like execution order, which changes were made to models, how to retrieve and store models, and the like.

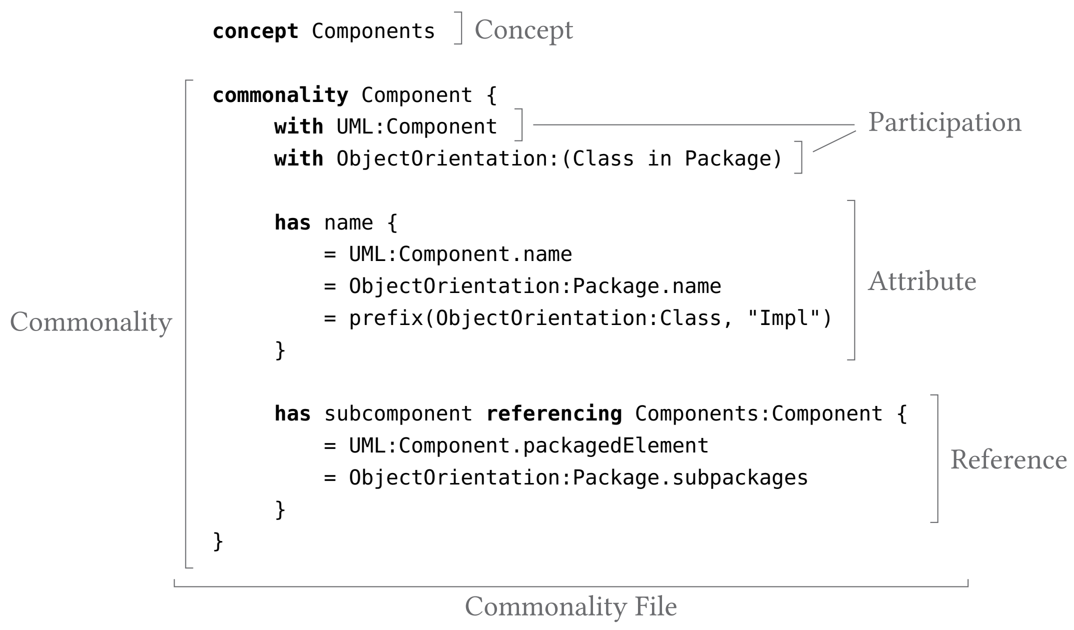
5.1.2 Implicitly Defining the Intermediate Metamodel

Creating an intermediate metamodel is a new and additional step for transformation developers; one that is not present in other workflows. Metamodels are usually built with different tools than transformations. However, the processes of designing the intermediate metamodel and creating the transformations are closely interrelated. It is unlikely that developers will first create the intermediate metamodel they need and then specify the transformations. On the contrary, we expect that specifying the transformations will often reveal additional requirements for the intermediate metamodel or unveil semantic overlap of which the developers were not aware. So the intermediate model will not be in its final form when the transformations are created. It is, on the other hand, not possible to specify the transformations without an existing intermediate metamodel. Hence, transformations and intermediate metamodel will evolve together. A language for the process should account for that.

Ideally, specifying the transformations and defining the intermediate metamodel will not feel like two different steps for developers, but like one process. We thus aimed for a language that does not impose constant context switching on developers. This can only be achieved if developers do not have to switch technologies for the two tasks. Both the intermediate metamodel and the transformations should be created through the language. Because changing one of the two artefacts will usually require adapting the other, it would be even better if both could be specified together in the same file. We realised that with the Commonalities Language.

5.1.3 Keeping the Basic Case Simple

When designing features and the syntax of the language, we aimed to make specifying the simplest case of consistency preservation as easy as possible. This primarily means that the programmer has to handle the least amount of concepts as possible, but also to provide a compact syntax for it. In particular, the Commonalities Language should be fit to be used for cases where consistency only needs to be preserved for two models. The features that make multi-model consistency possible should not make two-model consistency preservation more complicated—or at least not more than necessary. The vision for the Commonalities Language is that it can be applied to any consistency preservation problem, not just the multiary case.



```
concept Components

commonality Component {
    with UML:Component
    with ObjectOrientation:(Class in Package)

    has name {
        = UML:Component.name
        = ObjectOrientation:Package.name
        = prefix(ObjectOrientation:Class, "Impl")
    }

    has subcomponent referencing Components:Component {
        = UML:Component.packagedElement
        = ObjectOrientation:Package.subpackages
    }
}
```

Figure 5.1: A simple Commonality File. Annotated are the most important classes from the language's abstract syntax.

5.2 Overview

Figure 5.1 provides a first impression of the Commonalities Language. It shows code in the Commonalities Language that could be used to preserve consistency of components in the running example. Without going into detail about the exact semantics, we can see that a common notion of a "component" is established. Such a metaclass of objects which are common in multiple models is called a *Commonality*. We can furthermore see from the statements starting with "`with`" that the component Commonality is shared with the metaclass "Component" from UML and a class and a package from object orientation. These statements starting with "`with`" are called *Participations*. They declare how instances of metaclasses map to an instance of a Commonality. In this case, there should be a UML component as well as a package and a class from object orientation for every component Commonality. The metaclasses listed in a Participation are called *Participation Classes*. The second Participation has two Participation Classes that reference a "Class" and a "Package" from "ObjectOrientation". These are two other Commonalities which are not shown in the example. "ObjectOrientation" is the Commonalities' *Concept*. Our component Commonality also has a Concept. It is called "Components" and declared at the top of the Commonality File.

After the Participations, the Commonality contains an *attribute* and a *reference*. The attribute is called "name". The block surrounded by braces contains *attribute mapping specifications*. These define how attribute values of model objects map to attribute values of the Commonality. In our case, the component Commonality's name is equal to the name of the participating UML component and the package from object orientation. The suffix "Impl" is appended to a component's name to form the name of the participating object-oriented class. The equals sign used in the attribute mapping specification declares that the relation between the attributes should be kept consistent in both directions.

Finally, the reference in our example is called “subcomponent” and references the very component Commonality we are declaring. It also contains mapping specifications, so-called *reference mapping specifications*. They define model objects whose corresponding Commonalities should be contained in the reference. In our case, the “subcomponent” reference points to those Commonalities that correspond to UML components that are in the “packagedElement” reference of the participating UML component as well as to the object-oriented packages that are in the “subpackages” reference of the participating object-oriented package. Like the attribute mapping specifications, these reference mapping specifications create bidirectional transformations.

5.3 Common Constructs

5.3.1 Comments and Whitespace Handling

The Commonalities Language allows developers to include comments in Commonality Files. This makes it possible to annotate non-obvious ideas behind definitions, mark places where more work is needed, or quickly disable a declaration without deleting it. The language uses the well-known syntax from C-like languages: “`//`” starts a comment that reaches until the next newline symbol, and “`/*`” starts a comment that reaches until the next occurrence of “`*/`” and may contain newline symbols.

Comments starting with “`//`” are the only instance in the Commonalities Language where a newline symbol has semantics. Apart from that, newlines and whitespace characters—like spaces and tab characters—are only needed to separate identifiers and keywords. They carry no further semantics. Whitespace characters, as well as “`/*`,`*/`”-comments, may be inserted wherever an identifier or keyword starts or ends.

5.3.2 Referencing Metaclasses and Properties

When defining mappings for properties, metaclasses and their properties need to be referenced. The Commonalities Language uses a uniform syntax for such references, depicted in figure 5.2. This syntax is used for three different cases: Referencing existing metaclasses, referencing Commonalities, and referencing Participation Classes. In all cases, the first part provides the name of a metamodel-like object and the second part the name of a class-like object that is contained in the first object. If referencing properties, the third part gives the name of a property of the second element. The three different cases are shown in table 5.3.

5.4 Expressions*

Throughout the Commonalities Language, expressions are used to define mappings or to express conditions. Before the language’s features are introduced, we will look at how such can look like. The Commonalities Language knows three different types of expressions:

```
Metamodel = ID;
Metaclass = ID;
Property = ID;
Qualified Metaclass = Metamodel, ':', Metaclass;
Fully Qualified Property = Qualified Metaclass, '.', Property;
```

Figure 5.2: The syntax to reference a metaclass or Commonality, or a property thereof, given in the extended Backus-Naur Form [ISO96]. The rule “ID” is the rule for valid identifiers.

	FirstPart	:	SecondPart	.	ThirdPart
existing metaclass	metamodel	:	metaclass	.	property thereof
Commonality	Concept	:	Commonality	.	
Participation Class	Participation	:	Participation Class	.	

Table 5.3: The different cases of the reference syntax and what is referenced by name in each case. The third part is only given when referencing properties.

invertible expressions, *enforceable condition expression*s, and *predictable expressions*. The first type can be executed “in both directions”; that is, input and output parameters can switch their roles. The second type are expressions that can be interpreted as a condition if all variables are given, but also as an expression generating values for the variables. The third type, predictive expressions, are “normal” expressions, like they are known from general-purpose programming languages, with one relatively small restriction: the result of an expression may only change if its input parameters change.

The Commonalities Language does not prescribe any specific syntax or semantics for any expression type. Instead, it builds on existing expression languages with the desired properties and integrates their syntax and semantics. For this thesis, we will focus on expression languages developed by Kramer [Kra17], but other languages could be used. The method of using foreign code for expressions in domain-specific languages is common practice [Kar+09; FP10, pp. 309 ff.]. It helps to keep the specification of a domain-specific language small and prevents it from “reinventing the wheel”. If the used expression languages are well-known or imitate the syntax of popular languages, the technique makes it easier for programmers to adopt the domain-specific language: they only have to learn the other concepts of the domain-specific language and can use their experience from other languages to write expressions. Because we expect the Commonalities Language to be mainly used by programmers, embedding well-known expression languages follows the general goal of domain-specific languages to adopt existing notations [MHS05]. For example, the expression languages by Kramer, which we present in this thesis, use a syntax that is a subset of Java’s syntax.

5.4.1 Simple but Extendible Expressions

One central goal in the development of the Commonalities Language was to provide developers clarity about created transformations. Creating Commonalities should be straightforward, but it was more important for us to ensure that *understanding* them is possible effortlessly. The Commonalities Language consists of relatively few elements and is thus, hopefully, easy to understand. However, this expressiveness could be undermined by allowing arbitrary statements from general-purpose languages to be embedded in the language. Because of this, expression languages in the Commonalities Language are simple but extensible.

In particular, the Commonalities Language does not offer any way to define routines or functions. It is also not possible to create multiple statements or code blocks in an expression. This is deliberate. The language encourages developers to keep Commonality Files compact and readable, because we regard this to be the most effective way to ensure consistency preservation rules stay maintainable and, ultimately, correct. Specifying consistency rules is inherently domain-specific. Thus, developers will need the possibility to implement custom logic. We argue that this should never happen in Commonality Files, but in external files. Instead of allowing the declaration of new functions or complex expressions in the language, the Commonalities Language enables developers to use operators that are defined in a general-purpose programming language in external files. All expression languages offer the possibility to extend their set of operators. This way, developers can use existing libraries or implements their own helper functions.

In 4.1.1, we presented an example where component names, which can be arbitrary strings, need to be transformed into valid Java identifiers. Although the conversion is not particularly complex, it requires several steps: To obtain a valid Java identifier, any impermissible character must be removed. It will usually also be desirable also to convert the string to camel case notation; that is, to convert every letter after a space to uppercase. This might lead to the situation that two different names are mapped to the same value, so a de-duplication strategy must also be applied. When converting Java names back to component names, a best-effort strategy might be applied to obtain a more readable name. A space could be added in front of every uppercase letter, for example.

Adding the logic for these conversions to a Commonality would distract from the transformations that are defined in the Commonality. Instead, developers should program the logic in an external file, give it a descriptive name, and then use it as an operator in attribute mappings. This is how the development process is envisioned for the Commonalities Language: Developers use existing, mature programming languages to define operators; which they then use to create expressive Commonality declarations.

5.4.2 Invertible Expressions

Intuitively, defining how a property of a metaclass maps to a property of a Commonality often already includes all information needed to determine how to transform values of the Commonality's property back to the metaclass. For example, we might have a metaclass that stores an execution time in seconds in a property *m*. However, we decide to store that time in the Commonality's corresponding attribute *c* in milliseconds (for example because all other participating metamodels also use milliseconds). Hence, we declare that $c = m \cdot 1000$. Using a simple equivalence transformation, we directly see that $m = \frac{c}{1000}$ follows. However, usual expression languages force us to write down both versions explicitly. Using them would thus mean that programmers have to create a lot of superfluous expressions.

This problem presents itself in any language for bidirectional transformations that uses expressions. Therefore, Kramer and Rakhman [KR16] present a set of 30 invertible operators that address the issue. The operators cover fundamental operations used in programming languages—like arithmetic, boolean operators and string processing—and all have an according inverter. By combining the inverters, a given expression build out of the operators can be transformed into an inverted expression. The inverted expression has the source expression's output parameters as input parameters, and vice versa.

The operators are not only concerned with bijective cases, where no information is lost, but also handle operations that lose information. Kramer and Rakhman show that, wherever possible, the operators perform *well-behaved transformations*, as introduced by Foster et al. [Fos+07]. That means that a round trip through an operator op and its inverse op^{-1} in either direction does not change a value: $\mathrm{op}^{-1}(\mathrm{op}(s), s) = s$ for all source values s (Get-Put law), and $\mathrm{op}\left(\mathrm{op}^{-1}(t, s)\right) = t$ for all source values s and target values t (Put-Get law). The inverted operator op^{-1} has two arguments and takes the initial model as its second parameter to realise the principle described in section 2.4.2, that one direction of bidirectional transformations takes its target model as input to better adapt to existing values.

If an operator's transformation cannot be well-behaved due to its nature—i.e. it is not surjective—Kramer and Rakhman make sure that it is a *best-possible behaved transformation*. That means it fulfils the Get-Put law in every case and the Put-Get law whenever possible [KR16]. 14 out of the 30 operators are well-behaved, the others are best-possible behaved.

To illustrate, we look at some of the operators. For the floating point operations addition, multiplication, subtraction and division can easily be realised as well-behaved invertible operators, as already discussed above. Even integer division $\mathrm{intdivison}(s_1, s_2) = \left\lfloor \frac{s_1}{s_2} \right\rfloor$ with integer arguments s_1, s_2 is well-behaved, despite the loss of the decimal places, because the target value can be taken into account. The inverse operator for the first argument s_1 is defined as:

$$\mathrm{intdivision}_1^{-1}(t, s_1, s_2) = \begin{cases} s_1 & \text{if } \left\lfloor \frac{s_1}{s_2} \right\rfloor = t \\ t \cdot s_2 & \text{otherwise} \end{cases}$$

and the inverse operator for the second argument s_2 is given by:

$$\mathrm{intdivision}_2^{-1}(t, s_1, s_2) = \begin{cases} s_2 & \text{if } \left\lfloor \frac{s_1}{s_2} \right\rfloor = t \\ \left\lfloor \frac{s_1}{t} \right\rfloor & \text{otherwise} \end{cases}$$

Finally, we look at the string concatenation operator $\mathrm{concat}(\mathrm{s}_1, \mathrm{s}_2) = \mathrm{s}_1 \cdot \mathrm{s}_2$ for two string parameters s_1 and s_2 ($|\cdot|$ shall denote the word concatenation), for which we will briefly write $s_1 \frown s_2$. The operator cannot be well-behaved, because there is, for example, no string x such that `"solution"` = `"bad"`$^\frown x$. The inverters are therefore only best-possible behaved. The inverse operators for the arguments s_1 and s_2 are respectively given by:

$$\operatorname{concat}_1^{-1}(t, s_1, s_2) = \begin{cases} s_1' & \text{if } t = s_1'{}^\frown s_2 \\ \bot & \text{otherwise} \end{cases}$$

$$\operatorname{concat}_2^{-1}(t, s_1, s_2) = \begin{cases} s_2' & \text{if } t = s_1{}^\frown s_2' \\ \bot & \text{otherwise} \end{cases}$$

$c_1 = x^\frown c_2$ or $c_1 = c_2{}^\frown x$ is, if solvable, unambiguous for constant strings c_1, c_2 and variable string x. What to do in the error case $\bot$ is up to the implementation of the operators and will vary by use case [KR16]. In interactive scenarios, the user could be asked.

5.4.3 Enforceable Conditions

An expression that can be true or false can be interpreted in two different ways. On the one hand, it can be seen as a condition: If the expression is filled with values, it evaluates to either true or false. On the other hand, it can also be seen as *prescribing* the values of variables used in the expression. The second form is common in maths, in sentences like "let there be an $\varepsilon > 0$". The expression "$\varepsilon > 0$" is not used as a condition in that case, but rather to describe the valid values of ε.

In consistency preservation, this duality can be helpful when defining bidirectional transformations. If a certain condition must apply to a model object before it is transformed into the other object, that condition should also be true at the end of transformations from the other model. For example, if only public Java classes are considered in the running example when looking whether a new component needs to be created in UML or PCM, then classes that are created for UML or PCM components should also be public. Kramer [Kra17] has created an expression language consisting of operators that can be *checked* to see if a variable fulfils a condition, and *enforced* to generate a value for the variable according to the conditions. Most of the operators have a form similar to our "$\varepsilon > 0$" example: One operand must be a literal value, while the other operand is a feature of a metaclass [Kra17, pp. 143-146]. The simplest operator is the equals operator: When checked, it returns true if a metaclass feature is equal to the literal value. When enforced, the metaclass feature is set to the literal value. The set of operators also covers more complicated use cases, like containment in a list, checking and enforcing a condition for all elements in a list and number inequality. Number inequality is enforced by adding the smallest value possible to the attribute, such that the condition holds. So for our "$\varepsilon > 0$"-example, the operator would set the attribute ε to the smallest number greater than 0 that is supported by the runtime system [Kra17, pp. 148-150].

5.4.4 Predictable Expressions

Invertible expressions can be used in the Commonalities Language for all bidirectional cases. But if only unidirectional mappings are needed, or the intended cannot be expressed with the invertible expression language, developers can use predictable expressions. These

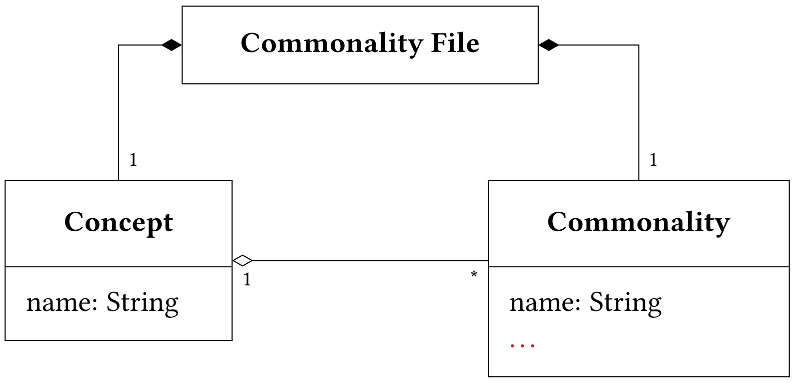


Figure 5.4: Abstract Syntax of a Commonality File, a Commonality and a Concept.

are expressions, preferably in a syntax known from general-purpose languages, with only one restriction: An expression's result only changes if its input arguments' values change. Notably, this is less restrictive than demanding the expressions to be pure functional: predictable expressions may have side effects. Predictable expressions guarantee that the Commonalities Language can be used for incremental consistency preservation. It is important to know up front which expressions have to be evaluated after changes to a model. For predictive expressions, the implementation knows that it is only necessary to evaluate them if their input arguments changed.

5.5 Commonalities

The central element of the Commonalities Language is a Commonality. It is used to both declare metaclasses of the intermediate metamodel and host the transformations from the existing models to the intermediate model. A Commonality is declared in a Commonality File. A Commonality File must contain exactly one Commonality, and its file name must match the Commonality's name (ignoring the file extension). These constraints were copied from Java to make Commonalities easy to discover.

A Commonality File must also declare a Concept (see figure 5.4). Concepts create a namespace but have no further semantics from a technical point of view. Instead, they are meant as a way for developers to organise Commonalities and better communicate their intentions. In this chapter's introductory example we already showed how concepts could be used for the running example: We might intuitively see the two concepts of component-based software architecture on the one hand and object orientation on the other hand. The former would contain Commonalities that concern themselves with mappings to PCM and the component-related parts of UML and the latter would contain mappings to Java and the object-oriented parts of UML.

5.5.1 Visibility

Every Commonality is visible to every other Commonality. Commonalities can be referenced using the reference syntax, without requiring any form of imports. This is because we think that there will never be more than a few hundred Commonalities defined for the same system, so they can easily be managed using the namespacing provided by Concepts. A global namespace and no requirement for imports simplifies the languages further.

5.5.2 Deriving the Intermediate Model

A Commonality constitutes a metaclass in the intermediate metamodel. This metaclass has the same properties as a Commonality has. There is, however, a noteworthy difference between attributes and references in the Commonalities Language and attributes and references in other metamodelling tools like Ecore. In Ecore, everything holding scalar values or plain Java objects is an attribute, while everything pointing to another Ecore model object is a reference. In the Commonalities Language, an attribute can hold both scalar values, Java objects (if applicable) and model objects. References in the Commonalities Language, on the other hand, point to other Commonalities. So a Commonality attribute would be translated to an Ecore attribute or an Ecore reference, depending on whether it holds a non-Ecore object or an Ecore model object. Commonality references would always be translated to Ecore references that point to the respective metaclass of the Commonality the Commonality reference points to.

As we described in the design goals (section 5.1), declaring the intermediate model should blend in the process of creating transformations. We thus do not want developers to think of Commonalities as a way of declaring metaclasses, but rather as a mean to group transformations.

5.6 Participations

After a Commonality has been declared, it needs to be specified how metaclasses from the metamodels for which consistency is to be preserved map from and to the Commonality. The first part of this task, defining which instances correspond to each other, is done through *Participations.* Participations are declared inside of a Commonality and relate the Commonality to other metaclasses. One Participation captures the metaclasses of one metamodel that together share the semantic overlap described by the Commonality. The meaning of the most basic form of a Participation, with one metaclass, is:

> For every instance of the metaclass, there should be a corresponding instance of the Commonality. For every instance of the Commonality, there should be a corresponding instance of the metaclass.

Such Participations hence realise a one-to-one relationship for model objects. Because transformation is executed transitively through the intermediate metamodel, one-to-one relationships between model objects are realised by creating a Participation for each of the objects' metaclass in the same Commonality.

One metaclass mentioned in a Participation is called a *Participation Class.* A Participation can declare more than one Participation Class, but all Participation Classes' metaclasses in a Participation must come from the same metamodel. In particular, it is possible to declare the same metaclass multiple times in one Participation using different Participation

Classes. This realises a one-to-n relationship (with fixed arity n) between the Commonality and that metaclass. Through transitive propagation, the technique allows creating m-to-n relationships (with fixed m and n) between model objects. The meaning of a Participation with $n \in \mathbb{N}^+$ Participation Classes, where the i-th Participation Class references the metaclass M_i, is:

> For every combination $(o_1, \dots, o_n) \in \{x \mid x : M_1\} \times \dots \times \{x \mid x : M_n\}$ of instances of the the metaclasses $M_1, \dots, M_n$ there should be a corresponding instance of the Commonality. For every instance of the Commonality, there should be a corresponding combination $(o_1, \dots, o_n) \in \{x \mid x : M_1\} \times \dots \times \{x \mid x : M_n\}$ of instances of the metaclasses $M_1, \dots, M_n$.

The definition is further restricted: First, the same model object may only be used once in a Participation. So in all occurrences in the above definition, it follows that $o_i \neq o_j$ for $1 \leq i, j \leq n$ and $i \neq j$. Second, all model objects used in a combination must come from the same model. Third, the same model object may only correspond to one Commonality instance of a specific Commonality. So if C is a Commonality, $c_i, c_j \in \{c \mid c : C\}$, $c_i \neq c_j$ different instances of this Commonality, and O_i and O_j the sets of model objects that correspond to c_i and c_j, then $O_i \cap O_j = \emptyset$. These restrictions realise the important parts of the semantics of Participation Classes: They are meant to give the participating model objects a particular role (like the different Java packages used to represent a repository in the running example). One model object should only ever have one role for one Commonality. Furthermore, the restrictions are necessary to make references (see section 5.8) unambiguous.

Other Commonalities can also participate in a Commonality. This feature can be used as a technique to split Commonality Files and separate concerns. For example, the component Commonality in the example at the beginning of this chapter has two other Commonality from the concept "ObjectOrientation" as a Participation, instead of directly using the appropriate classes from UML and Java. That way, the component Commonality is only concerned with how a component translates to elements from the concept of object orientation. The (often technical) details of how these elements are represented in concrete metamodels can be encapsulated in the concept "ObjectOrientation". The graph formed by Participation relations must, however, be free of circles, as creating a Commonality instance would cause endless recursion otherwise.

Attribute and reference mapping specifications need to reference the metaclasses that participate in a Commonality; i.e. the Participation Classes. If nothing else is specified, Participation Classes are referenced by the name of the metamodel and the name of the metaclass they represent. Nevertheless, because the same metaclass can be mentioned multiple times in a Participation, a different name can also be assigned to a Participation Class. There can also be more than one Participation in a Commonality that has Participation Classes referencing the same metamodel. This covers use cases in which the same metamodel plays different roles. Because of that, Participations also have a name, which can also be changed. A Participation's name is is the qualifier when referencing its Participation Classes (see section 5.3.2). Examples of different combinations of Participations and Participations Classes having or not having an explicit name set can be found in table 5.6.

```
concept Components

commonality Repository {
    with Java:(
        Package called "MainPackage",
        Package called "ContractsPackage",
        Package called "DataTypePackage"
    )

    ...
}
```

Listing 5.5: Three Participation Classes for a Java package.

Participation Class	referenced as
`Java:Class`	`Java:Class`
`Java:(Class called "Implementation")`	`Java:Implementation`
`Java:Class as "Language"`	`Language:Class`
`Java:(Class called "Implementation") as "Language"`	`Language:Implementation`

Table 5.6: Examples of Participation Classes and how to reference them in mapping specifications. The reference syntax is explained in general in section 5.3.2.

Participations are declared by starting with the keyword “`with`”, followed by the name of the metamodel the Participation’s Participation Classes come from and a colon. If the Participation has only one Participation Class, its metaclass can follow directly. Otherwise, the names of the Participation Classes’ metaclasses follow in a comma-separated list surrounded by brackets (see, for example, listing 5.5).

If we were, for instance, to create a Commonality for a repository in our running example, it might have a Participation with three Participation Classes for the three Java packages a repository is represented by. Each Participation Class would be for the Java package metaclass: One for the main package, one for the contracts package and one for the data type package. Consequently, we might name the Participation Classes “MainPackage”, “ContractsPackage” and “DataTypePackage” (listing 5.5). They would be referenced as “`Java:MainPackage`”, “`Java:ContractsPackage`” or “`Java:DataTypePackage`”, respectively.

5.6.1 Optional Participation Classes*

In some situations, not all instances of a metaclass share semantic overlap with other model objects. In that case, they should only participate in a Commonality under certain conditions. To support this, Participation Classes can be marked optional. The concrete syntax for that is to place a question mark behind them. Unlike for non-optional Participation Classes, no instance of the Participation Class’ metaclass is created by default when a Commonality is created. Instead, expressions in mapping specifications or conditions can use the existence of the Participation Class as a predicate. If true is assigned to the predicate, an instance of the Participation Class is created. Optional Participation Classes are declared by appending a question mark to the name of the metaclass. In expressions, the existence of an optional Participation Class can be queried and set by appending a question mark to a Participation Class reference.

5.6.2 Enforceable Conditions*

A Participation can have a condition that defines under which circumstances there should be a corresponding Commonality for it. These conditions are declared by using the keyword "`whereat`" after a Participation. They are expressed through enforceable condition expressions (see 5.4.3). The operators of the enforceable condition expression language can be combined using the logical and-operator, or-operator or xor-operator. The enforceable condition expressions combined in such a way will be enforced until the whole condition is fulfilled. All metaclass features used with the operators must come from metaclasses that have a Participation Class in the same Participation. Because it is clear from the context, the metamodel of metaclasses referenced in the expressions does not need to be given. The semantics of enforceable conditions is:

> There should only be an instance of the Commonality corresponding to the Participation if the condition holds true.

When a model of the Participation is authoritative, the enforceable condition is *checked* to see whether or not there should be an instance of the Commonality for a given combination of model objects. If another model is authoritative, and new model objects are being created to preserve consistency, the enforceable condition is *enforced* to make sure that the new model objects conform to it.

In our running example, components are, amongst others, represented by UML classes. These UML classes are public and final. Consequently, all UML classes corresponding to a component Commonality must be public and final. On the other hand, only final and public classes need to be considered to create new Commonality instances. This can be expressed in an enforceable condition, as shown in listing 5.7. The equals operator was introduced in section 5.4.3. The constant "public" would need to be imported by an implementation-specific mechanism.

5.6.3 Participation Class Relations

When more than one Participation Class is declared in a Participation, these Participation Classes are often related in a specific way. The Commonalities Language honours this by allowing to declare relations between Participations directly. So-called *Participation Class relations* are declared by placing a relation operator between two Participation Classes in a Participation. The operator replaces the comma that would usually separate the Participation Classes. A relation then applies to the Participation Classes left and right of it. If a relation should apply to more than two Participation Classes, the whole expression can be put in parentheses. The semantics of Participation Class relations are the same as enforceable conditions: When creating new model objects to map the Commonality to,

```
concept ObjectOrientation

commonality Component {
    with UML:Class
        whereat Class.visbitiy equals public
            and Class.isFinalSpecialization equals true

    ...
}
```

Listing 5.7: A Participation with an enforceable condition, specifying that the participating UML class must be public and final.

```
concept ObjectOrientation

commonality Component {
    with ObjectOrientation:(Class in Package)

    ...
}
```

Listing 5.8: A class and a package related with the "in"-operator, specifying that the class must be contained in the package.

the relation is *enforced*. When considering model objects for creating a new Commonality, the relation is *checked*. Relation operators can, like all operators, be provided by the user.

We present two Participation Class relation operators that we assume to be helpful for typical use cases. The first is the "in"-operator. It prescribes that the instance of the left Participation Class must be contained in the instance of the right Participation Class. In Ecore metamodels, this would mean that the left instance is in a containment reference of the right instance. "Contained in" will not always be uniquely defined, as there could be multiple matching containment references. The "in"-operator can only be used if "contained in" is unambiguous. In our running example, the notion of a component is translated into object-oriented constructs by representing it with a class and a package, such that the class is contained in the package The "in"-operator could be used to express this relation between the Participation Classes in a compact manner.

The other operator we present, the "xor"-operator, can be used when two exclusive alternatives are given for a Participation. It requires that the Participation Classes related by it are both optional. Somewhere in the Commonality, there will be a specification declaring when each of the Participation Classes should exist. Without the operator (or an equivalent enforceable condition), it would be necessary to specify two very similar expressions, one for the existence of one of the Participation Classes and one for the existence of the other. Because one Participation Class should only be present if the other is not, the two expressions would express the same condition, with one being the negation of the other. The "xor"-operator now declares that if one Participation Class is present, the other is not. Because of that, it is sufficient to give an expression for the existence of one of the Participation Classes, and the other will always be present when the first is not.

Keyword	Changes Commonality	Changes Participation Classes	Expression Type
`->`		x	predictive
`<-`	x		predictive
`=`	x	x	bidirectional

Table 5.9: The different types of attribute mapping specifications, how they affect the language elements and which type of expressions may be used with them.

5.7 Attributes

A Commonality can contain attributes. An attribute is introduced by the keyword "`has`", followed by the attribute's name. After an attribute's name follows a block of attribute mapping specifications, which define how values from the Participation Classes map to the attribute and back. There are three types of attribute mapping specifications: From the Commonality to the Participation, from a Participation to a Commonality, and equality specifications. The first type, to the Participation, can set one attribute of a Participation Class. It is introduced with the keyword "`->`", followed by a predictable expression that can only use the attribute's name as variable. The result of the expression is assigned to a property of a Participation Class, which must be given after the expression, separated by another "`->`". The second type, to the Commonality, also uses a predictable expression and is only used to set the attribute. It is introduced with the keyword "`<-`", followed by a predictable expression that may use the Participation Classes of one Participation as variables. The result of the expression is assigned to the attribute. The last type, the equality specification, uses an invertible expression and is used to both set the attribute's values and set values on the Participation Class.

The semantics of an equality attribute mapping specification is that the relation which is expressed by the invertible expression should be upheld all the time. In other words, if at any state of the models the variables are filled in with their current values, the expression should evaluate to the current value of the attribute. For the one-directional versions, this is weakened: the expression is only evaluated and applied if the side the mapping specification is "pointing away from" is authoritative. This means for an attribute mapping specification to the Commonality that it is evaluated and its result is set on the attribute when, and only when, any instance of one of the Participation Classes used in the expression becomes authoritative. An attribute mapping specification from the Commonality is evaluated if the instance it would set an attribute on is not authoritative and the attribute's value has been modified because of changes to another model.

Attributes have a type and multiplicity, which are both derived from its attribute mapping specifications. The attribute mapping specifications are invalid if the type or multiplicity cannot be derived without causing contradictions. Attributes can not only hold scalar values but also other model objects (see also section 5.5.2 for the distinction between attributes in the Commonalities Language and other metamodelling tools).

5.7.1 Checked Attribute Mappings*

Attribute mapping specifications can be marked to be *checked* when considering whether a new Commonality has to be created. In that case, the attribute mappings also function as conditions. The semantics is as follows: When a combination of model objects is considered for a Participation of a Commonality and passed all previous conditions, the Commonality is virtually created, and the checked attribute mappings are executed. Only if all checked attribute mappings can be executed—that is, the invertible operator is always defined, see section 5.4.2—and all checked attribute mappings of the same attribute yield the same result, an instance of the Commonality is created. The short, informal description of this behaviour would be: “Only create the Commonality if all checked mappings are defined and do not contradict each other”. Checked attribute mappings prevent duplication of logic in Commonality declarations.

In our running example, we want to create a Java class whose name ends with “Impl” for every component. At the same time, only Java classes ending with “Impl” are considered to form a new component. We could already express this in the Commonalities Language by giving a condition for the Java class Participation, assuming we have or define an according operator. This operator would, one way or another, express that the class’ name must end with “Impl”. We would furthermore create an attribute “name” in the Commonality and specify that it is equal to the Java class’ name when removing “Impl” (listing 5.10 (a)—we assume that this always yields a valid Java identifier for the moment). This solution works, but has a practical flaw: We have defined very similar logic at two different points. This makes the code harder to maintain. If we were, for instance, to change the suffix of the Java class from “Impl” to “Implementation”, we would have to remember to change it at both places. Using a checked attribute mapping instead removes the duplication (listing 5.10 (b)). In listing 5.10, the operator “prefix” is meant to be defined as $\mathrm{prefix}\,(\mathrm{s}_1, \mathrm{s}_2) = t \iff t = \mathrm{concat}_1^{-1}\,(s_1, x, s_2)$ for an arbitrary string x and concat_1^{-1} as introduced in section 5.4.2. The operator is not defined if s_1 (here: the Java class’ name) does not end with s_2 (here: “Impl”). So when using a checked attribute mapping like in listing 5.10 (b), no Commonality instance would be created for Java classes that do not end with “Impl”.

5.7.2 Error Handling

Checked attribute mapping specifications are not the default case but have to be explicitly requested by the programmer by writing “`check`” in front of a mapping specification. This is a deliberate feature. If all attribute mappings were checked by default, mistakes made by the programmer could be hidden. If mappings that are contradictory or not defined in all cases were specified by accident, a checked attribute mapping would lead to a Commonality instance not being created under certain circumstances. This would be difficult to detect in automated tests. Even if it was detected, the cause for the missing Commonality instances could not be tracked down easily. Because of that, mapping specifications are *not* checked per default. If normal attribute mapping specifications yield undefined or contradictory results, an error is raised at runtime. This eases debugging.

```
concept Components

commonality Component {
    with Java:(Class in Package)
        whereat Class.name endsWith "Impl"
    ...

    has name {
        = prefix(Java:Class.name, "Impl")
    }

    ...
}
```

(a) without a checked attribute mapping.

```
concept Components

commonality Component {
    with Java:(Class in Package)
    ...

    has name {
        check = prefix(Java:Class.name, "Impl")
    }

    ...
}
```

(b) with a checked attribute mapping.

Listing 5.10: Checked attribute mappings remove duplicated information.

5.8 References

Additionally to attributes, a Commonality can contain references, which point to other Commonalities. References make it possible to connect model elements that are managed by the consistency preservation process. Model objects are not kept consistent in isolation. Indeed, the goal of consistency preservation is usually to transform a whole graph of model objects. Not only the graph's nodes—the model objects—but also its edges—references between the model objects—need to be addressed.

A reference in a Commonality has a name and a *converting Commonality*. Like an attribute, a reference can contain mapping specifications. Expressions in reference mapping specifications must always return model objects. The semantics of an equality reference mapping specification is that the value of the reference is always the Commonality instance(s) that correspond(s) to the model object(s) that are returned by the invertible expression. There are also reference mapping specifications from and to the Commonality, which have the same execution semantics as they have for attributes.

We have already used a reference in the example used in the introduction of this chapter. In our running example, a reference would also be used for the components in a component repository (listing 5.11). Here, the reference has the Commonality for a Component as converting Commonality. An instance of the repository Commonality will reference all component Commonalities that correspond to UML components that are placed in the "packagedElement" reference of the UML Model that corresponds to the current repository Commonality instance. If, on the other hand, a new component Commonality instance is placed in the "components" reference of a repository Commonality instance, the UML component corresponding to it will be placed in the "packagedElement" reference of the UML Model that corresponds to the repository Commonality instance. The same mechanism applies to the PCM components in the "components__Repository" reference, of course.

Metaclasses can participate in a Commonality multiple times. Sometimes, a reference should only hold those Commonalities that correspond to model objects that have the role of a specific Participation Class in the correspondence. For such cases, a reference mapping

```
concept Components

commonality Repository {
    with UML:Model
    with PCM:Repository

    has components referencing Components:Component {
        = UML:Model.packagedElement
        = PCM:Repository.components__Repository
    }

    ...
}
```

Listing 5.11: Excerpt of a Commonality for a component repository. It uses a reference to preserve consistency of the pointers to the contained components.

specification can be appended with the keyword “`via`” and the name of a Participation Class. Only model objects that correspond to the converting Commonality because of this Participation Class will then be used for the reference.

Unlike for attributes, no type is inferred for references. Instead, the converting Commonality and, thus, the reference’s type, must be stated explicitly. This violates our design goal of “keeping the basic case simple”: If the combination of metaclasses that are the types of the reference’s mapping specifications uniquely identifies a Commonality, the converting Commonality could be inferred. However, if another Commonality was added that also included matching Participation Classes, the language would be forced to raise an error for the reference, as it is now not clear which Commonality should be used. In effect, adding code—like an alternative implementation—could break (possibly a lot of) existing code. This is a situation that should be avoided. Because of that, the Commonalities Language requires developers to always specify the converting Commonality, even if it could be inferred. Nevertheless a reference’s multiplicity *is* inferred. Reference mapping specifications can also be checked, with the same semantics it has for attribute mapping specifications. The error handling of unchecked references is also the same. Additionally, it is a compile-time error if a reference mapping specification returns a type which is neither a subtype nor a supertype of any Participation Class in the converting Commonality.

5.9 Design Decisions

5.9.1 External Domain-Specific Language

The Commonalities Language is implemented as an external domain-specific language, meaning that it defines its syntax and semantics completely independently instead of using a host language (see section 2.3). This decision is deliberate, but not self-evident. For example, Hinkel et al. show how model transformation languages can be realised as internal domain-specific languages without a significant loss of conciseness. The solution has the advantage that existing tooling can be used for the languages, which eases their adoption [Hin+17].

We decided to use an external domain-specific language mainly for two reasons: it allows creating more sophisticated compile-time checks and offers greater syntactical freedom. There are multiple instances where compile-time checks should be introduced in the Commonalities Language. For instance, Participations between Commonalities must never form a circle. Internal domain-specific languages are limited to the compile-time checks that can be realised in their host language. All other errors can only be detected at runtime, which we think is less user-friendly. Regarding syntactical freedom, we think that the Commonalities Language profits from bespoke syntactic features. The syntax for defining Participations, for example, has many different cases and could probably not be realised in any existing host language.

It is often mentioned in favour of internal domain-specific languages that they are cheaper and less time-consuming to build [Hin+17; FP10, pp. 106 f.]. While still true, this argument is extenuated by the advances of language workbenches like Xtext. Generating required artefacts for a given grammar can be automated largely, which makes realising an external domain-specific language significantly easier. We have profited from Xtext in our implementation.

5.9.2 Collecting Mappings at one Point

The development process for consistency preservation rules does not only involve specifying transformations but also understanding and verifying existing ones, as already discussed in chapter 4. This includes comprehending how changes to one model will affect other models. Over time, more and more metamodels will be added to the transformation rules. So the challenge might start even sooner, with identifying which models are influenced by an attribute of the intermediate metamodel. The Commonalities Language therefore collects the mappings of all metamodels to a Commonality in the same Commonality File. Because of that, it is easy to see which metamodels share semantic overlap[1]. It is also easy to verify that the mapping specifications realise a coherent notion of consistency, because they are gathered at the same place.

Yet, this decision comes with a price. It means that one metamodel will be referenced from many different places. We assume it likely that work on transformations will be shared between developers by making developers responsible for a specific metamodel. Metamodels being spread across files will thus make merge conflicts in version control systems more likely. It will also be more complicated to remove a metamodel from Commonality Files, because all files referencing it must be found. The latter process can, however, be completely automated by appropriate refactoring functionality in an integrated development environment. Overall, we think that the advantages of collecting related consistency preservation rules at one place exceed the disadvantages this brings in practice.

[1]Commonalities can also participate in other Commonalities, which makes discovering the metamodel sharing semantic overlap a little bit more involved. However, we expect such hierarchies to be comparably flat. When using an integrated development environment, developers can also easily jump from a Commonality to a participating Commonality.

5.10 Feature Overview

The previous sections have presented the features of the Commonalities Language. We conclude the chapter with an overview of all constructs available in the Commonalities Language. Table table 5.12 lists all constructs in an informal way and references the sections which describe the respective feature.

Table 5.12: An overview of the Commonalities Language's features. Names that can be freely chosen are marked green, identifiers referencing existing elements are marked blue. Brown text in italics is a placeholder for an expression in the denoted expression language (see section 5.4).

Construct	Concrete Syntax

Commonality named `Name` in a Concept named `ConceptName`
see section 5.5
	`concept ConceptName`
	
	`commonality Name {`
	`    ...`
	`}`

—inside a Commonality—	
Participation Classes for the metaclasses `Metaclass1` and `Metaclass2` from the metamodel `Metamodel` see section 5.6	`with Metamodel:(Metaclass1, Metaclass2)`

Optional Participation Class for `Metaclass1` and two Participation Classes for `Metaclass2` (both from the metamodel `Metamodel`), named `ClassName1` and `ClassName2`
see section 5.6.1
	`with Metamodel:(`
	`    Metaclass1?,`
	`    Metaclass2 called "ClassName1",`
	`    Metaclass2 called "ClassName2"`
	`)`

Participation with an enforceable condition see section 5.4.3	`with Metamodel:Metaclass` `  whereat` *<enforceable condition expr.>*
Two Participation Classes for the metaclasses `MC1` and `MC2` from the metamodel `MM`, related by the Participation Relation Operator *<op>* see section 5.6.3	`with MM:(MC1` *<op>* `MC2)`

Attribute called `AttributeName`
see section 5.7
	`has AttributeName {`
	`    ...`
	`}`

Reference called `ReferenceName`, having the Commonality `Commonality` from the Concept `Concept` as referencing Commonality
see section 5.8
	`has ReferenceName`
	`    referencing Concept:Commonality {`
	`    ...`
	`}`

—inside an Attribute—	
Equality attribute mapping specification see section 5.7	`=` *<invertible expression>*

Construct	Concrete Syntax
Attribute mapping specification to the Commonality see section 5.7	`<- <predictive expression>`
Attribute mapping specification from the Commonality, setting the property `prop` on the Participation Class `PC` from the Participation `P` see section 5.7	`-> <predictive expression> -> P:PC.prop`

Checked attribute mapping specifications
see section 5.7.1
	`check = <invertible expression>`
	`check <- <predictive expression>`
	`check -> <invertible expression>`
	`-> P:PC.prop`

—inside a Reference—	
Equality reference mapping specification see section 5.8	`= <invertible expression>`
Reference mapping specification to the Commonality see section 5.8	`<- <predictive expression>`
Reference mapping specification from the Commonality, setting the reference `ref` on the Participation Class `PC` from the Participation `P` see section 5.8	`-> <predictive expression> -> P:PC.ref`

Checked reference mapping specifications
see section 5.8
	`check = <invertible expression>`
	`check <- <predictive expression>`
	`check -> <invertible expression>`
	`-> P:PC.ref`

6 Implementation

We have developed a prototypical implementation of the Commonalities Language for this thesis. It realises the language's core features and allows to preserve consistency of Ecore models in Eclipse using Vitruvius. It contains a compiler that generates an intermediate metamodel and transformations in the Reactions Language out of Commonality Files. The implementation also contains an editor for Eclipse with syntax highlighting and extended functionality like code suggestions. The language is implemented using Xtext.

Our implementation does not cover the whole Commonalities Language as it was described in the previous chapter, but only a subset of integral features. The implementation can be used for consistency preservation of simple cases (see section 7.2). Specifically, the implementation allows to declare Commonalities, concepts, Participation, attributes and references. Participations can only contain one Participation Class. Optional Participations and conditions are not supported. In attribute or reference mapping specifications, the only supported expression is referencing a feature of a Participation Class. All presented directions of mapping specifications are supported, but not checked mapping specifications.

6.1 Platform Requirements

As explained in chapter 5, the Commonalities Language does not prescribe any specific technology, as long as object-oriented metamodels are available. The language specification does, however, imply that certain mechanisms need to be provided to implement it. We make those technical requirements explicit by listing them here and describe how they were realised in the prototypical implementation:

- Machine-readable metamodels. The compiler and the editor (if developed) need to access information about the available metaclasses and their attributes. This is provided by EMF with Ecore metamodels.

- Code-accessible model instances. The model instances the consistency preservation is executed on must be accessible from the language the Commonalities Language has been translated to. In particular, it must be possible to know at compilation time how to read and write model instances' properties at runtime. The presented implementation compiles the Commonalities Language to the Reactions Language, which compiles to Java code, which uses the Java classes EMF generates for Ecore metamodels. These generated Java classes offer methods that allow access to model instance's property by providing the property's name.

- Consistency preservation execution. The Commonalities Language does not define how and when consistency preservation routines are executed. There must be a facility that actually executes the code generated from the Commonalities Language. The facility must also provide the model instances to the preservation rules and handle their results. Our implementation targets Vitruvius, which executes consistency preservation rules after models in the VSUM have been changed.

- Correspondence Model. If incremental preservation of consistency is desired (and this will likely be the case), the platform must provide means to store which tuples of model elements are already consistent with each other (see section 2.7.1). The Vitruvius framework provides the correspondence model that is used by code generated by our implementation.
- Extending expression languages' set of operators through general-purpose languages. As discussed in section 5.4.1, the expression languages used in the Commonalities Language should be extendible, so users can provide custom operators tailored to their domain. Our implementation realises an extension mechanism that allows developers to create new operators in Java.

6.2 Platform-Specific Language Features

We described the Commonalities Language independent of any concrete platform for metamodelling and consistency preservation. There are, however, features we have created for the Commonalities Language that address issues specific to Vitruvius and EMF. We suspect that any implementation would have to solve similar tasks in one way or another.

6.2.1 Vitruvius Domains

A number of technical issues arise when trying to operate on models in a uniform manner. For instance, it is necessary to be able to uniquely identify model elements in order to store correspondences for them. Furthermore, model information eventually needs to be stored in a file system. These and some other technical tasks are dependent on information about the metamodel at hand and the system and user preferences the transformations are being executed in. The Vitruvius framework hides these implementation specifics and offers transformations to solely operate on so-called *Vitruvius Domains*. They represent one or more metamodels that are closely related and provide information which is specific for these metamodels. Thereby, they allow Vitruvius to handle the aforementioned tasks transparent to clients.

In the implementation of the Commonalities Language, metamodels are exclusively referenced through Vitruvius Domains. Metaclass references have the name of a Vitruvius Domain as the first part and the name of a metaclass as the second part. Not only does this solve technical problems, it also has advantages for users of the language. Ecore models are identified using Uniform Resource Identifiers (URIs). The URIs' authority is usually set to a internet domain that is owned by the metamodel's creators. This avoids name clashes between metamodels. EMF has become relatively popular in the Eclipse community, and as a consequence, there exist a lot of Ecore metamodels which are used by various plugins. Developers in Model-Driven Software Development usually have a lot of Ecore metamodels installed in their development environment. Using URIs makes

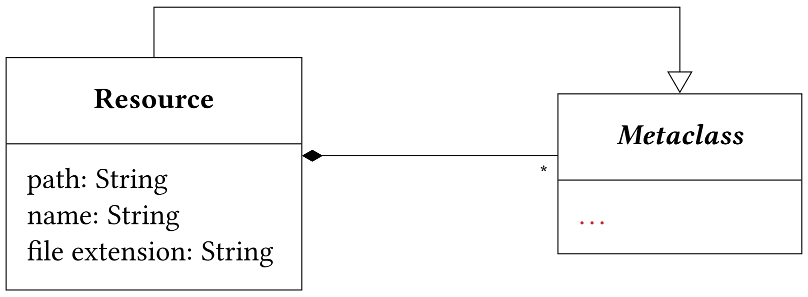


Figure 6.1: The virtual resource metaclass added to every Vitruvius Domain.

this situation technically manageable. In consistency preservation languages, however, it would be impractical to always use a lengthy URI when referencing metamodels. A common solution is thus to first import metamodels and give them an alias. None of this is necessary in the Commonalities Language, because all Vitruvius Domains are always accessible. This simplifies development and has the advantage that the same identifier always refers to the same set of metamodels because there is no mechanism need to give aliases to metamodels, which could be differ between different files.

6.2.2 Resource HandlingTu

Because our implementation acts on Ecore metamodels, every created model object must eventually be placed in a Resource, either by putting in a Resource or in a containment reference. Model objects not contained in any Resource would not be persisted and thus lost. For objects that are created during consistency preservation and not placed in a containment reference, an appropriate Resource needs to be determined. Hence, the implementation of the Commonalities Language must offer means to place model objects in Resources.

Like model objects, Resources can also share semantic overlap with other elements. For example, files containing a public Java class must be named according to the class' name and be placed in a folder hierarchy representing the class' package [Gos+15, p. 189]. Because of that, we chose to make Resources model objects that can also be used in the Commonalities Language. The implementation adds a virtual metaclass called "Resource" to every Vitruvius Domain (figure 6.1). This metaclass can be used like any other metaclass inside the language. If an instance of this resource metaclass is created by the Commonalisies Language at runtime, and a name is set on it, it will persist all model elements in its containment tree. Vitruvius determines the actual folder the Resource will be placed in and makes sure the file is updated with changes to the contained objects. The framework also chooses an appropriate file extension for the Resource if no file extension was set explicitly.

6.3 Expressions

Developing the concrete expression languages to be used by the Commonalities Language (see section 5.4) was not in the scope of this thesis. The current implementation does not support any expression apart from property references and the “in” Participation Relation Operator. However, we show how the implementation could be extended by an easily extendible system of operators to form an expression language. We have already applied this mechanism to the implementation of the “in” Participation Relation Operator.

6.3.1 Embedding Languages in Xtext

The Xtext framework allows to integrate existing language grammars into new languages. This mechanism is called *Grammar Mixins*. Because Xtext grammar specifications also contain mappings to the abstract syntax model, the abstract syntax model of the importing language is automatically extended [ES17]. Both the invertible expressions and the enforceable conditions expressions developed by Kramer were implemented in Xtext [Kra17, p. 178]. They can hence be added to the Commonalities Language through Xtext’s Grammar Mixins feature. Of course, extending the concrete and abstract syntax of the Commonalities Language with the expression languages is not sufficient. The languages need to link references in the language against the actual objects. Furthermore, the code generated for the expression languages must be integrated with the code generated for the Commonalities Language to inherit the expression language’s dynamic semantics. Both expression languages were used by Kramer in the Mappings Language, a language for bidirectional model transformations (more on the Mappings Language follows in section 8.1.1) [Kra17]. The Mappings Language also operates on Ecore models and is translated into the Reactions Language. Because of that, the expression languages already link against the same types the Commonalities Language uses. Furthermore, the code generated by the expression languages is already integrated with the Reactions Language. This makes the process of integrating the expression languages easier. The main work will be to provide an appropriate scope of objects which the expression languages can link against and to adapt the code generated for them, such that it is integrated in the Reactions that are created for the Commonalities Language.

Predictive expressions, on the other hand, can be added to the Commonalities Language using the Xbase grammar developed by Efftinge et al. [Eff+12]. It is written with Xtext and realises an expression language that is very similar to Java. Once again, the grammar could be imported into the Commonalities Language. The expressions must then also be given a scope of objects to link against. Xbase links against Java objects, so the expressions could link against the Java types [Eff+12] that are generated for the Ecore metamodels. There would be, however, no guarantee that the expressions that can be expressed with this implementation are predictable. Instead, developers would need to take care to only specify predictable expressions. Whether predictiveness of Xbase expressions can be checked statically or be enforced by a well-defined scope requires further research.

6.3.2 Extending Operators through Java Classes

In section 5.4.1 we explained why it is important for the Commonalities Language to have expression languages whose operators can be extended by developers to match their specific needs. In our implementation, such operators can be defined in Java (or Xtext, a programming language which compiles to Java). We have already implemented the principle for Participation Relation Operators and provided the "in" operator using it.

Providing a new operator for one of the expression languages in our implementation is done by implementing an interface. The interface offers template methods for the different cases in which the operator needs to be applied. For Participation Relation operators, for example, the interface contains a template method that will be called when a new combination of model objects has been found and must be checked for whether a Commonality instance should be created for it. The interface also contains a template method for the other direction, which can modify newly created model instances to realise the operator's semantics. Operators implemented that way are called at runtime. However, the language implementation has to have certain information about the operator at compile time, like, for example, its name. This compile-time information is provided through Java annotations, which are provided on the annotation. There is, for example, the annotation "`RelationName`", which provides the name by which a Participation Relation Operator can be referenced. Consequently, our implementation of the "in"-operator is annotated with "`@RelationName('in')`".

To use this mechanism for invertible expressions and enforceable condition expressions, those expression languages need to be adapted to call the Java implementations of the operators. In particular, the inversion mechanism of invertible expressions needs to be adapted to call appropriate template methods instead of inverting pre-defined operators directly. Xbase, on the other hand, is already constructed in a way that allows to provide new operations that can be linked against. Even the built-in operators could be overwritten [Eff+12].

6.4 Compilation

The implementation developed for this thesis compiles consistency preservation rules written in the Commonalities Language into the Reactions Language and creates an Ecore metamodel for the intermediate metamodel defined by the Commonalities. We have taken care to make the generated code as readable as possible. We have restraint from calling only library functions from the Reactions Language—which would be easier to implement, but harder to read—and left much code in the reactions. The reason is that we want to enable developers to use the generated code as an artefact on its own. In particular, we want to make it possible to extend the generated code with other reactions.

7 Evaluation

7.1 Validity

As we have shown in chapter 4, the approach of using an intermediate model for multi-model consistency preservation can be applied to any set of models that could also be kept consistent using only binary transformations. Applying it will thus not restrict transformation developers in their possibilities. If there are a lot of—i.e. more than three—models sharing semantic overlap, the approach has remains scalable without compromising the possibility to add models to or remove models from the system. If there are not that many models with semantic overlap yet. The approach can still be applied. It may require more effort, but assures that transformations are prepared for future requirements. We have explained that identifying semantic overlap will always be part of the process of specifying transformations. To use intermediate metamodels, the overlap only needs to be made explicit. Additionally, the Commonalities Language aims to minimise the effort to declare an intermediate model.

The Commonalities Language itself has been designed based on assumptions of how it will be used and what typical tasks will be. It does not claim to be suitable for every use case. Nevertheless, we have provided small examples that suggest that there are use cases in which it will be a helpful tool. We focused on explaining how the designed features allow to create compact definitions that remain understandable and do not duplicate logic. Our implementation translates the Commonalities Language to the Reactions Language. If the Commonalities Language proves to be only useful for a subset of the problems of declaring model transformations, the generated reactions can be combined with reactions written directly in the Reactions Language. Kramer already used this approach for a declarative language that is also translated to the Reactions Language. He argued that the declarative language can be used were appropriate, and the reactions generated from it can be extended with reaction written in the Reactions Language where necessary [Kra17, pp. 90 f.].

7.2 Implementation Functionality

Our implementation of the Commonalities Language allows to keep models consistent if the consistency specification only requires one-to-one correspondences of instances and only equality relations between attributes.

The implementation we have created for this thesis has been tested with unit tests for basic functionality. Our implementation thus shows that it is possible to derive an Ecore metamodel from Commonality Files and to use it as an intermediate model in the consistency preservation process. It also shows that it is possible to realise the basic semantics of the Commonalities Language by generating reactive consistency preservation rules in the Reactions Language.

In the tests, we have executed the code generated for two Commonalities on four metamodels. We asserted that corresponding model instances were created as expected and that changes to values on either of them are correctly represented in the corresponding instances. We also checked that references work as expected; that is, that model objects contained in a property were translated to the correct corresponding model objects and then set on the respective participation object’s property. All tests were executed for both single and multi valued attributes and references. There were some cases that could not be covered, namely removing created model objects and setting attributes that are marked as identifying in the metamodel. This was due to unexpected behaviour in the Vitruvius framework, which has to be fixed first.

7.3 Threats to Validity

The Commonalities Language has not yet been used for consistency preservation of models in a realistic scenario. Because of that, there exists no verification that the assumptions we made about the language’s usage match reality. While we gave arguments why the language’s features solve problems in a desirable manner, these problems could not be the ones that are important in practice. Models that are used in practice, like UML, PCM or Jamopp’s abstract syntax tree cover many use cases and therefore consist of many metaclasses, attributes and reference. Only a case study can provide certainty on whether the Commonalities Language is suitable to create understandable transformations for such metamodels.

In section 4.2 we explained that intermediate models do not need to be free of semantic overlap. We also showed how a fully connected graph of bidirectional transformations can be converted to use intermediate models without having direct transformations between the existing models. However, we do not have a prove that such a conversion would yield an intermediate model that is free of semantic overlap. We assume that it will always be feasible to find intermediate models with only a reasonable amount of semantic overlap. This assumption is supported by the fact that Atkinson et al. have already created semantic-overlap-free SUMs for real-life applications in software engineering [Atk+13] and Malavolta et al. created a central metamodel to preserve consistency of architecture models [Mal+10]. Nevertheless, it is possible that, if our language is applied to a large amount of metamodels, it will become impossible or infeasible to define useful intermediate models and preserve their consistency. The effort to keep the intermediate models consistent might, for example, exceed advantages they bring. In that case, our approach and language would not be applicable for systems at such scale.

8 Related Work

The fundamental problem of managing consistency of models has been extensively studied in various publications, albeit using differing terminology [MJC17]. Macedo et al. [MJC17] recently published an overview of the many works on the topic. This chapter will hence not look at consistency preservation and model transformations in general—with the exception of the Mappings Language, which this thesis lent several ideas from. Instead, we fill focus on publications that explicitly address multidirectional consistency preservation of more than two models.

8.1 Transformation Languages

8.1.1 The Mappings Language

Additionally to the Reactions Language, the Vitruvius framework contains another language to specify consistency preservation rules, the Mappings Language. It is a problem-oriented language for bidirectional transformations and was also presented by Kramer [Kra17, pp. 137 ff.]. The Commonalities Language proposed in this thesis was heavily inspired by the Mappings Language. For instance, both the idea and the implementation of invertible expressions and enforceable condition expressions were developed for the Mappings Language (see section 5.4). In the Mappings Language, consistency preservation rules are declared between two metamodels. A rule starts by listing the metaclasses that will be used for each metamodel and assigning names to them. Next, developers can declare one-sided conditions for the metaclasses using enforceable condition expressions, which may only access the metaclasses of one metamodel. Finally, developers can provide invertible expressions to transform properties of one metamodel into the other and back. For both the conditions for each side and the transformations, developers can also provide unidirectional expressions if the bidirectional expressions are not sufficiently expressive [Kra17, pp. 137 f.].

Although the Commonalities Language uses solutions from the Mappings Language, it also introduces new ideas; the most obvious being that consistency can be specified for more than two metamodels at once. The Commonalities Language also introduces an explicit model of the consistency specification—the intermediate model—while this information is only contained implicitly in transformations written in the Mappings Language. We also think that it is an advantage that the Commonalities Languages allows to specify transformations in a more structured way: In the Mapping Language, all bidirectional transformations for a given pair of combinations of metaclasses are specified in one list. In the Commonalities Language, on the other hand, the transformations are automatically grouped by the attribute or reference of the Commonality they apply to.

Finally, the Commonalities Language discourages the excessive use of expressions more than the Mapping Language does. While conditions and transformations are specified in code blocks in the Mapping Language, the Commonalities Language does not allow code blocks and encourages developers put logic in a new operator rather than specifying it directly in the transformation file.

8.1.2 QVT-R

The the Object Management Group (OMG) defines a metamodelling infrastructure, the Meta Object Facility (MOF) [Obj16b]. To complement it, the Object Management Group (OMG) specifies QVT, a set of transformation languages for MOF-metamodels [Obj16a]. QVT consists of a rather technical "core" language, which can be compared to Java bytecode, and two developer-targeting languages: the declarative, bidirectional "relations" language and the imperative, unidirectional "operational" language [Obj16a, p. 9 f.]. Both languages can take an unlimited amount of model objects as input or output parameters [Obj16a, pp. 13, 91] and could hence be used for multi-model consistency preservation. We will focus on the declarative language, QVT-R, in this section, as we only cover multidirectional languages in this chapter.

Fundamentally, QVT-R allows declaring relations between models. These relations specify when the targeted model objects are to be considered consistent. The relations have to modes: they can be checked and enforced. The check mode checks whether model instances are consistent according to the consistency specification the QVT-R relations express. In enforce mode, on the other hand, a transformation is executed in one direction to restore this consistency.

Macedo et al. have studied the implications of using QVT-R for multiary consistency preservation [MCP14]. They found that the language as it is specified is not sufficiently expressive. They give an example of a simple consistency specification that cannot be expressed in QVT-R. They show that the essential problem, which prevents the example from being specified, is that QVT-R all-quantifies all input models for conditions. Consequently, Macedo et al. propose an extension to QVT-R and show how it would allow to specify relations for their example and similar situation. They conclude that more research and case studies are required to understand QVT-R's capabilities for multiary consistency preservation [MCP14]. This statement certainly also applies to the Commonalities Language.

8.1.3 Triple Graph Grammars

Models can be interpreted as a graph, with the model objects as nodes and the references between them as edges. Because of that, Triple Graph Grammars, introduced by Schürr [Sch94], can also be used for bidirectional model transformations. Triple Graph Grammars are graph-based grammars that can be used to produce two graphs and an according correspondence graph. The correspondence graph has a function that is very similar to the correspondences used in model transformations (see section 2.7.1): it relates the nodes of the two graphs. The graphs are constructed in productions, which define a pattern to match and modification to apply if the pattern matches. Productions are monotonic; that is, they cannot remove existing edges or nodes. There are, however, also approaches that remove this limitation [Lau+12].

Additionally to producing graphs, Triple Graph Grammars can also be used to generate one of the two graphs if the other is given, which makes them suitable to be used as model transformation definitions. It has even been shown that QVT core (the bytecode-like transformation language, see section 8.1.2) can be implemented using Triple Graph Grammars [GK10]. A main difference of Triple Graph Grammars to other transformation languages is that they are usually defined graphically. If tools offer a textual syntax at all, it is a fallback solution [Hil+13].

Trollmann and Albayrak have extended Triple Graph Grammars so they can also be used to preserve consistency of multiple models [TA16]. They show how their approach preserves the strong formal understanding of classical Triple Graph Grammars. In particular, they are able to prove its correctness in terms of the existing definitions for Triple Graph Grammars.

8.2 Orthographic Software Modelling

We already introduced Orthographic Software Modelling, an approach presented by Atkinson et al. [ASB10], in section 2.6.1. It uses a single underlying model (SUM) that holds all information about a software system and is only edited through projective views. Because the SUM is free of semantic overlap per construction [ASB10; ATM15], Orthographic Software Modelling has no need for consistency preservation. Nevertheless, the SUM seems to be similar to the intermediate models we propose, which is why we will explore some of the differences. First, intermediate models do not contain all information about the system, but only those parts that have semantic overlap in the models and are addressed by the consistency preservation process. Any other informations remains exclusively in the existing models. Furthermore, the data in the intermediate models is never authoritative. While a SUM is a single source of truth, intermediate models only contain deliberately introduced copies of information. Second, even though it seems to be desirable not to have much semantic overlap in intermediate models, our approach allows it. We even assume that sometimes, semantic overlap can be helpful to separate concerns. A SUM, on the other hand, must not contain any semantic overlap. Finally, a virtual single underlying model (VSUM) that is kept consistent using intermediate models has the advantages described by Kramer et al., namely that existing metamodels and editors can be re-used [KBL13].

8.3 Preserving Consistency through a Central Model

Malavolta et al. use a central metamodel for consistency preservation of architecture description languages; i.e. modelling languages for the domain of software architecture [Mal+10]. They define a central metamodel, called A_0, for software architecture and build a framework, Dually, around it to integrate existing metamodels. The approach does not prescribe any specific technology for the transformations, although it contains a generic mechanism to preserve consistency. The A_0 metamodel does not contain all semantic overlap of the metamodels that are kept consistent. It only models a specific kernel of architecture description languages. This kernel is frozen—meaning that it is not meant to be changed in the future—but extensible. Malavolta et al. argue that because of that, the approach remains scalable, as the A_0 does not have to be modified for every added metamodel. On the other hand, the small kernel means that important information could be lost because it is not represented in the kernel. In those cases, the kernel should be extended, but not modified, which allows to retain backwards-compatibility [Mal+10].

Di Ruscio et al. add to this idea by describing how a kernel metamodel like A_0 can be extended in a systematic process. They define four extension operators and and show that when using their process, extensions can be combined and also be applied to different kernels [Di +12]. These considerations might also enrich our approach of using intermediate models. While there are notable differences between a kernel metamodel and an intermediate metamodel, the initial situation is similar in both approaches: Multiple metamodels need to be kept consistent, but it is not known up front which metamodels might be added to the consistency preservation process in the future. Being able to evolve intermediate metamodels using reusable extensions and without breaking backwards compatibility is also desirable for our approach.

8.4 Theoretical Work

The theoretical properties of multiary consistency preservation were recently discussed by Stevens [Ste17]. She takes a general look at the expressiveness of consistency preservation rules and gives an example for a consistency specification for which consistency preservation is not possible if only binary transformations are used. She then shows that consistency preservation is related to “constraint networks” known from the constraint satisfactory problem. The constraint network problem searches for an assignment to a number of variables that fulfils all of a given set of constraints. For this problem, it is known that any network of multiary constraints can be reduced to a network of binary constraints if new variables are introduced [RPD90]. Stevens hence suggests that adding new models allows binary transformations to also preserve consistency for arbitrary multiary consistency specifications. It is worth studying whether this also applies to the intermediate metamodels as they can be defined with the Commonalities Language.

9 Future Work

The results presented in this thesis add to the overall goal of being able to preserve consistency of multiple models. They are, however, far from sufficient to achieve the goal. This chapter shows the areas where we already identified shortcomings and outlines steps that could be taken to overcome them.

9.1 Implementation

To improve our implementation of the Commonalities Language and make it usable in practice, the foremost goal should be to bring it on par with the feature set described in this thesis. We regard the lack of invertible expressions and enforceable condition expressions as the most pressing issue. Nevertheless, there are other features that we could not approach but developers would profit from. There could be, for example, more sophisticated compile-time checks. For instance, it would be worth introducing checks that Participations between Commonalities never form a circle, that there is no Participation Class which can never be in a Resource (because it is never put in a Resource or containment relation), that used properties are not marked as deprecated, and so forth. The editors should also support more advanced features which developers have come to expect form other languages. The editors should, for example, propose valid identifiers wherever possible and allow to jump from a property or metaclass reference to the definition of that property or metaclass.

9.2 Evaluation in Practical Use Cases

Once an acceptable implementation of the Commonalities Language is available, it could be applied to use cases that have already been identified by other publications. There have been, for example, a doctoral thesis by Langhammer [Lan17] and two Bachelor's theses by Chen [Che17] and Klatte [Kla17] about consistency preservation with Vitruvius for UML, PCM and Java models. Each thesis focuses on one of the three combinations of two of the metamodels. The rules were all implemented in the Reactions Language. Realising them with the Commonalities Language could be beneficial for two reasons: First, implementing rules created by others assures that the Commonalities Language can solve problems in a meaningful way. Second, the implementation in the Commonalities Language could be compared to those in the Reactions Language. It could then be assessed whether the Commonalities Language actually outperforms the other approach in terms of development effort. Further properties, like runtime, could be compared, too.

9.3 Language Evolution

What functionality is needed additionally in the Commonalities Language will likely reveal itself when the language is used in practice. Nevertheless, we already see one feature that might be worth adding: Commonality inheritance. As Commonalities translate directly to metaclasses, it seems intuitive to also introduce inheritance, a key feature of object orientation, for them. Defining inheritance for Commonalities is, however, not straightforward, because a meaningful strategy regarding how transformations are inherited and overridden needs to be found. Inheritance could help to further reduce duplicated transformation logic. For example, models often have a notion of a "named element" [Obj15, pp. 47 f.][Reu+11, p. 99] from which most metaclasses inherit. In the Commonalities Language's current state, practically every Commonality would need to define the attribute "name" and map it to the Participation Classes' name. Inheritance could help here by allowing developers to define the transformation once for all "named elements" of a metamodel and inherit from this definition. However, any solution should also allow overriding inherited behaviour.

9.4 Theoretical Evaluation

Once the feature set of the Commonalities Language is sufficiently mature and has proven itself in case studies, it would be interesting to explore the language's theoretical capabilities. We see two areas that could be studied in particular: First, it has already been discovered that binary consistency rules, like they are for example created by bidirectional transformations, are not sufficient to preserve every desirable consistency specification (see section 8.4). However, Stevens already showed that this shortcoming can be overcome if new models are introduced [Ste17]. Thus, the question arises if the intermediate models as they can be defined through the Commonalities Language also allow to preserve consistency for any multiary consistency specification. Studying this question might also gain new insight into how intermediate metamodels should be designed to require no or only a few changes when new requirements arise in the future.

Second, it could be examined whether the Commonalities Language can be used to transform arbitrary change sequences. For example, Kramer showed for the Reactions Language that reactions can be triggered to for any possible change and that reactions can execute any computable transformation [Kra17, p. 211 f.]. The Commonalities Language can also execute Turing complete programs through the extensible operator mechanism. However, operators should never have to modify the models directly, as we explained in section 5.4. So for an evaluation of the language's completeness, it should preferably be studied whether arbitrary changes to models can be transformed to arbitrary changes to other models under the restriction that operators never modify the models that are kept consistent directly and that they also conform to the specific restrictions that were introduced for them.

10 Conclusion

This thesis presented an approach and an accompanying, new programming language to preserve consistency of multiple models. We argued that it is desirable to preserve consistency of multiple models using binary transformations. We then analysed different approaches to specify multi-model consistency using binary transformations. Subsequently, we explained how consistency of multiple models can be preserved using an intermediate metamodel. It makes the semantic overlap of metamodels explicit. All transformations go "through" it, such that there are no direct transformations between the models that are kept consistent. We analysed which advantages this solution has. We came to the conclusion that, unlike the other solutions, using an intermediate metamodel allows to add and remove models without needing to modify existing transformations, while still requiring only a reasonable amount of development effort.

To support the approach, we thereupon presented a novel, domain-specific, declarative programming language, the Commonalities Language. It can be used to declare an intermediate metamodel together with the transformations from existing metamodels to the intermediate metamodel. Developers can use the language to create the metaclasses of the intermediate metamodel—so-called Commonalities—and their properties. The mapping of existing metaclasses to these constructs is provided directly with the declaration of each construct. We presented the language's syntax and semantics and explained how we expect it to be used. We also showed how we aimed to design the language in a way that makes transformation rules written in it understandable and free of logic duplication.

Finally, we have developed a prototypical implementation of the Commonalities Language, which supports an essential subset of the language's features. It uses the Vitruvius framework and transforms the Commonalties Language into a pre-existing, reactive language for model consistency, the Reactions Language.

All in all, we have shown how an approach for multi-model consistency with desirable properties can look like. We have presented a programming language that is designed to support developers in realising this approach and create easily understandable transformation rules. We implemented the language's essential features.

The contributions of this thesis are one step towards an all-encompassing solution for consistency preservation of multiple models. Using an intermediate metamodel together with bidirectional transformations promises to be a scalable solution. It is supported by the fact that similar strategies have already been successfully applied to real-world scenarios [Mal+10; Atk+13]. The Commonalities Language already incorporates numerous features to handle specifics of consistency preservation problems that we deem typical. It will, however, likely need further research and development to become a tool that equips developers for all requirements real-life applications pose. Our implementation of the language is only rudimentary. It can be seen as a proof of concept but will require further development before it will be useful in practice.

Glossary

compiler
Software that translates code written in one programming language into another programming language.

consistency 6
A state of a set of models in which the set does not contain contradictory information.

dependency injection
A design pattern, stating that classes do not obtain or create their dependencies by themselves, but rather have them provided by a central facility.

domain
“A bounded field of interest or knowledge” [VS06, p. 56].

domain-specific language 5
A (programming) language that is tailored to be used for a particular domain.

Eclipse
An open-source integrated development environment that is very extensible because of its modular architecture.

Eclipse Modelling Framework (EMF) 7
An infrastructure to define metamodels and generate code for them, including graphical editors.

Ecore 8
The meta-metamodel used in EMF.

Java
A statically typed, object-oriented programming language. In this thesis, Java is most of the time manipulated through a model, like it is possible with Jamopp.

Java Model Parser and Printer (Jamopp) 9
An Ecore metamodel of the Java programming language, allowing to treat Java source code as a model.

metaclass 4
Part of an object-oriented metamodel, defines which properties model objects can have.

metamodel 4
A model describing the permissible values of a model.

Model-Driven Software Development 3
A paradigm putting models at the centre of the software development process.

Palladio Component Model (PCM) 9
A metamodel for component-based software architecture. By also capturing software's abstract behaviour and context of use, it allows to run simulations and carry out experiments on models conforming to it.

Reactions Language 12
A domain-specific, reactive language that is used to specify transformations for consistency preservation in Vitruvius.

semantic overlap 6
When multiple models—explicitly or implicitly—contain the same piece of information about their common original.

serialisation
The process of reversibly converting objects in memory into a stream of bytes, usually to save them to disk or to transmit them to another machine.

single underlying model (SUM) 10
A central model, without semantic overlap, holding all information about the system it describes.

Turing completeness
Property of a programming language or system meaning that any computable function can be computed with it.

Unified Modelling Language (UML) 8
A metamodel for different aspects of software development, like use cases, architecture, object-oriented design or business processes.

Uniform Resource Identifier (URI)
A syntax to identify arbitrary resources, standardised by the Internet Engineering Task Force (IETF) [**RFC3986**]. Addresses used in the world wide web, like "`https://sdqweb.ipd.kit.edu/wiki/Vitruvius`", are an example.

version control system
Software that manages versions of files, allowing multiple users to edit the same set of files. Usually contains means to combine ("merge") changes. Popular version control systems are git and SVN.

virtual single underlying model (VSUM) 10
A single underlying model tha

Vitruvius 10
A tool for Model-Driven Software Development, offering a framework to specify consistency preservation for models and to rapidly create on-the-fly views on them. Uses a virtual single underlying model.

Xtext 9
A framework to create domain-specific languages, generating various artefacts like the lexer, the parser, an abstract syntax model, editors, and more for it.

Bibliography

[AK03] Colin Atkinson and Thomas Kühne. 'Model-Driven Development: A Metamodeling Foundation'. In: *IEEE Software* 20.5 (2003), pp. 36–41. ISSN: 0740-7459. DOI: `10.1109/MS.2003.1231149`.

[ASB10] Colin Atkinson, Dietmar Stoll and Philipp Bostan. 'Orthographic Software Modeling: A Practical Approach to View-Based Development'. In: *Evaluation of Novel Approaches to Software Engineering*. Ed. by Leszek A. Maciaszek, César González-Pérez and Stefan Jablonski. Vol. 69. Communications in Computer and Information Science. Berlin/Heidelberg: Springer, 2010, pp. 206–219. ISBN: 978-3-642-14819-4.

[Atk+13] Colin Atkinson et al. 'A Prototype Implementation of an Orthographic Software Modeling Environment'. In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. VAO '13. New York, NY, USA: ACM, 2013, 3:1–3:10. ISBN: 978-1-4503-2070-2. DOI: `10.1145/2489861.2489862`.

[ATM15] C. Atkinson, C. Tunjic and T. Möller. 'Fundamental Realization Strategies for Multi-View Specification Environments'. In: *2015 IEEE 19th International Enterprise Distributed Object Computing Conference*. 2015 IEEE 19th International Enterprise Distributed Object Computing Conference. Sept. 2015, pp. 40–49. DOI: `10.1109/EDOC.2015.17`.

[BKR09] Steffen Becker, Heiko Koziolek and Ralf Reussner. 'The Palladio Component Model for Model-Driven Performance Prediction'. In: *Journal of Systems and Software*. Special Issue: Software Performance - Modeling and Analysis 82.1 (1st Jan. 2009), pp. 3–22. ISSN: 0164-1212. DOI: `10.1016/j.jss.2008.03.066`.

[BMK16] Erik Burger, Victoria Mittelbach and Anne Koziolek. 'View-Based and Model-Driven Outage Management for the Smart Grid'. In: *Proceedings of the 11th International Workshop on Models@run.Time*. 11th International Workshop on Models@run.Time Co-Located with 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016). Saint Malo, France, 4th Oct. 2016. URL: `http://ceur-ws.org/Vol-1742/MRT16_paper_1.pdf` (visited on 12th Oct. 2017).

[Bur+14] Erik Burger et al. 'View-Based Model-Driven Software Development with ModelJoin'. In: *Software & Systems Modeling* 15.2 (2014). Ed. by Robert France and Bernhard Rumpe, pp. 472–496. ISSN: 1619-1374. DOI: `10.1007/s10270-014-0413-5`.

[Bur14] Erik Burger. ‘Flexible Views for View-Based Model-Driven Development’. Karlsruhe, Germany: Karlsruhe Institute of Technology, July 2014. DOI: `10.5445/KSP/1000043437`.

[Che+15] James Cheney et al. ‘Towards a Principle of Least Surprise for Bidirectional Transformations’. In: *Proceedings of the Fourth International Workshop on Bidirectional Transformations (Bx 2015)*. Ed. by A. Cunha and E. Kindler. L’Aquila, Italy, 24th July 2015, pp. 66–80. URL: `http://ceur-ws.org/Vol-1396/p66-cheney.pdf`.

[Che+17] James Cheney et al. ‘On Principles of Least Change and Least Surprise for Bidirectional Transformations’. In: *The Journal of Object Technology* 16.1 (2017), 3:1–31. ISSN: 1660-1769. DOI: `10.5381/jot.2017.16.1.a3`.

[Che17] Fei Chen. ‘Änderungsgetriebene Konsistenzhaltung zwischen UML-Klassenmodellen und Java-Code’. Bachelor’s Thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT), Faculty of Informatics, 24th May 2017.

[Dam+16] Hoa Khanh Dam et al. ‘Consistent Merging of Model Versions’. In: *Journal of Systems and Software* 112 (Supplement C 1st Feb. 2016), pp. 137–155. ISSN: 0164-1212. DOI: `10.1016/j.jss.2015.06.044`.

[Dev16] DevBoost. *JaMoPP*. 2nd Feb. 2016. URL: `http://www.jamopp.org/index.php/JaMoPP` (visited on 8th Oct. 2017).

[Di +12] Davide Di Ruscio et al. ‘Model-Driven Techniques to Enhance Architectural Languages Interoperability’. In: *Fundamental Approaches to Software Engineering*. International Conference on Fundamental Approaches to Software Engineering. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 24th Mar. 2012, pp. 26–42. ISBN: 978-3-642-28871-5 978-3-642-28872-2. DOI: `10.1007/978-3-642-28872-2_2`.

[Eff+12] Sven Efftinge et al. ‘Xbase: Implementing Domain-Specific Languages for Java’. In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*. GPCE ’12. New York, NY, USA: ACM, 2012, pp. 112–121. ISBN: 978-1-4503-1129-8. DOI: `10.1145/2371401.2371419`.

[ES17] Sven Efftinge and Miro Spönemann. *Xtext Reference Documentation*. 7th Sept. 2017. URL: `https://eclipse.org/Xtext/documentation` (visited on 20th Sept. 2017).

[EV06] Sven Efftinge and Markus Völter. ‘oAW xText: A Framework for Textual DSLs’. In: *Eclipsecon Summit Europe 2006*. Vol. 32. Nov. 2006, p. 118. URL: `http://eclipsecon.org/summiteurope2006/presentations/ESE2006-EclipseModelingSymposium12_xTextFramework.pdf`.

[Fin+92] Anthony Finkelstein et al. ‘Viewpoints: A Framework for Integrating Multiple Perspectives in System Development’. In: *International Journal of Software Engineering and Knowledge Engineering* 2.1 (Mar. 1992), pp. 31–58. ISSN: 0218-1940. DOI: `10.1142/S0218194092000038`.

[Fos+07] J. Nathan Foster et al. 'Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem'. In: *ACM Trans. Program. Lang. Syst.* 29.3 (May 2007). ISSN: 0164-0925. DOI: `10.1145/1232420.1232424`.

[FP10] Martin Fowler and Rebecca Parsons. *Domain Specific Languages.* 1st. Addison-Wesley, Reading, MA, USA, 2010. ISBN: 0-321-71294-3 978-0-321-71294-3.

[FR07] Robert France and Bernhard Rumpe. 'Model-Driven Development of Complex Software: A Research Roadmap'. In: *2007 Future of Software Engineering.* FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 37–54. ISBN: 978-0-7695-2829-8. DOI: `10.1109/FOSE.2007.14`.

[GK10] Joel Greenyer and Ekkart Kindler. 'Comparing Relational Model Transformation Technologies: Implementing Query/View/Transformation with Triple Graph Grammars'. In: *Software & Systems Modeling* 9.1 (1st Jan. 2010), p. 21. ISSN: 1619-1366, 1619-1374. DOI: `10.1007/s10270-009-0121-8`.

[Gos+15] James Gosling et al. *The Java® Language Specification.* Java SE 8 Edition. 13th Feb. 2015. URL: `https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf` (visited on 11th Oct. 2017).

[Hei+09] Florian Heidenreich et al. 'Closing the Gap between Modelling and Java'. In: *Software Language Engineering.* International Conference on Software Language Engineering. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 5th Oct. 2009, pp. 374–383. ISBN: 978-3-642-12106-7 978-3-642-12107-4. DOI: `10.1007/978-3-642-12107-4_25`.

[Hil+13] Stephan Hildebrandt et al. 'A Survey of Triple Graph Grammar Tools'. In: (1st Jan. 2013). DOI: `10.14279/tuj.eceasst.57.865`.

[Hin+17] Georg Hinkel et al. 'Using Internal Domain-Specific Languages to Inherit Tool Support and Modularity for Model Transformations'. In: *Software & Systems Modeling* (30th Jan. 2017), pp. 1–27. ISSN: 1619-1366, 1619-1374. DOI: `10.1007/s10270-017-0578-9`.

[ISO11] ISO/IEC/IEEE 42010. *Systems and Software Engineering — Architecture Description.* Geneva, Switzerland: International Organization for Standardization, 1st Dec. 2011.

[ISO96] ISO/IEC 14977. *Information Technology — Syntactic Metalanguage — Extended BNF.* Dec. 1996.

[Kap+09] Tim Kapteijns et al. 'A Comparative Case Study of Model Driven Development vs Traditional Development: The Tortoise or the Hare'. In: 4th European Workshop on "From Code Centric to Model Centric Software Engineering: Practices, Implications and ROI". Vol. WP09-07. CTIT Workshop Proceedings Series. Enschede, Netherlands, 26th Apr. 2009, pp. 22–33.

[Kar+09] Gabor Karsai et al. 'Design Guidelines for Domain Specific Languages'. In: *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM' 09).* Vol. 1409.2378. eprint. Orlando, Florida, USA: Helsinki School of Economics, 2009. ISBN: 978-952-488-372-6. URL: `http://arxiv.org/abs/1409.2378`.

[KBL13] 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 '13. Montpellier, France: ACM, 2013, 5:1–5:6. ISBN: 978-1-4503-2070-2. DOI: `10.1145/2489861.2489864`.

[Ker88] Brian W. Kernighan. *The C Programming Language.* Ed. by Dennis M. Ritchie. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0-13-110370-9.

[Kla17] Matthias Klatte. 'Konsistenzhaltung zwischen UML- und PCM-Komponentenmodellen'. Bachelor's Thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT), Faculty of Informatics, 13th June 2017.

[KR16] 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. Eindhoven, The Netherlands: CEUR-WS.org, 2016, pp. 61–76. URL: `http://nbn-resolving.org/urn:nbn:de:0074-1571-4`.

[Kra17] Max E. Kramer. 'Specification Languages for Preserving Consistency between Models of Different Languages'. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT), 2017. DOI: `10.5445/IR/1000069284`.

[Lan17] Michael Langhammer. 'Automated Coevolution of Source Code and Software Architecture Models'. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT), Feb. 2017. URL: `https://publikationen.bibliothek.kit.edu/1000069366`.

[Lau+12] Marius Lauder et al. 'Efficient Model Synchronization with Precedence Triple Graph Grammars'. In: *Graph Transformations.* International Conference on Graph Transformation. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 24th Sept. 2012, pp. 401–415. ISBN: 978-3-642-33653-9 978-3-642-33654-6. DOI: `10.1007/978-3-642-33654-6_27`.

[LK15] 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. URL: `http://fg-sre.gi.de/fileadmin/gliederungen/fg-sre/wsre2015/WSRE2015-Proceeedings-preliminary.pdf#page=40`.

[Mal+10] I. Malavolta et al. 'Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies'. In: *IEEE Transactions on Software Engineering* 36.1 (Jan. 2010), pp. 119–140. ISSN: 0098-5589. DOI: `10.1109/TSE.2009.51`.

[McG16] Tyler McGinnis. *Imperative vs Declarative Programming.* 13th July 2016. URL: `https://tylermcginnis.com/imperative-vs-declarative-programming/` (visited on 8th Oct. 2017).

[MCM12] Yulkeidi Martínez, Cristina Cachero and Santiago Meliá. 'Evaluating the Impact of a Model-Driven Web Engineering Approach on the Productivity and the Satisfaction of Software Development Teams'. In: *Web Engineering*. International Conference on Web Engineering. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 23rd July 2012, pp. 223–237. ISBN: 978-3-642-31752-1 978-3-642-31753-8. DOI: `10.1007/978-3-642-31753-8_17`.

[MCP14] Nuno Macedo, Alcino Cunha and Hugo Pacheco. 'Towards a Framework for Multi-Directional Model Transformations'. In: *3rd International Workshop on Bidirectional Transformations*. Vol. 1133. CEUR-WS.org. Athens, Greece, Mar. 2014. URL: `http://haslab.uminho.pt/nmacedo/files/bx14mx.pdf` (visited on 5th Oct. 2017).

[MHS05] Marjan Mernik, Jan Heering and Anthony M. Sloane. 'When and How to Develop Domain-Specific Languages'. In: *ACM Computing Surveys* 37.4 (Dec. 2005), pp. 316–344. ISSN: 0360-0300. DOI: `10.1145/1118890.1118892`.

[MJC17] Nuno Macedo, Tiago Jorge and Alcino Cunha. 'A Feature-Based Classification of Model Repair Approaches'. In: *IEEE Transactions on Software Engineering* 43.7 (July 2017), pp. 615–640. ISSN: 0098-5589. DOI: `10.1109/TSE.2016.2620145`.

[NER01] Bashar Nuseibeh, Steve Easterbrook and Alessandra Russo. 'Making Inconsistency Respectable in Software Development'. In: *Journal of Systems and Software* 58.2 (1st Sept. 2001), pp. 171–180. ISSN: 0164-1212. DOI: `10.1016/S0164-1212(01)00036-X`.

[Obj15] Object Management Group (OMG). *OMG Unified Modeling Language™ (OMG UML)*. version 2.5. Mar. 2015. URL: `http://www.omg.org/spec/UML/2.5` (visited on 15th May 2017).

[Obj16a] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0—Query/View/Transformation Specification*. version 1.3. June 2016. URL: `http://www.omg.org/spec/QVT/1.3` (visited on 15th May 2017).

[Obj16b] Object Management Group (OMG). *OMG Meta Object Facility (MOF) Core Specification*. version 2.5.1. Nov. 2016. URL: `http://www.omg.org/spec/MOF/2.5.1` (visited on 15th May 2017).

[Pet13] Marian Petre. 'UML in Practice'. In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 722–731. ISBN: 978-1-4673-3076-3. URL: `http://dl.acm.org/citation.cfm?id=2486788.2486883` (visited on 17th Oct. 2017).

[Reu+11] Ralf Reussner et al. 'The Palladio Component Model'. In: *Karlsruhe Reports in Informatics* 14 (2011). ISSN: 2190-4782. DOI: `10.5445/IR/1000022503`.

[RPD90] Francesca Rossi, Charles Petrie and Vasant Dhar. 'On the Equivalence of Constraint Satisfaction Problems'. In: *In Proceedings of the 9th European Conference on Artificial Intelligence*. 1990, pp. 550–556.

[Sch06] Douglas C. Schmidt. ‘Guest Editor’s Introduction: Model-Driven Engineering’. In: *Computer* 39.2 (Feb. 2006), pp. 25–31. ISSN: 0018-9162. DOI: 10.1109/MC.2006.58.

[Sch94] Andy Schürr. ‘Specification of Graph Translators with Triple Graph Grammars’. In: *Graph-Theoretic Concepts in Computer Science.* International Workshop on Graph-Theoretic Concepts in Computer Science. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 16th June 1994, pp. 151–163. ISBN: 978-3-540-59071-2 978-3-540-49183-5. DOI: 10.1007/3-540-59071-4_45.

[Sel03] Bran Selic. ‘The Pragmatics of Model-Driven Development’. In: *IEEE Software* 20.5 (Sept.-Oct. 2003), pp. 19–25. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231146.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie.* Wien: Springer Verlag, 1973. ISBN: 3-211-81106-0.

[Ste07] Perdita Stevens. ‘A Landscape of Bidirectional Model Transformations’. In: *Generative and Transformational Techniques in Software Engineering II.* International Summer School on Generative and Transformational Techniques in Software Engineering. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2nd July 2007, pp. 408–424. ISBN: 978-3-540-88642-6 978-3-540-88643-3. DOI: 10.1007/978-3-540-88643-3_10.

[Ste17] Perdita Stevens. ‘Bidirectional Transformations in the Large’. In: ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems. 12th June 2017. URL: http://www.research.ed.ac.uk/portal/en/publications/bidirectional-transformations-in-the-large(39e0a66b-ffc8-4359-ae19-79789483875e).html (visited on 28th Sept. 2017).

[TA16] Frank Trollmann and Sahin Albayrak. ‘Extending Model Synchronization Results from Triple Graph Grammars to Multiple Models’. In: *Theory and Practice of Model Transformations.* International Conference on Theory and Practice of Model Transformations. Springer, Cham, 4th July 2016, pp. 91–106. DOI: 10.1007/978-3-319-42064-6_7.

[VS06] Markus Völter and Thomas Stahl. *Model-Driven Software Development – Technology, Engineering, Management.* Chichester, England: John Wiley & Sons, Ltd, 2006. ISBN: 978-0-470-02570-3.

[WHR14] John Whittle, John Hutchinson and Mark Rouncefield. ‘The State of Practice in Model-Driven Engineering’. In: *IEEE Software* 31.3 (May 2014), pp. 79–85. ISSN: 0740-7459. DOI: 10.1109/MS.2013.65.