View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by KITopen

Methodology for Evaluating a Domain-Specific Model Transformation Language

Master'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:	Prof. Dr.-Ing. Anne Koziolek
Advisor:	M. Sc. Heiko Klare
Second advisor:	Dr.-Ing. Erik Burger

19th October 2020 – 18th April 2021

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

This document is licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0): https://creativecommons.org/licenses/by/4.0/deed.en

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, 18th April 2021

. . . .
(Joshua Gleitze)

Abstract

When using multiple models to describe a system, the different descriptions can get out of synchronization and, hence, contradict themselves. Model transformations are a means to ensure that models remain consistent even while they are being edited by multiple parties. Although strong results have been achieved for transforming changes between two models, the challenge of creating multidirectional model transformations is still insufficiently addressed. The Commonalities Language is a declarative, domain-specific programming language allowing to develop multidirectional model transformations by combining bidirectional transformation specifications. Since it has not yet been validated empirically, it is an open question whether the language is suitable for developing realistic model transformations and whether it brings benefits over an alternative model transformation language.

In this thesis, I design a case study evaluating the Commonalities Language. I discuss both the methodology of the case study and its validity. Furthermore, I introduce a novel property for bidirectional transformations, called congruency. It ensures a notion of compatibility between the two directions of a transformation. I motivate from practical examples why we can expect congruency to hold usually. Afterwards, I present the design decisions of a test strategy for two model transformation implementations that realize the same consistency specification. The test strategy also provides a practical application of congruency. Finally, I contribute improvements to the Commonalities Language.

Together, the contributions of this thesis enable researchers to conduct a case study on programming languages for model transformations, and, thus, gain a better understanding of their benefits. Congruency can make any transformation better usable and might proof useful for constructing model transformation networks. The test strategy can be applied to any acceptance test suite for model transformations.

Zusammenfassung

Sobald ein System durch mehrere Modelle beschrieben wird, können sich diese verschiedenen Beschreibungen auch gegenseitig widersprechen. Modelltransformationen sind ein geeignetes Mittel, um das selbst dann zu vermeiden, wenn die Modelle von mehreren Parteien parallel bearbeitet werden. Es gibt mittlerweile reichhaltige Forschungsergebnisse dazu, Änderungen zwischen zwei Modellen zu transformieren. Allerdings ist die Herausforderung, Modelltransformationen zwischen mehr als zwei Modellen zu entwickeln, bislang unzureichend gelöst. Die Gemeinsamkeiten-Sprache ist eine deklarative, domänenspezifische Programmiersprache, mit der multidirektionale Modelltransformationen programmiert werden können, indem bidirektionale Abbildungsspezifikationen kombiniert werden. Da sie bis jetzt jedoch nicht empirisch validiert wurde, stellt es eine offene Frage dar, ob die Sprache dazu geeignet ist, realistische Modelltransformationen zu entwickeln, und welche Vorteile die Sprache gegenüber einer alternativen Programmiersprache für Modelltransformationen bietet.

In dieser Abschlussarbeit entwerfe ich eine Fallstudie, mit der die Gemeinsamkeiten-Sprache evaluiert wird. Ich bespreche die Methodik und die Validität dieser Fallstudie. Weiterhin präsentiere ich Kongruenz, eine neue Eigenschaft für bidirektionale Modelltransformationen. Sie stellt sicher, dass die beiden Richtungen einer Transformation zueinander kompatibel sind. Ich leite aus praktischen Beispielen ab, warum wir erwarten können, dass Transformationen normalerweise kongruent sein werden. Daraufhin diskutiere ich die Entwurfsentscheidungen hinter einer Teststrategie, mit der zwei Modelltransformations-Implementierungen, die beide dieselbe Konsistenzspezifikation umsetzen, getestet werden können. Die Teststrategie beinhaltet auch einen praktischen Einsatzzweck von Kongruenz. Zuletzt stelle ich Verbesserungen der Gemeinsamkeiten-Sprache vor.

Die Beiträge dieser Abschlussarbeit ermöglichen gemeinsam, eine Fallstudie zu Programmiersprachen für Modelltransformationen umzusetzen. Damit kann ein besseres Verständnis der Vorteile dieser Sprachen erzielt werden. Kongruenz kann die Benutzerfreundlichkeit beliebiger Modelltransformationen verbessern und könnte sich als nützlich herausstellen, um Modelltransformations-Netzwerke zu konstruieren. Die Teststrategie kann auf beliebige Akzeptanztests für Modelltransformationen angewendet werden.

Contents

1 Introduction **1**

2 Foundations **3**
2.1 Domain-Specific Languages . . . 3
2.2 Model-Driven Software Development . . . 4
2.3 Model Consistency . . . 5
2.4 Model Transformations . . . 6
2.5 The Commonalities Approach . . . 9
2.6 The Vitruvius Framework . . . 10

3 A Case Study Evaluating the Commonalities Language **15**
3.1 Goals and Design . . . 15
3.2 The Selected Consistency Preservation Case . . . 16
3.3 Methodology . . . 17
3.4 Validity . . . 18

4 Congruent Bidirectional Transformations **25**
4.1 Relationship to the Consistency Relation . . . 28
4.2 Evaluation . . . 32

5 A Test Strategy for Different Implementations of a Consistency Specification **35**
5.1 Black Box Acceptance Tests . . . 35
5.2 Exploiting Congruency . . . 38
5.3 A Test Framework for the Test Strategy . . . 39

6 Improvements to the Commonalities Language **49**
6.1 Consistent Syntax for Renaming Participations . . . 49
6.2 Simplified Operator Imports . . . 50
6.3 Clearer Syntax for Operator Chains . . . 53

7 Related Work **57**
7.1 Evaluating Model Transformation Languages . . . 57
7.2 Properties of Model Transformations . . . 59
7.3 Model Transformation Testing . . . 65

8 Conclusion **67**

Bibliography **71**

List of Figures

2.1 An example for code written in the Reactions Language. 11
2.2 An example for code written in the Commonalities Language. 12

3.1 The Goal-Question-Metric plan for the presented case study. 20

4.1 The two types of changes in a congruent transformation. 28

5.1 The Test Pyramid. 36
5.2 A schematic test plan exploiting congruency. 39
5.3 A Java parameter printed without and with custom formatting. 42
5.4 Execution time measurements for two test suites. 44
5.5 A test specification written with the test framework. 45
5.6 Test cases derived by the test framework. 47
5.7 Example for a difference printed by the test framework. 47

6.1 The new syntax for Participation aliases. 50
6.2 Demonstration of the new mechanism to resolve operators. 52
6.3 The current and the proposed, new syntax for defining operator chains. . 54

7.1 Graphic representation of a bidirectional transformation example. 61
7.2 The relationships between properties of transformations. 63

1 Introduction

Throughout the sciences, models are used to make complex situations manageable. Software engineering, more than any other discipline, could profit from adopting model-based techniques more widely [Sel03]. Nevertheless, practitioners still use models mainly in an informal fashion, if at all [Bad+18]. Thus, they miss out on the benefits tool-based, model-driven development techniques promise; like simulating properties of a system before it is built, generating parts of the system's implementation, or avoiding architecture drift [Sch06; VS06; FR07]. One central reasons for the scarce adoption of model-driven development is inadequate support by tools [WHR14; Bad+18].

As soon as software developers start modelling more aspects of a software system, they face a dilemma: On one hand, they want to use the best-suited models for each aspect of their project. On the other hand, as soon as they use multiple models, the models can get out of synchronization and start to contradict each other. Any benefits they hope to gain from their models can only be achieved if the models are free of contradictions. But ensuring manually that the models stay consistent is laborious and can outweigh the increases in productivity that the developers hoped to gain by introducing the models in the first place [Bad+18]. A solution to resolve this dilemma is using model transformations that propagate changes made to one model to the other models. There has been much research on model transformations, resulting in good support for keeping two models in synchronization [Kah+19]. However, more research is needed on multidirectional model transformations. State-of-the-art solutions do not adequately support propagating changes between more than two models [Ste17; Cle+19].

Klare and Gleitze [KG19] proposed the Commonalities Approach as a programming model to support building multidirectional model transformations out of bidirectional ones. It allows to re-use the well-established body of knowledge on bidirectional transformations for the multidirectional case. Gleitze [Gle17] developed a programming language to support the Commonalities Approach, called the Commonalities Language. Although Klare and Gleitze give reasons why the Commonalities Approach and Language will improve how multidirectional model transformations can be developed, it remains an open question whether these benefits manifest themselves in practice. Furthermore, some of the design decisions on the Commonalities Language are based on not-yet-validated assumptions about how the language will be used. By conducting a case study on a realistic case of consistency preservation for multiple models, we can provide the missing validation regarding these open questions. This thesis lays the groundwork for such a case study.

The case study will use the case of consistency between a Java [Gos+21], a UML [Obj15], and a PCM [Reu+16] model. These models are different views on a software system: Java is the system's implementation, UML captures the system's architecture, and PCM models the system's performance attributes. Hence, the case addresses the dilemma that arises when using multiple models, as I described above. The case study will validate whether the

Commonalities Language is suitable for building multidirectional model transformations. Furthermore, it will compare the implementation in the Commonalities Language to an implementation in an alternative model transformation language. The comparison shall reveal which benefits using the Commonalities Language brings.

Before the case study can be conducted, challenges of both technical and conceptual nature need to be overcome. After having presented the case study design, I will address some of these challenges. Concretely, I will make the following contributions in this thesis:

A case study design Chapter 3 describes a design for a case study evaluating the Commonalities Language. It presents the study's methodology and analyses the design's validity.

A useful property for bidirectional transformations Chapter 4 presents a novel property for bidirectional transformations: congruency. I will describe under which conditions bidirectional transformations can be congruent. I will give practically motivated reasons for why I expect that all bidirectional transformations that the Commonalities Language is concerned with can be congruent.

A test strategy comparing transformation implementations Chapter 5 derives a strategy to test that two model transformation implementations realize the same consistency specification. It gives reasons for why the test suite should be designed in a particular way, and checks potential drawbacks of this design. The test strategy also presents a practical application of congruency.

Improvements to the Commonalities Language Chapter 6 presents improvements to the Commonalities Language that make the language easier to use and better adapted to its domain of application. The chapter presents justifications for the design decisions behind the improvements.

All contributions facilitate conducting a case study on the Commonalities Language. The case study will validate both whether the assumptions on which the Commonalities Language was designed are accurate, and whether its expected benefits manifest in practice. Thus, the case study and this thesis contribute to finding an appropriate programming model for multidirectional model transformations in Vitruvius and in general. Congruency enables developers to create better usable model transformations. In conjunction with the presented test strategy, congruency also helps testing model transformations.

The thesis will start by introducing the most relevant concepts that this thesis is based on (Chapter 2). After presenting the contributions in Chapters 3–6, I will relate them to the existing body of research in Chapter 7. Chapter 8 concludes the thesis and gives an outlook on future work.

2 Foundations

This thesis is evaluates a domain-specific languages for programming model transformations, the Commonalities Language. In this chapter, I will introduce the concepts that are relevant to the discussions in this thesis. We will start by looking at what domain-specific languages are and why one might consider using them. We will then turn to model-driven software development and how it raises the need to maintain model consistency. Finally, we will see how the Commonalities Approach and the Vitruvius Approach address this need with the help of the Commonalities Language.

2.1 Domain-Specific Languages

Most software is typically written using a general-purpose programming language, like C. C is 'not specialized to any particular area of application' [Ker88, p. xi] and can, hence, be used for almost any task. On the other hand, there are programming languages which are tailored to a specific software domain. These languages are called *domain-specific languages* [Völ13, p. 28]. They deliberately sacrifice claims of generality to better fit the requirements of their usage context.

Latex, for example, is a domain-specific language for typesetting documents. It offers syntactic features and commands that address typical use cases for this task. While documents could, theoretically speaking, be typeset with C as well, it would be significantly more cumbersome than using Latex. For instance, Latex allows writing the text of a document directly into the source code file. Text and commands can be mixed freely. Arguments to commands do not always need to be enclosed by special characters. Strings in C, on the other hand, must always be delimited by double quotes. String arguments to functions must also be given in double quotes, and the return value of functions must be combined with other strings using a function like '`strcat`'. That alone makes Latex a better fit to produce documents, which contain a lot of text.

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]. The languages are valuable tools for their domains. However, not every software domain benefits from having its domain-specific language. Domain-specific languages are laborious to develop and first need to be learned by their potential users before they can bring benefits. Whether or not to develop a domain-specific language needs to be decided carefully on a case-to-case basis and the right answer 'may become clear only after a sizeable investment in domain-specific software development using a [general-purpose language] has been made' [MHS05, p. 320].

When implementing a domain-specific language, there is a fundamental design decision to be made: whether to develop it as an *internal* or an *external* domain-specific language. Internal domain-specific languages do not define a new syntax, but instead re-use the syntax of an existing—usually general-purpose—language, the so-called host language. Code written in an internal domain-specific language is still valid code in the host language. The domain-specific language is merely a specific way to use the host language. External domain-specific languages, on the other hand, use their own syntax and, thus, need their own parser, compiler, type system, etcetera [FP10, p. 28]. Internal domain-specific languages can be implemented faster, and existing tools can be used for them. External domain-specific languages, on the other hand, can freely choose the syntactic constructs that make the most sense for them and are not limited by the syntax of the host language [FP10, p. 150 ff.]. Whether an internal or an external domain-specific language fits an intended use case best needs to be decided, once again, on a case-by-case basis [FP10, p. 29]. Both approaches have been used successfully. All previously mentioned examples are external domain-specific languages. The software build system Gradle, for instance, is configured using internal domain-specific languages hosted by either Groovy or Kotlin. Software build systems illustrate that there is no one-size-fits-all answer for internal versus external domain-specific languages, not even within the same domain. Because for the domain of software build systems, both approaches have been successful: The external domain-specific language of Make as well as the internal domain-specific languages of Gradle.

2.2 Model-Driven Software Development

Software engineering suffers from a gap between the concepts of the problem and the concepts of 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 proposed to reduce this gap. Developers practising model-driven software development describe software in terms of the problem domain. This 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 can also be tailored to the specific domain the software is being built for. Either way, they do not merely document the software but are made to be a part of it, for example by being automatically transformed into executable code [FR07]. Thereby, models get equal to programming languages. In fact, we can regard a programming language as just another model that is used to describe and create the software. In consequence, the software development process becomes similar to the practice in other engineering disciplines: Just like 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 just by modelling it. The comparison to other engineering disciplines goes further: with model-driven software development, software engineers can execute analyses, predict

properties of the software system, and evaluate the design before it is implemented, solely based on the models—just like mechanical engineers can [VS06, p. 6].

Practising model-driven software development promises several advantages. First, it can increase development productivity. Because they are working on a higher level of abstraction, developers can ignore details and focus on the software's 'core'. The productivity improvements have been found in empirical studies, however, only if adequate tooling and code generators are available [Kap+09; MCM12; WHR14]. Second, model-driven software development can 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, it is easy to implement suboptimal solutions: The system's intended architecture might be violated, or a feature may fail to meet its requirements precisely [Sch06]. If the system's architecture, its requirements or its domain-specific properties are modelled explicitly, and the software is derived from these models, the architecture, requirements and other properties are enforced. Additionally, since they have been formulated in a domain-specific way, it will be easier to reason about whether they have been stated correctly.

For our means, models are digital representations of a part of the modelled system. A model focuses on the aspects that are relevant for a particular point of view and represents the aspects according to this viewpoint [Sta73, p. 131 ff.]. All models we will be concerned with in this thesis will conform to a *metamodel.* A metamodel is a model that defines the possible structure of another model [VS06, p. 85]. A model's metamodel is what a class is to an object in object-oriented programming. We will formalize metamodels simply as the set of all valid model instances; that is, a model m conforms to the metamodel M if, and only if, $m \in M$. We will assume that a metamodel is never an empty set.

2.3 Model Consistency

If multiple models describe the same system, it is possible for the same piece of information to be contained in multiple models. In such a situation, we say that the models share *semantic overlap* [Bur+14]. Semantic overlap does not imply that the piece of information is contained as a syntactic copy in the concerned models. It can be represented in very different ways, even just implicitly. When editing a model which shares semantic overlap with other models, it can become out of synchronisation with the other models. While the models are still expected to share the same piece of information, they now contradict each other. We say the models are *inconsistent.* If, on the other hand, no model contradicts another—either because models share no semantic overlap or because the 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 across multiple models, that will often be from different metamodels. It is an external property. In addition, there is never an inherent or unique definition of when a set of models is consistent, although there may be an intuitive one. Consistency is, thus, relative to a *consistency specification* that declares which model states should be considered consistent [Kra17, p. 38].

Finding and resolving inconsistencies is a central activity in software engineering [NER01]. For example, implementing a new feature of a software system can be seen as resolving inconsistency, because the software implementation has become inconsistent with its requirements. Model-driven software development can help to find inconsistencies, because models can be checked automatically. It can also assist with resolving inconsistencies.

Different approaches have been proposed to manage consistency in model-driven software development. Orthographic software modelling [ASB10], for example, uses a central model that is free of semantic overlap and can, hence, never become inconsistent. The software is only edited through views that are automatically derived from the central model. This guarantees that no inconsistencies can arise, but also restricts the models that can be used. Furthermore, orthographic software modelling does not allow to model inconsistent states when they exist in reality. For example, product management might have decided that a feature should be implemented, but implementation has not started yet. Representing such inconsistencies in the models can be valuable [NER01], but is not possible in orthographic software modelling.

2.4 Model Transformations

If model inconsistency cannot, or should not, be avoided, then it needs to be resolved at one point to reach a state where all models are consistent again. If done manually, this 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. For these reasons, it is desirable to automate the process at least partially. A prominent means for automating consistency preservation are *model transformations*. They derive a new target model from an existing source model. In the applications we are interested in, model transformations create an updated version of the target model after the source model has been changed, such that the target model reflects the changes in the source model and the models are consistent again.

2.4.1 Bidirectional Model Transformations

In our use cases—that is, model consistency preservation—we usually allow users to modify each of the transformations' two models, and want to keep the respective other model consistent with the users' modifications. Hence, we require that the transformation between the two models can transform changes in both directions. We call such a transformation a *bidirectional* transformation. This is the class of model transformations that we will be concerned with in this thesis.

To make it easier to reason about bidirectional transformations, I will use a light-weight formalization which was put forward by Stevens [Ste10]. It is based on the notion of a consistency relation $T \subseteq A \times B$ for two metamodels A and B. The consistency relation

defines which models are consistent with each other[1]. For our purposes, the consistency relation will always be left-total and right-total. This means that every model state has at least one consistent 'partner' state. In practice, the consistency relation T will often not be stated explicitly [Kla+20]. Instead, it will usually be understood implicitly by those that create a model transformation. Nevertheless, the consistency relation is a useful tool for reasoning about transformations.

In our formalization, a bidirectional model transformation $\overleftrightarrow{T}$ for a consistency relation T consists of one transformation for each direction, named $\vec{T}$ and and $\overleftarrow{T}$. The transformations create a new target model from their source model. Because there are practically relevant consistency relations that can only be realized if a transformation can also examine the current state of the target model, the transformations receive both models as their input. We formalize the transformation directions as functions.

Definition 2.1 ([Ste10])**.** A bidirectional transformation $\overleftrightarrow{T} =: (T, \vec{T}, \overleftarrow{T})$ between the metamodels A and B is given by

$$
\begin{aligned}
&T \subseteq A \times B \\
&\vec{T} : A \times B \rightarrow B \\
&\overleftarrow{T} : A \times B \rightarrow A.
\end{aligned}
$$

The most basic requirement we have for bidirectional transformations is that the transformation functions adhere to the consistency relation.

Definition 2.2 ([Ste10])**.** A bidirectional transformation $\overleftrightarrow{T} =: (T, \vec{T}, \overleftarrow{T})$ is *correct* if, and only if, its results are always consistent:

$$
\begin{aligned}
\forall a \in A \; \forall b \in B\colon \quad & (a, \vec{T}(a, b)) \in T \\
& \wedge (\overleftarrow{T}(a, b), b) \in T.
\end{aligned}
$$

We want to use bidirectional transformations to resolve inconsistencies. If our models are already in a consistent state, however, there is nothing to do and we expect the transformations not to modify the models at all. After all, there might be good reasons why the models are in this particular consistent state, and not in another consistent one. Inspired by the Hippocratic Oath—which demands 'first, do no harm'—Stevens calls transformations that fulfil this property 'Hippocratic'.

Definition 2.3 ([Ste10])**.** A bidirectional transformation $\overleftrightarrow{T} =: (T, \vec{T}, \overleftarrow{T})$ is *Hippocratic* if, and only if, it does not modify consistent model states:

$$
\begin{aligned}
\forall (a, b) \in T\colon \quad & \vec{T}(a, b) = a \\
& \wedge \overleftarrow{T}(a, b) = b.
\end{aligned}
$$

[1]We already saw a similar term in Section 2.3: 'consistency specification'. How are the two terms related, and what is the difference? In this thesis, I will use 'consistency relation' in the specific meaning introduced here; that is, a relation $A \times B$ that defines which model states are consistent. With 'consistency specification', on the other hand, I will refer to any, not necessarily formal, specification of when models are consistent. A consistency specification may additionally encompass an operationalization; that is, a description of *how* to reach consistency. A consistency relation is, thus, one possible form of a consistency specification that does not include an operationalization.

2.4.2 Lenses

One specific type of bidirectional model transformations are lenses, as introduced by Foster et al. [Fos+07]. Lenses transform between a concrete and a more abstract model. They can lose information when transforming from the concrete to the abstract model. This operation is called Get. When transforming in the other direction, they use the current state of the concrete model to restore the lost information best possible [Fos+07]. This operation is called Put.

Definition 2.4 ([Fos+07])**.** A lens $l := (\textsc{Get}, \textsc{Put})$ between two metamodels A and B consists of two functions[2], Get and Put, which are defined as

$$\begin{aligned} \textsc{Get}\colon \quad & A \longrightarrow B \\ \textsc{Put}\colon & A \times B \longrightarrow A. \end{aligned}$$

If we compare this definition to the definition of a general bidirectional transformation (Definition 2.1), we notice that they coincide, except that the Get operation has only one argument. A lens can, indeed, be regarded as a bidirectional transformation where one transformation direction ignores the current state of the target model. With this view, the basic requirements for lenses defined by Foster et al. [Fos+07] are equivalent to correctness (Definition 2.2) and Hippocraticness (Definition 2.3) [Ste10]. Lenses that fulfil these requirements are called *well-behaved.*

Since the GET operation cannot take the current state of the target model into account, lenses are more restricted than general bidirectional transformations. On one hand, this means that they can be used in fewer contexts. On the other hand, the restrictions provide useful guarantees that general bidirectional transformations do not. The Commonalities Language uses the fact that lenses can be chained. Two general bidirectional transformations $\overleftrightarrow{T} := (T \subseteq A \times B, \vec{T}, \overleftarrow{T})$ and $\overleftrightarrow{U} =: (U \subseteq B \times C, \vec{U}, \overleftarrow{U})$ between A and B, and B and C, cannot be composed into a bidirectional transformation between A and C, because both transformation directions need access to the current state of the model B. Lenses, in contrast, *can* be composed this way.

Proposition 2.1 ([Fos+07])**.** *Two well-behaved lenses $l =: (\textsc{Get}_l, \textsc{Put}_l)$ between A and B, and $m =: (\textsc{Get}_m, \textsc{Put}_m)$ between B and C can be composed to a well-behaved lens $c =: (\textsc{Get}_c, \textsc{Put}_c)$ between A and C by*

$$\begin{aligned} \textsc{Get}_c\colon & A \longrightarrow C \\ & a \longmapsto \textsc{Get}_m\left(\textsc{Get}_l\left(a\right)\right) \\ \textsc{Put}_c\colon & A \times C \longrightarrow A \\ & (a, c) \longmapsto \textsc{Put}_l\left(a, \textsc{Put}_m\left(\textsc{Get}_l\left(a\right), c\right)\right). \end{aligned}$$

The 'trick' that allows lenses to be chained is that $\textsc{Get}_l$ can derive the old B-state from the old A-state without any further information.

[2] Foster et al. define lenses with *partial* functions, a detail I have omitted to simplify our discussion, since it is not relevant for us.

2.4.3 Multidirectional Model Transformations and Transformation Networks

How can we preserve consistency if there are more than two models? When using model transformations, we have two options: Either combine multiple bidirectional transformations or use multidirectional transformations, which generalize bidirectional transformations to transform more than two models. Both approaches have merits. Advantages of combining bidirectional transformations are that bidirectional transformations are well understood and there are mature tools supporting them. Additionally, as experience has shown, reasoning about bidirectional transformations is difficult enough on its own. Understanding and correctly creating multidirectional transformations will, thus, be even more difficult [Ste17]. We can expect that breaking the transformation up into multiple bidirectional transformations and combining them correctly is the better manageable approach. It is also the approach which this thesis focuses on.

When we combine multiple bidirectional transformations, we create a *transformation network*. Its topology is defined by the metamodels we use and by the bidirectional transformations we use between the metamodels. To actually restore consistency in an instance of such a network, we additionally need a transformation execution strategy. The strategy defines which transformations will be executed in which order. Finding an appropriate execution strategy is an open problem [GKB21]. For this thesis, we will presuppose that there is a suitable execution strategy. For all concrete consistency preservation constellation we will be concerned with, a naive execution strategy suffices.

2.5 The Commonalities Approach

In the previous section, I motivated that we want to preserve consistency among multiple models using a network of bidirectional transformations. Klare and Gleitze [KG19] propose the *Commonalities Approach* for designing such networks. By examining prototypical network topologies, they argue that any transformation network topology has inherent disadvantages if it only involves the models that shall be kept consistent. These disadvantages concern how laborious it is to write all required transformations, how open for change the resulting network is, and how prone the transformations are to be inconsistent with each other. Klare and Gleitze [KG19] demonstrate that depending on the chosen topology, the disadvantages manifest to different extents, but can never be avoided entirely. If, however, we introduce an additional metamodel to the network, we can escape the dilemma. Klare and Gleitze [KG19] propose that this additional metamodel—the *concept metamodel*—should capture all concepts that at least two models of the network have in common. In other words, the concept metamodel should model the other model's semantic overlap.

To build the transformation network with the Commonalities Approach, transformation developers define transformations between the concrete metamodels and the concept metamodels. The transformations translate between the common concept in the concept metamodel and its manifestation in the concrete metamodel. This results in a transformation network in star topology, with the concept metamodel at its centre.

Using the Commonalities Approach promises three benefits [KG19]: First, it makes the transformations more understandable because common concepts of the concrete metamodels are made explicit in the concept metamodel. In other transformation networks, these concepts are encoded only implicitly in the bidirectional transformations. Second, the approach reduces the potential for errors when writing the transformations because transformations do only need to translate between the concept metamodel and the concrete metamodel. In other transformation networks, it is often necessary to take care that a transformation is compatible to all other transformations. Third, the Commonalities Approach makes reusing transformation networks easier. It allows to remove a metamodel from the network without affecting the consistency preservation for the rest of the network. That is not always possible in other transformation networks.

This thesis evaluates the Commonalities Language, a programming language for writing model transformations with the Commonalities Approach. The Commonalities Language is part of the Vitruvius framework, which we will now turn our attention to.

2.6 The Vitruvius Framework

Vitruvius is an approach and framework for maintaining consistency in model-driven development [Kla+20]. Vitruvius allows to combine view-based modelling and consistency preservation in one development process. When applying the Vitruvius approach, users define a transformation network out of potentially pre-existing models. This transformation network is used as a central datastore for all modelled data, called 'virtual single underlying model' (VSUM). The data in the VSUM is edited only through views, which allow to select and transform the data so it fits the editor's use case best. This way, Vitruvius promises to combine the benefits of consistency preservation and view-based editing: Through consistency preservation, users can integrate pre-existing models and their tooling into their development process without losing consistency. Through view-based editing, users can edit the modelled data in a form that fits their point of view best, without every time having to integrate a new model into the network.

Vitruvius offers users external domain-specific (cf. Section 2.1) programming languages to program the transformations that make up the VSUM network. Two of them are the Reactions Language and the Commonalities Language.

2.6.1 The Reactions Language

In the way we formalized them in Section 2.4.1, model transformations are state-based: they derive a new model based on the state of the input models after one of them was modified. This is also how most bidirectional transformation languages operate [Hid+16]. However, state-based transformations have a drawback: they cannot know *how* its input model states were reached. This information can be relevant. For example, it might be necessary to differentiate whether some object's features changed, or whether the object was actually deleted and a replacement with different features was created afterwards. Thus, state-based transformations are not able to realize all practically relevant consistency specifications [Dis+11]. Because of that, model transformation in Vitruvius happens based

```
1  reaction {
2      after element adl::Component inserted in adl::Repository[components]
3      call {
4          val component = newValue
5          createClass(component)
6      }
7  }
8
9  routine createClass(adl::Component component) {
10     match {
11         require absence of oo:Class corresponding to component
12         val componentsPkg = retrieve oo::Package corresponding to component.repository
               tagged with "componentsPackage"
13     }
14     action {
15         val class = create oo::Class and initialize {
16             class.package = componentsPkg
17             class.name = component.name + "Impl"
18         }
19         add correspondence between component and class
20     }
21 }
```

Figure 2.1: An example for code written in the Reactions Language [Kla+20]. Additionally to the Reaction Language's syntax constructs, programmers can use the general-purpose language Xbase at well-defined sections in the the language; like in the `call` and in the `initialize` block.

on changes. Additionally to the current model states, transformations can access the sequence of changes that lead to the states. Vitruvius offers the Reactions Language to program transformations that react to the changes that occurred.

When writing transformations in the Reactions Language, programmers declare the change event they want to react to, and how the models should be changed in consequence of the change. The language offers syntax elements for matching the change events and executing the most common consequential modifications. For more advanced actions, developers can also write code in Xbase, a general-purpose programming language similar to Java. Xbase code can be written directly into a Reactions Language file. We can see an example of this in Figure 2.1. Because the Reactions Language allows embedding a Turing-complete, general-purpose programming language, and because it allows to match any possible change, it is guaranteed that the language can be used to realize any consistency specification (assuming the specification is computable) [Kla+20].

The Reactions Language also gives developers access to a correspondence model. Correspondences store model elements that are in some kind of relationship with each other. Later invocations of a transformation can use the information in the correspondence model to recover such relationships after they have been established by earlier invocations. If, for example, an object b in model B was created in reaction to the object a being created in model A, developers could create a correspondence between a and b. If b was deleted at a later point, the correspondence could be used identify a and delete it as well. In Figure 2.1,

```
concept ComponentBasedDesign

commonality Component {
    with PCM:Component
    with UML:Component
    with ObjectOrientation:Class

    has name {
        = PCM:Component.name
        = UML.Component.name
        = prefix(ObjectOrientiation:Class.name, "Impl")
    }
}
```

Figure 2.2: An example for code written in the Commonalities Language [KG19]. It declares a Commonality for a software component. The name attribute of the Commonality is mapped to the name attribute of an ObjectOrientation Class by appending 'Impl'. Because `prefix` is a bidirectional operator, the suffix 'Impl' will also be removed from names of ObjectOrientation Classes when they are mapped to the component Commonality.

we can see how a correspondence is created using the '`add correspondence`' keyword and how correspondences are queried with the '`corresponding to`' keyword.

2.6.2 The Commonalities Language

The Commonalities Language is another programming language for writing transformations for Vitruvius. It was created to give developers two benefits compared to the Reactions Language: First, the Commonalities Language aims to have a more declarative style. That is, programmers should have to focus less on *how* consistency can be achieved, but rather only have to define *when* models are consistent [Gle17, p. 29]. Second, the Commonalities Language shall allow developers to address multi-model consistency better by realizing the Commonalities Approach that was introduced in Section 2.5 [KG19].

To create a transformation with the Commonalities Language, developers first define Commonalities. Commonalities are the model objects of the concept metamodel (cf. Section 2.5) and capture the semantic overlap (cf. Section 2.3) of the models that shall be kept consistent. Developers can then define how the concrete models map to the Commonalities. To declare these mappings, developers can use bidirectional operators. Such operators define the mapping in both directions at the same time. An example of this can be seen in Figure 2.2.

Operators can be defined in Java (or any other JVM programming language) and then be imported into the Commonalities Language. This allows developers to define and re-use bits of mapping logic. Unlike the Reactions Language, the Commonalities Language does not allow to embed code in a general-purpose programming language. It is unclear whether the Commonalities Language can be used to realize all computable consistency specifications like the Reactions Language can.

The Commonalities Language knows three different types of operators [Gle17, p. 32 f.]. The main type are *bidirectional operators*, which can be executed in both directions. They have been adopted from Kramer and Rakhman [KR16], who extend the lens formalism (cf. Section 2.4.2) to cases where the right-hand model can assume values that are outside the image of the GET function. In these cases, lenses cannot be well-behaved. Kramer and Rakhman [KR16] show how to construct lenses that are 'best-possible-behaved' in such situations. The resulting operators are more flexible, while retaining the desirable properties of lenses where possible. The second type of operators are *enforceable conditions*. Such operators act as conditions when executed in one direction, and produce a value fulfilling the condition when executed in the other direction. We can view them as lenses between an arbitrary metamodel and the metamodel $\{\text{true}, \text{false}\}$. Finally, the Commonalities Language allows using *predictable expressions*. These are usual, unidirectional expressions that can be used as a fallback when bidirectional operators cannot be used. Their only restriction is that they must be a deterministic function of their input values.

On a technical level, Commonalities Language code is compiled to Reactions Language code. Because of that, it is possible to use the Commonalities Language together with the Reactions Language in the same transformation.

3 A Case Study Evaluating the Commonalities Language

This thesis facilitates a case study that examines the Commonalities Language. The case study shall reveal whether the Commonalities Language is suitable developing for real-world model transformations, whether the language's operator mechanics can be used the way that they are intended to be used, and whether the Commonalities Language offers benefits over the Reactions Language. To those ends, the study analyses code written with in the Commonalities Language, and compares the code to an implementation of the same case in the Reactions Language. By examining two implementations of the same consistency specification we want to reveal relevant differences between the two languages. This chapter will describe the case study and discuss its design.

3.1 Goals and Design

This thesis is part of an overarching objective to find good programming models for consistency preservation with the Vitruvius approach. The Commonalities Language was proposed to address two shortcomings of the existing consistency programming language for Vitruvius, the Reactions Language: First, the Commonalities Language shall make programming transformations more declarative, and thereby shorter and easier to read and write. Second, the Commonalities Language shall make programming multidirectional transformations less error-prone, more comprehensible and altogether easier. While the language was developed with these goals, no empirical evaluation has been conducted yet to verify whether the language fulfils its claims in practice. This thesis contributes to such an empirical test.

To understand the Commonalities Language's usefulness, we first need to analyse its suitability; that is, the degree to which it can be used to program model transformations. There might be constellations in which the Commonalities Language cannot be used to maintain model consistency. Afterwards, we can turn to usefulness, i.e. the question of *how well* the Commonalities Language allows programming model transformations. Unlike for suitability, it is difficult to find absolute indicators for answering this question. Whether a programming language is useful can, instead, be best understood in comparison to another programming language. Since the Commonalities Language is developed for the Vitruvius framework, we compare it to the other model transformation programming language for Vitruvius, the Reactions Language. Developers using Vitruvius will be faced with the decision whether to use the Reactions Language or the Commonalities Language[1].

[1]There is another model transformation programming language for Vitruvius: the Mappings Language. However, the implementation of the Mappings Language is currently not in a state where it could be

Hence, we can expect the comparison to yield results that directly give practically relevant advice to developers.

Crucially, picking a programming language for developing a model transformation for Vitruvius is not an either-or question. Instead, the languages have been developed to complement each other. Developers can, for example, pick the Commonalities Language as their default language, but fall back to using the Reactions Language for constellations where the Commonalities Language cannot be used or is less useful[2]. Because of that, the case study described in this thesis should be designed to yield results that indicate *under which circumstances* which language is more useful.

With these considerations in mind, I derived a 'Goal-Question-Metric' plan [BW84] for the case study. I first broke the research goal up into quantifiable questions. Then I found appropriate metrics that can be used to answer the questions. The 'Goal-Question-Metric' approach ensures that the selected metrics are adequate for evaluating the research goal. The plan itself is given in Figure 3.1 (on page 20). We will discuss the chosen metrics in detail in Section 3.4.3, which focuses on construct validity.

3.2 The Selected Consistency Preservation Case

Subject of the proposed case study will be an implementation of a consistency specification for UML, PCM, and Java models. The consistency specification was designed by Langhammer [Lan17] and synchronizes the representation of concepts from object-oriented and component-based software design between the three model types. Where concepts exist in multiple models—for example a class in UML and Java, or a component in UML and PCM—the specification defines the intuitive mapping for those concepts. It specifies, for example, that for every Java class there shall be a UML class with equivalent properties; and vice versa. When a concept does not exist in a target model but shall still be represented, the specification establishes conventions to express the source concept through the concepts of the target model. Java, for example, has no concept for software components. So a component that was created in UML or PCM is represented in Java using a Java package and a Java class. Hennig [Hen20, p. 30 ff.] gives a detailed description of all the mappings that encompass the consistency specification.

The selected consistency preservation case is a good fit for the case study goal for several reasons. First, the case preserves consistency among models with much semantic overlap. For instance, almost all structural features of Java can be represented in UML. The Commonalities Language was developed to allow building transformations that synchronize concepts among model types with semantic overlap. Hence, the case is one that the language claims to address. Second, the consistency specification is multidirectional. Since the Commonalities Language was developed to address multidirectional transformations

used for real-world case studies. Furthermore, the Commonalities Language might be considered a superset of the Mappings Language, meaning that there might be no use case at all where the Mappings Language is better suited than the Commonalities Language. This is, of course, a subject for another empirical evaluation.

[2] There has been no case study combining different languages in Vitruvius yet. However, since the Commonalities Language generates code in the Reactions Language, we can reasonably expect that the two languages can be combined easily.

better than bidirectional solutions do, the case must include at least three models. Third, the selected case involves structurally diverse models. For example, PCM's objects all have a dedicated identifier feature. UML also uses identifiers for objects, but does not model them explicitly. Instead, identifiers are assigned by the serialization mechanism. Java objects, lastly, have no singular identifier, but are identified by their name and their topological position in the model tree. As a second example, packages in UML are model objects containing all other objects that are in the package. In Java, on the other hand, a package is conceptually created by all types declaring it. Types are stored in different compilation units, so there cannot be a single package instance that points to all contained objects. Such structural peculiarities pose technical challenges for model transformation languages and implementations. The fact that the models of the selected case have a diverse structure and contain such peculiarities means that the case study tests the Commonalities Language in a wider range.

3.3 Methodology

To conduct the the case study, Langhammer's consistency specification for UML, PCM, and Java needs to be implemented using the Commonalities Language. Hennig [Hen20] has already started such an implementation, but there are significant parts of the specification that are not covered. While some of those parts could not be realized because of time constraints, other parts could not be realized because the Commonalities Language did not allow it [Hen20, p. 119-122]. Furthermore, Hennig identified issues with the Commonalities Language that led to suboptimal code in the implementation [Hen20, p. 95-105]. Hence, the Commonalities Language needs to be adapted and extended to allow, where possible, realizing the parts of the specification that can currently not be realized, and to allow writing better code for the parts where the code currently has clear deficiencies. We will discuss the validity of this approach in Section 3.4.

Chen [Che17], Klatte [Kla17] and Syma [Sym18] have implemented Langhammer's consistency specification in the Reactions Language. Other developers have since maintained and improved the implementation. It shall be used as a baseline to compare the implementation in the Commonalities Language against. To make sure that the comparison is valid, both implementations need to implement the exact same behaviour. Even small differences in behaviour threaten the validity of the study, because the difference may be in an area where one of the two languages is more useful. The differences would then alter the outcome of the study.

Because of the size and complexity of the case study, we should not trust that both implementations will exhibit the same behaviour just because they were implemented against the same specification. Instead, there needs to be an extensive test suite ensuring that both implementations behave the same in all relevant aspects. The test suite also needs to make sure that the exhibited behaviour complies with Langhammer's specification exactly. Any differences between the specification and the implementations might, once again, hide relevant differences between the languages. The test strategy and the technical realization of this test suite is the topic of Chapter 5.

After the Commonalities Language has been adapted to allow conducting the case study, the consistency specification has been implemented in the Commonalities Language, and both implementations have been tested to accurately implement the same and the right specification; the final step in the case study will be to evaluate the metrics specified in Figure 3.1 against the implementations. Since not all of the metrics are objective, the evaluation needs to be conducted by a domain expert.

3.4 Validity

In this section, we will examine the validity of the case study. Since the case study was not conducted completely yet, there are insufficient results and we cannot draw any conclusions yet. Nevertheless, we can already judge the validity of the presented methodology and metrics. We will, in turn, examine whether the study's results can be generalized outside of the scope of the case (external validity); whether the values we obtain for our metrics can be attributed to the treatment (internal validity); and whether the selected metrics accurately capture the concepts we are interested in (construct validity) [Woh+12, p. 102 f.].

3.4.1 External Validity

When interpreting the results of a case study, we can usually not separate the measured phenomenon as clearly from the case study's context [Yin13, p. 13] as we could, for example, in a controlled experiment. Hence, we can generalize from case studies worse than from other types of studies [Woh+12, p. 55]. Without further empirical evaluation, there will be no sure way to tell whether similar results can be expected when implementing other consistency specifications with the Commonalities Language. The case study mainly serves as a proof of existence; that is, it shows that there are practically relevant consistency specifications for which the Commonalities Language is, or is not, suitable and useful.

In Section 3.3, I described that the Commonalities Language should be adapted while the case study is being implemented. Since the language will be adapted with the specific case in mind, chosen solutions may not be sufficiently generalized, that is, they may only apply to the case at hand. This further reduces the external validity of the study.

Generalizability of case studies can be improved by 'strategic selection of cases' [Fly06]. To that end, we study a 'critical case' with this case study, such that we know that if we cannot find positive results for this case, it is unlikely that we will find positive results in other cases [Fly06]. The Vitruvius framework and the Commonalities Language have been developed with the case of consistency of UML, PCM, and Java in mind. Both the framework [Kla+20] and the language [Gle17, p. 13 f.] used a simplified version of the case as a running example. Furthermore, the case study lets domain experts implement the transformations and encourages them to make the best use of the respective transformation language. This amplifies the impact of potential advantages or disadvantages of the languages. Furthermore, as detailed in Section 3.2, the case offers a variety of model structures that need to be transformed, so that the languages need to prove themselves under diverse circumstances.

3.4.2 Internal Validity

The case study implementations will have been written by different developers. This precludes any biases that are common in intra-subject studies, like maturation or sequencing effects. On the other hand, having different developers for the implementations means some of the differences in the implementations may have been caused by the developers' different programming styles instead of the different languages. Differences can, thus, not be entirely attributed to the different treatments. We try to minimize this limitation by using domain experts to implement the case study. While implementing, the experts try to find not just any solution, but rather one that makes the best use of the respective language's features.

The implementers are, in part, aware of the case study the implementation is being created for. This may cause subject effects, i.e., it may influence the code the implementers write in a way that makes it conform to the study's hypothesis. However, since the implementers try to program optimal (instead of 'realistic') implementations anyway, these effects should not be able to change the outcome. All in all, the case study's results will relate more to an ideal implementation in the respective language, than to what an average developer might program. Similar to how the study can only provide a proof of existence and cannot be generalized (see Section 3.4.1), the study informs on what is possible with the languages, but not necessarily on what to expect when the languages will be used in practice.

A similar threat to validity stems from adapting the Commonalities Language itself to facilitate the case study. The threat is that changes to the Commonalities Language may be tailored to optimize the metrics of the case study, instead of optimizing the usefulness of the language. Whether that is possible, however, is a matter of whether the study has good construct validity. If optimizing the Commonalities Language according to the metrics of the study did not optimize usefulness as well, the study's metrics would be unfit to measure the intended constructs. As long as the design of the study provides sufficient construct validity, this threat should, thus, not reduce the study's validity further. On the other hand, adapting the Commonalities Language before evaluating it is essential for the study's internal validity. Since we want to measure how suitable and useful the language is for developers wanting to use it, the study needs to be conducted with a version of the language that developers would realistically use. As long as the language misses essential features, such that it does not allow implementing certain consistency specifications, or only allows to do so in a cumbersome way, the language cannot be evaluated in the case study. If it was evaluated in such a premature state, the results would *not* inform on the experience developers will have in practice because developers would not use the language in that state to begin with.

3.4.3 Construct Validity

Let us now turn to assessing how well the chosen metrics (cf. Figure 3.1) can represent the constructs we want to measure. In other words: How well do the chosen metrics answer our questions? We will examine each question in turn.

Goal Understand whether and how the Commonalities Language helps developers develop model transformations for Vitruvius.

- **Question 1** How suitable is the Commonalities Language for developing model transformations?
  - **Metric 1** A characterisation of consistency specifications that cannot be reasonably preserved with the Commonalities Language.
  - **Metric 2** The ratio of model features that can be reasonably transformed with the Commonalities Language to the those that cannot.
- **Question 2** Do the operator mechanics of the Commonalities Language offer good abstraction and reusability?
  - **Metric 3** The number of usages per used operator.
  - **Metric 4** The number of operators used per position where operators can be used.
- **Question 3** To what extent is it easier to develop multidirectional transformations in the Commonalities Language than in the Reactions Language?
  - **Metric 5** The ratio of the size of the implementation in the Commonalities Language, including operators and Reactions Language parts with their called Java code; to the size of the implementation in the Reactions Language, including called Java code; all measured in source lines of code.
  - **Metric 6** The number of classes of the abstract syntax tree of both the Commonalities Language and the Reactions Language.

Figure 3.1: The Goal-Question-Metric plan for the presented case study.

All metrics we discuss in this section will rely on the assumption that, as we established in Section 3.4.2, the case study implementation uses the Commonalities Language's features to find one of the best implementations. Hence, we preclude that any shortcomings measured by the metrics are due to a suboptimal implementation, and attribute the shortcomings to the language. I will refer to this assumption as the 'optimal code assumption'.

3.4.3.1 Suitability

As mentioned in Section 2.6.2, it is unknown whether the Commonalities Language can be used to implement every computable consistency specification. Hence, Question 1 shall establish in which situations the Commonalities Language can and and in which situations it cannot be used. Here, we are not only interested in evaluating the language, but also in exploratively learning more about the language's capabilities. To answer the question, I define two metrics: First, a description of the situations where the Commonalities Language cannot be used (Metric 1). Unusually for the Goal-Question-Metric approach,

this is a qualitative, rather than a quantitative metric. Nevertheless, this is appropriate for the explorative nature of the question. The second metric (Metric 2) is quantitative and evaluates the language's suitability by taking the ratio of model features that could to model features that could not be transformed. Together, the two metrics establish a clear construct: They show when the Commonalities Language was unsuitable for the case study, and how often that happened.

Both metrics are objective except for one complication: Since the Commonalities Language allows to use operators that are programmed in Java—a Turing-complete, general-purpose programming language—it is not unlikely that all computable consistency specifications can be realized with it technically, but not in a way that programmers would consider reasonable. It could, for example, be possible to write an operator that implements the whole transformation and only uses the Commonalities Language to be triggered initially. With such a solution, the language could be used for the specification in question, but no programmer would ever want to use it this way. To avoid such 'solutions', I specified the metrics to only consider *reasonable* implementations of consistency specifications. Determining what is reasonable and was is not, however, introduces a subjective component into the construct. I argue that this subjectiveness does not harm the precision or understandability of the construct significantly because developers will have similar definitions of when the constructs of the Commonalities Language have been used reasonably.

3.4.3.2 Operator Mechanics

The Commonalities Language allows programmers to define operators externally and re-use them in the language. This is the central mechanism to specify transformation logic in the Commonalities Language. If the mechanics are used as intended, programmers need to define new operators seldom and can specify most pieces of logic by combining existing operators. With Question 2 I examine whether the language was built in a way that allows this use. The construct that that the question aims at is twofold: Its first component is abstraction, i.e. whether the operator mechanics allow writing operators that are sufficiently abstract, such that they can be re-used well. Its second component is combinability, that is, whether operators can be combined well. It might be possible that sufficiently abstract operators can be defined (good abstraction) but the language does not allow to combine them well (bad combinability).

Metric 3 measures abstraction: A sufficiently abstract operator can be used in different contexts, and we expect such an operator to be used multiple times. A concrete operator with insufficient abstraction, on the other hand, can not be used in different contexts and we expect that it will not be used often. There are two caveats: First, even if it is possible to define sufficiently abstract operators, there might be no need to re-use every single one of the defined operators. Thus, we cannot conclude from the fact that an operator was not re-used that it is was not defined abstractly enough. Nevertheless, the only other possible reason—except bad abstraction—why no (or few) operators are re-used is that implementing the consistency specification does not allow re-using logic. This is not the case. The specification contains multiple mappings with similar logic. For example, there are multiple occasions where the transformations need to convert strings between upper and lower case, append or prepend strings by other strings, or convert

between enumerations. Hence, should we find with Metric 3 that only few operators are re-used, we can attribute this to bad re-usability of the operators, which we can, per the optimal code assumption, attribute to the Commonalities Language not allowing to define sufficiently abstract operators. The other caveat is that operators might be re-used in the implementation, but not in different contexts. This would be a form of code duplication. In this case, the fact that the operator was used multiple times would not indicate that it is a well-abstracted operator, but only that the same context occurred multiple times. Per the optimal code assumption, there will be no code duplication that could have been avoided with better code. There might, however, be code duplication because the Commonalities Language lacks mechanisms to avoid it. This would harm the precision of Metric 3. Hence, while developing the case study, the Commonalities Language needs to be extended such that whenever code should be de-duplicated, the language allows to do that. Luckily, extending the Commonalities Language is part of this case study's methodology (cf. Section 3.3), so we can be sure that the language will have sufficient mechanisms to avoid code duplication before it is evaluated. We can, in conclusion, expect Metric 3 to give a good measure of the abstraction that the Commonalities Language allows operators to have.

To measure how well operators can be combined with the Commonalities Language, we will count how many operators were used per code location that allows using operators (Metric 4). If operators can be combined well, we would expect that there are multiple locations where more than one operator was used. This suffers from a similar caveat as Metric 3: If there are not many code locations where operators are combined, we cannot know whether this is due to bad combinability of operators, or due to the implementation not needing to combine operators. Unlike before, there are no strong arguments to decide a priori which will be the case. Hence, the Metric 4 can be used to provide evidence for combinability. If, however, the metric shows only few operator combinations, the case study implementations needs to be examined carefully to judge whether this is due to a shortcoming of the language, or due to the selected consistency specification.

3.4.3.3 Comparison to the Reactions Language

With Question 3, we ask whether the Reactions Language or the Commonalities Language is the better choice to implement the case study's case. Unlike the previous questions, this one targets a construct that is not defined precisely. The study defines the 'better choice' in terms of 'development effort', meaning the amount of work that is required to write and maintain the transformations. This measure is both subjective—because different developers need different amounts of time for different tasks—and difficult to measure—because especially the necessary tasks for maintaining transformations are unknown a priori and could only be evaluated ex post after having conducted a long-running study[3]. We will, hence, have to settle on a metric that correlates with development effort, but will not be able to capture to whole picture.

[3] A study that only focuses on implementation effort while ignoring maintenance effort would skew its results. Maintenance is often the most cost- and time-consuming activity in a software project. Programming languages could, theoretically, trade easiness of maintenance against easiness of implementation. Such a study would not detect such trade-offs.

Subramanyam and Krishnan [SK03] analysed different code metrics for object-oriented software regarding their power to predict defects. They formulated the hypothesis that 'larger classes will be associated with a higher number of defects, all else being equal' [SK03], which they could support with the results of the study. The study was conducted on both Java and C++ code, which shows that the correlation is not specific to a programming language (although it might still be specific to object-oriented code). Their findings suggest that for a given problem, the solution with less code will be less prone to bugs. Hence, implementation size qualifies as a measurement for maintenance effort. It also qualifies as a measure for implementation effort, since we can expect that writing more code takes more time, all else being equal. Thus, we will compare the size of the implementations measured in source lines of code[4] (Metric 5). When counting the lines, we need to make sure to count all source code files that are part of the respective implementation. For the implementation in the Reactions Language, this includes all Reactions Language files and all Java code[5] that is transitively called from the Reaction Language code. For the implementation in the Commonalities Language, the count includes all Commonalities Language files and the Java code of called operators, including Java code that is transitively called by the operators. When counting Java code lines, only the lines of the source code should be counted, not the lines of libraries the source code might be linked against. As described in Section 2.6.2, code written in the Commonalities Language can be mixed with code written in the Reactions Language. If the Commonalities Language implementation contains code in the Reactions Language, these lines of this code shall be counted as they are counted for the Reactions Language implementation. The code of both implementations must be entirely disjunct to make sure the count is accurate. If the two implementations shared common code, it would be unclear how to count it, since some of the code might only be required for one of the two implementations.

Implementation size can, however, not be the only measure for development effort. As Subramanyam and Krishnan warn: 'This is because, if we reduce the size of all classes in the application, it may influence other design complexity measures.' [SK03]. The statement refers to the fact that an implementation might trade complexity for length, i.e. write shorter, but more complex code that is not actually easier to maintain. A similar argument can be made for programming languages: A language could allow writing short code by being more difficult to use. Language developers could, for example, introduce many specialized features that allow to write short code but make learning and using the language difficult. Unlike for implementations in object-oriented languages, there are no metrics that have been shown to measure the complexity of a language (see also Section 7.1 where we discuss related studies). In this study, we will use the number of classes in the abstract syntax tree of a language (Metric 6) as a measure for how large the language is. The reasoning is that the more classes an abstract syntax tree has, the more concepts the language contains, and the more effort it is to learn and use the language.

[4] 'Source lines of code' refers to the number of source code lines that are neither comments nor contain only white space.

[5] For reasons of brevity, I will refer to all code that can be run on the JVM and was written in a multi-purpose programming language as 'Java code'; even though some of the code was written in the Java dialect Xtend.

In effect, the combination of Metric 5 and Metric 6 tests how well the languages make use of the strengths of domain-specific languages (cf. Section 2.1). The language that is better adapted to the domain should require less code to solve the problem, without having to introduce many different concepts to the language. Notably, since the Commonalities Language allows to fall back to Reactions Language, the comparison will show to which extend it is beneficial to use the Commonalities Language *in addition* to the Reactions Language. Only if the Commonalities Language can be used well to implement the whole case study will the comparison yield insights into how beneficial it is the use the Commonalities Language exclusively.

4 Congruent Bidirectional Transformations

The two transformation directions $\vec{T}$ and $\overleftarrow{T}$ of a bidirectional transformation $\overleftrightarrow{T}$ are not uniquely defined by their consistency relation T. If T is not bijective, the directions have the freedom to choose among different target model states for a given input model state. For practical use, however, it is desirable that both directions of a transformations make use of this freedom in a compatible way. In this chapter, I will present a novel property for bidirectional transformations that formalizes this expectation: *congruence*. The property can guide transformation developers when building transformations. It ensures a notion of compatibility between the two directions of a bidirectional transformation and makes transformations more predictable for users. It will also become relevant in Chapter 5 for testing transformations.

Let us begin with an example: Java does not have a native construct to represent software components. To maintain consistency between PCM models and Java source code, transformation developers have to define a convention that specifies how to represent software components in Java. Langhammer [Lan17, p. 69], for instance, proposes to map every new PCM component to a new Java package containing an implementation class for the component. This is not the only possible solution, developers can choose among multiple possible conventions that would all be considered valid by users. Nevertheless, users expect that in any chosen solution, both directions of the transformation adhere to *the same* convention. To illustrate why this is important to users, imagine the following scenario: We use a consistency specification based on Langhammer's [Lan17, p. 69] convention for mapping components. However, our specification does not give details on how to handle the names of components or implementation classes, respectively (more on this assumption will follow in Section 4.1). Now assume we defined the following bidirectional transformation between PCM and Java: The PCM→Java transformation creates a package and a class for every component. The package has the name of the component in lowercase letters; and the class has the name of the component, with 'Impl' appended to it. The Java→PCM transformation creates a PCM component for every Java package containing a class ending with 'Impl'[1] and gives the component the name of the class. If an architect modelled a component called 'UserRepository' in PCM, the PCM→Java transformation will create the Java package 'userrepository' containing a class called 'UserRepositoryImpl'. If, however, a Java developer created a package called 'userrepository' and put a class named 'UserRepositoryImpl' into it, the Java→PCM transformation will create a component called 'UserRepositoryImpl'. We can see a discrepancy: When the developer applies the changes

[1] In practice, the transformation would need to differentiate more carefully whether the class should really represent a component, for example by asking the user for input. But this is not relevant to our discussion.

that resulted from the architect's changes manually (i.e. adds the package and the class), the resulting changes are different from the initial changes by the architect. The architect's component was named 'UserRepository', while the component that results from the developer making the corresponding changes in Java will be called 'UserRepositoryImpl'. Users will be surprised by this behaviour. If creating a component named 'UserRepository' results in a class named 'UserRepositoryImpl', then it stands to reason that creating a class named 'UserRepositoryImpl' creates a component called 'UserRepository'.

If we abstract from this example, we can observe that we expect bidirectional transformations to have 'interchangeable changes'. That means that if a user changes model a_0 to a_1 (e.g. adds a component in PCM) and the transformation transforms model b_0 to be b_1 as a consequence (e.g. adds a package and a class to Java); then we expect that if the user changes model b_0 to b_1, the transformation will change a_0 to be a_1. In other words: changes made by transformations should have the same effect as changes made by users.

Definition 4.1. Given a bidirectional transformation $\overleftrightarrow{T} =: (T, \overrightarrow{T}, \overleftarrow{T})$, a change from $(a_0, b_0) \in T$ to a_c is *user-transformation-interchangeable* if, and only if, transforming a_c by $\overleftrightarrow{T}$ leads to a new model b_c, and transforming b_c back by $\overleftrightarrow{T}$ results in a_c. More formally, $a_c \in A$ is user-transformation-interchangeable in state $(a_0, b_0) \in T$ iff

$$\overleftarrow{T}\left(a_0, \overbrace{\overrightarrow{T}(a_c, b_0)}^{b_c}\right) = a_c.$$

Dually, $b_c \in B$ is user-transformation-interchangeable in state $(a_0, b_0) \in T$ iff

$$\overrightarrow{T}(a_c, \overleftarrow{T}(a_0, b_c)) = b_c.$$

Definition 4.2. A bidirectional transformation $\overleftrightarrow{T} =: (T, \overrightarrow{T}, \overleftarrow{T})$ between A and B is *user-transformation-interchangeable* if, and only if, all changes are user-transformation-interchangeable. More formally, $\overleftrightarrow{T}$ is user-transformation-interchangeable iff

$$\begin{aligned}\forall (a_0, b_0) \in T\colon \quad & \forall a_c \in A\colon \overleftarrow{T}(a_0, \overrightarrow{T}(a_c, b_0)) = a_c \\ & \land \forall b_c \in B\colon \overrightarrow{T}(\overleftarrow{T}(a_0, b_c), b_0) = b_c.\end{aligned}$$

If a user-transformation-interchangeable transformation between A and B is also Hippocratic and correct, it is uniquely defined by a bijection between A and B. We will prove this later in Proposition 4.1 (on page 29). It justifies calling the whole transformation 'bijective'.

Definition 4.3. A correct, Hippocratic and user-transformation-interchangeable transformation is called *bijective*.

It would, however, not be realistic to expect all transformations to consist exclusively of user-transformation-interchangeable changes (and, hence, be bijective) [Ste10]. To see why, we extend our example above. Langhammer suggests that, because his proposed mapping 'is hard to match for [Java] developers' [Lan17, p. 70], a component could already be created in PCM after a new Java package was created[2]. This is a sensible proposal; but the resulting

[2]Langhammer proposes to ask the user whether the new package should, in fact, represent a new component. But once again, this is not relevant for our discussion.

transformation will not be user-transformation-interchangeable. After adding a package in Java, for example, the new transformation will create a component in PCM; but after adding a component in PCM, the new transformation will create a package *and* a class in Java. Hence, adding a package in Java is not user-transformation-interchangeable. Nevertheless, we observe that the change, although not being user-transformation-interchangeable itself, *resulted* in a user-transformation-interchangeable change. As users, we can still accept that: Adding a package is an alias for adding a package and a class. Hence, we are not surprised that the PCM→Java transformation would create a package and a class in reaction to a new PCM component, even though we just added a package to create a PCM component.

Abstracting from our extended example again, we derive that we should tolerate not only user-transformation-interchangeable changes, but also changes that lead to user-transformation-interchangeable changes after being transformed once. I even argue that we should *only* tolerate such changes. In other words, we should build transformations in a way that they always produce user-transformation-interchangeable changes. Let us call such transformations *congruent*.

Definition 4.4. A bidirectional transformation $\overleftrightarrow{T} =: (T, \overrightarrow{T}, \overleftarrow{T})$ between A and B is *congruent* if, and only if, transforming any change results in a user-transformation-interchangeable change. More formally, $\overleftrightarrow{T}$ is congruent iff

$$\begin{aligned} \forall (a_0, b_0) \in T\colon \quad & \forall a_c \in A\colon \overrightarrow{T}\left(\overleftarrow{T}\left(a_0, \overrightarrow{T}\left(a_c, b_0\right)\right), b_0\right) = \overrightarrow{T}\left(a_c, b_0\right) \\ & \wedge \forall b_c \in B\colon \overleftarrow{T}\left(a_0, \overrightarrow{T}\left(\overleftarrow{T}\left(a_0, b_c\right), b_0\right)\right) = \overleftarrow{T}\left(a_0, b_c\right). \end{aligned}$$

Just as I implied, congruency is a weakening of user-transformation-interchangeability:

Lemma 4.1. *Every user-transformation-interchangeable transformation is congruent.*

Proof. Since $\overrightarrow{T}$ and $\overleftarrow{T}$ are deterministic, wrapping them around the Definition 4.2 retains equality and yields Definition 4.4:

$$\begin{aligned} & \forall (a_0, b_0) \in T\colon \quad && \forall a_c \in A\colon \overleftarrow{T}\left(a_0, \overrightarrow{T}\left(a_c, b_0\right)\right) = a_c \\ & && \wedge \forall b_c \in B\colon \overrightarrow{T}\left(\overleftarrow{T}\left(a_0, b_c\right), b_0\right) = b_c \\ \Longrightarrow \quad & \forall (a_0, b_0) \in T\colon && \forall a_c \in A\colon \overrightarrow{T}\left(\overleftarrow{T}\left(a_0, \overrightarrow{T}\left(a_c, b_0\right)\right), b_0\right) = \overrightarrow{T}\left(a_c, b_0\right) \\ & && \wedge \forall b_c \in B\colon \overleftarrow{T}\left(a_0, \overrightarrow{T}\left(\overleftarrow{T}\left(a_0, b_c\right), b_0\right)\right) = \overleftarrow{T}\left(a_0, b_c\right). \end{aligned}$$ □

Furthermore, we can conceptualize congruent transformations as user-transformation-interchangeable transformations that additionally allow alias changes. Alias changes lead to user-transformation-interchangeable changes when transformed (see Figure 4.1).

Lemma 4.2. *Every congruent transformation has user-transformation-interchangeable changes.*

Proof. Let $\overleftrightarrow{T} =: (T, \overrightarrow{T}, \overleftarrow{T})$ be a bidirectional transformation between the metamodels A and B. If we start with any consistent state $(a_0, b_0) \in T$ and transform any change $a_c \in A$

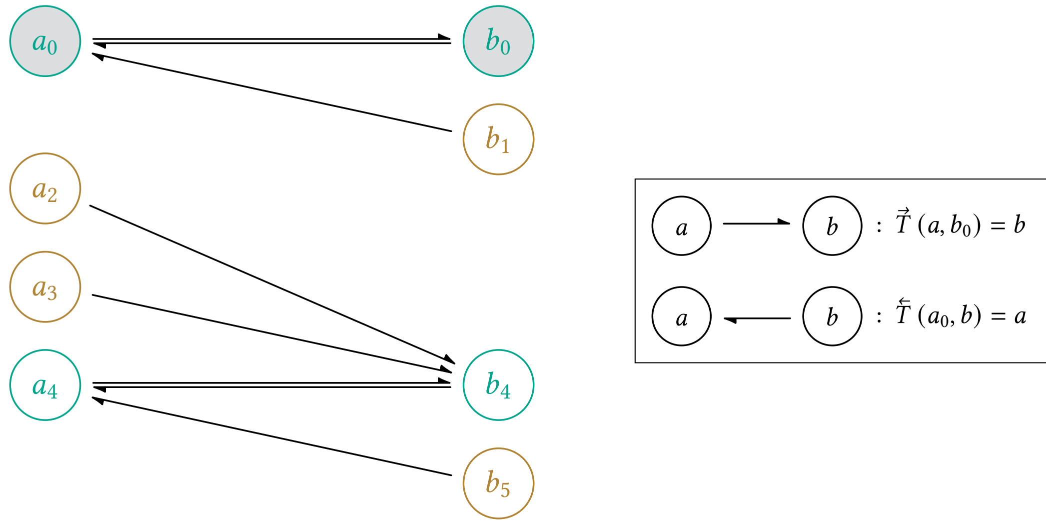


Figure 4.1: Relative to a consistent state pair $(a_0, b_0) \in T$ (grey background) of a congruent bidirectional transformation $\overleftrightarrow{T} =: (T, \vec{T}, \overleftarrow{T})$, any change to another model state is either a user-transformation-interchangeable change (green), or an alias change (brown). Alias changes are transformed into user-transformation-interchangeable changes by $\overleftrightarrow{T}$.

(or $b_c \in B$) to $b_t := \vec{T}(a_c, b_0)$ (or $a_t := \overleftarrow{T}(b_c, a_0)$), the result will be a user-transformation-interchangeable change. We can see this by filling the result into Definition 4.4, which directly yields Definition 4.2:

$$\vec{T}(\overleftarrow{T}(a_0, \vec{T}(a_c, b_0)), b_0) = \vec{T}(a_c, b_0) \implies \vec{T}(\overleftarrow{T}(a_0, b_t), b_0) = b_t$$
$$\overleftarrow{T}(a_0, \vec{T}(\overleftarrow{T}(a_0, b_c), b_0)) = \overleftarrow{T}(a_0, b_c) \implies \overleftarrow{T}(a_0, \vec{T}(a_t, b_0)) = a_t. \qquad \square$$

4.1 Relationship to the Consistency Relation

While considering the above example regarding consistency of software components between PCM and Java, we assumed that the consistency specification did not define how names of components and implementation class should be mapped onto each other. This was necessary to construct the example. If we assumed a more sensible consistency specification that specifies how to handle component names and implementation class names, we would not have been able to construct an example that is not congruent, but still correct and Hippocratic. This is consistent with the introduction of this chapter, where I described congruency as, intuitively speaking, a property of how transformations 'use the freedom' that the transformation relation leaves them. If there is no freedom, any correct transformation will also be congruent. Let us formalize this intuition.

We will begin with the strictest of consistency relations: bijective ones. If the consistency relation is bijective, then there is no freedom at all and any correct bidirectional transformation will be user-transformation-interchangeable. Conversely, all user-transformation-interchangeable transformations that are also Hippocratic have bijective consistency relations.

Proposition 4.1. *There is a correct, Hippocratic, and user-transformation-interchangeable bidirectional transformation for a consistency relation T if, and only if, T is bijective.*

Proof. Let A and B be the metamodels that are related by T. Recall that neither A nor B is empty (cf. Section 2.2).

'$\Longrightarrow$' Assume that T is not bijective. T is left-total and right-total per definition (see Section 2.4.1). Hence, T is either not left-unique or not right-unique. Assume without loss of generality that T is not left-unique (else swap the roles of A and B). Hence, there are $(a_1, b) \in T$ and $(a_2, b) \in T$ with $a_1 \neq a_2$. Let $\overleftrightarrow{T} = (T, \overleftarrow{T}, \overrightarrow{T})$ be any correct and Hippocratic transformation for T. By its Hippocraticness, we know that $\overrightarrow{T}(a_1, b) = b$ and $\overleftarrow{T}(a_2, b) = a_2$. Consequently, we see that

$$\overleftarrow{T}(a_2, \overrightarrow{T}(a_1, b)) = a_2 \neq a_1.$$

Thus, $\overleftrightarrow{T}$ is not user-transformation-interchangeable.

'$\Longleftarrow$' Let T be bijective. The only correct bidirectional transformation is the induced transformation $\overleftrightarrow{T} = (T, \overrightarrow{T}, \overleftarrow{T})$, with

$$\begin{aligned} \overrightarrow{T}&\colon\ (a, b) \mapsto T(a) \\ \overleftarrow{T}&\colon\ (a, b) \mapsto T^{-1}(b)\,. \end{aligned}$$

$\overleftrightarrow{T}$ is well-defined, correct and Hippocratic [Ste10]. It is also user-transformation-interchangeable:

$$\begin{aligned} \forall (a_0, b_0) \in T\colon \quad & \forall a_c \in A\colon \overleftarrow{T}(a_0, \overrightarrow{T}(a_c, b_0)) = T^{-1}(T(a_c)) = a_c \\ & \wedge \forall b_c \in B\colon \overrightarrow{T}(\overleftarrow{T}(a_0, b_c), b_0) = T\left(T^{-1}(b_c)\right) = b_c. \end{aligned}$$

□

Because congruency is strictly weaker than user-transformation-interchangeability, there are consistency relations that do not allow the latter, but do allow the former. To get a better understanding of which consistency relations allow congruency, we define 'indirect consistency'.

Definition 4.5. Given a consistency specification $T \subseteq A \times B$ between A and B, we define the additional relations for all $i \in \mathbb{N}^+$:

$$\begin{aligned} T^0 &:= T \\ T^i &:= \left\{(a, b) \mid \exists (a', b') \in \left(T^{i-1} \setminus \{(a, b)\}\right) :\ (a, b') \in T \wedge (a', b) \in T\right\} \\ T^* &:= \bigcup_{n \in \mathbb{N}_0} T^n \end{aligned}$$

We say that $a \in A$ and $b \in B$ are *i-step consistent* iff $(a, b) \in T^i$, and *indirectly consistent* iff $(a, b) \in T^*$.

Congruency, just like user-transformation-interchangeability can be trivial:

Proposition 4.2. *Let $T \subseteq A \times B$ be a consistency relation between the metamodels A and B. If there is no pair of model elements $(a, b) \in A \times B$ that is 2-step consistent, then any correct, Hippocratic bidirectional transformation between A and B is congruent.*

Proof. Let T have no model pair that is truly 2-step consistent. Assume to the contrary that there exists a bidirectional transformation $\overleftrightarrow{T} := (T, \vec{T}, \overleftarrow{T})$ that is correct and Hippocratic, but not congruent. Because $\overleftrightarrow{T}$ is not congruent, we know that there exist $(a_0, b_0) \in T$ such that either

$$\exists a_c \in A\colon \underbrace{\vec{T}\left(\overbrace{\overleftarrow{T}\left(a_0, \underbrace{\vec{T}\left(a_c, b_0\right)}_{b_c}\right)}^{a_x}, b_0\right)}_{b_x} \neq \underbrace{\vec{T}\left(a_c, b_0\right)}_{b_c};$$

or

$$\exists b_c \in B\colon \overleftarrow{T}\left(a_0, \vec{T}\left(\overleftarrow{T}\left(a_0, b_c\right), b_0\right)\right) \neq \overleftarrow{T}\left(a_0, b_c\right).$$

Let, without loss of generality, the former be the case (else swap the roles of A and B). Let $b_c := \vec{T}(a_c, b_0)$, $a_x := \overleftarrow{T}(a_0, b_1)$ and $b_x := \vec{T}(a_x, b_0)$, as indicated in the formula above. Per assumption, we know that $b_c \neq b_x$. It follows that $a_x \neq a_c$, because otherwise we would have $\vec{T}(a_x, b_0) = \vec{T}(a_c, b_0)$, which violates our assumption. Since $\overleftrightarrow{T}$ is correct, we conclude that $(a_c, b_c) \in T$, $(a_x, b_c) \in T$, and $(a_x, b_x) \in T$. But then a_c and b_x are 2-step consistent; that is, $(a_c, b_x) \in T^2$. This is a contradiction. □

Notably, this means that every lens is congruent:

Lemma 4.3. *Every well-behaved lens is congruent.*

Proof. Let $\overleftrightarrow{L} =: (L, \vec{L}, \overleftarrow{L})$ be a well-behaved lens between A and B; that is, $\overleftrightarrow{L}$ is correct and Hippocratic, and there exists a function $\textsc{Get}\colon A \to B$ such that $\vec{L}(a, b) = \textsc{Get}(a)$ for all $a \in A$ and $b \in B$. We show that L has no 2-step consistent pairs. Hence, assume to the contrary that $a \in A$ and $b \in B$ are 2-step consistent. Hence, there are $(a', b') \in L$, such that $(a, b') \in L$ and $(a', b) \in L$. Furthermore $b \neq b'$. But then $\overleftrightarrow{L}$ is not Hippocratic, because $\vec{L}(a', b) = \textsc{Get}(a') = \vec{L}(a', b')$, so either $\vec{L}(a', b) \neq b$ or $\vec{L}(a', b') \neq b'$. This is a contradiction. □

By now, we have found that, depending on the consistency relation, user-transformation-interchangeability can be impossible, and congruency can be trivial. Now we will find a relaxed, sufficient condition for the existence of a congruent transformation for a given consistency relation. We will show later, in Example 7.1 on page 61, that the condition is sufficient, but not still not necessary. Nevertheless, there are also consistency relations for which there are no congruent, Hippocratic, bidirectional transformations; even though there is a correct and Hippocratic one. Example 7.2 on page 63 provides such a consistency relation.

Proposition 4.3. *Let $T \subseteq A \times B$ be a between two metamodels A and B. If any indirectly consistent pair of models $(a, b) \in A \times B$ is also consistent, then there is a correct, Hippocratic and congruent bidirectional transformation $\overleftrightarrow{T} =: (T, \vec{T}, \overleftarrow{T})$.*

Proof. We construct such a bidirectional transformation. T has subsets that form a left-unique, right-unique (but not necessarily bijective) relation. Let S be a maximal of these subsets (that is, no tuple $t \in T$ can be added to S without making it not left-unique or not

right-unique). Recall that neither A nor B is empty and that T is left-total (cf. Section 2.2). Thus, S is also not empty. We define the transformations as

$$\vec{T}(a,b) = \begin{cases} b & \text{if } (a,b) \in T \\ b_s & \text{if } (a,b_s) \in S \\ b_t & \text{where } (a,b_t) \in T \end{cases}$$

$$\overleftarrow{T}(a,b) = \begin{cases} a & \text{if } (a,b) \in T \\ a_s & \text{if } (a_s,b) \in S \\ a_t & \text{where } (a_t,b) \in T \end{cases}$$

We can see directly from the definition that $\overleftrightarrow{T}$ must be Hippocratic (because of the respective first case) and correct (because we always select a result from T). Since T is left-total and right-total, the functions are also well-defined. We will show one direction of congruency, the other direction is dual and can be shown analogously.

Let $(a_0, b_0) \in T$ and $a_c \in A$. We differentiate three cases:

1. Assume $(a_c, b_0) \in T$. We get:

$$\begin{aligned} &\vec{T}\left(\overleftarrow{T}\left(a_0, \vec{T}\left(a_c, b_0\right)\right), b_0\right) \\ =\ &\vec{T}\left(\overleftarrow{T}\left(a_0, b_0\right), b_0\right) \\ =\ &\vec{T}\left(a_0, b_0\right) \\ =\ &b_0 \\ =\ &\vec{T}\left(a_c, b_0\right). \end{aligned}$$

2. Else, assume that there is a $b_c \in B \setminus \{b_0\}$ such that $(a_c, b_c) \in S$. Since we are not in case 1, we know that $(a_c, b_0) \notin T$ and, hence, $\vec{T}(a_c, b_0) = b_c$. We also know that $(a_0, b_c) \notin T$. Because else, a_c and b_0 would be indirectly consistent via (a_c, b_c), (a_0, b_c), (a_0, b_0); although a_c and b_0 are not consistent. Hence, $\overleftarrow{T}(a_0, b_c) = a_c$. We get:

$$\begin{aligned} &\vec{T}\left(\overleftarrow{T}\left(a_0, \vec{T}\left(a_c, b_0\right)\right), b_0\right) \\ =\ &\vec{T}\left(\overleftarrow{T}\left(a_0, b_c\right), b_0\right) \\ =\ &\vec{T}\left(a_c, b_0\right). \end{aligned}$$

3. Else, let $b_c := \vec{T}(a_c, b_0)$. Since we are neither in case 1 nor 2, we know that $(a_c, b_c) \notin S$ and $b_c \neq b_0$. We also know that there is no other $b_x \in B$ such that $(a_c, b_x) \in S$ because else we would be in case 2. It follows that there must be an $a_s \in A$ such that $(a_s, b_c) \in S$. Otherwise, S would not be maximal because we could add (a_c, b_c) to it. By the same argument as in case 2, $(a_0, b_c) \notin T$. Hence, $\overleftarrow{T}(a_0, b_c) = a_s$ and $(a_s, b_c) \in T$. Similarly, $(a_s, b_0) \notin T$, because else a_c and b_0 would be indirectly consistent via (a_c, b_c), (a_s, b_c) and (a_s, b_0); although a_c and b_0 are not consistent.

Hence, $\vec{T}(a_s, b_0) = b_c$. We get:

$$\begin{aligned}
&\vec{T}\left(\overleftarrow{T}\left(a_0, \vec{T}\left(a_c, b_0\right)\right), b_0\right) \\
&= \vec{T}\left(\overleftarrow{T}\left(a_0, b_c\right), b_0\right) \\
&= \vec{T}\left(a_s, b_0\right) \\
&= b_c \\
&= \vec{T}\left(a_c, b_0\right).
\end{aligned}$$

□

The condition that indirect consistency shall imply consistency is a natural one. For example, imagine that we did not represent UML comments in Java. Hence adding a comment to a UML model does not change whether or not the model is consistent with Java. Similarly, the body of Java methods is not represented in UML by Langhammer's consistency specification. Hence, changing the body of a constructor does not make a consistent Java model inconsistent with UML. Now consider consistent Java and UML model states, both representing a software component. We know that, first, the Java state with a different constructor body is consistent with the UML state, and, second, the UML state with an added comment is consistent with the Java state. Given these facts, it is only reasonable to expect that the Java state with a different constructor body is also consistent with the UML state with an added comment . Otherwise, a correct bidirectional transformation would be forced to remove the modified constructor body after the comment was added to Java. This is obviously not desirable.

4.2 Evaluation

In this chapter, I introduced a novel property for bidirectional transformations, which I called congruency. I propose that additionally to correctness and Hippocraticness, we should expect congruency of all transformations that are used for maintaining consistency between models with semantic overlap. I have explained with examples why I expect transformations to become more predictable and easier to use if they are user-transformation-interchangeable. Then I showed why this is too strict to be a general requirement, but that we can still gain the proposed advantages if we require transformations to be congruent. Additionally, I will show in Chapter 5 how congruent transformations are easier to test than non-congruent ones.

There are three reasons why I think that it is realistic to expect that it will always be possible to use congruent transformations when maintaining consistency of models with semantic overlap. First, as shown in Proposition 4.3, every consistency relation can be maintained by a congruent bidirectional transformation if all indirectly consistent model pairs are also consistent with each other. As I explained, this is a intuitive requirement and it is reasonable to expect that all practical transformation relations will fulfil it. Second, congruency maps cleanly onto the idea of semantic overlap. Two models sharing semantic overlap means that there is a common concept that can be represented in both models. For every initial model state, we can designate a pair of a representations in each model that map onto each other. This pair will be the respective user-transformation-interchangeable change for each model. The pair would be in the subset S in the proof of Proposition 4.3.

In our example above, the state after adding a PCM component and the state after adding a Java package with a Java class would form such a pair. Additionally, congruency allows aliases for these states in both models. These aliases are either other ways to express the same concept, like the simplification of only adding a Java package, that we discussed; or they capture additional model data which is irrelevant to the common concept, like comments in UML. All such aliases would be in $T \setminus S$ in the proof of Proposition 4.3. This way, semantic overlap can naturally lead to a congruent bidirectional transformation. Last, as far as I was able to test the transformations in the case study of this thesis, they were all congruent or meant to be congruent. This gives the previous two theoretical arguments some early empirical validation. Nevertheless, more practical experience will be needed to be sure that congruency is a good requirement for all bidirectional transformations when maintaining consistency between models with semantic overlap.

5 A Test Strategy for Different Implementations of a Consistency Specification

A part of the case study presented in Chapter 3 compares two implementations of the same consistency specification, one in the Reactions Language, and the other in the Commonalities Language. As discussed in Section 3.3, it is crucial for the internal validity of the case study that the compared implementations both implement the same behaviour, and that this behaviour conforms to the specification. However, when examining the pre-existing implementation in the Reactions language by Chen [Che17], Klatte [Kla17] and Syma [Sym18] and the implementation in the Commonalities Language by Hennig [Hen20], it quickly became apparent that they did not realise the same logic. Instead, they had interpreted Langhammer's requirements [Lan17] differently. The implementations were covered by tests, but those tests were written for each implementation separately. Hence, the tests could not detect the discrepancies between the implementations. To ensure that both implementations fulfil the requirements to be used in the case study, I developed a test strategy and a framework for a test suite that can be applied to both implementations. In this chapter, I will present and discuss the design decisions for this new test suite.

5.1 Black Box Acceptance Tests

Software test cases are typically categorized into at least two categories: Whether they are unit, integration, or system tests; and whether they apply black box, white box, or grey box testing. The first category concerns the subject of the tests: either a small unit of the software in isolation, or the interplay of several units, or the whole software system at once. The second category differentiates how much knowledge about the implementation is used to define the test cases. In white box testing, any information about the implementation can be used to improve the test cases and focus them on the areas that are deemed in particular need of testing. In black box testing, the opposite is the case: No knowledge about the implementation is used, and test cases are defined only based on information that users have about the system. The software system is, hence, treated as a 'black box'. In grey box testing, only specific information about the implementation is used to define test cases.

The pre-existing tests for the Reactions Language implementation are mostly white box unit tests. They focus on small parts of the transformations and test the properties that developers considered most relevant. The tests are accompanied by some white box

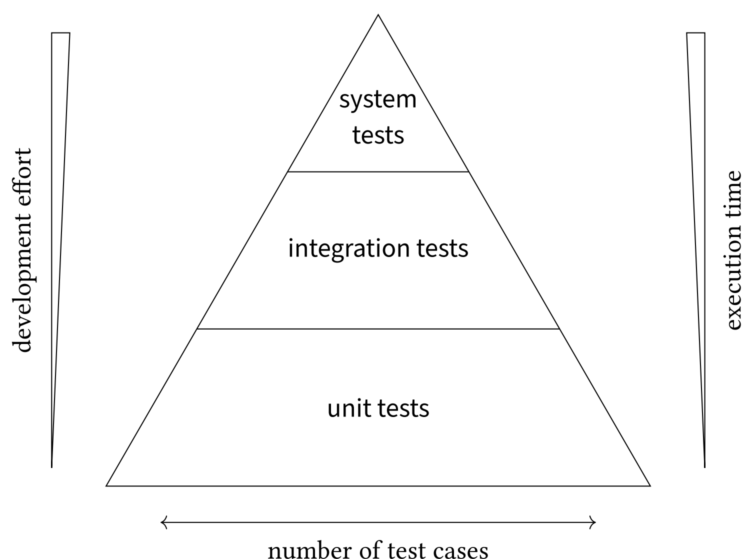


Figure 5.1: The Test Pyramid. Because tests at higher levels require more effort to develop and take longer to be executed, there should be more test cases at lower levels than at higher levels. To achieve this, tests at higher levels can rely on the behaviour that was already tested at lower levels. This illustration is analogous to the one by Fowler [Fow12].

integration tests, which cover relevant behaviour when combining multiple Reactions. The pre-existing tests for the implementation in the Commonalities Language are different. They test the transformations by providing user changes and checking the result models after applying the transformation. Since they test the whole transformation from the users' perspective, they are black box system tests. The new tests suite for the case study will check the whole respective implementation against its specification. Hence, it is an acceptance test suite. Acceptance tests are a subcategory of system tests. The test suite will be applied to two significantly different implementations, so it can not rely on implementation details. It must, thus, be a black box test suite.

When defining a test strategy, it is usually recommended to combine unit, integration and system tests such that they form an imagined pyramid, the so-called 'Test Pyramid' (cf. Figure 5.1) [Coh09; Fow12]. Unit tests form the foundation of the pyramid. They are the cheapest tests to develop, the fastest to execute, and they provide the most actionable information when they fail [Coh09]. The middle layer of the pyramid contains integration tests. These tests focus on whether the software units can be properly integrated with each other. The integration tests can rely on the behaviour that has already been asserted by the unit tests. Thus, they only need to test integration, and do not need to check the behaviour of individual units. The top layer of the Test Pyramid is formed by system tests. They can, once again, rely on the behaviour that has been asserted on the layers below and only need to check the remaining requirements for the system as a whole. Since tests typically get more expensive to develop, take longer to execute, and provide less information to debug errors as we progress from unit, to integration, to system tests; adhering to the Test Pyramid promises a better use of resources and more robust tests [Coh09].

The test suite for the case study cannot contain unit tests or integration tests that test both implementations at the same time. The two implementations are too heterogeneous to allow breaking them into parts among the same lines. Thus, the only category of test cases that can be executed against both implementations are system tests. There can, of course, be individual unit and integration tests for each implementation. However, they run the risk of interpreting the consistency specification slightly differently. This is what already happened with the pre-existing tests. Hence, the case study acceptance tests should *not* rely on unit or integration tests. This way, the consistency specification only gets interpreted once (instead of twice, once per implementation), reducing the potential for errors. Furthermore, should any misinterpretation of the specification occur, it will affect both implementations in the same way, giving none an advantage in the case study.

Without any lower level tests to rely on, the case study acceptance test suite needs to contain a high number of test cases to cover the specification well. For reference, the existing test suite covering the Reactions Language implementation contained 511 test cases. Nevertheless, developers generally agreed that the test suite was not sufficiently comprehensive and needed to cover more cases. Once a comprehensive test acceptance test suite has been created, it will be desirable to remove the implementation-specific tests that cover the same cases as the acceptance test suite. Implementation-specific tests will only make sense to check implementation-specific invariants or functionality of implementation-specific helpers. Keeping implementation-specific tests that check the same functionality as the acceptance test suite would mean a higher maintenance effort. However, the resulting test strategy—elaborate system tests and only few unit and integration tests—will then be the opposite of what is recommended with the Test Pyramid pattern. Such a strategy—dubbed the 'Test Ice Cream Cone' [Fow12]—is considered an anti-pattern for the same reasons that the Test Pyramid is recommended.

Given the above, we can derive the following requirements for the acceptance test suite:

Test Requirement 1 The same acceptance test suite covers all relevant aspects of the consistency specification.

Test Requirement 2 The same acceptance test suite can be applied to the implementation in the Commonalities Language and the implementation in the Reactions Language.

Test Requirement 3 It must be easy to write and maintain acceptance test cases, such that it does not require significantly more effort than writing unit tests.

Test Requirement 4 The acceptance tests must provide good debug information in case of failures, such that developers can understand the issue well and quickly.

Test Requirement 5 The test suite implementation must be performant, such that hundreds of test cases can be executed in only a few seconds.

If requirements , 3, 4, and 5 are fulfilled, the test suite will not suffer from not following the Test Pyramid pattern. As Fowler puts it: 'The pyramid is based on the assumption that broad-stack tests are expensive, slow, and brittle compared to more focused tests, such as unit tests. While this is usually true, there are exceptions. If my high level tests are fast, reliable, and cheap to modify—then lower-level tests aren't needed' [Fow12]. The following two sections will describe how the test suite can fulfil its requirements.

5.2 Exploiting Congruency

In Chapter 4, I introduced congruency and explained why we should expect it of bidirectional transformations. In this section, I will describe how we can use congruency to make specifying test cases easier. This contributes to realizing Test Requirement 3. At the same time, the technique tests whether the transformations are congruent.

When writing a (black box) acceptance test for a transformation network, the general procedure is as follows: Start with a consistent state of the models, apply a user change, transform the changed models through the network, and check that the resulting models are in the expected state afterwards. To write a test case, we need to specify three model states per transformation: The initial state, the state after the user change, and the expected state after applying the transformations. This effort can be reduced if we remember that bidirectional transformations are correct. We can use the result state of other test cases—which must be consistent—as the initial state for new test cases. The first test cases will simply start with empty models. With this technique, we only need to specify two model states per transformation.

Congruent bidirectional transformations have two type of changes: user-transformation-interchangeable changes and alias changes (see Lemma 4.2 and Figure 4.1). If we transform a user-transformation-interchangeable change in one direction, we know that the transformation result can also be an input change for the other direction, which should, after being transformed, result in the initial input change. By specifying two model states, one for each metamodel of the bidirectional transformation, we can, hence, derive two test cases. Apart from user-transformation-interchangeable changes, congruent transformations only have alias changes. The result of transforming an alias change is a user-transformation-interchangeable change (Lemma 4.2). Hence, to test an alias change, we only need to specify the alias change and can re-use the model specifications of the resulting user-transformation-interchangeable changes from other test cases. Altogether, we can follow the following procedure to specify test cases for bidirectional transformations:

1. Specify an initial state of the models by either referencing the result of another test case, or by using empty models.
2. Derive from the consistency specification a user-transformation-interchangeable change to the initial state.
3. Program the user-transformation-interchangeable change and the result change of transforming it.
4. Derive from the consistency specification relevant alias changes for the user-transformation-interchangeable change.
5. Program the alias changes.
6. Let a test framework derive one test case for each changed model state we have specified.

Figure 5.2 shows schematically how a test plan can be derived from specified input model states. In effect, we need to specify one model state per transformation and test case. This halves the amount of model states that need to be specified compared to if we did not

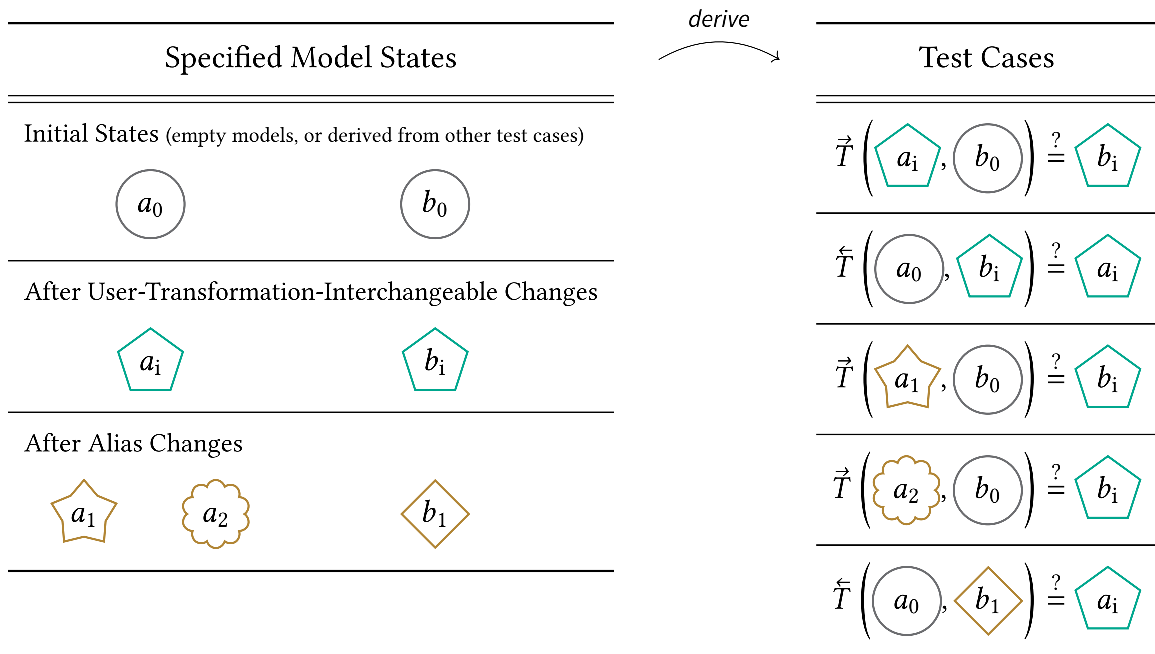


Figure 5.2: A schematic test plan exploiting congruency. Left: Examples for the initial and the changed model states of the metamodels A and B. Right: The test plan that will be derived for those model states. It tests the bidirectional transformation $\overleftrightarrow{T} =: (T, \vec{T}, \overleftarrow{T})$ between A and B.

make use of congruency. Furthermore, all user-transformation-interchangeable changes coincided between the transformations of the tested transformation network. That is, a change to the metamodel B that was user-transformation-interchangeable for some transformation $\overleftrightarrow{T}$ between A and B was also always user-transformation-interchangeable any transformation $\overleftrightarrow{U}$ between B and C. This reduces the effort for specifying test cases even further.

5.3 A Test Framework for the Test Strategy

We saw in the previous section that test cases for the case study acceptance test suite can have a specific structure. Test cases can be defined by specifying which other test case to use to obtain the initial model states from, then providing the model states, and finally marking whether the model states are user-transformation-interchangeable changes or alias changes. Since a lot of acceptance test cases will have to be written (see Section 5.1), it is worthwhile to develop a specialized test framework that derives the test cases automatically from this input, and reduces the boilerplate code that needs to be written to a minimum. The test framework needs to setup the Vitruvius framework, obtain the initial states from the previous test case, apply the input model state, apply the tested transformations, compare the transformation result to the other model states, and fail the test if any differences are found. I implemented the test framework as part of this thesis. In this section, I will

discuss the framework's design, focussing on how it helps fulfilling the test requirements we found in Section 5.1.

5.3.1 Specifying Model States

The case study test suite must implement the consistency specification by Langhammer [Lan17] correctly, otherwise the case study's validity will be harmed. This is Test Requirement 1. The importance of this requirement is underlined by the fact that the pre-existing tests already had interpreted the consistency specification differently. One way to help avoiding mistakes in the test suite is to ensure that the tests are easy to read. The easier developers can understand written test cases, the less likely it becomes that discrepancies to the consistency specification will be missed. When programming changes for the test cases, using EMF's API directly to create and modify model objects leads the programmer to write in an imperative style. The resulting code is verbose and does not reflect the structure of the resulting model well. To address this, the test framework provides an annotation that uses Xtend's code generation features to generate helpers for creating model objects. If used together with Xtend's lambda functionality, developers can use the helpers to program model changes in a declarative style. The resulting code is significantly shorter and represents the structure of the model objects. This should make it easier to read the specified changes and verify their correctness.

5.3.2 Comparing Model States

After the input changes have been transformed, the test framework needs to compare the transformation results to the expected model states. When writing unit test, developers usually specify tailored assertions, verifying that relevant portions of the result model are correct. These assertions can be very specific. If they fail, the failure in and of itself already provides a lot of information about what went wrong. Plus, developers can provide additional context-dependent information with the assertion, further helping to debug the issue. This approach is not available to the test framework because it is agnostic to the specific changes that are being verified. Instead, the test framework needs to compare the whole result models with the whole expected models. This raises two challenges: First, model comparison needs to account for specifics of the metamodels whose instances are being compared. Second, model comparison needs to provide rich error messages. To fulfil Test Requirement 4 (debug information), the result of the model comparison cannot simply be 'pass' or 'fail'. After a test failure, this would force developers to manually examine the differences between the whole models. Instead, the comparison needs to provide a detailed and readable description of the differences between the models that failed the tests. This section will describe how the framework compares models and finds differences, while the next section will explain how the framework generates readable messages.

Comparing EMF models is not a straightforward task, but 'intrinsically complex' [BP08]. For instance, the comparison needs to account for different types of features, like derived features, identifying attributes, containment references and non-containment attributes. The EMF Compare project implements a metamodel-agnostic and extensible algorithm for comparing EMF models [BP08]. The test framework uses EMF Compare to compare the

transformed model states against the expected ones. However, without further adaptions, EMF Compare generates false negative results; i.e., marks models as different even though they should be considered equal. Hence, the framework uses EMF Compare's extensibility to make the following adaptions to its comparison algorithm:

- Consider the roots of model trees to be equal. Otherwise, EMF Compare will not report detailed differences between the models.
- Ignore ordering changes of unordered references, even if they are containment references. EMF Compare reports ordering changes of unordered containment references per default, which can lead to false negatives.
- Allow ignoring certain features during the comparison. For example, the Java metamodel contains layout information which cannot be reproduced reliably and which is irrelevant for the semantics of the model. The PCM metamodel contains identifiers which are randomly assigned and can thus differ between otherwise semantically equal models. Without ignoring such features the comparison can yield false negatives.
- Allow specifying how model objects that are referenced via non-containment references should be compared. This is relevant if the referenced objects have no unique identifying attribute (like in Java) or if the identifying attribute cannot be used for the comparison (like in PCM). Without these specifications, the comparison can be false positive or false negative.

5.3.3 Model Printing

The acceptance test checks compare whole model trees, which can become quite large even for simple test examples. We expect the tests to give developers detailed and actionable, yet easy to read information about test failures (Test Requirement 4). To achieve this, the test framework introduces its own human-readable notation for model objects. The notation aims to make scanning the printed objects as easy as possible. To that end, it establishes the following conventions:

- Every model feature and every collection item is printed on its own line. The only exception are collections with only one element, which are kept on a single line to avoid unnecessary visual clutter.
- The identifying feature of a model object is always printed first, since it establishes the object's identity and allows to detect quickly whether the object has been moved.
- Do not print unset features, since this conveys very little information, while taking up much space for objects with many features.
- Print objects with all their features if they are in a containment reference, since the features are being compared. Print other referenced objects only in a form that establishes their identity, since those objects' features are not being compared.

```
OrdinaryParameter#1(
    name="clients"
    typeReference=NamespaceClassifierReference#1(
        classifierReferences=[ClassifierReference#1(
            typeArguments=[QualifiedTypeArgument#1(
                typeReference=NamespaceClassifierReference#2(
                    classifierReferences=[ClassifierReference#2(
                        target=Class#1
                    )]
                )
            )]
            target=Interface#1
        )]
    )
)
```

```
OrdinaryParameter#1(
    name="clients"
    typeReference=java.util.List<org.example.Client>
)
```

Figure 5.3: The model of a Java parameter printed without and with custom formatting. Top: The default model printing algorithm prints type references verbosely and hides relevant information. Bottom: With custom formatting, the parameter's type can be read easily.

One particularity for the notation are objects whose metaclass does not define an identifying feature. Developers need to understand whether two instances of such objects were considered equal by the comparison algorithm, and there should also be a way to print short references to such objects when they appear in non-containment references. Hence, the model printing facility assigns artificial identifiers to such objects and uses the same identifier for all instances that are considered equal by the comparison algorithm.

To improve the model notation even further, the test framework allows to customize the representation of specific metaclasses. This makes sense if the default notation would be verbose and there is a sensible, domain-specific, shorter notation. For example, references to other classes in Java are represented using two nested objects, with the inner object pointing to the class. If the type reference involves type parameters, the model structure representing it gets even more involved. Instead of printing this structure directly, we can replace it with the fully qualified type reference, like we would write in Java. Figure 5.3 shows a comparison. When providing custom notations, developers must take care not to hide details which might become relevant in model comparisons.

After a test failure has occurred, the test framework first prints the expected model tree using the notation described above. Afterwards, it prints a list of all differences between the expected and the actual model state. The differences give a reference path to the affected objects. Developers can follow the path from the model root to identify the affected object. Additionally, the affected object's identifier is printed. After the list of differences, the framework prints the model state of the transformed object using the notation described above. The expected interaction with this error message is that developers first skim over

the expected model to understand what the test case was testing. Afterwards, they read the detailed differences to understand what exactly went wrong. Finally, they can consult the transformed object to get more context for the detected differences.

5.3.4 Performance

Having addressed development effort (Test Requirement 3) and detailed debug information (Test Requirement 4), the remaining potential disadvantage of the test strategy's 'ice cream cone' is execution time (Test Requirement 5). To estimate whether the acceptance test suite will run considerably longer than a unit test suite, I conducted a performance benchmark comparing both suites. The benchmark measured the time it took to execute n test cases for different values of n. This allows us to build a linear regression model from which we can extrapolate how long executing a higher number of test cases would take.

The benchmark was executed using JMH [Ale+21], a state-of-the-art benchmarking tool for the JVM [Ste+17]. For each test suite s and each number n of test cases, 20 measurements were taken. Each measurement executed the respective test cases repeatedly for 60 seconds. The measurements were preceded by 10 warm-up iterations of 20 seconds each. The warm-up and measurement iterations were all executed in the same JVM instance. Both test suites have a startup time of multiple seconds, caused by initializations in the underlying frameworks and libraries. This startup time affects both suites equally, but would add considerable noise to the measurements. By using exactly one JVM instance per (s, n) combination, this effect was avoided and the initializations only affected the warm-up iterations, while the measurements captured the net execution time. The benchmark was executed on a machine equipped with an Intel® Core™ i7-7500U CPU, having 4 virtual cores with a base frequency of 2.7 GHz. The benchmark was running in the OpenJDK 64-Bit Server VM 18.9, build 11.0.9.1+1, on the Ubuntu 20.10 operating system.

The results of the benchmark are shown in Figure 5.4. The standard error of the mean was less than 2 milliseconds for each (s, n) measurement series. The p-value of the linear regressions is less than 1 ‰ for both test suites, indicating very high correlation. The regressions predict an execution time for 1000 test cases of 13.1 seconds for the acceptance test suite and 6.4 seconds for the unit test suite. Both test suites had an execution time that was approximately proportional to the number of test cases. The regressions predict that the execution time for the acceptance test suite is approximately double as long as the execution time of the unit test suite.

The validity of this performance evaluation is primarily threatened by the small sets of test cases that the benchmark could operate on. To ensure that we compare comparable test cases, I selected a single transformation as test subject, namely the PCM↔Java transformation written in the Reactions Language. The acceptance test suite contained 26 test cases for this transformation. The unit test suite contained 15 test cases testing behaviour that was also tested by the acceptance test suite. I grouped the test cases by the concept they test, namely consistency of PCM repositories, PCM components, and PCM systems. This resulted in three groups, with $n \in \{9, 19, 26\}$ for the acceptance test suite and $n \in \{3, 12, 15\}$ for the unit test suite. Grouping the test cases in more or other groups would have resulted in misleading results, since some test cases take inherently longer than others. For example, renaming a component requires creating a component first, so

Figure 5.4: Execution time measurements for the acceptance test suite (green) and the unit test suite (brown). The linear regressions are superimposed with dashed lines.

the test for renaming takes inherently longer than the test for creation. The grouping I chose assured that each group was equally affected by such effects.

Even with the aforementioned limitations, the performance analysis allows us to draw approximate conclusions about the performance impact of the presented test strategy. A doubled execution time per test case is noticeable, but acceptable. We should take into account that an acceptance test case performs more checks than a unit test case. Hence, we can expect the acceptance test suite to require less test cases to check the same behaviour. The absolute predicted value of 13.1 seconds for 1000 test cases (plus framework initialization) is also acceptable. This value was predicted for serial execution. The test cases can be parallelized to improve the absolute execution time.

5.3.5 Conclusion

We can see everything discussed in this chapter in effect in Figure 5.5. The code uses the test framework to test the creation of a conceptual repository. This corresponds to the creation of a PCM repository, the creation of three Java packages, or the creation of a UML model and three UML packages, respectively [Hen20, p. 47 f.]. The specification starts with empty models and defines seven changes: one user-transformation-interchangeable change in every metamodel, one alias change in Java and PCM, and two alias changes in UML. The specification also exploits the fact that the consistency specification specifies overlapping user-transformation-interchangeable changes. Consequently, the test framework will derive seven test cases, as shown in Figure 5.6. Figure 5.7 shows an example of a failure message generated by the framework.

All in all, the test framework described in this section helps realizing the Test Requirements 1 (correctness), 3 (development effort), 4 (debug information), and 5 (performance). Thus, it facilitates a comprehensive, yet efficient test strategy.

```
@TestFactory
def creation(extension EquivalenceTestBuilder builder) {
    stepFor(pcm.domain) [ extension view |
        resourceAt('model/Test'.repository).propagate [
            contents += pcm.repository.Repository => [
                entityName = 'Test'
            ]
        ]
    ]

    inputVariantFor(pcm.domain, 'lowercase name') [ extension view |
        resourceAt('model/test'.repository).propagate [
            contents += pcm.repository.Repository => [
                entityName = 'test'
            ]
        ]
    ]

    stepFor(java.domain) [ extension view |
        resourceAt('src/test/package-info'.java).propagate [
            contents += java.containers.Package => [
                name = 'test'
            ]
        ]

        resourceAt('src/test/contracts/package-info'.java).propagate [
            contents += java.containers.Package => [
                name = 'contracts'
                namespaces += #['test']
            ]
        ]

        resourceAt('src/test/datatypes/package-info'.java).propagate [
            contents += java.containers.Package => [
                name = 'datatypes'
                namespaces += #['test']
            ]
        ]
    ]

    inputVariantFor(java.domain, 'creating only the root package') [ extension view |
        resourceAt('src/test/package-info'.java).propagate [
            contents += java.containers.Package => [
                name = 'test'
            ]
        ]
    ].alsoCompareToMainStepOfSameDomain()
```

⋮

Figure 5.5: A test specification written with the test framework. The `stepFor` method defines user-transformation-interchangeable changes and the `inputVariantFor` method defines alias changes. *Continued on the next page.*

⋮

```
48      stepFor(uml.domain) [ extension view |
49          resourceAt('model/model'.uml).propagate [
50              contents += uml.Model => [
51                  name = 'model'
52                  packagedElements += uml.Package => [
53                      name = 'test'
54                      packagedElements += uml.Package => [
55                          name = 'contracts'
56                      ]
57                      packagedElements += uml.Package => [
58                          name = 'datatypes'
59                      ]
60                  ]
61              ]
62          ]
63      ]
64
65      inputVariantFor(uml.domain, 'creating only the root package') [ extension view |
66          resourceAt('model/model'.uml).propagate [
67              contents += uml.Model => [
68                  name = 'model'
69                  packagedElements += uml.Package => [
70                      name = 'test'
71                  ]
72              ]
73          ]
74      ].alsoCompareToMainStepOfSameDomain()
75
76      inputVariantFor(uml.domain, 'creating only the root package (uppercase name)') [
            extension view |
77          resourceAt('model/model'.uml).propagate [
78              contents += uml.Model => [
79                  name = 'model'
80                  packagedElements += uml.Package => [
81                      name = 'Test'
82                  ]
83              ]
84          ]
85      ].alsoCompareToMainStepOfSameDomain()
86
87      return testsThatStepsAreEquivalent
88  }
```

Figure 5.5: *Continued from the previous page.*

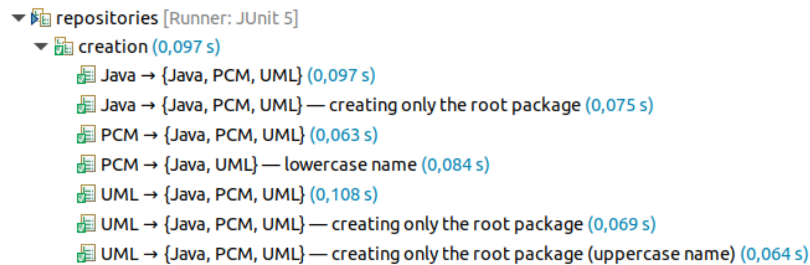


Figure 5.6: The test cases that the test framework derives from the specification code in Figure 5.5, as visualized by the Eclipse IDE.

```
1  Expected: a resource containing a Model deeply equal to <Model#1(
2      name="model"
3      packagedElement={Package#1(
4          name="test"
5          packagedElement={
6              Package#2(name="contracts"),
7              Package#3(name="datatypes")
8          }
9      )}
10 )>
11   but: found the following differences:
12     • .packagedElement{0}.packagedElement{1} (Package#4).name had the wrong value: "data"
13   for object <Model#2(
14     name="model"
15     packagedElement={Package#5(
16         name="test"
17         packagedElement={
18             Package#2(name="contracts"),
19             Package#4(name="data")
20         }
21     )}
22 )>
23   in the resource at <[test view]/model/model.uml>
```

Figure 5.7: Example for a difference printed by the test framework. The output was taken from an execution of the test shown in Figure 5.5 against a faulty implementation that assigned a wrong name to the UML dataype package.

6 Improvements to the Commonalities Language

Before we can expect to find relevant insights with a case study, the Commonalities Language needs to have matured enough to be realistically usable. Otherwise, the internal validity of the study would be threatened (see Section 3.3 and Section 3.4.2). To that end, I realized two improvements to the language and propose a third one. The changes bring the Commonalities Language closer to being ready to be examined in the case study.

6.1 Consistent Syntax for Renaming Participations

In the Commonalities Language, the set of model classes that correspond to a Commonality is called a ‘Participation’. The participating classes are called ‘Participation Classes’, and the metamodel they stem from is called the ‘Participation Domain’. When defining Participations, it is desirable—and sometimes even necessary—to change the name of a Participation Class or the Participation Domain. Per default, Participation Classes have the same name as their type, and Participation Domains are called like the domain they reference. Giving either a Participation Class or Participation Domain another name can be desirable if the default name does not convey the function of the element well. It becomes necessary if there are two Participation Classes of the same type in the same Participation, or if there are two Participations of the same domain in the same Commonality.

The initial design of the Commonalities Language recognized these use cases and allowed to rename Participation Domains and Participation Classes [Gle17, p. 39 f.]. However, the concrete syntax for this feature was not intuitive: First, there were two different keywords for renaming: ‘`called`’ to rename a Participation Class, and ‘`as`’ to rename a Participation Domain. Second, the new name for a Participation Domain needed to be provided at the very end of the declaration, after the Participation Classes. This made it ambiguous whether the name belonged to the Participation Domain or the last Participation Class (cf. the second example in Figure 6.1). Third, the new name was to be given in quotes. This broke the usual conventions of the language, since the new name is an identifier, but nowhere else in the language are quotes needed to declare or reference an identifier. In the current version of the Commonalities Language, the syntax is streamlined. Renaming is now always done with the keyword ‘`as`’, the new name always follows directly after the renamed object, and quotes are no longer needed. Figure 6.1 contrasts the old with the new syntax. The change makes the Commonalities Language easier to use because its syntax is more consistent.

old syntax [Gle17, p. 40]	new syntax	reference
`Java:(Class called "Impl")`	`Java:(Class as Impl)`	`Java:Impl`
`Java:Class as "Lang"`	`(Java as Lang):Class`	`Lang:Class`
`Java:(Class called "Impl")as "Lang"`	`(Java as Lang):(Class as Impl)`	`Lang:Impl`

Figure 6.1: The syntax for renaming Participation Classes and Participation Domains before (left) and after (centre) the change. The right column lists how the resulting Participation Class can be referenced.

6.2 Simplified Operator Imports

Operators are the building blocks for mapping logic in the Commonalities Language. They can be defined in any JVM language and then be used in the Commonalities Language to specify how features map onto each other. To use any operator that is not defined in the standard library of the language, developers need to import the operators into the Commonalities Language file using an `import` statement. I realized three improvements to this import mechanism. Two of them make the feature easier to use, while the last is an important technical improvement.

Developers can create operators for the Commonalities Language by offering a class that implements a specific interface and by annotating it with a special annotation. This annotation also specifies the name by which the operator can be referenced from the Commonalities Language. Giving operators their names via an annotation instead of just using the class name has two advantages: First, classes can have names according to general and implementation-specific conventions for Java class names. Additionally, operators can be identified by strings that are not valid Java identifiers, like '`+`' or '`%`'.

The first improvement concerns the names that developers use to import operators into the Commonalities Language. Before, imports used to refer to the name of the JVM class. An operator like '`firstUpper`' was imported by importing its implementing class, which was called '`FirstUpperOperator`'. This could be especially irritating if the operator's identifier differed much from the Java class name, like the operator '`%`', which might be implemented by a class called '`ModuloOperator`'. Since operators are meant to be reusable, the developers using operators have not necessarily written them. The import mechanism forced those developers to remember two identifiers for every operator: the one by which they can use it in the language, and the one by which they can import it. Consequently, I adapted the language such that imports now refer to the language identifier of an imported operator. Instead of

```
import tools.vitruv.commonalities.operators.ModuloOperator,
```

developers can now write

```
import tools.vitruv.commonalities.operators.%.
```

This change makes the import system of the Commonalities Language easier to use because developers using operators now only need to remember one identifier for each operator.

Additionally to importing each used operator individually, developers can also import all operators from a JVM package ('wildcard imports'). The Commonalities Language used the

syntax ‘<package name>.*’ for this feature. Unfortunately, this meant that the identifier ‘*’ was not available to be used for an operator, as there would have been no possibility to import it. Its import would have coincided with the syntax for a wildcard import. The identifier ‘*’ is commonly used for multiplication in programming, and multiplication is a sensible bidirectional operator [KR16]. Hence, I changed the syntax for wildcard imports to ‘<package name>._’. This is the syntax that the Scala programming language uses for wildcard imports [Ode+06]. The change allows developers to use ‘*’ as an identifier for operators.

The third change to operator imports improved how operators are resolved on a technical level. The old implementation scanned all JVM classes that were defined in a project and filtered them for potential operator implementations. The resulting set of operators could then be imported from the language. This approach only worked inside the Eclipse IDE, where all classes of a project where available for inspection. The approach would have been difficult to realize for the stand-alone compiler of the Commonalities Language. It would have required scanning the whole classpath to find all potential operators. Since the classpath of Vitruvius projects is considerably large, the approach would also have been inefficient. Without a working stand-alone compiler, the Commonalities Language could only be compiled from inside the Eclipse IDE[1]. To allow automated tests to run in continuous integration, the tests started an Eclipse IDE and compiled the Commonalities Language code to Java in that IDE before testing it. In consequence, the tests had long execution times, which slowed down the development-test feedback cycle for all developers using the language.

To overcome these disadvantages, I changed how the Commonalities Language resolves imports. I established a bijection between operator identifiers and the name of the implementing Java class. For instance, an operator that can be imported as ‘`example.package.%`’ will be mapped by the bijection to the fully qualified name ‘`example.package.cl_operators__.mod_`’. The language will lookup the implementing class at this name. As before, developers should be able to pick the names for operator implementation classes freely. Hence, I used Xtend’s code generation feature to generate facade classes for annotated operator implementations. The facade classes are named according to the bijection and delegate to the actual implementation classes. The Commonalities Language can then derive the location of an operator’s facade class from the operator name and the imports via the bijection. Figure 6.2 demonstrates this mechanism. The new logic removes the need to query the whole classpath for all defined operators and allows implementing a stand-alone compiler for the Commonalities Language. The availability of a stand-alone compiler is a big step towards a mature version of the Commonalities Language, as the code can now be checked in the build process just like code in any other language.

[1] There were also other technical reasons for why the Commonalities Language could not be compiled without running an Eclipse IDE. However, these other reasons were far less substantial and relatively easy to resolve.

```
1  package tools.vitruv.applications.cbs.commonalities.domaincommon.operators
                                   ⋮
14 @AttributeMappingOperator(
15     name='firstUpper',
16     commonalityAttributeType = @AttributeType(multiValued=false, type=String),
17     participationAttributeType = @AttributeType(multiValued=false, type=String)
18 )
19 class FirstUpperOperator extends AbstractAttributeMappingOperator<String, String> {
                                   ⋮
43 }
```

```
1  package tools.vitruv.applications.cbs.commonalities.domaincommon.operators.cl_operators__;
                                   ⋮
11 public class firstUpper implements IAttributeMappingOperator<String, String> {
12   private final FirstUpperOperator delegate;
13 
14   public firstUpper(final ReactionExecutionState executionState) {
15     this.delegate = new FirstUpperOperator(executionState);
16   }
17 
18   public String applyTowardsCommonality(final String arg0) {
19     return  this.delegate.applyTowardsCommonality(arg0);
20   }
21 
22   public String applyTowardsParticipation(final String arg0) {
23     return  this.delegate.applyTowardsParticipation(arg0);
24   }
25 }
```

```
1  import tools.vitruv.applications.cbs.commonalities.domaincommon.operators._
2  
3  concept ComponentBasedSystems
4  
5  commonality Repository {
                                   ⋮
26     has name {
27         = PCM:Repository.entityName
28         -> PCM:Resource.name
29 
30         = firstUpper(ObjectOrientedDesign:RepositoryPackage.name)
31     }
                                   ⋮
52 }
```

Figure 6.2: Demonstration of the new mechanism to resolve operators. Top: Excerpt of the implementation of the `firstUpper` operator. Centre: The facade that is generated for the operator. It will be looked up by the Commonalities Language. Bottom: Import and use of `firstUpper` in the Commonality for a repository.

6.3 Clearer Syntax for Operator Chains

The operator system of the Commonalities Language is in large parts inspired by the system Kramer designed for the Mappings Language [Gle17, p. 34 ff. Kra17, p. 151 ff. KR16]. The Commonalities Language also adopted the concrete syntax for operators proposed by [Kra17, p. 179]; namely using the well-known notation of mathematical functions with parentheses, for example ‘$a(b(1, 2), c(4))$’. In the Mappings Language, the syntactic choice is also motivated by the fact that the language grammar embeds the Xbase grammar [Kra17, p. 179], which uses this function notation. The Commonalities Language does not re-use the Xbase grammar and is, hence, not bound by its concrete syntax. The function notation has disadvantages when used for bidirectional operators, which makes revisiting the design choice for their concrete syntax worthwhile.

To discuss the operator syntax, let us consider an example from the case study presented in Chapter 3. It expands on the example of naming a Java implementation class for a software component, which we have seen throughout the thesis. Until now, we have discussed that the implementation class should be given the name of the software component, with ‘Impl’ appended to it. However this convention alone will not suffice. The names of components in PCM and UML can be any string. Java class names, on the other hand, may only use certain characters—in particular, no white space—must not start with a number, and must not coincide with a keyword [Gos+21, p. 24–26]. Furthermore, Java class names are conventionally given in upper camel case, that is, the name ‘administrative user repository’ becomes ‘AdministrativeUserRepository’.

We have discussed that the Commonalities Language was designed to allow developers to re-use bits of transformation logic through operators (see Section 3.4.3.2). Thus, developers will combine multiple operators to realize the name mapping described above. First, they would likely have a Java-specific operator that converts any string into a valid Java identifier when executed in one direction, and leaves a Java identifier unchanged in the other direction. This operator can be re-used throughout the case study whenever a Java identifier needs to be generated. Second, they would likely have an operator that converts a string into camel case notation in one direction, and in the other direction replaces any occurrence of a lower case character ‘`a`’ followed by an upper case character ‘`B`’ with ‘`a b`’. This operator can be re-used elsewhere as well, not just in the context of Java, because camel case notation is common throughout programming-related technologies. Finally, to append the suffix ‘Impl’, the developers would use an operator that appends a suffix to strings in one direction, and strips the suffix, if present, in the other direction. Putting it all together, the Commonality feature declaration for a component name might look like in Figure 6.3 (top), line 4; if we used intuitive names for the operators. The example assumes that the camel case operator takes a constant that can be either `UPPER` or `LOWER`, indicating whether to use upper camel case (‘UserRepository’) or lower camel case (‘userRepository’). Furthermore, it assumes that the component Commonality is directly mapped to a Java class, and not via an `ObjectOrientation:Class` Commonality, like it is done in the case study and in Figure 2.2.

Looking at the example, we can notice drawbacks of the syntax. First, the structure of the Commonalities Language forces us to define how we can obtain the component name from a Java class name. Until now, we found it more intuitive to think of the mapping

```
1 has name {
2     = UML:Component.name
3     = PCM:Component.name
4     = fromCamelCase(UPPER, stripSuffix("Impl", fromJavaIdentifier(Java:Class.name)))
5 }
```

```
1 has name {
2     <> UML:Component.name
3     <> PCM:Component.name
4     <> toCamelCase(UPPER) | addSuffix("Impl") | toJavaIdentifier | Java:Class.name
5 }
```

Figure 6.3: The current (top) and the proposed, new syntax (bottom) for defining operator chains in the Commonalities Language.

the other way: We described how to obtain a Java class name from a component name. Since we are dealing with bidirectional operators, this issue will always be present. We give textual representations of operator chains, so there is always one direction of an operator chain that follows the natural reading order (in the Commonalities Language's English-like syntax, that is left-to-right), while the other direction goes against it. A priori, we might assume that there is no reason to favour one direction over the other. However, practice has shown that developers tend to think about, and communicate about, Commonalities mappings in terms of how the concept model can be transformed to the concrete model. This is consistent with how the Commonalities Language uses the lens formalism (cf. Section 2.4.2) that makes up its bidirectional operators. The Get operation always transforms towards the concrete metamodel. Both Foster et al. [Fos+07] and Kramer and Rakhman [KR16] conceptualise and name lenses based on their Get operation, and consider Put the 'inverse' of that. We can infer that operator chains in the Commonalities Language will generally be easier to understand if they define how to obtain the concrete metamodel's value from the concept metamodel's value. Because then the direction of reading will be the same as the conceptually more intuitive transformation direction.

The second drawback we can notice is that by using function notation, the Commonalities Language suggests that its operators behave like other functions in programming, even though that is not the case. With normal functions, we can, for example, give variables or constants at any argument position. This is not possible for bidirectional operators. For example, '`stripSuffix(Java:Class.name, "Impl")`' would not compile, even though the concrete syntax suggests that this should be valid code. The reason is that the Commonalities Language has two inherently different types of arguments: The input and output value of the operator (which swap roles depending on the executed direction) and constants parametrising the operator (which stay the same regardless of the direction of execution). The concrete syntax should reflect that fact, lest it irritates developers.

Taking these considerations into account, I propose to modify the concrete syntax for operator expressions in the Commonalities Language. First, I propose to introduce the higher-order operator '`|`' that chains two operators together via lens composition, as described in Proposition 2.1 on page 8. The operator is inspired by the pipe operator of the UNIX shell [Tho76] and will, thus, hopefully feel familiar to most programmers. We

will call it ‘pipe operator’ in the Commonalities Language, as well. The pipe operator is not meant to replace the function notation entirely. Instead, it is only used to connect the input and output arguments. To parametrise operators by constants, developers still use function notation. The parenthesis can be omitted if there are no parametrising arguments. Unlike in other programming language, there is no chance of confusing operators and variables, since the Commonalities Language does not have variables.

Secondly, I propose to change the keyword used to introduce a bidirectional mapping from ‘=’ to ‘<>’. The reason for this change is that the = is well-known in programming, and almost always means ‘assign the result of the right-hand expression to the left-hand item’. This is misleading because when using bidirectional operators, data can flow in both directions, not just right to left. Using a different syntax stresses this fact. Furthermore, ‘<>’ is consistent with the existing symbols ‘->’ and ‘<-’ used to introduce a unidirectional left-to-right or right-to-left mapping, respectively.

In effect, our example would be written as in Figure 6.3 (bottom), line 4 if we used intuitive names for the operators. It is apparent that the new syntax does not suffer from the two drawbacks described above. The new syntax also looks less familiar to programmers that are used to typical imperative language like Java and C. Although this might be considered a disadvantage, I argue that it is actually an advantage. Since bidirectional operators behave fundamentally different from functions, it is appropriate that they have a fundamentally different syntax. The new syntax also has the benefit that operators are now applied in the order they are read, left to right.

The idea of an operator that allows better readable chaining of operators or functions is not new. It has been realized in different programming languages like F# [F S21, ‘|>’] or Clojure [Hic19, ‘as->’]. There also is a current proposal to introduce such an operator into the widely used web programming language Ecma Script [Ehr21]. The Firefox browser has already implemented experimental support for it. This shows that the need for chaining operators with a better readable syntax than function notation also occurred in other languages. Arguably, this need is even greater for bidirectional operators, which also have to be read in the reverse order.

This section has focused exclusively on the syntax for bidirectional operators. Their new syntax is now different from the one used for unidirectional operators. Although this might be considered inconsistent, I propose to leave the syntax for unidirectional operators unchanged. None of the drawbacks we have discussed above apply to unidirectional operators, since they do indeed behave like normal functions. Hence, using function notation is appropriate for them.

7 Related Work

Before concluding, I will relate the contributions presented in this thesis to the current state of research. We will look at three major areas: First, research that compares model transformation languages. The case study presented in Chapter 3 contributes to and is based on such research. Second, theoretical findings on model transformations and how congruency (presented in Chapter 3) relates to them. Third, we will consider works in the area of validating and verifying model transformations.

7.1 Evaluating Model Transformation Languages

Model transformations have been the subject of intensive research, and numerous tools for developing transformations have been presented. Naturally, the model transformation community is interested in understanding the relevant differences between the tools. Consequently, multiple papers have surveyed the available model transformation tools and categorized them [MV06; CH06; Hid+16]. Macedo, Jorge and Cunha [MJC17] present a comprehensive taxonomy for model repair tools, of which transformations are a subcategory. The most recent and largest survey is by Kahani et al. [Kah+19], who compare 60 model transformation tools using 46 facets. These works provide an overview of the available tools and introduce a taxonomy that allows to categorize the tools and understand their differences. Hence, they enable users to select the right tool based on the features they need. Compared to the case study presented in this thesis, the surveys differ in intent: They aim to categorize as many tools as possible in a useful way, but do not directly judge fitness for a particular purpose. This thesis, on the other hand, focuses on comparing only two tools, but aims at finding whether their functional differences lead to actual improvements for developers.

The problem of having to pick the right tools is, of course, not specific to model transformations. General-purpose programming languages have been the subject of studies as well. The chosen programming language can have significant influence on programmer productivity and solution quality [DKC07; NF15; AOG16]. This suggests that studying the differences between model transformation languages will also reveal significant differences. The insights from general-purpose languages can guide us when designing evaluations for model transformations languages, even though we must take care whether the finding can still apply in this domain. For example, the size in source lines of code has proven to be a good predictor of maintainability across programming languages [Lip82; SK03; GFS05], even when controlling for other factors. We used this experience when designing the case study in this thesis.

The works most similar to Chapter 3 are those that evaluate model transformation languages empirically. Kolahdouz-Rahimi et al. [Kol+14] use a structured approach to

asses the suitability of model transformation languages for model refactoring. They start from the *ISO/IEC 9126-1* standard [ISO01] (superseded by *ISO/IEC 25010* [ISO11]), which defines a quality model for software. The authors select from the quality model the characteristics they deem relevant for their use case. Like in this thesis, they apply the Goal-Question-Metric approach [BCR02] to derive the metrics that measure the characteristics. This method provides some evidence for why their metrics cover all relevant aspects for the overall assessment. By evaluating five implementations of their case study using their approach, they yield statistically significant differences between the examined transformation languages. The case study presented in this thesis evaluates three of the six characteristics the authors chose: functionality, usability, and maintainability.

Grønmo, Møller-Pedersen and Olsen [GMO09] compare three transformation languages, two graphical and one text-based. They implement a case study in all three languages and discuss several characteristics of the implementations and how they are influenced by the languages' design. The main metric they evaluate is implementation size. Unlike in the case study presented in this thesis, Grønmo, Møller-Pedersen and Olsen do not control with any metric whether the languages 'buy' smaller implementations by making the language more complex. Nevertheless, the authors report the insights they gathered in the case study regarding how complex the languages are.

Samimi-Dehkordi, Khalilian and Zamani [SKZ14] present a method to assess model transformation languages. They use four main criteria—readability, writability, reliability and cost—and specify several characteristics for each of these criteria. The characteristics are judged by experts. After applying their method to five transformation languages, they conclude that yields no clear ranking of the languages, but rather helps selecting the right language for a given task. The authors' approach differs from the one of the case study in Chapter 3, as experts judge the languages directly, instead of gathering metrics from implementations written with the languages.

This thesis focused on the Commonalities Language and Reactions Language, which are specific for the Vitruvius framework. In the context of Vitruvius, Klare et al. [Kla+20] have applied transformations written in the Reactions Language to the case of consistency of UML, Java and PCM (which we also used in this thesis) and a case of consistency in embedded automotive software architectures. They showed that the transformations were suitable for these cases and that they were significantly smaller in terms of lines of code than an implementation in plain Java. Werle [Wer16] compared the Mappings Language—another domain-specific model transformation language for Vitruvius—to Triple Graph Grammar tools [Hil+13]. The comparison focused on differences in functionality.

Empirical evaluation of model transformation languages has, so far, been limited to conducting case studies. The contributions in this thesis are no different in this regard. Researchers may currently still prefer case studies over experiments or user studies because many tools still lack relevant features [Kah+19]. Hence, qualitative feedback gathered from case studies might be considered more valuable than quantitative results that may be invalidated as soon as the language changes. This is certainly the case for the Commonalities Language. Kramer et al. [Kra+16] present a template for an experiment that compares the understandability of model transformation languages. This is the first description of an experiment on model transformation languages. However, the template has not been used in an experiment yet.

Additionally to evaluating model transformation languages, there has been research on evaluating model transformations themselves, and finding metrics that correlate with well-maintainable implementations [vALvdB09; Kap+10; vAms+11; RRA17]. These results are promising in the sense that maintainability for model transformations can, at least partly, be predicted. However, the found metrics are usual specific to the used transformation language and cannot be transferred to other languages. Even if they can be defined analogously in other transformation languages, it is unclear whether they have the same predictive power there. This means that they cannot be used to compare different languages, which we have been interested in.

7.2 Properties of Model Transformations

Stevens [Ste10; Ste12] and Diskin [Dis08] have proposed algebraic models to formalize bidirectional transformations. I have adopted Stevens' model [Ste10] in this thesis (see Section 2.4.1). The model corresponds to Diskin's di-systems [Dis08], except that di-systems use partial functions for the transformation directions. Together with their algebraic models, the authors also define desirable properties for bidirectional transformations. They recognize that without further requirements, bidirectional transformations do not behave as expected. However, although they present several useful properties, only two of them have been widely assumed to always hold for realistic examples. These are correctness (Definition 2.2) and Hippocraticness (Definition 2.3) [Ste10].

Stevens [Ste10] starts the discussion about requirements for transformations by noting that, although one might tend to think about bidirectional transformations as being bijective, most most of them are not. We have seen an example of this in Chapter 4. Most requirements for bidirectional transformations can be understood as an attempt to recover some of the useful properties of bijective transformations, without forcing the transformations to actually be bijective. For instance, Diskin [Dis08] and Stevens [Ste12] present history-ignorance, which ensures a certain compatibility between different executions of the same transformation direction. Undoability [Ste10] is a weaker requirement in the same spirit. Both requirements are useful for users and tool developers alike [Ste10; Ste17]. Unfortunately, many practically relevant transformations are neither history-ignorant nor undoable [Ste10].

Another desirable property for bidirectional transformations is that the two directions of the transformation are compatible in some sense, similar to how they are in bijective transformations. Congruency (cf. Chapter 4) is such a property; mandating that each transformation direction produces a change that has an interchangeable change in the other direction. [Ste12] provides two properties ensuring compatibility between directions, as well: 'matching' and 'simply matching'. They are based on the consistency relations that bidirectional transformation induce, and require the resulting equivalence classes to map onto each other. These two properties have many similarities to congruency. We will examine the relationships more closely in the following.

To introduce Stevens' properties, let me first specify what I mean by transformations inducing equivalence relations.

Definition 7.1 ([Ste12]). Given a bidirectional transformation $\overleftrightarrow{T} =: (T, \overrightarrow{T}, \overleftarrow{T})$ between A and B, the two equivalence relations $\sim_{\overrightarrow{T}} \subseteq A \times A$ and $\sim_{\overleftarrow{T}} \subseteq B \times B$ are defined as

$$\begin{aligned} a_1 \sim_{\overrightarrow{T}} a_2 &\iff \forall b \in B\colon \overrightarrow{T}(a_1, b) = \overrightarrow{T}(a_2, b) \\ b_1 \sim_{\overleftarrow{T}} b_2 &\iff \forall a \in A\colon \overleftarrow{T}(a, b_1) = \overleftarrow{T}(a, b_2)\,. \end{aligned}$$

Second, we need the notion of a transversal of an equivalence relation as a helpful tool.

Definition 7.2. Given a set S and an equivalence relation $\sim$ on S, a *transversal* $S|\sim$ of S is a set containing exactly one representative of each equivalence class of S under $\sim$.

We are now equipped to formulate Stevens' definition of matching and simply matching:

Definition 7.3 ([Ste12]). A bidirectional transformation $\overleftrightarrow{T} =: (T, \overrightarrow{T}, \overleftarrow{T})$ between A and B is *matching* if, and only if, there is a bijection $f\colon A|\sim_{\overrightarrow{T}} \to B|\sim_{\overleftarrow{T}}$ and

$$\forall a_r \in A|\sim_{\overrightarrow{T}}\colon\ (a_r, f(a_r)) \in T.$$

Furthermore, $\overleftrightarrow{T}$ is *simply matching* iff additionally

$$\forall a_r \in A|\sim_{\overrightarrow{T}}\ \forall b \in B\colon\ (a_r, b) \in T \iff b = f(a_r)\,.$$

We might recognize similarities to the discussion in Chapter 4. The equivalence classes of $\sim_{\overrightarrow{T}}$ and $\sim_{\overleftarrow{T}}$ are similar to the alias changes we described. The bijection f between the transversals resembles the definition of user-transformation-interchangeable changes. And indeed, there are transformation relations for which the simply matching transformations are exactly the congruent transformations. Nevertheless, congruency is strictly weaker than simply matching.

Proposition 7.1. *Every correct, simply matching, bidirectional transformation is congruent.*

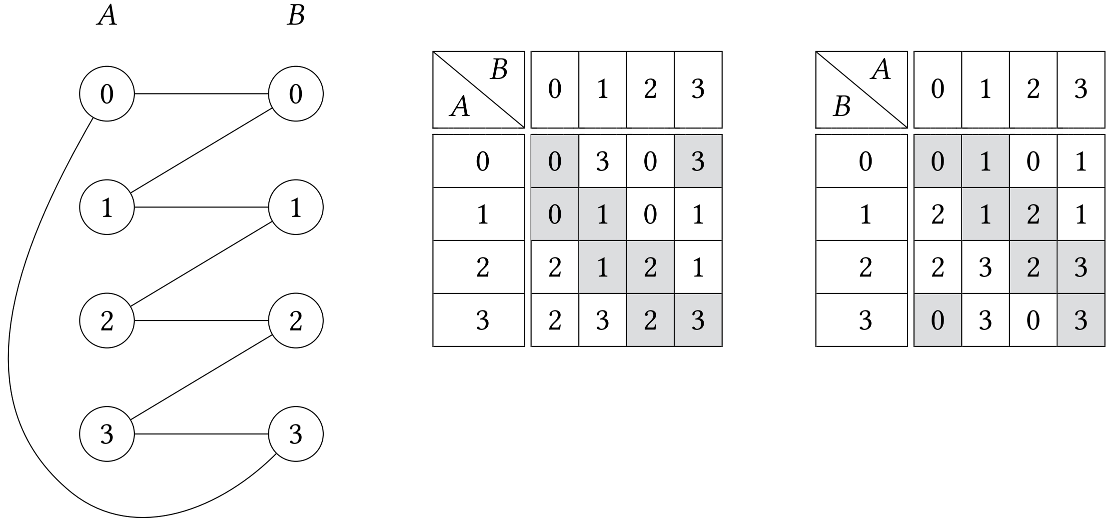
Proof. Let $\overleftrightarrow{T} =: (T, \overrightarrow{T}, \overleftarrow{T})$ be a correct, simply matching, bidirectional transformation between A and B. Let $A|\sim_{\overrightarrow{T}}$ be a transversal of A under $\sim_{\overrightarrow{T}}$, $B|\sim_{\overleftarrow{T}}$ be a transversal of B under $\sim_{\overleftarrow{T}}$, and $f\colon A|\sim_{\overrightarrow{T}} \to B|\sim_{\overleftarrow{T}}$ be the bijection between the two transversals. We show one direction of congruency, the other is dual. Let $(a_0, b_0) \in T$ and $a_c \in A$. Furthermore, let $a_c^r \in A|\sim_{\overrightarrow{T}}$ be the representative of a_c in $A|\sim_{\overrightarrow{T}}$; that is, $a_c \in \left[a_c^r\right]_{\sim_{\overrightarrow{T}}}$.

Because $\overleftrightarrow{T}$ is simply matching and correct, and by the definitions of $\sim_{\overrightarrow{T}}$ and $\sim_{\overleftarrow{T}}$, we see

$$\begin{aligned} \overrightarrow{T}(\overleftarrow{T}(a_0, \overrightarrow{T}(a_c, b_0)), b_0) &= \overrightarrow{T}(\overleftarrow{T}(a_0, b_x), b_0)\,, && \text{for some } b_x \in \left[f\left(a_c^r\right)\right]_{\sim_{\overleftarrow{T}}} \\ &= \overrightarrow{T}\left(\overleftarrow{T}\left(a_0, f\left(a_c^r\right)\right), b_0\right) \\ &= \overrightarrow{T}(a_x, b_0)\,, && \text{for some } a_x \in \left[f^{-1}\left(f\left(a_c^r\right)\right)\right]_{\sim_{\overrightarrow{T}}} \\ &= \overrightarrow{T}\left(f^{-1}\left(f\left(a_c^r\right)\right), b_0\right) \\ &= \overrightarrow{T}\left(a_c^r, b_0\right) \\ &= \overrightarrow{T}(a_c, b_0)\,. \end{aligned}$$

□

To show that there are congruent transformations that are not simply matching, we construct the following example. Its consistency relation contains a pair of model states that are indirectly consistent but not consistent. Hence, it also shows that this is not a necessary condition for the existence of a congruent bidirectional transformation.



A \ B	0	1	2	3
0	0	3	0	3
1	0	1	0	1
2	2	1	2	1
3	2	3	2	3

B \ A	0	1	2	3
0	0	1	0	1
1	2	1	2	1
2	2	3	2	3
3	0	3	0	3

Figure 7.1: Graphic representations of the consistency relation T_4 (left), the right direction $\overrightarrow{T}_4$ (centre), and the left direction $\overleftarrow{T}_4$ (right) of the bidirectional transformation $\overleftrightarrow{T}_4 := (T_4, \overrightarrow{T}_4, \overleftarrow{T}_4)$ from Example 7.1. Values that are pre-determined by Hippocraticness are coloured grey in the tables of the the transformation directions.

Example 7.1. Let $A = B = \mathbb{Z}/4\mathbb{Z}$ and let T_4 be a consistency relation between them, such that every element in A is paired with itself and its predecessor (see Figure 7.1, left):

$$(a, b) \in T \iff a = b \vee a = b - [1]_4 .$$

We define the bidirectional transformation $\overleftrightarrow{T}_4 := (T_4, \overrightarrow{T}_4, \overleftarrow{T}_4)$ between A and B as follows: First, $\overleftrightarrow{T}_4$ is Hippocratic. Second, every value that is not defined by Hippocraticness is obtained by choosing the old value plus two (see Figure 7.1, centre and right):

$$\begin{aligned} \overrightarrow{T}_4 \colon A \times B &\to B \\ (a, b) &\mapsto \begin{cases} b & \text{if } (a, b) \in T \\ b + [2]_4 & \text{else} \end{cases} \end{aligned}$$

$$\begin{aligned} \overleftarrow{T}_4 \colon A \times B &\to A \\ (a, b) &\mapsto \begin{cases} a & \text{if } (a, b) \in T \\ a + [2]_4 & \text{else} \end{cases} \end{aligned}$$

Lemma 7.1. *There exists a correct, Hippocratic, congruent, bidirectional transformation that is not simply matching.*

Proof. We can see directly from the definition of Example 7.1 and in Figure 7.1 that $\overleftrightarrow{T}_4$ is correct and Hippocratic. The equivalence classes of $\sim_{\overrightarrow{T}_4}$ can be read from the rows of the centre table in Figure 7.1, and the equivalence classes of $\sim_{\overleftarrow{T}_4}$ can be read from the

rows of the right-hand table in Figure 7.1. All elements with equal rows are in the same equivalence class [Ste12]. We see that there are no non-trivial equivalence classes for both relations. Hence, $A|{\sim_{\vec{T}_4}} = A$ and $B|{\sim_{\overleftarrow{T}_4}} = B$. There are two bijections between $A|{\sim_{\vec{T}_4}}$ and $B|{\sim_{\overleftarrow{T}_4}}$ which are also a subset of T: $f\colon a \mapsto a$ and $g\colon a \mapsto a - [1]_4$. Hence, $\overleftrightarrow{T}_4$ is matching. But $\overleftrightarrow{T}_4$ is not simply matching because there are other consistent pairs for both bijections; for example $([2]_4, [1]_4) \in T$, although $[1]_4 \neq f([2]_4)$, and $([2]_4, [2]_4) \in T$, although $[2]_4 \neq g([2]_4)$.

To show that $\overleftrightarrow{T}_4$ is congruent, we observe two properties of T_4. First, we notice that values with the same distance to each other are equal with regard to T_4:

$$\forall a \in A\, \forall b \in B\, \forall n \in \mathbb{Z}/4\mathbb{Z}\colon\ (a, b) \in T_4 \iff (a + n, b + n) \in T_4.$$

Second, values with a distance of two are opposites with regard to T_4:

$$\forall a \in A\, \forall b \in B\colon\ (a + [2]_4, b) \notin T_4 \iff (a, b) \in T_4 \iff (a, b + [2]_4) \notin T_4$$

Let $(a_0, b_0) \in T_4$ and $a_c \in A$. We differentiate two cases:

1. $(a_c, b_0) \in T_4$. Then:

$$\begin{aligned}
\vec{T}_4\left(\overleftarrow{T}_4\left(a_0, \vec{T}_4\left(a_c, b_0\right)\right), b_0\right) &= \vec{T}_4\left(\overleftarrow{T}_4\left(a_0, b_0\right), b_0\right) \\
&= \vec{T}_4\left(a_0, b_0\right) \\
&= a_0 \\
&= \vec{T}_4\left(a_c, b_0\right).
\end{aligned}$$

2. $(a_c, b_0) \notin T_4$. Then:

$$\begin{aligned}
\vec{T}_4\left(\overleftarrow{T}_4\left(a_0, \vec{T}_4\left(a_c, b_0\right)\right), b_0\right) &= \vec{T}_4\left(\overleftarrow{T}_4\left(a_0, b_0 + [2]_4\right), b_0\right) \\
&= \vec{T}_4\left(a_0 + [2]_4, b_0 + [2]_4\right) \\
&= b_0 + [2]_4 \\
&= \vec{T}_4\left(a_c, b_0\right).
\end{aligned}$$

Conversely, let $(a_0, b_0) \in T_4$ and $b_c \in B$. Once again, we differentiate two cases:

1. $(a_0, b_c) \in T_4$. Then:

$$\begin{aligned}
\overleftarrow{T}_4\left(a_0, \vec{T}_4\left(\overleftarrow{T}_4\left(a_0, b_c\right), b_0\right)\right) &= \overleftarrow{T}_4\left(a_0, \vec{T}_4\left(a_0 + b_0\right)\right) \\
&= \overleftarrow{T}_4\left(a_0, b_0\right) \\
&= a_0 \\
&= \overleftarrow{T}_4\left(a_0, b_c\right).
\end{aligned}$$

2. $(a_0, b_c) \notin T_4$. Then:

$$\begin{aligned}
\overleftarrow{T}_4\left(a_0, \vec{T}_4\left(\overleftarrow{T}_4\left(a_0, b_c\right), b_0\right)\right) &= \overleftarrow{T}_4\left(a_0, \vec{T}_4\left(a_0 + [2]_4, b_0\right)\right) \\
&= \overleftarrow{T}_4\left(a_0 + [2]_4, b_0 + [2]_4\right) \\
&= a_0 + [2]_4 \\
&= \overleftarrow{T}_4\left(a_0, b_c\right).
\end{aligned}$$

□

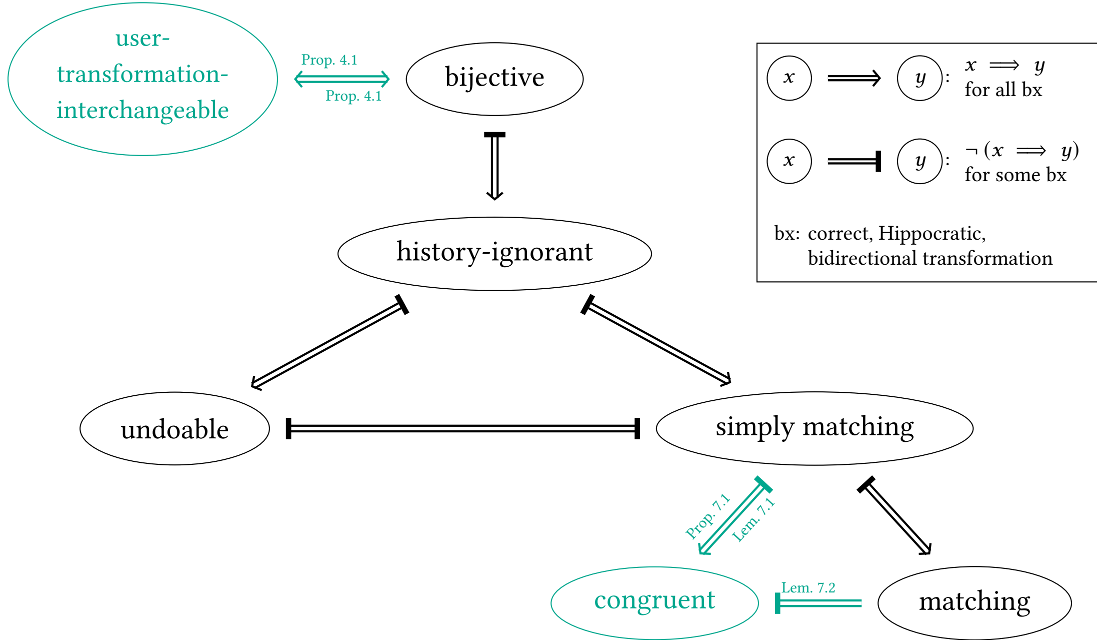


Figure 7.2: The relationships between properties of correct, Hippocratic, bidirectional transformations. This figure extends a similar one by Stevens [Ste12] with the properties presented in this thesis. The additions are marked green.

Congruency is a strictly weaker property than simply matching. Matching is also strictly weaker than simply matching. Nevertheless, congruency and matching are not equivalent. There are transformations that are matching, but not congruent. Whether congruency implies matching is an open question.

Example 7.2 ([Ste12])**.** Let $A = B = \{0, 1\}$ and $T_2 := \{(a, b) \mid a \cdot b = 0\}$. The bidirectional transformation $\overleftrightarrow{T}_2 = (T_2, \vec{T}_2, \overleftarrow{T}_2)$ between A and B is uniquely defined by correctness and Hippocraticness.

Lemma 7.2. *There exists a correct, Hippocratic, matching, bidirectional transformation that is not congruent.*

Proof. $\overleftrightarrow{T}_2$ is matching: It has only trivial equivalence classes and the bijection $f\colon a \mapsto a+1 \mod 2$ is a subset of T_2 [Ste12]. However, $\overleftrightarrow{T}_2$ is not congruent:

$$\begin{aligned}\vec{T}\left(\overleftarrow{T}\left(0, \vec{T}\left(1,1\right)\right), 1\right) &= \vec{T}\left(\overleftarrow{T}\left(0,0\right), 1\right)\\ &= \vec{T}\left(0,1\right)\\ &= 1\\ &\neq \vec{T}\left(1,1\right) = 0.\end{aligned}$$ □

The relationships between the properties we have discussed in this section are summarized in Figure 7.2.

Chapter 4 and Chapter 5 of this thesis add to the existing algebraic descriptions of desirable properties for bidirectional transformations. Notably, congruency has been

motivated in this thesis from observations of practical examples. Chapter 5 also presents a practical use case for congruency. The properties most similar to congruency, simply matching and matching, have not yet been reported to have practical uses. Furthermore, congruency can be tested locally, as shown in Section 5.2: We can take individual model states and test whether congruency holds for them. (Simply) matching, on the other hand, can only be tested globally, if at all. Testing for those properties would require, first, to determine all equivalence classes of both transformation directions. Then we would need to check whether the transformation relation contains a bijection between those equivalence classes.

Stevens writes that they 'have the impression that a large class of practical examples [...] are in fact simply matching' [Ste12]. This conjecture supports the one stated in Chapter 4, namely that all transformations that are used to maintain consistency for models with semantic overlap can, and should, be congruent.

Bidirectional transformations have been analysed theoretically from numerous other angles. In this thesis, we have abstracted the actual transformation logic into black box functions in this thesis and only set requirements for them. Diskin [Dis11] proposes an algebraic framework for specifying the actual transformation logic based on tile algebra and category theory. It builds complex transformations out of simpler building blocks and also offers a visual representation for the mapping logic. In our formalism, we also only considered state-based model transformations. Diskin et al. [Dis+11] and Diskin, Xiong and Czarnecki [DXC11] show that certain consistency specifications cannot be modelled without considering the changes that have been made to models and offer algebraic tools to model such changes. Similar issues can occur if traces between model elements are not modelled. We have described that Vitruvius uses correspondences (cf. Section 2.6.1), but have ignored them in the formalism. Diskin, Gómez and Cabot [DGC17] show how traceability mappings, like Vitruvius' correspondences, can be modelled algebraically and which benefits it brings. The way we modelled bidirectional transformations, they can also not handle the case that both models have been changed before the transformation was executed. This is often referred to as model synchronization. Supporting this use case requires a broader theoretical framework and poses challenges of its own [Ore+13]. When lifting existing bidirectional transformations to make them synchronizing, properties we have discussed in this chapter—like history-ignorance—are, once again, relevant [Xio+13].

In recent years, the attention of the model transformations research community has been broadened from the bidirectional case to the multidirectional case; i.e. to transformations between more than two metamodels. There are two fundamental approaches to this issue: First, extending the existing tools and formalisms to directly support transforming between more metamodels [MCP14; TA15; TA16]. Second, combining existing transformations into a transformation network that spans across more models than the transformations it is built out of [Ste17; Ste18; GKB21]. The two approaches are not mutually exclusive; transformation networks can, at least in theory, incorporate multidirectional transformations. The Vitruvius approach uses model transformation networks and the discussed case study (Chapter 3) focuses on a case of a model transformation network between UML, PCM and Java. The Commonalities Language allows defining a model transformation networks [Gle17; KG19]. So far, the theoretical results for transformation networks have been limited. It is known that a network of bidirectional transformations can express

any multiary consistency relation if adding an auxiliary model is allowed [Ste17; RPD90; Cle+19]. However, finding an appropriate strategy to execute the transformations in a network prooves challenging and has only been achieved in limited settings [Di +17; Ste17; Ste18; GKB21]. Finding sensible properties for bidirectional transformations may be decisive in this regard. For example, history-ignorant transformations can be executed easily [Ste17]. Unfortunately, most transformations are not history-ignorant [Ste10; Ste17]. As detailed in Section 4.2, it seems more realistic to expect transformations to be congruent. In Section 5.2 we saw that the tested transformations were compatible in a sense related to congruency, as their user-transformation-interchangeable coincided. Maybe this can be exploited to make progress on finding execution strategies.

7.3 Model Transformation Testing

Model transformations need to be validated just like other software artefacts. However, model transformations also pose challenges that are specific to their domain and demand specialized validation techniques [AW15]. Ab. Rahim and Whittle [AW15] classify the research around such techniques into five categories: theorem proving, proving properties with graph theory, applying model checking, evaluating metrics, and testing. We already touched on quality assessment based on software metrics in Section 7.1. The contributions of Chapter 5 fall into the last category: testing.

One stream of research considers techniques that establish a sense of test coverage. For model transformations, it is more insightful to check how much of the transformed metamodels has been covered, as opposed to checking line coverage of the transformations [FSB04; WKC06]. However, this requires developers to establish which parts of the metamodels should be part of the transformation in the first place—the so-called 'effective metamodel'. Other works study how mutation testing can be brought to model transformation testing [MBL06; DPV08; Tro+15]. These results are also relevant for the test suite of the case study. To make the transformations comparable, we need to verify that they implement the same behaviour. Establishing the test coverage of the acceptance test suite helps understanding how complete it is.

The models between which transformations transform conform to a metamodel. We can use the rules of the abstract and concrete syntax imposed by the metamodel to generate inputs for testing model transformations. Various works have developed techniques based on this idea [Bro+06; Lam07; SBM09; GC12; WAS14]. This is also relevant for the case study acceptance test suite. Generated models could be used to verify that both transformation implementations yield the same result for any input.

Many researchers have also presented test frameworks for the testing techniques they propose. Such frameworks usually realize the presented technique in the context of a specific model transformation tool [LZG05; Stu+07; DPV08; GP09]. The framework presented in Section 5.3 is no different: it enables the congruency-based acceptance test approach in the context of Vitruvius.

8 Conclusion

In this thesis, I have lain the foundation for conducting a case study on the Commonalities Language. The case study will provide empirical insights into the differences between the model transformation programming languages that are available for the Vitruvius framework. The case study will help finding a good programming model for the framework, and, ultimately, contribute to the much-needed [Bad+18], better tool support for model-driven engineering.

I started the thesis by presenting the design of the case study (Chapter 3). Using the Goal-Question-Metric approach, I showed how the studied metrics are derived from the study's goals. I discussed why the design offers sufficient internal validity and construct validity. I described why the studied case is relevant and can be considered a 'critical case'. Nevertheless, the external validity of the case study will be limited.

In Chapter 4, I introduced a novel property for bidirectional transformations: congruency. It ensures that the two directions of a bidirectional transformation are, in a certain sense, compatible. I showed how the property derives from intuitive expectations users have for transformations and explained why it is realistic to expect congruency to hold for all model transformations that are used for maintaining consistency of models with semantic overlap. Congruency is not tied to the context of the case study. Instead, it can be added to the toolbox of properties we have to assure that model transformations behave as expected. In Chapter 7, I proved that congruency is similar, but not equivalent, to two properties presented by Stevens [Ste12]. Compared to Stevens' properties, congruency is easier to test.

One practical use case of congruency was shown in Chapter 5, where I described a test strategy for black box acceptance tests that check whether two inherently different model transformation implementations realize the same consistency specification. The test strategy will be used on the implementations for the case study. I listed and justified requirements for the test strategy, presented a framework with which the requirements can be met, and showed how the framework meets the requirements. The insights from the described test strategy, especially how to use congruency to reduce the test development effort, can be applied to any test suite for bidirectional transformations. The accompanying test framework is specific to the Vitruvius framework, but not specific to the case study. It can, thus, be used for testing any transformation for the Vitruvius framework.

Finally, I showcased three improvements to the Commonalities Language in Chapter 5, of which I have implemented two. The improvements are necessary to conduct the test study. They also improve the usability of the Commonalities Language in general.

Future work based on this thesis should focus on three areas: The Commonalities Language, the case study, and congruency.

Commonalities Language I have described that the Commonalities Language is not yet ready to be meaningfully evaluated in a case study (Section 3.3) and have already presented three improvements to the language (Chapter 6). While working on the case study, I found the following areas that also need to be addressed before the case study can be conducted:

- The bidirectional operators for the Commonalities Language are currently not defined as lenses. Hence, they cannot be combined and the Commonalities Language currently only allows to use at most one operator per mapping. The interfaces of bidirectional operators should be changed to be that of a lens. Then the proposed new syntax for bidirectional operators (Section 6.3) can be realized.
- The Commonalities Language currently forces every listed Participation Class to exist before a Commonality is established. To implement use cases where some participating objects are optional (like creating a component for a Java package, without requiring the implementation class to exist, as we discussed in Chapter 4), optional Participations should be implemented. The design for them is provided by Gleitze [Gle17, p. 40 f.].
- It should be possible to define Participations in the Commonalities Language that are mapped 1:n to the Commonality, for a variable $n \in \mathbb{N}_0$. Currently, only fixed ns are supported. This is necessary, for example, to establish proper correspondences between Commonality for object-oriented packages, and packages in Java, which are implicitly defined by all compilation units that declare the same package.
- It should be possible to import values from models into the language, and use them to create mappings between concrete values. This is useful, for example, to specify directly in the language how enums and other special values map onto each other. Currently, users are forced to create a new operator for each mapping, which increases the lines of code that need to be written, introduces unnecessary indirection, and hides the relevant logic.

Case Study The case study implementation should be finalized for both the Reactions Language and the Commonalities Language. The test strategy and framework I presented in Chapter 5 will help to do this in a test-driven fashion. This will also ensure that the remaining cases where the different transformations in the Reactions Language contradict each other are found. Afterwards, the case study can be conducted as described in Chapter 3.

Congruency Studying congruency more deeply may reveal further useful results. For example, we have seen in Chapter 5 that the different transformations shared their user-transformation-interchangeable changes. When building transformations networks, maybe we can demand that all involved transformations are congruent and that any user-transformation-interchangeable change of one transformation is also a user-transformation-interchangeable change of the others. If the expectation could be met in realistic use

cases, it could help address the open problem of finding a suitable transformation network execution strategy [GKB21]. Generally speaking, congruency seems to be flexible enough to be fulfilled in realistic transformations, while still being restricting enough to make use of it.

Congruency should also be kept in mind when improving the Commonalities Language's design. When not using any operators, the Commonalities Language establishes congruent bidirectional transformations between the concept metamodel and the concrete metamodels. The user-transformation-interchangeable changes are given by the minimal model states of the concrete metamodels that suffice to establish a Participation. It would be worth investigating whether the transformations remain congruent when using operators. Here it could pay off that the Commonalities Language restricts its bidirectional operators to lenses. Together with the insights we got from congruency and transformation networks, one could perhaps even show that the Commonalities Language establishes transitively congruent transformations—for a yet-to-be-defined meaning of the term.

Finally, it might be worth examining the differences between congruency and simply matching more closely. Maybe simply matching—a stronger property than congruency—also always holds in practice. Example 7.1, which we used to differentiate the two properties, does not seem like a practically relevant one. Stevens [Ste12] proved that simply matching transformations are exactly the ones that can be obtained by putting two lenses 'back to back'. If this theorem could be carried over to the multidirectional case, it would have interesting implications for the Commonalities Language: We would not have to store the concept model any more. The concept model could then be a construct that only helps developing multidirectional transformations, and is only instantiated ad-hoc at runtime when executing the transformation.

While implementing the case study, it will be interesting to see whether all transformations are indeed congruent. This will add more empirical validation to the conjecture that we can expect congruency of all bidirectional transformations used to keep models with semantic overlap consistent.

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.

[Ale+21] Aleksey Shipilev et al. *Openjdk/Jmh*. OpenJDK, 24th Mar. 2021. URL: https://github.com/openjdk/jmh (visited on 11/04/2021).

[AOG16] D. Alic, S. Omanovic and V. Giedrimas. 'Comparative Analysis of Functional and Object-Oriented Programming'. In: *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). May 2016, pp. 667–672. DOI: 10.1109/MIPRO.2016.7522224.

[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.

[AW15] Lukman Ab. Rahim and Jon Whittle. 'A Survey of Approaches for Verifying Model Transformations'. In: *Software and Systems Modeling* 14.2 (2015), pp. 1003–1028. ISSN: 16191374. DOI: 10.1007/s10270-013-0358-0.

[Bad+18] Omar Badreddin et al. 'A Decade of Software Design and Modeling: A Survey to Uncover Trends of the Practice'. In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS '18. New York, NY, USA: Association for Computing Machinery, 14th Oct. 2018, pp. 245–255. ISBN: 978-1-4503-4949-9. DOI: 10.1145/3239372.3239389.

[BCR02] Victor R. Basili, Gianluigi Caldiera and H. Dieter Rombach. 'Goal Question Metric (GQM) Approach'. In: *Encyclopedia of Software Engineering*. Ed. by John J. Marciniak. In collab. with Rini van Solingen. 15th Jan. 2002. ISBN: 978-0-471-02895-6. DOI: 10.1002/0471028959.sof142.

[BP08] Cédric Brun and Alfonso Pierantonio. 'Model Differences in the Eclipse Modelling Framework'. In: *European Journal for the Informatics Professional* 9.2 (2nd Apr. 2008), pp. 29–34. ISSN: 1684-5285.

[Bro+06] E. Brottier et al. 'Metamodel-Based Test Generation for Model Transformations: An Algorithm and a Tool'. In: *2006 17th International Symposium on Software Reliability Engineering*. 2006 17th International Symposium on Software Reliability Engineering. Nov. 2006, pp. 85–94. DOI: 10.1109/ISSRE.2006.27.

[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.

[BW84] V. R. Basili and D. M. Weiss. 'A Methodology for Collecting Valid Software Engineering Data'. In: *IEEE Transactions on Software Engineering* SE-10.6 (Nov. 1984), pp. 728–738. ISSN: 1939-3520. DOI: 10.1109/TSE.1984.5010301.

[CH06] Krzysztof Czarnecki and Simon Helsen. 'Feature-Based Survey of Model Transformation Approaches'. In: *IBM Systems Journal* 45.3 (2006), pp. 621–645. ISSN: 0018-8670. DOI: 10.1147/sj.453.0621. pmid: 15866344.

[Che17] Fei Chen. 'Änderungsgetriebene Konsistenzhaltung zwischen UML-Klassenmodellen und Java-Code'. Bachelor's Thesis. Karlsruhe, Germany: Karlsruher Institut für Technologie (KIT), 24th May 2017.

[Cle+19] Anthony Cleve et al. 'Multidirectional Transformations and Synchronisations (Dagstuhl Seminar 18491)'. In: *Dagstuhl Reports* 8.12 (2019), pp. 1–48. ISSN: 2192-5283. DOI: 10.4230/DagRep.8.12.1.

[Coh09] Mike Cohn. *The Forgotten Layer of the Test Automation Pyramid*. Mountain Goat Software. 17th Dec. 2009. URL: https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid (visited on 31/03/2021).

[DGC17] Zinovy Diskin, Abel Gómez and Jordi Cabot. 'Traceability Mappings as a Fundamental Instrument in Model Transformations'. In: *Fundamental Approaches to Software Engineering*. Ed. by Marieke Huisman and Julia Rubin. Springer Berlin Heidelberg, 2017, pp. 247–263. ISBN: 978-3-662-54494-5. DOI: 10.1007/978-3-662-54494-5_14.

[Di +17] Juri Di Rocco et al. 'Consistency Recovery in Interactive Modeling'. In: *Proceedings of MODELS 2017 Satellite Event: EXE, Co-Located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems*. Austin, TX, USA, 17th Sept. 2017, pp. 116–122. URL: http://ceur-ws.org/Vol-2019/exe_6.pdf.

[Dis+11] Zinovy Diskin et al. 'From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case'. In: *Model Driven Engineering Languages and Systems*. Ed. by Jon Whittle, Tony Clark and Thomas Kühne. Springer Berlin Heidelberg, 2011, pp. 304–318. ISBN: 978-3-642-24485-8. DOI: 10.1007/978-3-642-24485-8_22.

[Dis08] Zinovy Diskin. 'Algebraic Models for Bidirectional Model Synchronization'. In: *Model Driven Engineering Languages and Systems*. Ed. by Krzysztof Czarnecki et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 21–36. ISBN: 978-3-540-87875-9. DOI: 10.1007/978-3-540-87875-9_2.

[Dis11] Zinovy Diskin. 'Model Synchronization: Mappings, Tiles, and Categories'. In: *Generative and Transformational Techniques in Software Engineering III*. Ed. by João Fernandes et al. Vol. 6491. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 92–165. ISBN: 978-3-642-18022-4. DOI: 10.1007/978-3-642-18023-1_3.

[DKC07] D. P. Delorey, C. D. Knutson and S. Chun. 'Do Programming Languages Affect Productivity? A Case Study Using Data from Open Source Projects'. In: *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*. First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007). May 2007, pp. 8–8. DOI: 10.1109/FLOSS.2007.5.

[DPV08] Andrea Darabos, András Pataricza and Dániel Varró. 'Towards Testing the Implementation of Graph Transformations'. In: *Electronic Notes in Theoretical Computer Science*. Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006) 211 (28th Apr. 2008), pp. 75–85. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.04.031.

[DXC11] Zinovy Diskin, Yingfei Xiong and Krzysztof Czarnecki. 'From State- to Delta-Based Bidirectional Model Transformations: The Asymmetric Case'. In: *Journal of Object Technology* 10 (2011), 6:1–25. ISSN: 1660-1769. DOI: 10.5381/jot.2011.10.1.a6.

[Ehr21] Daniel Ehrenberg. *Pipeline Operator*. 9th Mar. 2021. URL: https://tc39.es/proposal-pipeline-operator (visited on 10/04/2021).

[F S21] F# Software Foundation. *Operators (FSharp.Core)*. 3rd Mar. 2021. URL: https://fsharp.github.io/fsharp-core-docs/reference/fsharp-core-operators.html (visited on 10/04/2021).

[Fly06] Bent Flyvbjerg. 'Five Misunderstandings About Case-Study Research'. In: *Qualitative Inquiry* 12 (1st Apr. 2006), pp. 219–245. DOI: 10.1177/1077800405284363.

[Fos+07] J. Nathan Foster et al. 'Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29.3 (May 2007). ISSN: 0164-0925. DOI: 10.1145/1232420.1232424.

[Fow12] Martin Fowler. *TestPyramid*. martinfowler.com. 1st May 2012. URL: https://martinfowler.com/bliki/TestPyramid.html (visited on 31/03/2021).

[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.

[FSB04] F. Fleurey, J. Steel and B. Baudry. 'Validation in Model-Driven Engineering: Testing Model Transformations'. In: *Proceedings. 2004 First International Workshop on Model, Design and Validation, 2004.* Proceedings. 2004 First International Workshop on Model, Design and Validation, 2004. Nov. 2004, pp. 29–40. DOI: 10.1109/MODEVA.2004.1425846.

[GC12] Carlos A. González and Jordi Cabot. 'ATLTest: A White-Box Test Generation Approach for ATL Transformations'. In: *Model Driven Engineering Languages and Systems*. Ed. by Robert B. France et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 449–464. ISBN: 978-3-642-33666-9. DOI: 10.1007/978-3-642-33666-9_29.

[GFS05] T. Gyimothy, R. Ferenc and I. Siket. 'Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction'. In: *IEEE Transactions on Software Engineering* 31.10 (Oct. 2005), pp. 897–910. ISSN: 1939-3520. DOI: 10.1109/TSE.2005.112.

[GKB21] Joshua Gleitze, Heiko Klare and Erik Burger. 'Finding a Universal Execution Strategy for Model Transformation Networks'. In: *Fundamental Approaches to Software Engineering*. Ed. by Esther Guerra and Mariëlle Stoelinga. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 87–107. ISBN: 978-3-030-71500-7. DOI: 10.1007/978-3-030-71500-7_5.

[Gle17] Joshua Gleitze. 'A Declarative Language for Preserving Consistency of Multiple Models'. Bachelor's Thesis. Karlsruher Institut für Technologie (KIT), 2017. DOI: 10.5445/IR/1000076905.

[GMO09] Roy Grønmo, Birger Møller-Pedersen and Gøran K. Olsen. 'Comparison of Three Model Transformation Languages'. In: *Model Driven Architecture - Foundations and Applications*. Ed. by Richard F. Paige, Alan Hartman and Arend Rensink. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 2–17. ISBN: 978-3-642-02674-4. DOI: 10.1007/978-3-642-02674-4_2.

[Gos+21] James Gosling et al. 'The Java® Language Specification'. In: (12th Feb. 2021), p. 844. URL: https://docs.oracle.com/javase/specs/jls/se16/jls16.pdf (visited on 10/04/2021).

[GP09] Pau Giner and Vicente Pelechano. 'Test-Driven Development of Model Transformations'. In: *Model Driven Engineering Languages and Systems*. Ed. by Andy Schürr and Bran Selic. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 748–752. ISBN: 978-3-642-04425-0. DOI: 10.1007/978-3-642-04425-0_61.

[Hen20] Lukas Hennig. 'Describing Consistency Relations of Multiple Models with Commonalities'. Master's Thesis. Karlsruhe, Germany: Karlsruher Institut für Technologie (KIT), 2020.

[Hic19] Rich Hickey. *Clojure.Core - Clojure v1.10.2 API Documentation.* 2019. URL: `https://clojure.github.io/clojure/clojure.core-api.html` (visited on 10/04/2021).

[Hid+16] Soichiro Hidaka et al. 'Feature-Based Classification of Bidirectional Transformation Approaches'. In: *Software & Systems Modeling* 15.3 (1st July 2016), pp. 907–928. ISSN: 1619-1374. DOI: `10.1007/s10270-014-0450-0`.

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

[ISO01] ISO/IEC. *ISO/IEC 9126-1 Software Engineering — Product Quality — Part 1: Quality Model.* June 2001. URL: `https://www.iso.org/standard/22749.html`.

[ISO11] ISO/IEC. *ISO/IEC 25010 Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models.* Mar. 2011. URL: `https://www.iso.org/standard/35733.html`.

[Kah+19] Nafiseh Kahani et al. 'Survey and Classification of Model Transformation Tools'. In: *Software & Systems Modeling* 18.4 (1st Aug. 2019), pp. 2361–2397. ISSN: 1619-1374. DOI: `10.1007/s10270-018-0665-6`.

[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.

[Kap+10] Lucia Kapová et al. 'Evaluating Maintainability with Code Metrics for Model-to-Model Transformations'. In: *Research into Practice – Reality and Gaps.* Ed. by George T. Heineman, Jan Kofron and Frantisek Plasil. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 151–166. ISBN: 978-3-642-13821-8.

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

[KG19] Heiko Klare and Joshua Gleitze. 'Commonalities for Preserving Consistency of Multiple Models'. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C).* Sept. 2019, pp. 371–378. ISBN: 978-1-72815-125-0. DOI: `10.1109/MODELS-C.2019.00058`.

[Kla+20] Heiko Klare et al. 'Enabling Consistency in View-Based System Development – The Vitruvius Approach'. In: *Journal of Systems and Software* (9th Sept. 2020), p. 110815. ISSN: 0164-1212. DOI: `10.1016/j.jss.2020.110815`.

[Kla17] Matthias Klatte. 'Konsistenzhaltung zwischen UML- und PCM-Komponentenmodellen'. Bachelor's Thesis. Karlsruhe, Germany: Karlsruher Institut für Technologie (KIT), 13th June 2017.

[Kol+14] Shekoufeh Kolahdouz-Rahimi et al. 'Evaluation of Model Transformation Approaches for Model Refactoring'. In: *Science of Computer Programming* 85 (June 2014), pp. 5–40. ISSN: 0167-6423. DOI: 10.1016/j.scico.2013.07.013.

[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://ceur-ws.org/Vol-1571/.

[Kra+16] Max E. Kramer et al. 'A Controlled Experiment Template for Evaluating the Understandability of Model Transformation Languages'. In: *2nd International Workshop on Human Factors in Modeling*. Ed. by Miguel Goulão. Vol. 1805. Saint Malo, France: CEUR-WS, Aachen, Oct. 2016, pp. 11–18.

[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.

[Lam07] Maher Lamari. 'Towards an Automated Test Generation for the Verification of Model Transformations'. In: *Proceedings of the 2007 ACM Symposium on Applied Computing*. SAC '07. New York, NY, USA: Association for Computing Machinery, 11th Mar. 2007, pp. 998–1005. ISBN: 978-1-59593-480-2. DOI: 10.1145/1244002.1244220.

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

[Lip82] M. Lipow. 'Number of Faults per Line of Code'. In: *IEEE Transactions on Software Engineering* SE-8.4 (July 1982), pp. 437–439. ISSN: 1939-3520. DOI: 10.1109/TSE.1982.235579.

[LZG05] Yuehua Lin, Jing Zhang and Jeff Gray. 'A Testing Framework for Model Transformations'. In: *Model-Driven Software Development.* Ed. by Sami Beydeda, Matthias Book and Volker Gruhn. Berlin, Heidelberg: Springer, 2005, pp. 219–236. ISBN: 978-3-540-28554-0. DOI: 10.1007/3-540-28554-7_10.

[MBL06] Jean-Marie Mottu, Benoit Baudry and Yves Le Traon. 'Mutation Analysis Testing for Model Transformations'. In: *Model Driven Architecture – Foundations and Applications.* Ed. by Arend Rensink and Jos Warmer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 376–390. ISBN: 978-3-540-35910-4. DOI: 10.1007/11787044_28.

[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 05/10/2017).

[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.

[MV06] Tom Mens and Pieter Van Gorp. 'A Taxonomy of Model Transformation'. In: *Electronic Notes in Theoretical Computer Science*. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005) 152 (27th Mar. 2006), pp. 125–142. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.10.021.

[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.

[NF15] S. Nanz and C. A. Furia. 'A Comparative Study of Programming Languages in Rosetta Code'. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Vol. 1. May 2015, pp. 778–788. DOI: 10.1109/ICSE.2015.90.

[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 15/05/2017).

[Ode+06] Martin Odersky et al. *Identifiers, Names & Scopes | Scala 2.13*. Scala Language Specification. Mar. 2006. URL: https://scala-lang.org/files/archive/spec/2.13/02-identifiers-names-and-scopes.html (visited on 06/04/2021).

[Ore+13] Fernando Orejas et al. 'On Propagation-Based Concurrent Model Synchronization'. In: *Electronic Communications of the EASST* 57.0 (0 16th Sept. 2013). ISSN: 1863-2122. DOI: 10.14279/tuj.eceasst.57.871.

[Reu+16] Ralf H. Reussner et al. *Modeling and Simulating Software Architectures – The Palladio Approach*. Cambridge, MA: MIT Press, Oct. 2016. 408 pp. ISBN: 978-0-262-03476-0. URL: http://mitpress.mit.edu/books/modeling-and-simulating-software-architectures.

[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.

[RRA17] G Ramesh, T V Rajini Kanth and A Ananda Rao. 'Metrics for Consistency Checking in Object Oriented Model Transformations'. In: *Computer Modelling & New Technologies* 21.2 (29th Mar. 2017), pp. 29–36.

[SBM09] Sagar Sen, Benoit Baudry and Jean-Marie Mottu. 'Automatic Model Generation Strategies for Model Transformation Testing'. In: *Theory and Practice of Model Transformations*. Ed. by Richard F. Paige. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 148–164. ISBN: 978-3-642-02408-5. DOI: 10.1007/978-3-642-02408-5_11.

[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.

[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.

[SK03] R. Subramanyam and M. S. Krishnan. 'Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects'. In: *IEEE Transactions on Software Engineering* 29.4 (Apr. 2003), pp. 297–310. ISSN: 1939-3520. DOI: 10.1109/TSE.2003.1191795.

[SKZ14] Leila Samimi-Dehkordi, Alireza Khalilian and Bahman Zamani. 'Programming Language Criteria for Model Transformation Evaluation'. In: *2014 4th International Conference on Computer and Knowledge Engineering (ICCKE)*. 30th Oct. 2014, pp. 370–375. DOI: 10.1109/ICCKE.2014.6993469.

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

[Ste+17] Petr Stefan et al. 'Unit Testing Performance in Java Projects: Are We There Yet?' In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ICPE '17. New York, NY, USA: Association for Computing Machinery, 17th Apr. 2017, pp. 401–412. ISBN: 978-1-4503-4404-3. DOI: 10.1145/3030207.3030226.

[Ste10] Perdita Stevens. 'Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions'. In: *Software & Systems Modeling* 9.1 (1st Jan. 2010), p. 7. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-008-0109-9.

[Ste12] Perdita Stevens. 'Observations Relating to the Equivalences Induced on Model Sets by Beidirectional Transformations'. In: *Electronic Communications of the EASST* 49.0 (0 12th July 2012). ISSN: 1863-2122. DOI: 10.14279/tuj.eceasst.49.714.

[Ste17] Perdita Stevens. 'Bidirectional Transformations in the Large'. In: ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems. 12th June 2017. ISBN: 978-1-5386-3492-9. DOI: 10.1109/MODELS.2017.8.

[Ste18] Perdita Stevens. 'Towards Sound, Optimal, and Flexible Building from Megamodels'. In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. Copenhagen, Denmark: ACM, 2018, pp. 301–311. ISBN: 978-1-4503-4949-9. DOI: 10.1145/3239372.3239378.

[Stu+07] I. Stuermer et al. 'Systematic Testing of Model-Based Code Generators'. In: *IEEE Transactions on Software Engineering* 33.9 (Sept. 2007), pp. 622–634. ISSN: 1939-3520. DOI: 10.1109/TSE.2007.70708.

[Sym18] Torsten Syma. 'Multi-Model Consistency through Transitive Combination of Binary Transformations'. Master's Thesis. Karlsruher Institut für Technologie (KIT), 2018. DOI: 10.5445/IR/1000104128.

[TA15] Frank Trollmann and Sahin Albayrak. 'Extending Model to Model Transformation Results from Triple Graph Grammars to Multiple Models'. In: *Theory and Practice of Model Transformations*. Ed. by Dimitris Kolovos and Manuel Wimmer. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 214–229. ISBN: 978-3-319-21155-8. DOI: 10.1007/978-3-319-21155-8_16.

[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.

[Tho76] Ken Thompson. 'The UNIX Command Language'. In: *Structured Programming* (1976).

[Tro+15] Javier Troya et al. 'Towards Systematic Mutations for and with ATL Model Transformations'. In: *8th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2015)*. IEEE, Apr. 2015, pp. 1–10. DOI: 10.1109/ICSTW.2015.7107455.

[vALvdB09] Marcel F. van Amstel, Christian F. J. Lange and Mark G. J. van den Brand. 'Using Metrics for Assessing the Quality of ASF+SDF Model Transformations'. In: *Theory and Practice of Model Transformations*. Ed. by Richard F. Paige. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 239–248. ISBN: 978-3-642-02408-5. DOI: 10.1007/978-3-642-02408-5_17.

[vAms+11] Marcel F. van Amstel et al. 'Quality Assessment of ATL Model Transformations Using Metrics'. In: *CEUR Workshop Proceedings*. Vol. 711. CEUR-WS.org, 2011. URL: https://research.tue.nl/nl/publications/quality-assessment-of-atl-model-transformations-using-metrics(b777d090-8eee-4796-8ad0-8e88a0850b43).html (visited on 14/04/2021).

[Völ13] Markus Völter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, 23rd Jan. 2013. 558 pp. ISBN: 978-1-4812-1858-0. URL: `http://voelter.de/dslbook/markusvoelter-dslengineering-1.0.pdf`.

[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.

[WAS14] Martin Wieber, Anthony Anjorin and Andy Schürr. 'On the Usage of TGGs for Automated Model Transformation Testing'. In: *Theory and Practice of Model Transformations*. Ed. by Davide Di Ruscio and Dániel Varró. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 1–16. ISBN: 978-3-319-08789-4. DOI: `10.1007/978-3-319-08789-4_1`.

[Wer16] Dominik Werle. 'A Declarative Language for Bidirectional Model Consistency'. Master's Thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT), 2016.

[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`.

[WKC06] J. Wang, S.- Kim and D. Carrington. 'Verifying Metamodel Coverage of Model Transformations'. In: *Australian Software Engineering Conference (ASWEC'06)*. Australian Software Engineering Conference (ASWEC'06). Apr. 2006, 10 pp.–282. DOI: `10.1109/ASWEC.2006.55`.

[Woh+12] Claes Wohlin et al. *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer, 2012. 236 pp. ISBN: 978-3-642-29043-5.

[Xio+13] Yingfei Xiong et al. 'Synchronizing Concurrent Model Updates Based on Bidirectional Transformation'. In: *Software & Systems Modeling* 12.1 (1st Feb. 2013), pp. 89–104. ISSN: 1619-1366, 1619-1374. DOI: `10.1007/s10270-010-0187-3`.

[Yin13] Robert K. Yin. *Case Study Research: Design and Methods*. 5th ed. SAGE Publications, Inc, 10th May 2013. 312 pp. ISBN: 978-1-4522-4256-9.