A Case Study for Networks of Bidirectional Transformations

Master's Thesis of

Timur Sağlam

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

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Prof. Dr. Anne Koziolek
Advisor:	M.Sc. Heiko Klare
Second advisor:	Dr.-Ing. Erik Burger

4. October 2019 – 3. April 2020

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

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

Karlsruhe, 25.3.2020

. . . . . .

(Timur Sağlam)

Abstract

The development of modern software systems often comprises multiple artifacts. However, these artifacts often share a particular overlap in redundant or dependent information, which needs to be kept consistent during the development of the software system. Doing this process manually is labor-intensive and prone to errors. Consistency preservation mechanisms allow keeping these artifacts consistent automatically. They are often based on bidirectional transformations, which update a target model if a source model is modified. While bidirectional transformations are a well-researched topic, consistency preservation of more than two models is yet to receive as much attention. Nevertheless, the development of software systems often involves more than two models. Consequentially, consistency preservation between more than two models is required, which can be achieved using networks of bidirectional transformations.

Such transformation networks combine multiple bidirectional transformations, each concerned with keeping two of the multiple models consistent. Since the development of each transformation requires individual domain knowledge, they are usually developed by several domain experts without each other in mind. Additionally, single transformations may be reused in other networks. However, this is not considered in previous work. This makes consistency preservation by means of networks of bidirectional transformations prone to compatibility errors. For example, in a network of transformations, there may be two or more concatenations of transformations that relate the same metamodels across different other metamodels. Yet, they may relate the elements in different ways, which we call an *incompatibility*. This can, for example, lead to a duplicate creation of the same elements across the different transformation concatenations. However, there is no systematic knowledge about the kinds of compatibility issues that may occur in networks of bidirectional transformations. Consequentially, it is also unclear how to systematically prevent the occurrence of such issues and how far this is possible in the first place.

This thesis conducts a case study to identify which types of issues can arise during consistency preservation through networks of bidirectional transformations. We derive a classification for these issues regarding the knowledge required to avoid them. We distinguish between the knowledge that the transformation may be used in a network and the knowledge about the contents of the other transformations. For issues that transformation developers can prevent, we propose strategies for their systematic prevention during the transformation construction. In our case study, 90% of the issues we found could have been prevented. The remaining issues cannot be avoided during the development of a single transformation, as this requires knowledge about the other transformations in the network. In consequence, this thesis helps transformation developers to systematically avoid faults during the creation of transformations and allows network developers to spot faults that can not be prevented when creating the transformation.

Zusammenfassung

Die Entwicklung moderner Softwaresysteme basiert oft auf mehreren Artefakten. Diese Artefakte teilen sich oft redundante oder abhängige Informationen, welche während der Entwicklung des Softwaresystems konsistent gehalten werden müssen. Die manuelle Durchführung dieses Prozesses ist arbeitsaufwendig und fehleranfällig. Konsistenzerhaltungsmechanismen ermöglichen diese Artefakte automatisch konsistent zu halten. Konsistenzerhaltung basiert oftmals auf bidirektionalen Transformationen, welche ein Zielmodell aktualisieren, wenn ein Quellmodell modifiziert wird. Während das Gebiet der bidirektionale Transformationen stark erforscht ist, hat Konsistenzerhaltung von mehr als zwei Modellen bisher weniger Aufmerksamkeit erhalten. Allerdings umfasst die Entwicklung von Softwaresystemen jedoch oft mehr als zwei Modelle. Folglich benötigt man Konsistenzerhaltung zwischen mehr als zwei Modellen, welche durch Netzwerke bidirektionaler Transformationen erreicht werden kann.

Solche Transformationsnetzwerke kombinieren mehrere Transformationen, wobei jede einzelne für die Konsistenzerhaltung zweier Modelle verantwortlich ist. Da die Entwicklung jeder Transformation individuelles Domänenwissen erfordert, werden sie in der Regel von mehreren Domänenexperten unabhängig voneinander entwickelt. Zusätzlich können einzelne Transformationen in anderen Netzwerken wiederverwendet werden. Dies wird jedoch in bisherigen Arbeiten nicht berücksichtigt, macht aber die Konsistenzerhaltung durch Netzwerke bidirektionaler Transformationen anfällig für Probleme. In einem Netzwerk von Transformationen kann es beispielsweise zwei oder mehr Verkettungen von Transformationen geben, die dieselben Metamodelle mit verschiedenen anderen Metamodellen in Beziehung setzen. Jedoch können sie die Elemente unterschiedlich miteinander in Beziehung setzen. Dies kann zum Beispiel zu einer doppelten Erstellung derselben Elemente über die verschiedenen Transformationsketten führen. Es gibt jedoch kein systematisches Wissen über die Problemarten, die in solchen Netzwerken auftreten können oder ob und wie derartige Probleme systematisch verhindert werden können.

Diese Thesis führt eine Fallstudie durch, die ermitteln soll, welche Arten von Problemen bei der Konsistenzerhaltung durch Netzwerke bidirektionaler Transformationen auftreten können. Für diese Probleme leiten wir eine Klassifizierung hinsichtlich des erforderlichen Wissens für ihre Vermeidung ab. Für Probleme, die Transformationsentwickler verhindern können, schlagen wir Strategien zur systematischen Vermeidung während ihrer Konstruktion vor. In unserer Fallstudie sind 90% der gefundenen Probleme verhinderbar. Die übrigen Probleme lassen sich während der Entwicklung einer einzelnen Transformation nicht ohne das Wissen über weitere Transformationen im Netzwerk vermeiden. Folglich hilft diese Thesis Transformationsentwicklern Fehler bei der Erstellung von Transformationen systematisch zu vermeiden und ermöglicht es Netzwerkentwicklern Fehler zu erkennen, die bei der Konstruktion der Transformation nicht verhindert werden können.

Contents

List of Figures

List of Tables

1. Introduction

Since modern software systems can grow very complex and large-scale, their development often comprises multiple artifacts. These artifacts might represent different parts of the software system. However, they often share a certain overlap in redundant or dependent information. This information needs to be kept consistent during the development and maintenance of the software system. If done manually, this process is labor-intensive and prone to errors. This motivates the need for an automatism that replaces this process. Consistency preservation is such an automatism that keeps artifacts consistent even when a developer changes a single artifact. Many of these consistency preservation mechanisms are based on model transformations. Model transformations update a target model based on a source model. This process is usually defined by multiple transformations rules, which define how to map model elements from the source model to the target model. Most of the research covers bidirectional transformations, which are transformations between precisely two models. Thus, the majority of model consistency preservation mechanisms mainly deal with keeping two models consistent [7].

The problem of keeping more than two models consistent, called *multi-model consistency preservation*, is significantly less researched. However, it is relevant, as more than two models can be used in the development of a single software system. Multi-directional transformations, which are transformations between more than two models, could be used to enable multi-model consistency. The alternative is using networks of bidirectional transformations, where multi-model consistency preservation is achieved by multiple bidirectional transformations, each concerned with keeping two of the multiple models consistent. The advantage of networks of bidirectional transformations is that they are a flexible approach for multi-model consistency preservation, as extending a network is straightforward and parts of the network can even be used for other networks. Networks of transformations are especially well-suited when trying to keep existing metamodels consistent, as pre-existing metamodels and transformations can be integrated into the network with little to no adaption.

Networks of bidirectional transformations can be developed by multiple experts, as their construction requires knowledge about the different domains of the target and source models [20]. The more models need to be kept consistent with each other, the less likely it is to find an expert that is proficient in all domains [43]. Thus, this makes consistency preservation prone to compatibility errors. Especially when the different bidirectional transformations of a network are not designed with each other in mind, or never have been considered for multi-model consistency preservation at all. Model transformations usually assume that their models can be modified by themselves or the user. Other transformations, however, are not considered. This makes the consistency preservation in networks of bidirectional transformations prone to compatibility errors. For example, in a network of transformations there may be two or more concatenations

of transformations that relate the same metamodels across different other metamodels. However, they may relate the elements in different ways, which we call an *incompatibility*. This can, for example, lead to a duplicate creation of the same elements across the different transformation concatenations. It is generally unclear what kind of issues can arise during multi-model consistency preservation through networks of bidirectional transformations. Consequentially, it is also unclear how to systematically prevent issues that arise in networks of bidirectional transformations. However, to make networks of bidirectional transformations a feasible approach for multi-model consistency preservation, it is crucial to identify these issues in order to allow their prevention during the construction of transformations as far as possible. This means we require systematic knowledge on which issues can arise in networks of transformations. Moreover, we need to know, on the one hand, which issues can be prevented before combining the transformations into a network, and on the other hand, which cannot be prevented at all and, thus, might need to be resolved when combining the transformations in a network.

This thesis conducts a case study on networks of bidirectional transformations to explore what types of issues arise during multi-model consistency preservation through networks of bidirectional transformations. We combine pre-existing, independently developed transformations into a network. Then, we use a set of small scenarios to test the consistency preservation on issues that arise when changes to a model are propagated in the network. We differentiate between *mistakes*, *faults*, and *failures*. The inconsistencies that arise during the case study execution are failures. Each fault is the cause of one or many failures and is manifested in the transformation definitions. It is the manifestation of a mistake made by a transformation developer during the planning or implementation of the transformations. From these mistakes, faults, and failures, we derive a classification for these issues concerning their avoidability with the knowledge that the transformations can be used in a network of transformations or with detailed knowledge about the other transformations. We propose systematic strategies on how these mistakes, faults, and failures can be prevented during the construction of the individual transformations. We also discuss some issues that cannot be prevented by construction and what they have in common. These issues might need to be resolved when assembling the network by combining the transformations. This thesis offers the two following envisioned benefits. First, it helps transformation developers to systematically avoid faults during the creation of the transformations as far as possible. Second, it allows network developers to spot faults that can not be prevented when creating the transformation in order to resolve them systematically. Previous work [22, 46] has explored issues with change propagation in simple, linear networks of bidirectional transformations. A linear network is a network where each metamodel is connected through bidirectional transformations with precisely two other metamodels so that the network has a linear topology. This thesis builds on this foundation and further analyzes more complex networks with redundant bidirectional transformations in a comprehensive case study. This means there can be many redundant paths in the network and cycles that are between multiple models.

1.1. Case Study

In this thesis, we conduct a case study on the issues that arise during multi-model consistency preservation with networks of bidirectional transformations. We base our case study on three pre-existing metamodels, namely a PCM metamodel, a UML metamodel, and a Java metamodel. PCM is a component-based model for performance prediction in software architectures. With these metamodels we then build a network out of the three pre-existing bidirectional model transformations $T_{PCM \leftrightarrow UML}$, $T_{UML \leftrightarrow Java}$, and $T_{PCM \leftrightarrow Java}$. These transformations are designed by different experts and therefore, not designed with each other in mind and without the knowledge that they are going to be used in a network of transformations. We utilize a pre-defined set of 39 fine-grained test cases that provide model instances for the metamodels of the network on which the consistency preservation is executed. A test modifies one of the models in the network and then calls the consistency preservation mechanism. The consistency preservation mechanism then executes the model transformation one-by-one until the network is stable, which means no transformation execution leads to any further changes.

We identify different types of mistakes, faults, and failures than can arise during multi-model consistency preservation through networks of bidirectional transformations. A mistake can be, for example, the duplicate creation of a model element. A possible fault that could cause this is the missing check if another transformation has already created the element. This fault is located in one or more transformations. It is also the manifestation of a mistake. In this example, the mistake could be not considering the use of a transformation in a network of transformations. During this case study, we encounter 119 failures, which are caused by 29 faults, which therefore, are the manifestations of 29 mistakes. Note that a single fault can cause multiple failures. We notice the failures during the execution of the test cases. We trace the underlying fault of failure, resolve that fault, and then match all failures that no longer occur to that fault. We reconstruct the mistake that manifested in the fault by analyzing which missing knowledge leads to such a fault. The time-consuming part of this process is tracing the underlying faults of the failures and then manually resolving that fault. We trace the faults by tracking how the initial changes made in the network of transformations are propagated through the network. We then back-track from the failure and analyze each transformation that had a part in the propagation chain until we understand which fault caused the failure. We resolve the fault by manually improving the affected transformations until the fault is fixed. We confirm that the fault is resolved by checking if the correlating failures still occur. We derive a classification for these mistakes, faults, and failures concerning their avoidability with the knowledge that the transformations can be used in a network of transformations or with detailed knowledge about the other transformations.

1.2. Research Questions

Generally speaking, the case study examines which mistakes, faults, and failures arise during multi-model consistency preservation through networks of bidirectional transformations. This serves two goals: For one, we want to find out how far it is possible to design bidirectional transformations in a way so that they will work with any other carefully designed transformation in a network of bidirectional transformations. For another, this case study is supposed to help avoid issues when building bidirectional transformations and networks of transformations. With this case study, we set out to answer the following research questions:

1. What are potential faults, failures, and mistakes that commonly appear during consistency preservation in networks of bidirectional transformations?

2. Which classification can be used to categorize these faults, failures, and mistakes with respect to their avoidability?

3. Which strategies allow the prevention of these mistakes, faults, and failures reliably during the transformation construction and what knowledge is required to do so?

4. Which mistakes cannot be prevented during the construction of bidirectional transformations? What do they have in common?

5. How do redundant transformation rules in networks of bidirectional transformations affect the mistakes, faults, and failures in a network?

6. How do these results compare to the results of previous work [21, 46]?

We answer *Research Question 1* by assembling a network of bidirectional transformations step-by-step out of different pre-existing transformations and checking on faults, failures, and mistakes through running pre-existing test cases for these transformations. The test cases create model elements in one model and then trigger the consistency preservation mechanism, which restores consistency by creating the correlating model elements in the other models. Since the transformations were initially designed for two models, the test cases might result in failures since they are now used with a network of transformations. We then identify the faults that caused the failures and the mistakes that led to the manifestation of the faults and fix the faults in the transformations. Last, we confirm that the fixes resolve the failures by making sure the failures no longer occur when rerunning the correlating tests. We answer *Research Question 2* by building a classification regarding avoidability with the knowledge that the transformations can be used in a network of transformations or with detailed knowledge about the other transformations. We categorize the failures according to the model state, which means the models are either missing elements, have too many elements, or have incorrect elements. Next, we categorize the faults according to their scope, meaning if the fault is technical, transformation-internal, or regards the transformation interaction. At last, we categorize the mistakes according to the knowledge scope, which means which knowledge is required to avoid a mistake. This classification is meant to help to answer the following research questions but also serves

as a catalog that assists with building new transformations that could be used in a network of bidirectional transformations. We answer *Research Question 3* by determining what knowledge is required to be able to implement the fixes we found for the faults during the construction of the transformations and therefore prevent the faults. We answer *Research Question 4* by listing all mistakes that can only be fixed with knowledge of the network topology and the details of how changes are propagated in the network. This is a direct result of the classification built for *Research Question 2*. We answer *Research Question 5* by finding common denominators between all faults, failures, and mistakes that are directly linked to transitive paths in the network. By finding differences and commonalities of our results with the results of previous case studies, we answer *Research Question 6*. We also classify the problems found in our case study according to the classification proposed in previous work.

1.3. Results

We observe that the most common failure class is the duplicate element creation, where two elements are created in a model that are semantically identical. They are often caused by the fault of not considering that, when creating an element, another transformation might have already created a semantically identical element. This fault class is the most common fault in this case study. These faults are manifestations of the most common mistake class, which are mistakes due to not considering transitive consequences in general. Our observations confirm that it is often possible to prevent faults in a network during the construction of transformations with minimal knowledge of the network. Even with just the knowledge that the transformation is used in a network of bidirectional transformations, it is possible to prevent 89.7% of all faults that we encountered. We identified three strategies that prevent different network knowledge mistakes encountered in this case study during the transformation construction. They prevent the manifestation of these mistakes in faults. Therefore, two-thirds of the encountered faults can be prevented with these three strategies alone. In total, only three mistakes we found cannot be prevented during the transformation construction. For these mistakes, it is not possible to predict transitive consequences at all. They all have one thing in common, which is the underlying cause of not being preventable by construction: Transformations are forced to make a decision where there is no inherently correct choice. Consequentially, different transformations in a network of transformations might make a different decision as it is not clear what option other transformations chose.

1.4. Thesis Structure

The remainder of this thesis is structured as follows. First, chapter 2 introduces the foundations regarding the topics of consistency preservation, metamodeling, and model transformations. It also details the notation used in this thesis and explains the difference between mistakes, faults, and failures in detail. Second, chapter 3 covers the topic of multi-model consistency preservation through networks of bidirectional transformations.

It introduces the challenges of such networks of bidirectional transformations and defines several terms that we use throughout this thesis. Third, chapter 4 presents the planning, execution, and results of the case study in detail. It introduces all the artifacts and tools used in the case study. Furthermore, it lists all encountered mistakes, faults, and failures. Fourth, chapter 5 analyzes the encountered mistakes, faults, and failures that we listed in the previous chapter. It also discusses our derived classification and sorts the mistakes, faults, and failures into their respective classes. Additionally, it reasons about the connections of mistakes, faults, and failures. Finally, we compare the classification and the analysis with previous work. Fifth, chapter 6 discusses the lessons that can be learned from the case study results. As a part of that, it covers the prevention strategies but also the unpreventable mistakes. Sixth, chapter 7 discusses the internal and external threats to the validity of this thesis and explains how we address them. Seventh, chapter 8 lists previous work that serves as the foundation for this thesis. Moreover, it discusses general related work to multi-model consistency preservation and networks of bidirectional transformations. Last, chapter 9 concludes this thesis by summarizing the case study and its result. It also details future work.

2. Foundations

This chapter introduces the foundations on which this thesis is based on. First, section 2.1 gives a brief introduction to the field of model-driven software development, which is the general research area to which this thesis is contributing. Second, section 2.2 explains the terminology regarding models and metamodeling. It also introduces the modeling-relation notation used in this thesis. Third, section 2.3 introduces consistency preservation, defines a basic notion of consistency, and explains the general problem of multi-model consistency. Fourth, section 2.4 covers the concept of model transformations and introduces the transformation-relation notation used in this thesis. Fifth, section 2.5 differentiates between the terms mistake, fault, and failure, which is heavily used throughout this thesis. It also explains how they are connected as causal chains.

2.1. Model-Driven Software Development

Model-driven software development applies model-driven engineering to software development and allows abstracting from the complexity of software engineering. This can increase the development speed through automation, increase the reusability through modularization, and improve the maintainability through redundancy avoidance. Model-driven software development is closely related to the Object Management Group's Model-Driven Architecture (MDA). The core idea of model-driven techniques is to work on a higher abstraction layer than traditional artifacts, such as code, by utilizing models of these artifacts. Compared to classic software engineering, which might be model-based, models become primary artifacts. For example, in model-driven software development, models are used to automatically generate code, while software development usually only uses models to generate an abstract view of the source code. In the context of model-driven software development, a model is an abstract representation of the structure of a system [40].

2.2. Metamodeling

A model is, generally speaking, a conceptual representation of something. While some might argue that "everything is model" [3], the concept of a model is commonly defined according to Stachowiak's three properties of a model [39]: First, a model is a *mapping* of the archetype it is modeling. Second, it is an abstraction or *reduction* of this archetype, meaning not every detail is modeled. Third, it has a *pragmatism*, as it was created for a specific purpose. A model can also describe other models. In this case, it is an abstract description of a model, and therefore the model of a model. For models of models, the

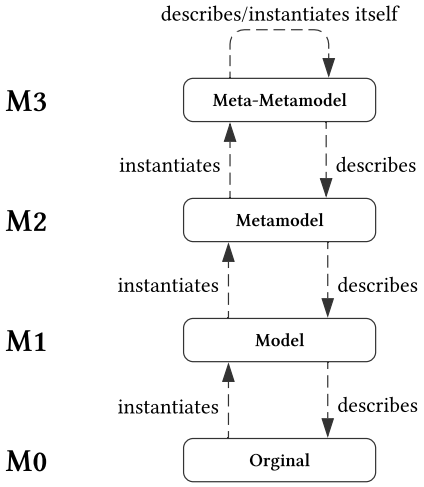

Figure 2.1.: The different model concepts according to the MOF meta-levels. Each model in a level describes the one below and instantiates the one above. The exception is the meta-metamodel which is self-describing and therefore self-instantiating.

term *metamodel* is used. A metamodel can be seen as a blueprint for specific models. Consequentially, any model is an instance of a metamodel. This can be taken one step further: When a model describes metamodels, it is a *meta-metamodel*. To differentiate between the abstraction level of the concepts mentioned above, the term *metalevel* is used. A model is one metalevel above its modeled original because its abstraction level is higher than the original. Thus, metamodels are one metalevel above their models, and meta-metamodels are one above their metamodels [40]. While one could endlessly define meta-levels above meta-levels, it is common practice to stop at the metalevel of meta-metamodels. Consequentially, meta-metamodels are seen as self-describing and, therefore, also instances of themselves. The Object Management Group defines four meta-levels: The instance level, the model level, the metamodel level, and the meta-metamodel level. This is depicted in Figure 2.1. They also define a meta-metamodel: The *meta object facility* (MOF). *Essential MOF* (EMOF) is a subset of the meta object facility. It allows the simplified creation of metamodels. The unified modeling language (UML), a well-known graphical modeling language developed by the Object Management Group, is an instance or application of the meta object facility [15].

Less formally, models can be seen as sets of elements [22]. We call these elements model elements. Multiple model elements can conform to a common description, a *model element type*. Sometimes a model element type is also referred to as the *metaclass* of a model element. In this notion, a metamodel describes all possible models, which means it is the set of all possible sets of model elements. Models can still be considered instances of their metamodels, as the model elements are considered as instances of their types. In order to apply this definition to the models we are observing, we need two additional concepts. First and foremost, model elements can have relations with other model elements. Second, a model element might have properties, also called attributes. Both the properties and the relations are defined by model element types but might differ in their values between model elements of the same types. These two concepts are in accordance with the EMOF

standard. When talking about the structure of a model, we are describing the specifics that relations between elements have due to their definitions in the metamodel. Containment relations are stronger relations that describe the concept of ownership. The contained element depends on the containing element. These containment relations define a partial order in the model. Generally, every model element needs to be contained in another model element, which forms a partial order in the model. The exception to this rule are root elements, also called model roots, which are not contained. This partial order of containment relations, in addition to the non-containment relations, is considered the structure of a model.

In this thesis, we use the following notation, which is similar to the notation used by Klare et al. [22]. A model is denoted as M, while a metamodel is denoted as $\mathcal{M}$. If a model is an instance of a metamodel, we express that as $M \in \mathcal{M}$, as it is one of the many possible models that the metamodel describes. A model element type $T \in \mathcal{M}$ is part of that metamodel, while the model element $t \in T$ is therefore part of the correlating model M, denoted as $t \in M$.

2.3. Consistency Preservation

Consistency preservation describes the problem of keeping different artifacts, or more specifically models, of a software system consistent. These models share some amount of *overlapping information*, but their representation of this overlapping information might differ. The definition of the term consistency itself depends on the specific models and how strict of a consistency notion is required. There is an ongoing discussion about different consistency terminologies, especially in regard to strictness. In some cases, inconsistencies might even be tolerated, requiring some sort of partial consistency.

To provide a framework for this thesis, let us define consistency for any two metamodels according to the following terminology, which is based on the definitions by Klare et al. [22, 20]. Consistency is defined between two metamodels by *consistency relations*, which describe the dependencies between the overlapping information of the two metamodels. *Consistency constraints* are derived from these consistency relations for any pair of models of these metamodels. Consequentially, there is a subset of all model pairs of these metamodels, which is the set of consistent model pairs. If this set is empty, no model pair can be consistent. *Consistency restoration* describes the process of restoring consistency between two models that do not satisfy one or more consistency constraints. This requires changing one or both models until the pair satisfies all consistency constraints. Finally, consistency preservation is defined as the process of keeping models consistent by utilizing consistency restoration as soon as one or more consistency constraints are no longer satisfied.

Multi-model consistency preservation describes the problem of keeping more than two models consistent. For multi-model consistency preservation, the previous definitions need to be extended to allow for multiple metamodels and, thus, multiple models. Consistency relations can now be n-ary relations, and consistency constraints might be defined for multiple models. Consequentially, consistency restoration needs to be able to restore consistency for more than two models. There are different approaches to realize multi-model consistency preservation. Some approaches tend to merge the different models into

a single underlying model. Other approaches keep the models separate and utilize model transformations for consistency preservation [28, 29].

2.4. Model Transformations

Model transformations were famously described as the "heart and soul of model-driven software development" [38]. According to Kleppe et al. [23], a model transformation is the automatic generation of a target model correlating to a source model. Furthermore, they describe a transformation definition as a set of transformation rules that define a model transformation rule. Each transformation rule is a description of how constructs can be transformed from the source model to the target model. These constructs, according to our notion of models, are either model elements, their properties, or their relations. The transformation rules are usually written in a transformation language and then executed by a transformation engine. This is depicted in Figure 2.2. Note that the transformation rules are written on the metalevel of the source and target metamodels. This means they describe the transformation of model element types. The transformation engine then executes the transformation rules on the metalevel of the models and transforms the actual model elements.

Consequentially, when we say a model is transformed by a transformation, we technically mean the model is transformed by the transformation engine according to the transformation rules. We also distinguish two terms regarding model transformations: *Mapping* and *matching*. When talking about *mapping* of elements, we mean that the transformation rules map elements of the metamodels, as in model elements types, to each other. When talking about *matching* of model elements, we mean pairing actual model elements during the transformation execution. A mapping defines that two model element types of the two source and target metamodels correspond to each other, while a matching defines that two model elements of the two source and target models correspond to each other. Correspondence, therefore, denotes an inter-metamodel relationship. So far, we only discussed that a model transformation transforms source models to target models. These transformations are *unidirectional*. For *bidirectional* transformations, both metamodels are source and target model at the same time, as models can be transformed in both directions. A bidirectional transformation can be broken down into two unidirectional transformations. One is called the forward transformation, and the other is called the backward transformation. Similarly, we only discussed binary transformations, meaning the transformation defined between two metamodels. In addition tho binary transformations, there are also *multiary* or *n-ary* transformations, that are defined between more than two metamodels. However, for this thesis, we mainly focus on binary transformations, as they are used in our case study in a network of transformations. There are many different kinds of transformations. Delta-based transformations are a specific category of transformations that transform a set of changes to a source model to corresponding changes for a target model. Instead of transforming the whole model every time the transformation is executed, only the changes are transformed. We call sets of changes also change sequences. Delta-based model transformations can be used for the consistency restoration of two models.

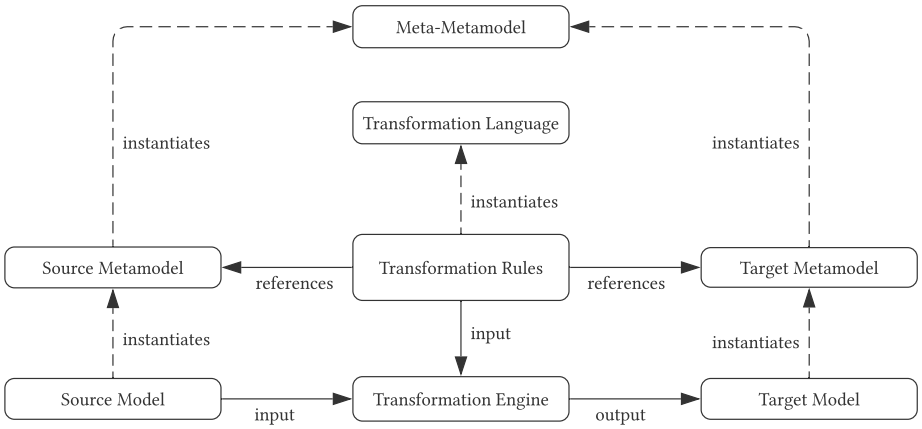

Figure 2.2.: Depiction of the model transformation concept [5]. A model transformation consists of multiple transformation rules, written in a transformation language, which are executed by a transformation engine.

When one model is changed, the other one is updated according to the transformation rules to keep them consistent.

In this thesis, we use the following notation. A transformation that transforms from a source metamodel $\mathcal{M}_{source}$ to a target metamodel $\mathcal{M}_{target}$ is defined as $T_{source \to target} : \mathcal{M}_{source} \mapsto \mathcal{M}_{target}$. If the transformations is bidirectional, the transformation can transform models in both direction. As a consequence, it is defined as $T_{source \leftrightarrow target} : \mathcal{M}_{source} \mapsto \mathcal{M}_{target} \wedge \mathcal{M}_{target} \mapsto \mathcal{M}_{source}$. When using the target model of a transformation as the source model of another transformation, we call this chaining of transformations. We abbreviate such chains as $T_{first \to second} \cdot T_{second \to third} : M_{first} \mapsto M_{third}$.

2.5. Mistakes, Faults, and Failures

In this thesis, we avoid the terms *problem* and *error* as they are too generic and could describe very different things depending on the context. While the terms used in related literature differ, we use the following terminology to differentiate problems according to their cause, manifestation, and impact in the context of software engineering:

1. ***Mistakes*** are a wrong judgment made by a person, for example, a developer.

2. ***Faults*** are the manifestation of the mistakes in an artifact, for example, the code.

3. ***Failures*** are how faults show themselves, for example, during code execution.

This means that when a mistake is made, it can potentially manifest itself in a fault. Moreover, when a fault is created, it can potentially show itself through the appearance of failures. It is important to realize that not every mistake leads to a fault, and not every fault causes a failure. Failures usually lead to a detection of such a mistake-fault-failure chain, which we call causal chains. The relations mentioned above are visualized in Figure 2.3.

Figure 2.3.: UML representation of the concept of a causal chain between a mistake, a fault, and several failures.

The fault is the part that needs to be fixed, and the mistake usually needs to be understood to implement a precise and compact fix correctly.

As an example, in the context of this case study, a mistake could be ignoring or forgetting a dependency between model elements while creating a transformation. The correlating fault would be the incorrect transformation routine that, as an example, should rename the depending element when renaming the element on which it depends. This leads to the failure where some elements end up with invalid or illegal namespaces.

For this case study, that means that any inconsistencies that arise during the test case execution are failures. The fault that causes such failure lies in the implementation of the bidirectional transformations. We try to reconstruct the mistakes made by the author of the bidirectional transformation where the fault is located, as it is the key to preventing such faults in the future.

3. On Networks of Bidirectional Transformations

In the previous chapter, we discussed the basics of consistency preservation and introduced the problem of multi-model consistency preservation. This chapter, however, discusses how networks of bidirectional transformations can achieve multi-model consistency preservation. It introduces the challenges of such networks of bidirectional transformations and defines several terms that we use throughout this thesis. First, section 3.1 explains how multi-model consistency preservation can be achieved using networks of bidirectional transformations and how changes to a model are propagated in the network. It also discusses what network topologies might exist and how they influence change propagation in the network. Second, section 3.2 gives two examples for issues that can occur in these networks of bidirectional transformations: Name transformation and duplicate element creation.

3.1. Achieving Multi-Model Consistency Preservation

As previously mentioned, multi-model consistency preservation describes the problem of keeping more than two models consistent. Consistency in this context describes the requirement that for any set of the models, all overlapping information is consistent. The overlapping information could be anything from only small parts of the models to the whole models themselves. Overlapping information can be represented in the different models very similarly or with a completely different representation. There are different approaches to realize multi-model consistency preservation. Some approaches tend to merge the different models into a single underlying model. Other approaches keep the models separate and utilize model transformations for consistency preservation [28, 29].

When keeping models consistent, consistency relations describe dependencies between the overlapping information. They define which model elements are mapped to each other. For multiple models, these relations are n-ary relations, since a model element might be mapped to multiple elements from different models. N-ary consistency relations can be represented through multiple binary relations by creating a binary relation for any pair of two models in the n-ary consistency relation. This is depicted in Figure 3.1, where three binary relations can replace the n-ary relation between three models.

This leverages the approach of achieving multi-model consistency preservation by defining multiple bidirectional transformations between models, each representing a binary consistency relation and, therefore, all together represent the n-ary consistency relation required to define the overlapping information. This approach forms networks of bidirectional transformations by combining multiple model transformations in one

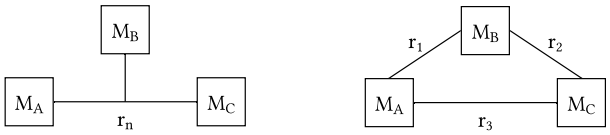

Figure 3.1.: The n-ary relation r_n between the three models M_A, M_B, and M_C (left) can be represented by the three binary relations r_1, r_2, and r_3 (right).

system. These networks of bidirectional transformations can be considered as (directed) graphs, where the models are the nodes of the graph, and the (unidirectional) binary transformations are the edges of the graph. This means that removing all fully redundant bidirectional transformations of the network results in the transitive reduction of the original network. Consistency is restored by propagating changes through the network, which means progressively executing the bidirectional transformations in a certain order until the network reaches a stable state where no further changes are made. Assuming the bidirectional transformations are all correctly implemented and have no side effects due to their use in the network, we assume that the stable state of the network is also a state where consistency is restored for all networks. Bidirectional transformations can be executed multiple times as cycles in the network are not uncommon. The execution order of the transformation affects how changes are propagated in the network, but in this thesis, we are abstracting from the execution order. Finding an approach to optimize the execution order to reduce the issues in a network of transformations is beyond the scope of this thesis.

Note that similarly to the distinction between metamodels and models as well as transformation definition and transformation execution, a distinction can be made for networks of transformations. Networks of bidirectional transformations are constructed with metamodels and transformation definitions. But during the consistency restoration, a network consists of the correlating models of the metamodels, and changes are propagated due to transformation execution.

It is important to mention that such a network does not need a bidirectional transformation from each model to each other model since transitive change propagation enables the chaining of model transformations. This means, assuming the existence of three models M_A, M_B, and M_C, changes can be propagated transitively from M_A to M_C by chaining two transformations $T_{A\to B} : \mathcal{M}_A \mapsto \mathcal{M}_B$ and $T_{B\to C} : \mathcal{M}_B \mapsto \mathcal{M}_C$. This network is depicted in Figure 3.2. Because of this transitive change propagation, the third transformation from $T_{A\to C} : \mathcal{M}_A \mapsto \mathcal{M}_C$ can be seen as redundant in this specific network because the chain $T_{A\to B} \cdot T_{B\to C} : M_A \mapsto M_C$ is equal to $T_{A\to C}$, which means it is technically not needed, assuming the $T_{A\to B}$ and $T_{B\to C}$ cover the same transformation rules as $T_{A\to C}$ would. This last requirement is important, as redundancy in such a network is not a binary property. Transformations and chains of those can be fully redundant, partly redundant, or not redundant at all. If a chain of transformations only allows the transformation of certain model elements, the correlating direct transformation cannot be removed from the network without altering the functionality of the network itself. Returning to the previous example, $T_{A\to C}$ would be partly redundant, if the intersection between the transformation rules by

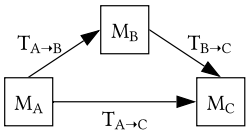

Figure 3.2.: Network of bidirectional transformations with the models M and the transformations T. Transitive chaining of the transformations $T_{A \to B}$ and $T_{B \to C}$ makes $T_{A \to C}$ redundant.

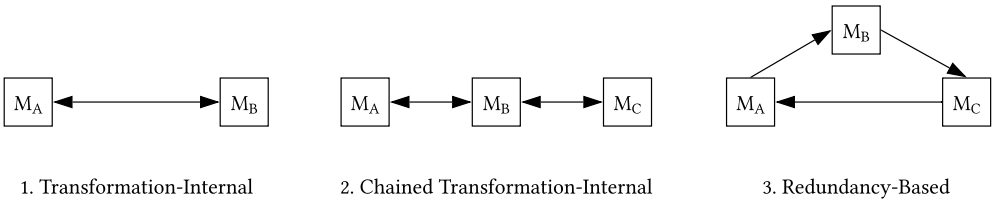

Figure 3.3.: Cycle types in networks of bidirectional transformations. First, transformation-internal cycles consisting of the backward and forward transformations. Second, chains of combined transformation-internal cycles. Third, redundancy-based cycles that do not depend on bidirectionality.

$T_{A \to B}$ and $T_{B \to C}$) is only a subset of the set of transformation rules by $T_{A \to C}$. This subset is the set of redundant transformation rules. This intersection can be characterized as compatibility between $T_{A \to B}$ and $T_{B \to C}$, which is indicative of how well these two transformations can be chained. While the previous example (see Figure 3.2) uses unidirectional transformations, transitive chaining also works with bidirectional transformations. In that case, changes can be propagated in both directions through the chain of transformations.

With this concept of redundancy, we can differentiate between two types of cycles in networks of transformations: First, every bidirectional transformation, by definition, forms a cycle between its two models. We call these kinds of cycles *transformation-internal cycles*. In addition, larger cycles may also exist that consist of multiple bidirectional transformations. These could just be multiple transformation-internal cycles chained together linearly, which we consider as an extension of transformation-internal cycles and call *chained transformation-internal cycles*. But larger cycles could also be a different second type of cycle that does not necessarily rely on the bidirectionally of its transformations. These cycles consist of multiple bidirectional transformations that are connected in a circular chain. Naturally, this means they rely on redundancy. Therefore we call them *redundancy-based cycles*. Examples for these cycle types can be seen in Figure 3.3: The transformation-internal cycle $M_A \to M_B \to M_A$ (1.) consists of a single bidirectional transformation $T_{A \leftrightarrow B}$. The chained transformation-internal cycles (2.) consist of two bidirectional transformations and relies on their bidirectionality. The redundancy-based cycle (3.) consists of multiple bidirectional transformations and relies on redundancy.

In general, these networks of transformations can have a very different topology (see Figure 3.4). The topology can have effects on the change propagation, mainly through

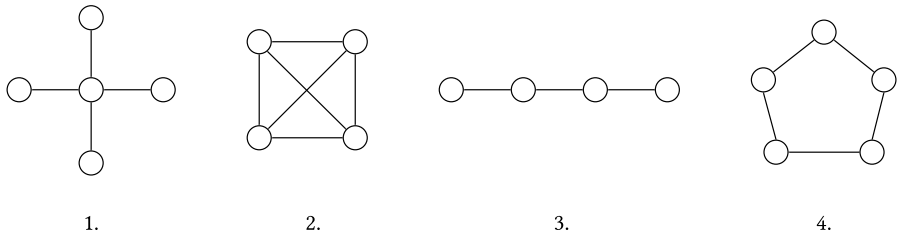

Figure 3.4.: Different network topologies: Star topology (1.), fully connected topology (2.), linear topology (3.), and circular topology (4.).

cycles, redundancy, and bottlenecks. Cycles and redundancy may increase the chance of side effects in the network because they increase the number of paths the change propagation can take in the networks. Technically, a cycle produces an infinite number of paths, but we expect to see only a finite number of paths since we expect the change propagation to terminate. A bottleneck in a network of transformations is a model or transformation which is always part of the path between two models. A bottleneck transformation equates to a minimum cut in the network. This means the transformation rules of the bottleneck transformation, or the overlapping information of the bottleneck model with its neighboring models on the path, limits what information can be shared between the two models at the beginning and the end of the path.

In Figure 3.4, we can see how the topology affects the number of cycles and bottlenecks. The star topology does not have transitive cycles but has a severe bottleneck in the center. The fully connected topology avoids bottlenecks but has a high number of transitive cycles. It also has a large number of transformations compared to its number of models, which increases the effort it takes to implement such a network. The linear topology does not have transitive cycles but has multiple bottlenecks, as every transformation and model on a path between two models in this topology is a bottleneck. Lastly, the circular topology has no bottlenecks and only one large transitive cycle that includes every model in the network.

It is important to mention that cycles and redundancy are not always avoidable in a real system, as every pair of models only has certain overlapping information, and therefore the transformation between them only specifies transformation rules for this overlapping information. This can be seen in Figure 3.5, where the network of transformations contains a transitive cycle, but none of the transformations are redundant, as the overlapping information of each model pair is exclusive to the pair. This also suggests that the topology of a network is not freely choosable since it depends on the degree of overlapping information between the different models. For example, the topology of the network in Figure 3.5 cannot be altered without breaking the functionality of the network.

3.2. Issues in Networks of Transformations

With this section, we illustrate some issues that can occur in networks of bidirectional transformations. These examples are meant to give an understanding of what kinds of

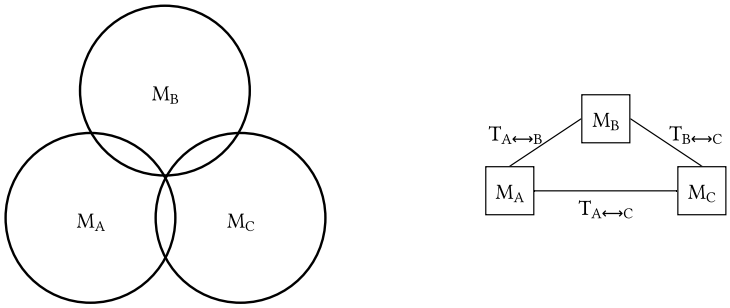

Figure 3.5.: Overlapping information between models M_A, M_B, and M_C on the left and a correlating network of binary transformations on the right. Only a subset of the information in the model is shared.

mistakes, faults, and failures we discuss in this thesis. First, we start with a basic example regarding name transformation, which is discussed in detail. Second, we address duplicate element creation, which is the most common failure we observed in this case study. This failure is explained on a more abstract level but closely related to our case study.

For the first issue, let us assume we transform between three metamodels in a network. For our issue, we only need to consider three element types. PCM repositories are named elements without any naming rules. UML packages are named elements where the first letter of the name is usually capitalized. Java packages are named elements with the same naming rules as UML packages. Figure 3.6 depicts the transformation rules for keeping the names of the three element types consistent. Between PCM and UML the bidirectional transformation $T_{PCM \leftrightarrow UML}$ enforces that the respective names start with lowercase and uppercase letters. This means, for example, a capitalized UML package name is transformed to a lowercase PCM repository name. Between Java and UML the bidirectional transformation $T_{UML \leftrightarrow Java}$ enforces that names start with a lowercase letter. However, both Java and UML packages have the same naming scheme anyways. Between PCM and Java, however, the bidirectional transformation $T_{PCM \leftrightarrow Java}$ does not enforce any naming schemes, which means names are transformed as they are. The fault lies in the fact that $T_{PCM \leftrightarrow Java}$ does not change the name, but $T_{PCM \leftrightarrow UML}$ capitalizes the first letter when transforming to PCM models. Consequentially, failures can occur due to this fault of mismatching transformation rules. There could be multiple failures that could occur to such a fault. For example, there could be alternating loops, where $T_{PCM \leftrightarrow UML}$ enforces a lowercase first letter in the repository name, followed by $T_{PCM \leftrightarrow Java}$ enforcing an uppercase first letter, which then repeats. This means the change propagation is non-terminating. Another failure that can occur due to this fault is the creation of two PCM repositories. The transformation $T_{PCM \leftrightarrow UML}$ might create one repository with a lowercase name, while $T_{PCM \leftrightarrow Java}$ might create one with a capitalized name.

This brings us to the second issue we want to discuss in this section. Many faults manifest themselves in failures of duplicate element creation. This means that two elements are created in the same model, that are semantically identical. Consequentially, only one should have been created, as both elements represent the same conceptual entity. An example of a fault that causes a duplication failure is the fault of not checking on the

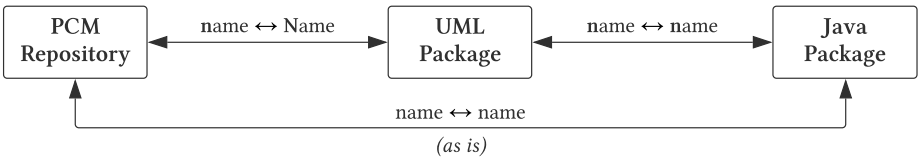

Figure 3.6.: Depiction of the transformation rules for keeping the names of the PCM Repositories, UML packages, and Java packages consistent. As $T_{PCM \leftrightarrow Java}$ does not change the name, but $T_{PCM \leftrightarrow UML}$ capitalizes the first letter, failures can occur due to the fault of mismatching transformation rules.

external creation of an element by another transformation. This can be the manifestation of the mistake of not considering the use of a transformation in a network. We differentiate between two types of duplicate creation: Duplicate creation with co-existence and duplicate creation with overwriting. Co-existence means while two semantically identical elements were created, they both exist without interfering with each other. Both can be changed or deleted, but depending on the access mechanism, meaning how the element is referenced or located, either one of them will be changed without much control on which one is changed. Overwriting means that two semantically identical elements exist, but the creation of the second one overwrites a reference to the first one or a means of locating the first one. This means that from the moment of the creation of the second one, the first one cannot be accessed anymore. Previous changes to the first element are lost, and future changes will only affect the second one. The big difference between the two types of duplicate creation is that co-existence is more easily detected. In the best case, it can be as simple as checking if two classes in a package have the exact same name. It is considerably harder if there are no unique properties to identify duplicates with. Overwriting, on the other hand, is harder to detect, as the access to the element is affected. Consequentially, duplicate creation with overwriting shows itself in this case study mainly through a failure caused by the overwriting and not through the overwriting itself.

4. Case Study

In this thesis, we conducted a case study on the mistakes, faults, and failures that arise during multi-model consistency preservation with networks of bidirectional transformations. We based our case study on three pre-existing metamodels, namely a PCM metamodel, a UML metamodel, and a Java metamodel, as well three bidirectional transformations between these metamodels. With this case study, we set out to answer the following six research questions. *Research Question 1* asks which mistakes, faults, and failures commonly appear in networks of bidirectional transformations. *Research Question 2* asks how these mistakes, faults, and failures can be classified with respect to their avoidability. This also enables gaining a better understanding of their connections. *Research Question 3* asks which prevention strategies can prevent these mistakes during the transformation construction and also asks which knowledge is required to do so. *Research Question 4* asks about the existence of mistakes that cannot be prevented by transformation construction. *Research Question 5* asks about the effects of redundancy in networks of bidirectional transformations. Last, *Research Question 6* asks about the comparison to previous work. To answer these questions, we conduct a case study on networks of bidirectional transformations. We identify different types of mistakes, faults, and failures that can arise during multi-model consistency preservation in networks of bidirectional transformations. This chapter presents the planning, execution, and results of the case study in detail.

First, section 4.1 explains the context of the case study. It discusses which assumptions were made and why they are reasonable in practical applications of consistency preservation mechanisms. Second, section 4.2 documents the artifacts and tools used in the case study. These artifacts and tools include the consistency preservation framework, three metamodels, six transformations, and a set of fine-grained test cases. Third, section 4.3 describes the individual stages of constructing the network of bidirectional transformations. Fourth, section 4.4 lists all encountered problems and how they manifest themselves in the test cases.

4.1. Context and Assumptions

This case study investigates problems that arise during multi-model consistency preservation based on networks of bidirectional transformations. There are several essential specifics to the use cases of such multi-model consistency preservation, as they considerably contribute to the challenge of keeping multiple models consistent. In software engineering, amongst other industries, companies and organizations tend to keep systems for a long time [30]. Legacy systems are, therefore, unlikely to be rebuilt for new use cases. This suggests that it is reasonable to expect that models and transformations are not newly

built for consistency preservation, but instead already exist as part of established systems. Therefore we make the following assumptions:

First, we assume the metamodels are pre-existing. In many cases, metamodels are already available or even in use. These metamodels could be explicit metamodels, such as performance metamodels, or implicit metamodels, such as domain models in software systems. Moreover, these metamodels could be standardized, such as the Object Management Group's UML [4, 33, 8]. On the opposite, these metamodels could also be highly proprietary and specific to a single company or organization, for instance, models for automotive electric and electronic architectures [35]. As these metamodels were designed for different tasks, they might have different structures, which makes keeping them consistent considerably harder. Second, we assume that some or all model transformations might be pre-existing. This means they were most likely not designed with each other in mind. Additionally, they were most likely designed by different experts, as their construction requires knowledge about the different domains of the target and source models [43, 20]. The more models need to be kept consistent with each other, the less likely it is to find an expert that is proficient in all domains. While at least many bidirectional transformations might be designed by a single expert, it might also be the case that two models need to be kept consistent with two independently designed unidirectional transformations, which could mean they are not even fully compatible with each other. Third, we assume that the amount of overlapping information might vary depending on the metamodels. Some may have minimal overlap, while others may nearly overlap completely. As a consequence, there might be some redundancy between transformations or even chains of transformations. This also means that the possibility of transitive cycles in the network increases. Moreover, the density of the network might be high in order to allow consistency preservation of all overlapping information. This makes it hard to predict the topology of such networks, as that depends highly on the metamodels that need to be kept consistent. Lastly, we try to abstract from the execution order of individual transformations in the network. It is reasonable to assume that strategies for finding a deterministic execution order might help to avoid problems during consistency preservation. However, as we want to find problems in our case study, the execution order of the transformations is random, and therefore non-deterministic.

4.2. Artifacts and Tools

The network of bidirectional transformations analyzed in this case study contains three metamodels and three bidirectional transformations. As previously mentioned, all metamodels and transformations are pre-existing and were mostly not designed with each other in mind. In this section, we discuss the case study artifacts and tools in detail: The consistency preservation framework, the metamodels, the bidirectional transformations, and the test case set.

4.2.1. Framework

The case study is conducted with the Vitruvius framework [24]. It is an Ecore-based framework for consistent system development and enables transformation-based consistency preservation. Vitruvius works delta-based, which means that fine-grained sequences of atomic changes to the source model are recorded by the framework. The change sequences are then transformed with the correlating bidirectional transformation to the resulting change sequence for the target model. As the last step, Vitruvius applies the resulting change sequence to the target model [24]. In the Vitruvius framework, bidirectional transformations are written in the Reactions language [19], an imperative, domain-specific language for defining unidirectional consistency preservation.

This means that a bidirectional transformation needs to be designed as two separate unidirectional ones. This makes each bidirectional transformation vulnerable to internal compatibility problems, as the transformations rules of the forward and backward transformation of each bidirectional transformation might not be fully compatible. For example, the forward transformation might cover additional element types with its transformation rules. Consequentially, elements of this type would only be transformed in one direction.

4.2.2. Metamodels

This case study bases its network of bidirectional transformations on three metamodels: The Palladio Component Model (PCM) [32, 2] is a model for the performance prediction of software by using component-based software architectures. It allows modeling software systems before actually implementing them to detect issues such as bottlenecks that impact the software's performance early on during the development. The PCM metamodel is EMF-based and can, therefore, be used directly with Vitruvius. It is the first domain metamodel used in our network. However, due to the transformations used in this case study, we only use repository models. They include elements like repositories, components, operation interfaces, operation signatures, data types, and assembly contexts.

The Unified Modeling Language (UML) [4, 33, 8] defined by the standards consortium called Object Management Group (OMG) is a heavily standardized language for the description and visualization of the design of a system in software engineering. UML offers different models for different purposes. They can be divided into structural and behavioral models. UML class models are structural models for the object-oriented architecture of software systems. They contain information about classes, their properties, and their relation. UML class models can be used to model code from any object-oriented language. The Eclipse Model Development Tools (MDT) [11] offer an EMF-based implementation of UML. The MDT UML class metamodel is the second domain metamodel used in our network. We only use UML class models, which contain elements like packages, types, operations, attributes, references, multiplicities, realization, and generalization.

Java [13, 1, 14] is a widely-used object-oriented, general-purpose programming language. It was initially developed in 1995 by James Gosling at Sun Microsystems, which is now owned by Oracle Corporation. Java code is compiled to Java bytecode and executed on the Java Virtual Machine. In itself, Java does not offer an EMF-based metamodel and does not conform to the EMOF standard. The language itself implicitly defines a metamodel

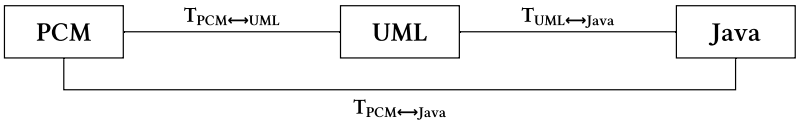

Figure 4.1.: The three bidirectional transformations used in the network of this case study. Each bidirectional transformation consists of two each other opposing unidirectional transformations.

through its language specification [14, 34]. However, in order to use Java with Vitruvius, an explicit EMF-based metamodel is required. The Java Model Printer and Parser (JaMoPP) [17, 16] offers such an EMF-compatible metamodel, and therefore we utilize it to bridge between Vitruvius and Java code. Consequentially, the JaMoPP Java metamodel is the third domain metamodel used in our network. However, due to the transformations used in this case study, we mainly use the parts of the metamodel that describe the structure of object-oriented code. To be specific, that means packages, compilation units, types, fields, method signatures with their parameters, constructors, references, generalization relations, and realization relations.

4.2.3. Transformations

All six model transformations used in this case study, namely $T_{PCM \to UML}$, $T_{UML \to PCM}$, $T_{UML \to Java}$, $T_{Java \to UML}$, $T_{PCM \to Java}$, and $T_{Java \to PCM}$, are developed with and for the Vitruvius framework using the Reactions language. They are pre-existing, which means they were not explicitly designed for this case study and especially not designed with the intention of combining them. While this case study analyzes networks of binary bidirectional transformations as depicted in Figure 4.1, the six transformations used are technically unidirectional. Two unidirectional transformations that transform the same models but in opposite directions, can form the backward and forward transformation of the bidirectional transformation [45]. In order to do so, it is important that their transformation rules cover the same elements of the models. If this is not the case, it can cause issues during consistency preservation. We experienced these issues with $T_{PCM \to Java}$, and $T_{Java \to PCM}$. We formed the bidirectional transformations $T_{PCM \leftrightarrow UML}$, $T_{UML \leftrightarrow Java}$, and $T_{PCM \leftrightarrow Java}$ out of pairs of of our six individual unidirectional transformations. For an overview of the element mappings of these transformations, see Table A.1.

The unidirectional transformations between the PCM and UML metamodels were developed together with each other in mind. Due to the fact that they were created as preparation of a previous case study [46], they employ some patterns that make them more flexible regarding existing model elements created by other transformations when used in a network. The unidirectional transformations between the UML and Java models were developed together with each other in mind as well. However, they were never intended for use in a network of transformations. The unidirectional transformations between the PCM and Java models were developed independently and therefore are not designed with each other in mind. They were also not designed by the same person. The transformation from PCM to Java is very comprehensive, while transformations from Java to PCM is

very minimalistic. That means they only partially complete each other to a bidirectional transformation, as the PCM to Java transformation covers more model elements. These differences in the three bidirectional transformations are beneficial for this case study, as different levels of compatibility allow us to find a broader set of issues.

Due to the unique characteristics in the structure of the models, the transformations deal with different problems. Both unidirectional transformations from UML and Java to PCM require their source elements to be contained in particular compositions to map those elements to a PCM element. For example, not every Java or UML package can be mapped to a repository, system, or component. While the two unidirectional transformations between the Java and UML models certainly profit from the similarities of the two models, they also need to deal with the atypical structure of the Java model, where a model consists of multiple model roots, which are packages and compilation units, due to the structure of the metamodel. This choice of metamodel structure is most likely influenced by the Java language specification [14], where packages are explicitly meant to be independent of each other, even if the namespaces suggest some sort of containment hierarchy. The UML model conforms to the more typical tree structure where everything is contained in a tree with a single root element. Both unidirectional transformations from PCM to the UML and Java models need to deal with the creation, adaption, or deletion of many target elements for one source element, as a single PCM element often maps to a large number of UML and Java element. The reason for this is that the PCM model elements mostly cover elements on a higher level of abstraction compared to UML and Java. As an example of this, a single assembly context in the PCM model is mapped to a package, a class in that package, a field in the class, a constructor in the class, and an initialization statement for the field in the constructor.

4.2.4. Test Cases

When keeping multiple models consistent, overlapping information is made explicit with consistency relations. While these relations define consistency dependencies for single elements, often, multiple elements have particular conceptual commonalities. We, therefore, define the term *concepts* as a set of element types which represent the same conceptual information in their respective metamodel. Therefore, a concept represents overlapping information, and the model element types of the concepts need to be kept consistent. As an example, Figure 4.2 shows the assembly context concept. To be specific, it shows on an instance level how the concept is depicted in each model. Note that the concept is not just the n-ary relation between a single element of each model. It rather contains multiple elements of each model.

The network of bidirectional transformations is used in different scenarios utilizing 39 fine-grained test cases that are part of the Vitruvius framework. These test cases were initially designed to ensure the functionality of the bidirectional transformation between PCM and UML. Each test case checks the consistency preservation of a pair of model elements or model concepts (see Table 4.1). They either create a simple UML model or a simple PCM model. As a preparation step, consistency is restored in the network, which results in the creation of the two missing models. Then the test cases make some changes to one of the models, which are the changes that are tested on. After that, consistency

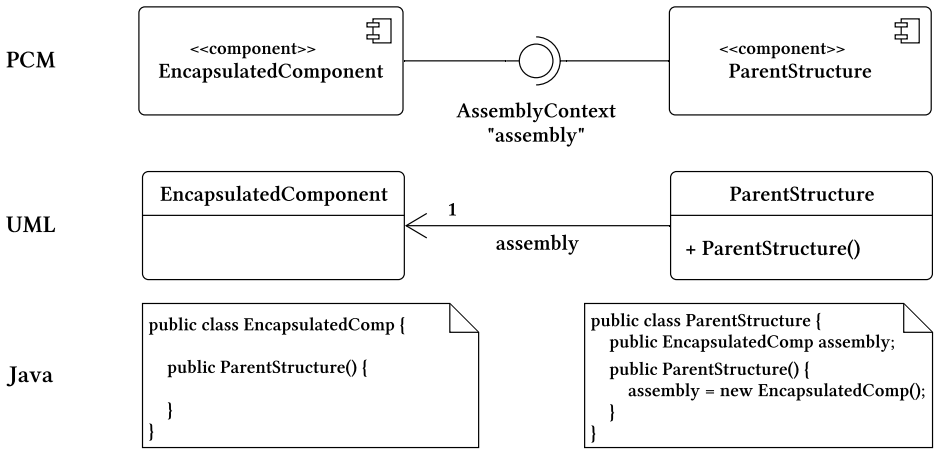

Figure 4.2.: Depiction of the assembly context concept on a model instance level. The assembly context between two components in the PCM model represents the same information as the UML classes and the relation between them and also as the Java classes and the field that references one class from another.

preservation is executed, which calls different transformations. Finally, the test cases verify the results on correctness. Additionally, they persist all models, with the Java model being persisted as regular Java code.

We extended the test cases to verify the Java model additionally. Since the Java and UML metamodels are very similar, this is done by comparing the UML elements to the Java elements. Assuming the PCM and UML models are consistent, it is sufficient only to make this comparison to guarantee consistency across all models. It is important to mention that these test cases, even with the Java extension, do not cover every possible scenario and also not every possible change to every possible model element. They only represent the common changes that one can reasonably expect to be made to those models.

4.3. Process

During the case study, the network of bidirectional transformations is constructed incrementally. We start with a single bidirectional transformation and extend it in four stages to a dense network with redundancy (see Figure 4.3). During each stage, we run the test cases and additionally inspect their persisted output manually. This ensures finding inconsistencies that are not being found by the test cases. Any failures are then analyzed to trace the fault that caused the failure. We then resolve the fault. That fault is responsible for causing any failures that now no longer occurs due to the fault being fixed. The time-consuming part of this process is tracing the underlying faults of the failures and then manually resolving that fault. We trace the faults by tracking how the initial changes made in the network of transformations are propagated through the network. We then back-track from the failure and analyze each transformation that had a part in the propagation chain until we understand what fault caused the failure. We resolve the fault by manually improving the affected transformations until the fault is fixed. We confirm

Test Case Group	Number of Test Cases
Repository Concept	4
Interface Concept	2
System Concept	2
Composite Data Type Concept	4
Repository Component Concept	2
Assembly Context Concept	2
Parameter Concept	6
Attribute Concept	6
Signature Concept	6
Required Role Concept	3
Provided Role	2

Table 4.1.: Overview of the test case categories and the number of test cases. We used a total of 39 test cases. The test case categories are divided into the core tests (upper part) and the additional tests (lower part).

that the fault is resolved by checking if the correlating failures still occur. Finally, we reconstruct the correlating mistake from the fault by analyzing what missing knowledge leads to such a fault. The stage-wise execution allows matching the failures to a particular state of the network. More specifically, it allows drawing conclusions on how the network topology correlates to the number of failures. As a foundation, we used the network that was used in a previous case study [46, 22]. It consists of the two metamodels PCM and UML, as well as two unidirectional transformations between them. In this network, there are no transitive cycles and no bottlenecks. As this network contains only a single bidirectional transformation, which already satisfies the test cases, no failures can be observed with this network.

The first stage adds the unidirectional transformation from the UML metamodel to the Java metamodel. In this stage, we observe the first failures. The network still has no transitive cycles, but the newly added transformation is a bottleneck. The second stage adds the unidirectional transformation from the Java metamodel to the UML metamodel. The resulting network has a linear topology. In the network, the UML metamodel is a bottleneck for the other models in both directions (Java to PCM and PCM to Java), but no transitive cycles exist. The third stage adds the unidirectional transformation from the PCM metamodel to the Java metamodel. The resulting network is not linear anymore, but not yet fully connected. It now contains the transitive cycle $PCM \rightarrow Java \rightarrow UML \rightarrow PCM$. The UML metamodel is now only a bottleneck for change propagation from Java to PCM. The fourth stage adds the unidirectional transformation from the Java metamodel to the PCM metamodel and therefore completes the network. The resulting network has a fully connected topology. There are multiple transitive cycles (for example $PCM \leftrightarrow UML \leftrightarrow Java \leftrightarrow PCM$), but no bottlenecks. This network has a high amount of redundancy, as every pair of incoming transformations for a metamodel have some degree of overlapping information.

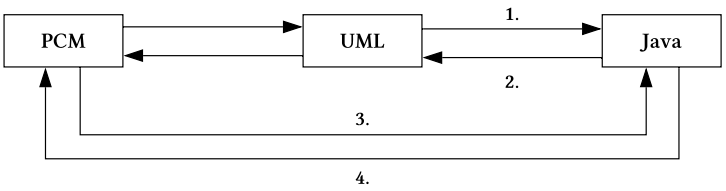

Figure 4.3.: Incremental construction of the network of bidirectional transformations, starting from the initial network that was already explored in previous case studies and adding unidirectional transformations step-by-step (1-4) until we reach the fully-connected network.

Because of the time constraints of this thesis, the last two stages were only conducted with a subset of the test cases. This subset of 16 out of 39 test cases covers the core concepts and model elements (see core tests in Table 4.1). Resolving the failures for all test cases would have taken more time than available, since the troubleshooting, analysis, and fault-removal is a very time-intensive task. This is the case because the high number of transformations in the network leads to even simple test cases producing a high number of change sequences that are propagated through the network. As an example, the creation of a single element results in the transformation of 25 to 50 change sequences in our specific network. Together these change sequences can include between 30 and 100 atomic changes. This makes tracing the correlating faults of the failures a task whose complexity does not scale well with the size and density of the network.

4.4. Encountered Mistakes, Faults, and Failures

In this section, we briefly discuss the mistakes, faults, and failures found during the case study and list them in causal chains to shows how mistakes lead to failures. These chains are created by matching the faults with the failures that no longer appear once the fault is fixed. The correlating mistake is reconstructed by analyzing the fault. A single chain can reveal itself through multiple failures, all caused by the same fault. We divide these problems into their respective stages of the case study and list them in the order of their occurrence. The number of failures is determined by temporarily removing each fix correlating to a fault one at a time after all faults of the stage were fixed and checking how many test cases fail. For the remainder of this thesis, we refer to mistakes, faults, and failures by number. The number references the correlating causal chain in Table 4.3, Table 4.4, Table 4.5, and Table 4.6

First Stage In the first stage, five faults were detected that lead to 11 test case failures (see Table 4.3). All except one fault in this stage regard the design of single transformations and not the inconsistencies between transformations. This means the faults can be avoided with the domain knowledge on the source and target models and careful transformation design. The last fault is a technical fault that could have been avoided by carefully implementing the transformation and proper testing.

Stage	Added Transformation	Faults	Failures
1	$T_{UML \rightarrow Java}$	5	11
2	$T_{Java \rightarrow UML}$	2	7
3	$T_{PCM \rightarrow Java}$	12	55
3	$T_{Java \rightarrow PCM}$	10	46

Table 4.2.: Distribution of the faults and failures over the four stages.

Second Stage In the second stage of the case study, only two faults were detected, which lead to seven test case failures (see Table 4.4). Both faults are technical faults located in the transformation $T_{Java \rightarrow UML}$, which we added to the network for this stage. As previously mentioned, technical faults can be avoided. The low number of faults can be explained by the fact that this stage makes no drastic changes to the network. While the first stage added a new model to the network through the addition of a transformation, this stage only completes the bidirectional transformation between two models that are already part of the network.

Third Stage In the third stage of the case study, 12 faults were detected, which lead to 55 test case failures (see Table 4.5). The high number of failures can be explained through the fact that the three *Faults 8, 10,* and *11* affect model elements that are required for even the most simple models and therefore are part of many test cases. The high number of faults can be explained by the fact that this is the first time that redundancy through redundant paths is introduced to the network of transformations. Both transformations $T_{PCM \rightarrow Java}$ and $T_{UML \rightarrow Java}$ are responsible for creating, changing and deleting Java elements. The transformation $T_{PCM \rightarrow Java}$ contains some redundancy with the chained transformations $T_{PCM \rightarrow UML}$ and $T_{UML \rightarrow Java}$. This is confirmed by the faults that occurred at this stage. The majority of them regard the transformation interaction and not the design of a single transformation or some technical issues. In this stage, we observe many failures of duplication. They are the manifestations of the mistake of not considering the external creation of an element by another transformation.

Fourth Stage In the fourth stage of the case study, ten faults were detected, which lead to 46 test case failures (see Table 4.6). Similar to the previous stage, the high number of failures can be explained through faults that affect model elements that are required for even the most simple models and therefore are part of many test cases. In total, 32 failures are caused by *Faults 20, 21,* and *25* alone. This stage is unique, as most faults are located in multiple transformations at the same time. This could be explained through the fact that the network is now fully connected, and every transformation in the network has two other transformations with whom it has some redundancy. The redundancy also explains, as in the previous stage, the high number of faults. Again, the majority of faults regard the transformation interaction, and we observe many failures of duplication.

When comparing the different stages, we observe an increased number of failures in the last two stages. Additionally, when tracing the faults that caused the failures, we observed an increased number of faults. The number of faults and failures per stage can be seen in Table 4.2. In total, 84.9% of the failures occurred in the last two stages. This makes sense, as these two stages introduce the redundant paths to the network, and therefore create redundancy-based cycles.

No.	Description	Location	Failures
1	**Mistake:** Incautious implementation **Fault:** Incorrect UML package deletion order (parent before child) **Failure:** Java child packages not deleted	$T_{PCM \to UML}$	1
2	**Mistake:** Overlooked intra-model dependency **Fault:** Java child packages not renamed with parent **Failure:** Invalid namespace of Java child packages	$T_{UML \to Java}$	2
3	**Mistake:** Overlooked relevant source model change **Fault:** Java parameters not created on UML parameter direction change **Failure:** Missing Java parameters (not created)	$T_{UML \to Java}$	4
4	**Mistake:** Overlooked relevant source model change **Fault:** Java types not updated when UML multiplicity changed **Failure:** Java parameter/return type is not a collection type	$T_{UML \to Java}$	2
5	**Mistake:** Overlooked intra-model dependency **Fault:** Java classes not updated when compilation unit inserted or moved **Failure:** UML classes located outside of their expected package	$T_{UML \to Java}$	2

Table 4.3.: Overview of the mistakes, faults, and failures encountered in the first stage of the case study. Contains a description of the mistake-fault-failure causal chain, the location which is the affected bidirectional transformations, and the number of failures in the 39 test cases.

No.	Description	Location	Failures
6	**Mistake:** Incautious implementation **Fault:** Correlating UML class is deleted when Java compilation unit is removed from its container **Failure:** Java classes are missing	$T_{Java \to UML}$	6
7	**Mistake:** Overlooked edge case (incautious implementation) **Fault:** Empty segment of package path not considered when renaming UML class for Java class **Failure:** Java package renamed to name of its child package	$T_{Java \to UML}$	1

Table 4.4.: Overview of the mistakes, faults, and failures encountered in the second stage of the case study. Contains a description of the mistake-fault-failure causal chain, the location which is the affected bidirectional transformations, and the number of failures in the 39 test cases.

No.	Description	Location	Failures
8	**Mistake:** Overlooked differences in naming schemes **Fault:** Java package name starting with lowercase not enforced **Failure:** Duplicate Java package with different spelling (co-existence)	$T_{PCM \to Java}$	13
9	**Mistake:** Overlooked relevant source model change **Fault:** Java packages mistakenly created for unnamed repositories **Failure:** Java packages misplaced (namespace of parent) **Note:** This fault alone technically causes no failures, since fixes for other faults cover this fault as well	$T_{PCM \to Java}$	0
10	**Mistake:** Inability to predict transitive consequences **Fault:** Difference in managing UML root models **Failure:** Duplicate creation of root models (co-existence)	$T_{PCM \to UML}$	14
11	**Mistake:** Not considering transitive scenarios at all **Fault:** Possibility of external creation by other transformation not considered **Failure:** Duplicate creation of UML packages (co-existence)	$T_{Java \to UML}$	12
12	**Mistake:** Not considering transitive scenarios at all **Fault:** Possibility of external creation by other transformation not considered **Failure:** Duplicate creation of UML interfaces (co-existence)	$T_{Java \to UML}$	2
13	**Mistake:** Not considering transitive scenarios at all **Fault:** Possibility of external creation by other transformation not considered **Failure:** Compilation unit not renamed due to duplicate creation (overwriting)	$T_{PCM \to Java}$	1
14	**Mistake:** Not considering transitive scenarios at all **Fault:** Possibility of external creation by other transformation not considered **Failure:** Java package not properly deleted due to duplicate creation (overwriting)	$T_{PCM \leftrightarrow UML}$ $T_{UML \leftrightarrow Java}$	1
15	**Mistake:** Incautious implementation **Fault:** Unescaped dots in regular expressions **Failure:** UML model element names are shortened or mutilated	$T_{PCM \to Java}$	1
16	**Mistake:** Not considering transitive scenarios at all **Fault:** Possibility of external creation by other transformation not considered **Failure:** Duplicate creation of UML and Java types (overwriting)	$T_{Java \leftrightarrow UML}$ $T_{PCM \to Java}$	6
17	**Mistake:** Not considering transitive scenarios at all **Fault:** Possibility of external creation by other transformation not considered **Failure:** Duplicate creation of Java package (co-existence)	$T_{PCM \to Java}$	1
18	**Mistake:** Overlooked differences in naming schemes **Fault:** Java package name starting with lowercase not enforced **Failure:** Duplicate creation of Java package (co-existence)	$T_{PCM \to Java}$	1
19	**Mistake:** Inability to predict transitive consequences **Fault:** Difference in managing UML root models **Failure:** Duplicate creation of UML root models (co-existence)	$T_{PCM \to Java}$	2

Table 4.5.: Overview of the mistakes, faults, and failures encountered in the third stage of the case study. Contains a description of the mistake-fault-failure causal chain, the location which is the affected bidirectional transformations, and the number of failures in the 16 test cases.

No.	Description	Location	Failures
20	**Mistake:** Not considering transitive scenarios at all **Fault:** Possibility of external creation by other transformation not considered **Failure:** Duplicate creation of PCM repositories (co-existence)	$T_{PCM \leftrightarrow UML}$ $T_{UML \leftrightarrow Java}$	14
21	**Mistake:** Inability to predict transitive consequences **Fault:** PCM repository naming schemes not consistent **Failure:** Duplicate PCM repositories with different spelling (co-existence)	$T_{UML \rightarrow PCM}$ $T_{Java \rightarrow PCM}$	7
22	**Mistake:** Incautious implementation **Fault:** Null as possible Java package name not considered **Failure:** Crashes due to null as Java package name	$T_{Java \rightarrow PCM}$	4
23	**Mistake:** Not considering transitive scenarios at all **Fault:** Possibility of external creation by other transformation not considered **Failure:** Duplicate creation of PCM operation interface (co-existence)	$T_{UML \rightarrow PCM}$ $T_{Java \rightarrow PCM}$	1
24	**Mistake:** Not considering transitive scenarios at all **Fault:** Possibility of external creation by other transformation not considered **Failure:** Duplicate creation of PCM systems (co-existence)	$T_{PCM \leftrightarrow UML}$ $T_{UML \leftrightarrow Java}$	1
25	**Mistake:** Not considering transitive scenarios at all **Fault:** Not limiting renaming to specific pairs of model elements **Failure:** Incorrect and invalid renaming, endlessly looping name change propagation	$T_{Java \rightarrow PCM}$	11
26	**Mistake:** Not considering transitive scenarios at all **Fault:** Possibility of external creation by other transformation not considered **Failure:** Duplicate creation of PCM components (co-existence)	$T_{UML \rightarrow PCM}$ $T_{Java \leftrightarrow PCM}$	4
27	**Mistake:** Overlooked differences in naming schemes **Fault:** Java and UML package and constructor names starting with lowercase not enforced **Failure:** Duplicate creation of PCM components with different spelling (co-existence)	$T_{PCM \rightarrow UML}$ $T_{PCM \rightarrow Java}$	1
28	**Mistake:** Not considering transitive scenarios at all **Fault:** Possibility of external creation by other transformation not considered **Failure:** Duplicate creation of PCM systems (co-existence)	$T_{PCM \leftrightarrow Java}$	1
29	**Mistake:** Not considering transitive scenarios at all **Fault:** Possibility of external creation by other transformation not considered **Failure:** Duplicate creation of PCM assembly contexts and correlating elements (co-existence and overwriting)	$T_{PCM \leftrightarrow UML}$ $T_{UML \leftrightarrow Java}$ $T_{PCM \leftrightarrow Java}$	2

Table 4.6.: Overview of the mistakes, faults, and failures encountered in the fourth stage of the case study. Contains a description of the mistake-fault-failure causal chain, the location which is the affected bidirectional transformations, and the number of failures in the 16 test cases.

5. Result Classification and Analysis

This chapter analyzes the encountered mistakes, faults, and failures listed in the previous chapter. We use them to derive the classification regarding avoidability and sort the mistakes, faults, and failures into their respective classes. We also reason about the connections of mistakes, faults, and failures. Finally, we compare the classification and the analysis with previous work. For a detailed list of all mistakes, faults, and failures we refer to Table 4.3, Table 4.4, Table 4.5, and Table 4.6. First, section 5.1 discusses the different classifications for the encountered mistakes, faults, and failures and, therefore, directly addresses *Research Question 2*. We illustrate how these classifications were constructed, and which conclusions can be drawn from the distribution of mistakes, faults, and failures over the classes. Second, section 5.2 analyzes the causal chains of mistakes, faults, and failures. It is meant to find the connections between the different mistakes, faults, and failures. Next, section 5.3 compares the results and the classifications of this case study with previous work. Additionally, it classifies the mistakes, faults, and failures according to the classifications proposed by previous work. This answers *Research Question 6*, which asks about the similarities and differences to previous work. Last, section 5.4 explains which measurements were taken to ensure the correctness and completeness of our results.

5.1. Classification

This section discusses several classifications for the mistakes, faults, and failures observed in this case study to answers *Research Question 2*. There are several reasons to develop such a classification. First, we are trying to understand what kind of mistakes, faults, and failures arise during multi-model consistency preservation through networks bidirectional transformations. Second, we want to understand how these different types of mistakes, faults, and failures are connected. Third, we need this classification to answer *Research Question 3* regarding the prevention by constructions, *Research Question 4* regarding the unpreventable mistakes, and *Research Question 5* regarding the effects of redundancy in networks of bidirectional transformations. We propose a high-level categorization and detailed classes for mistakes, faults, and failures separately. For all classifications, we discuss how they were constructed, and which conclusions can be drawn from them. We also list the number of failures for each category and class. In this section, we first classify the failures regarding the model state and analyze their distribution over the three models. Second, we classify the faults regarding their scope and inspect in which transformations they are commonly located. Third, we classify the mistakes by the knowledge required to prevent them and analyze which mistakes can be prevented in the first place.

Model	Failures	Failures by Fault
PCM	42	9
UML	59	11
Java	50	14
All	119	29

Table 5.1.: Distribution of the failures over the three models. Note that a single failure can occur in multiple models at once due to the change propagation. Failures by fault counts all failures caused by a fault as a single occurrence.

5.1.1. Classification of Failures

During this case study, a total of 119 failures were detected. While most of these failures only appear in one model, a few appeared in multiple models during a single test execution. As seen in Table 5.1, the most failures appeared in the UML model, while the least failures appeared in the PCM model. When counting all failures caused by the same fault as a single occurrence, the Java model shows the most occurrences.

To construct a classification for the failures, we look at the model state after a failure occurs, which describes how the failure affected the models in the network. Naturally, we can derive three categories, which are listed in Table 5.2. Either there were too many elements, too few elements, or the elements have incorrect properties. These three categories can be divided further into classes by looking at how the model state is reached. The category of too many elements contains the two classes: Duplicate element creation and missing element deletion. Duplicate element creation means the element is created twice by different transformations. This includes in this case study, as mentioned as in section 4.4, primarily duplicate creation with co-existence, and duplicate creation with overwriting. The latter mainly shows itself through other failures. The class of missing deletions summarizes all failures where the element was either not deleted at all or just partly deleted. The model state category of incorrect elements contains the encountered classes of incorrectly named elements, misplaced elements, and elements with the wrong type. The class of incorrect names summarizes all name-related problems, such as renaming the wrong elements, enforcing an incorrect naming scheme, or empty names. The class of misplaced elements contains all failures where either a model element was misplaced in the model structure or a root element was persisted in the wrong location. As a third option, both could be the case at the same time. The class of wrong element type groups failures where an element created by a transformation was not of the expected element type. As an example, it could be an instance of the wrong subclass, like an interface method instead of a class method. The model state category of too few elements contains the classes of unwanted deletions of elements and missing creation of elements. Unwanted deletion means an element was deleted that was not supposed to be deleted. Missing creation is the opposite of that, as it means the element was supposed to be created but was not created.

Table 5.2 shows that most of the failures in this case study are duplicate element creations. In total, 68.9% of all failures are duplicate element creations. Even when counting all failures of a single fault as one occurrence, duplicate creation accounts for half of the

Model State	Failures	Failure Class	Failures	Failures by Fault
Too Many Elements	84	Duplicate Creation	82	16
		Missing Deletion	2	2
Incorrect Elements	25	Incorrect Name	20	6
		Misplaced Elements	3	2
		Wrong Element Type	2	1
Too Few Elements	10	Unwanted Deletion	6	1
		Missing Creation	4	1

Table 5.2.: Distribution of the failures over the failure classes. Failures by fault counts all failures caused by a fault as a single occurrence.

occurrences. The second most occurring class is incorrect naming, which accounts for 16.8% of all failures and 20.7% when counting failures of a fault as one occurrence. This could be explained by the importance of names in these models. As not all models use unique identification numbers, elements are often located by their name and containment. For example, because Java models contain multiple root elements, compilation units are identified like this. The remaining classes only account for a small percentage of the failures.

The previously mentioned failure classes are only those we encountered. In this case study, there are potentially many more. We identified three additional classes by adding opposites and similar classes of the existing classes. Table 5.3 shows the completed classification. The class of unwanted creation is the opposite of missing creation. It describes the case where an element is created and inserted into one of the models but should not exist at all. This could potentially appear when the matching by a transformation is accidentally used by the correlating backward transformation to create an element that should not exist. The class of duplicate deletion is the opposite of duplicate creation. It describes the case where a transformation is trying to delete an element that does not exist anymore. The class of incorrect (non-name) properties completes the class of incorrect names. It describes the case where a property of a model element is incorrect. This case excludes names. We argue that the separation of incorrect name properties and incorrect non-name properties makes sense because names are frequently used for element identification and therefore play a particular role during the consistency preservation. Identifying naming-related failures is essential because identification-related properties seem to be often caused by different faults, which are also manifestations of different mistakes. Still, both classes could be merged into the sole class of incorrect properties.

We argue that these classes are complete for metamodels as those used in our case study. This means, more specifically, metamodels according to the essential meta object facility (EMOF) [15]. This is based on the following assumptions: When considering how a model can be changed, we can argue changes are either element additions, element deletions, or element adaptions, meaning a property or relations changes. According to EMOF, there

Model State	Failure Class
Too Many Elements	Duplicate Creation Missing Deletion Unwanted Creation
Incorrect Elements	Incorrect Name Incorrect (Non-Name) Property Misplaced Elements Wrong Element Type
Too Few Elements	Duplicate Deletion Unwanted Deletion Missing Creation

Table 5.3.: Extended classification of the failures. The failure classes can be grouped according to the model state. The added classes are unwanted creation, incorrect (non-name) property, and duplicate deletion.

are packages, classes, and data types. These are the model elements we are categorizing according to the model state. Too many model elements can only occur for three reasons: A missing deletion, an unwanted creation, or a duplicate creation. Technically, duplicate creation is a special case of unwanted creation. Too few model elements, again, can only occur for three reasons: A missing creation, an unwanted deletion, or a duplicate deletion. As before, a duplicate deletion is technically a special case of unwanted deletion. If there are incorrect model elements, there is only a limited number of aspects that can be incorrect, namely those defined by EMOF. First, properties can be incorrect, which we covered with two classes, one for names and one for other properties. The latter also covers non-containment relations, which can be seen as properties. Misplaced elements cover incorrect containment relations. Second, the class of wrong element types, which is the case when a model element is contained where it would be expected, but it technically has the wrong type. To summarize this argument, we covered combinations of all types of changes and all aspects defined by EMOF.

5.1.2. Classification of Faults

During this case study, 29 faults were discovered by analyzing the underlying cause of the mistakes. As previously mentioned, multiple failures can be caused by one fault. While we utilized these faults in Table 5.1 and Table 5.2 to count failures as singular occurrences, namely failures by fault, we did not look at the faults themselves. Some faults, for example *Faults 8, 9,* and *10*, are only located in a single transformation. Others, such as *Fault 20* and *21*, are located in multiple transformations at once. An extreme case of multiple locations is *Fault 29*, which appears in every single unidirectional transformation. It is the missing existence check for assembly context elements and correlating elements. To be specific, these elements may only be created if no other transformation created a semantically identical element yet in the same model. In this case study, only the faults

Bidir. Transformation	Faults
$T_{PCM \leftrightarrow UML}$	14
$T_{UML \leftrightarrow Java}$	15
$T_{PCM \leftrightarrow Java}$	22

Table 5.4.: Distribution of the 29 faults over the model transformations. Note that a fault can be located in multiple transformations at once.

Fault Scope	Faults	Fault Class	Faults
Technical	5	Technical Fault	5
Transformation-Internal	6	Missing Change Propagation	4
		Unwanted Change Propagation	2
Transformation Interaction	18	Creation Conflict	12
		Naming Conflict	4
		Root Element Management	2

Table 5.5.: Distribution of the faults over the fault classes.

regarding transformation interaction are located in multiple transformations at once. Transformation interaction means the fault is based on how chained transformations propagate changes through networks of bidirectional transformations. The distribution of the faults over the different transformations is listed in Table 5.4. It shows that the bidirectional transformation between PCM and Java contains most of the faults. 22 of the 29 faults are located in this bidirectional transformation. For this transformation, both unidirectional transformations that form the backward and forward transformations were written by different people without each other in mind. Thus, these unidirectional transformations are not entirely compatible. Over the bidirectional transformations the faults appear to be evenly distributed in both unidirectional transformations, therefore we limit Table 5.4 to the bidirectional transformations.

To classify the faults, we first separate the technical faults from the non-technical faults. Technical faults are faults that are introduced during the implementation of transformations. These faults are not connected to misunderstanding how elements should be transformed from the source to the target model. Instead, they stem from the incorrect usage of the transformation definition language, which is the Reactions language in this case study. These faults are the counterpart of programming errors in classical software engineering. While technical faults are less interesting in regards to issues in networks of bidirectional transformations, they are still important to acknowledge as they show the importance of careful implementation, especially in regard to unexpected edge cases. In this case study, 5 out of 29 faults are technical faults.

The remaining 24 faults can be divided further based on their scope into transformation-internal faults and faults regarding transformation interaction. Transformation-internal

faults can be explained in the scope of a transformation, which means the fault is independent of other transformations and can, therefore, be fixed without touching any other transformation. In this case study, we found transformation-internal faults where a change in the source model was not propagated to the target model and transformation-internal where a change in the source model was erroneously propagated to the target model. Transformation interaction faults can only be explained at the network level because multiple transformations are conflicting. It is essential to mention that this includes interactions of the two unidirectional transformations that form a bidirectional transformation. In this case study, this includes creation conflicts between transformations, naming conflicts between transformations, and mismatching root element management between transformations. As an example, a creation conflict fault is not checking for external creation by another transformation during the element creation. An example of a mismatching root element fault is when one transformation assumes the root element has a fixed location, while another transformation asks the user for the location and then tracks it by matching it with a correlating source element. These differences in the root element management lead to failures. An example of a naming conflict fault is not enforcing the correct naming scheme when transforming from a source element to a target element. All three examples can cause failures of duplicate model element creation. This separation between transformation-internal and interaction-related faults is confirmed by the fact that the faults classified as faults regarding transformation interaction are the only ones that are located in multiple transformations at once. In contrast, all faults classified as transformation-internal in this case study, are only located in a single transformation. Out of the 24 non-technical faults, only six are transformation-internal faults, and the other 18 are faults regarding transformation interaction. This classification shows that the majority of faults in this case study regard the interaction of transformations. It also shows that creation conflicts between transformations, namely duplicate element creation, are the most common faults that appeared. To be more specific, the fault of duplicate element creation is the manifestation of not considering the possibility of the external creation of a model element by another transformation. These creation conflicts account for 41,3% of all faults. This illustrates the importance of avoiding these creation conflicts by design.

5.1.3. Classification of Mistakes

Since we are interested in the prevention of mistakes during the construction of the transformations, we classify the mistakes according to the knowledge scope that is required to prevent them or to fix the correlating faults in the transformations. This classification is depicted in Table 5.6. Similarly to the classification of faults (see Table 5.5), the mistakes can be divided into technical and non-technical mistakes. Technical mistakes require no knowledge about the models or the transformations to prevent them. They might require some knowledge of the transformation specification language, but besides that, any person with some programming experience could be able to spot and fix the correlating faults. Five out of the 29 mistakes are technical mistakes, which are classified as incautious implementation. They directly correlate to the five technical faults. The remaining 24 mistakes can be further classified: Mistakes that require knowledge of a single bidirectional transformation and mistakes that require knowledge of the network of bidirectional

Mistake Knowledge	Mistakes	Mistake Class	Mistakes
Technical	5	Incautious Implementation	5
Transformation	8	Unconsidered Source Model Change	3
		Unconsidered Naming Scheme	3
		Unconsidered Intra-Model Dependency	2
Network	16	Not Considering Transitive Consequences	13
		Inability to Predict Transitive Consequences	3

Table 5.6.: Distribution of the mistakes over the mistake classes.

transformations. Knowledge of the bidirectional transformation means knowing the structure and the use of the models and knowing how to transform between the models. Knowledge of the network of bidirectional transformations means at least knowing about the possibility of a network of bidirectional transformations but might also contain detailed knowledge about how the network is constructed and how changes are propagated in the network. This case study encountered eight mistakes that only require transformation knowledge and 16 mistakes that require network knowledge.

The eight mistakes that require transformation knowledge can be classified into the following three classes. First, unconsidered source model changes, which means a change to the source model was not transformed into a change to the target model. Second, unconsidered naming schemes, which are mistakes where the naming schemes of the source and target elements are not considered when transforming names between two mapped elements. Last, unconsidered intra-model dependency, which means a change to one element should be accompanied by a change to another element, as the second element depends in some way on the first. For the class of intra-model dependencies, however, it is essential to mention that this should not be the responsibility of the model transformation, but instead of the model itself. Nevertheless, we count it here as a class, since in our case study, the transformation failed to achieve what was expected of it.

The 16 mistakes that require some sort of network knowledge can be classified into the following two mistake classes. The first one is the class of not considering transitive consequences, which contains 13 of the 16 mistakes. The second one, containing the other four mistakes, is the class of the inability to predict the transitive consequences. This categorization shows that 55.2% of all mistakes and 66.7% of all non-technical mistakes require some sort of network knowledge. This highlights the importance of considering networks of transformations and transitive change propagation during the construction of any model transformation.

While every mistake encountered in this case study can be fixed later on, not every mistake can be prevented during the construction of the transformations. Using the categorization regarding the knowledge, we can derive information about if and how the mistakes can be prevented. All technical mistakes can be prevented by definition. The only requirement for that is technical knowledge, such as being proficient in the language used to design the transformations. Moreover, all encountered mistakes regarding transforma-

tion knowledge can also be prevented, assuming the availability of expertise regarding the source and target domain. Out of the 16 encountered mistakes regarding network knowledge, only three cannot be prevented during the construction of the transformations. The other mistakes can be prevented, as they only require the knowledge that the transformation is going to be used in a network of transformations. With this knowledge, precautions can be made that avoid these mistakes. The three unpreventable mistakes are *Mistake 10* and *18* as well as *Mistake 21*. Even when knowing that the transformations are supposed to be used in a network of transformations, preventing would both require detailed knowledge about the network structure as well as insight in the interaction of the transformations and the change propagation. These issues are discussed in detail in section 6.2. To summarize, this means 18.8% of the mistakes of this case study that require network knowledge are not preventable, but only 10.3% of all encountered mistakes in this case study are not preventable.

5.2. Analysing the Causal Chain

The previous section proposed classifications for mistakes, faults, and failures. This section analyzes their correlation and the correlation of their high-level categories, namely the mistake knowledge scope, the fault scope, and the failure model state. During the case study, we encountered failures caused by faults that are manifestations of mistakes. These causal chains offer insight into the connections between mistakes, faults, and failures. We extracted the connections from the encountered problems (see section 4.4) for the categories introduced in the previous sections. These connections are listed in Table 5.7. Each column, therefore, matches one of the classification tables. In the following, we summarize the different categories of each column. The mistake column, which is taken from Table 5.6, represents the knowledge that led to the mistake. It also is the knowledge required to fix and, if possible, prevent the mistake and the fault that caused it. "Technical knowledge" means the required knowledge regards the tools and the framework, such as the transformation language. "Transformation knowledge" means only knowledge about the source and target model and how to transform between them is required. "Network knowledge" means at least the knowledge of the possibility of a network of transformation with transitive change propagation is required. However, this can also include detailed knowledge about the network topology and how changes are propagated in the network.

The fault column, which is taken from Table 5.5, contains the scope in which the fault manifests itself. "Technical" means the fault would have been avoided if the same functionality of the transformations would have been implemented correctly. "Transformation-internal" means the fault affects only the internal behavior of a single transformation. In contrast, "Transformation interaction" means the fault affects how different transformations interact in the network. The failure column, which is taken from Table 5.2, shows what in what model states the failure results in. The possible states are too many elements in the models, too few elements in the models, and elements with incorrect properties.

As expected, mistakes based on the lack of technical knowledge only manifest themselves in technical faults. These faults, however, can cause any failure. This correlation is not as interesting for networks of bidirectional transformations or multi-model consistency

Mistake (Knowledge)	Fault (Scope)	Failure (Model Elements)
Technical	Technical	Too Few
Transformation	Transformation-Internal	Incorrect
Network	Transformation Interaction	Too Many

Table 5.7.: Depiction of the connections between the mistake, fault, and failure categories abstracted from the failure chains listed in section 4.4. The failure chains are extracted from the case study by finding the causing faults of failures and then inferring the mistake that manifest in the fault.

preservation in general. However, it is noteworthy that despite the reduced complexity of the domain-specific language used for the transformation definition, technical issues still arise frequently. When analyzing the technical mistakes and faults in detail, it is apparent that about half of them regard edge cases, such as edge case values for attributes. This is similar to the transformation-internal faults and can be explained with the fact that the transitive scenario produces model states that are not expected or not intended. An example of this are elements that were not intended or expected to exist without a name being set. This is the case for technical faults *Fault 7* and *Fault 22* a well as the transformation-internal *Fault 9*.

Interestingly enough, the connection for the other two categories of mistakes and faults is not as linear (see bottom left of Table 5.7). This means that most mistakes based on transformation knowledge manifest themselves in transformation-internal faults and most mistakes based on network knowledge themselves in faults regarding the transformation interaction. However, this is not always the case. For one, mistakes based on transformation knowledge can lead to faults regarding the transformation interaction. This occurred in our case study in three causal chains: *Causal Chain 8*, *Causal Chain 18*, and *Causal Chain 27*. Semantically, such a chain means that while preventing the mistake by construction only requires knowledge about the bidirectional transformation and consequentially its source and target domain, the scope of the fault is the transformation interaction. This can be described as amplifying the mistake through the complexity of the transitive scenario. For another, mistakes based on network knowledge can manifest themselves in transformation-internal faults. This occurred in *Causal Chain 25*. Semantically, it means that while the fault is limited to the internals of a single transformation, preventing the mistake by construction still requires some knowledge of the network of transformations and transitive change propagation. In the case of *Causal Chain 25*, it is sufficient to know that the transformation is used in a network of bidirectional transformations.

The correlation between the fault scope and the failure model state is less informative. All fault categories can potentially lead to any failure. Generally, there is no pattern to be found for the correlation of the fault categories and the failure categories. Similar to the correlation of the categories in Table 5.7, we also analyzed the correlation of the classes introduced in the previous sections. This offers a less abstract view on the causal chains. These fine-grained class correlations are listed in Table 5.8. They lead to the same conclusions and, therefore, confirm the previous results. An additional connection that can be seen in Table 5.7 is the tendency of mistakes relating to network knowledge

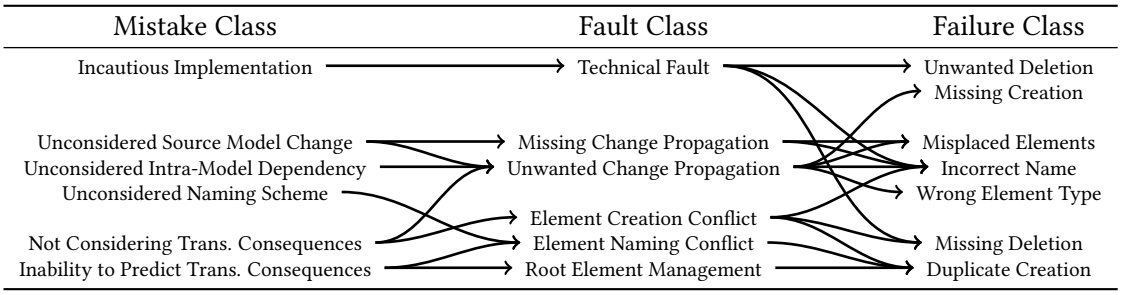

Table 5.8.: Connections between the mistake, fault, and failure classes abstracted from the failure chains listed in section 4.4. The classes are grouped by their respective categories (see Table 5.7).

causing failures that create too many elements rather than too few elements and the tendency of mistakes relating to transformation knowledge to cause failures that result in incorrect model elements. However, these two tendencies might also just be the result of over-interpretation.

5.3. Comparison with Previous Work

This thesis builds on a previous case study, whose results are published by Syma [46] and Klare et al. [22]. They conduct a similar case study regarding networks of bidirectional transformations. The previous case study uses the same framework for consistency preservation, and their network uses the same models. However, they only build a network with linear topology using the transformations $T_{PCM \leftrightarrow UML}$ and $T_{UML \leftrightarrow Java}$. This means their network only contains transformation-internal cycles and no redundancy-based cycles. It also does not contain any redundancy. In the linear network, the UML model is a bottleneck and therefore defines what changes can be propagated between PCM and Java. As an example, implementation details of methods cannot be shared between PCM and Java because UML only models the signatures of operations. The linear network is identical to the network of this case study at the second stage (see section 4.3). For a more detailed description of all contributions made by these publications, see chapter 8.

This section compares the results and classifications of our case study with the previous case study. Additionally, we classify the observed mistakes, faults, and failures according to the classifications proposed by Klare et al. Therefore, this section directly addresses *Research Question 6*, which asks about the differences compared to previous work. We compare our case studies to validate both their results and our results. We mainly reference Klare et al. [22], as this work builds on Syma [46] and offers a more comprehensive classification.

Klare et al. propose the concept of three consistency specification levels at which a consistency preservation mechanism can be conceptually defined. These three levels are:

1. Global: Knowledge about the n-ary relations between all models in a network.

2. Modularization: Separation into binary consistency relations between model pairs.

Failure Type by Klare et al.	Failures	Failures by Fault
Duplication	82	16
Inconsistent Termination	26	12
Non-Termination	11	1

Table 5.9.: Distribution of the failures observed in this case study over the failure types identified by Klare et al. [22]

3. Operationalization: Transitive change propagation or confluence of information in the network.

Our case study proposes categories for the knowledge scope of mistakes (see Table 5.6), which differ from the consistency specification levels as the categories are only meant for classifying mistakes and faults. The consistency specification levels by Klare et al. are meant as conceptual levels for the entire process of specifying consistency between more than two models.

Klare et al. identify different failure types that classify failures depending on how the failure leads to the termination of consistency preservation. These failure types are resulting in a *inconsistent termination (deterministic or non-deterministic)*, *non-termination (diverging or alternating loops)*, and *duplication (instantiation or insertion)*. In this thesis, we categorized the failures according to the model state after the termination of the consistency preservation. We identified the categories too many elements, too few elements, and incorrect elements (see Table 5.3). We listed the failures observed in this case study grouped by type according to the failure types by Klare et al. in Table 5.9. Most of the failures fall into the failure type duplication, as this is basically the same class as duplicate creation in our failure classification (see Table 5.2). All of them are multiple instantiations. We encounter non-termination only once with *Failure 25*, which results in a diverging loop. However, during the development of some fixes for the encountered faults, we produced non-termination a few times ourselves. The low number of non-terminations can be explained by the fact that networks of bidirectional transformations are susceptible to small deviations, and therefore, most loops lead to the change propagation crashing after a few iterations. All other failures encountered in this case study fall into the class of inconsistent termination, with some being deterministic and some being indeterministic.

Klare et al. identify four different fault types based on the state of the consistency specification. They identified the types *missing consistency constraint*, *additional consistency constraint*, *contradicting consistency constraint*, and *missing element matching*. This thesis identified six fault classes in Table 5.5. Comparing them to the fault types by Klare et al. is not entirely possible, as they classify faults on different levels. Our classification for faults is on a conceptual level closer to the failures, while their classification is closer to the mistakes. However, with this in mind, we can still describe some connections between the two classifications. In our classification, we explicitly listed technical faults. In the case study of Klare et al., technical faults are not listed as they operate under the assumption that each transformation is on its own correctly implemented. Our category of transformation-internal faults contains two classes: Missing change propagation and unwanted change

Fault Type by Klare et al.	Faults	Fault Class
Missing Element Matching	14	Creation Conflict, Root Management
Contradicting Consistency Constraint	4	Naming Conflict
Missing Consistency Constraint	4	Missing Change Propagation
Additional Consistency Constraint	2	Unwanted Change Propagation

Table 5.10.: Distribution of the faults encountered in this case study over the fault types identified by Klare et al. [22]. The corresponding fault classes from Table 5.5 are listed as well.

propagation. Missing change propagation can be connected to the fault type of missing constraints. Klare et al. Similarly, unwanted change propagation can be connected to the fault type of additional constraints. This is the case because the consistency constraints according to the definition of Klare et al. translate to the requirements that are fulfilled by the change propagation of a transformation. Our category of transformation-interaction related faults contains three classes: Naming conflicts, creation conflicts, and root element management. Naming conflicts are contradicting constraints, as for a naming conflict fault, the constraints regarding name transformations of two transformations are contradicting. Both creation conflict faults and root element management faults can be seen as faults of the fault type missing matching by Klare et al. In both cases, an additional matching would fix the fault. For the creation conflict fault, the matching is missing between the source element for the element to be created and the conflicting element of the element to be created. For the root management fault, the matching is missing for the different root elements. We listed faults according to the fault types by Klare et al. in Table 5.10. By far, the most failures fall into the fault type missing element matching, as this fault type is the one that contains the faults that did not check on the existence of elements that could have been created externally. These faults were the most common ones in this case study. Note that the five technical faults are not listed.

As already mentioned, Klare et al. identify different mistake types based on the three consistency specification levels. On the global level, they differentiate between incomplete and incorrect system knowledge. For the modularization level, they differentiate between incomplete and contradicting modular knowledge. For the operationalization knowledge, they only list unknown connection of modular specifications. Comparing our fault classes to the fault types by Klare et al. is even more difficult than for the faults. While both our and their mistake classification are based on the knowledge regarding the mistake, the classification by Klare et al. is, again, based on more assumptions. Our classification is based on bidirectional transformations and, therefore, not meant for n-ary relations, while the global level defined by Klare et al. explicitly captures these n-ary consistency relations. Furthermore, the modularization level depends on the global level and therefore represents the same relations, but this time decomposed as binary relations. With these differences in mind, we can make some connections between the two classifications. In our classification, we explicitly listed technical mistakes. In the case study of Klare et al., technical mistakes are similar to technical faults, not listed as they operate under the assumption that each transformation is on its own correctly implemented. However, they

Level	Mistake Type by Klare et al.	Mistakes
1	Incomplete System Knowledge	0
	Incorrect System Knowledge	0
2	Incomplete Modular Knowledge	0
	Contradicting Modular Knowledge	3
3	Unknown Connection of Modular Specifications	13

Table 5.11.: Distribution of the mistakes encountered in this case study over the mistake types identified by Klare et al. [22].

even base their classification on the assumption that a bidirectional transformations is non-conflicting, which means the transformation rules are correctly designed. We did not make these assumptions, and therefore we included the transformation knowledge mistakes. Consequentially, these mistakes cannot be classified according to Klare et al. The remaining mistake category is the category of network knowledge mistakes. It contains two mistake classes: Not being able to predict transitive consequences and not considering transitive consequences at all. The mistakes of not considering transitive consequences at all regards the operationalization level and, thus, can be connected to the mistake type of unknown connection of modular knowledge. The mistakes of not being able to predict transitive consequences mean there must be some problem on the modularization level and, therefore, can be connected to the mistake type of contradicting modular knowledge. This distribution of the mistakes encountered in our case study according to the classification by Klare et al. is depicted in Table 5.11. It shows that most of the mistakes that were not excluded by definition fall into the third level, which is the operationalization level. They are classified as mistakes due to the unknown connection of modular Specifications. Three mistakes, however, fall into the modularization level. These are the three unpreventable mistakes. Consequentially, these results suggest that we can expect the majority of mistakes to be made on the operationalization level.

To summarize, we now discuss the main differences between the results of this thesis and the case study results of Klare et al. [22]. While Klare et al. take a broader look at issues in networks of bidirectional transformations, this thesis mainly looks at issues at the scope of the operationalization level. Both classifications share similarities, especially for the fault classifications. A big difference is that Klare et al. exclude technical mistakes and faults by definition. However, there is not much insight gained from these technical-implementational mistakes and faults. Additionally, Klare et al. also exclude transformation-internal faults by definition, as they base their work on the assumption that each bidirectional transformation is itself correctly designed. We were able to classify all mistakes, faults, and failures that were not excluded by definition into the classification by Klare et al. This means we are not able to find mistakes, faults, or failures that we could not classify according to the classifications by Klare et al. Consequentially, we can confirm that their case study proposes a complete and correct classification which fulfills its purpose. Similarly, their results also confirm our classification due to the similarity of both classifications. The most significant difference lies in the classification of failures, as

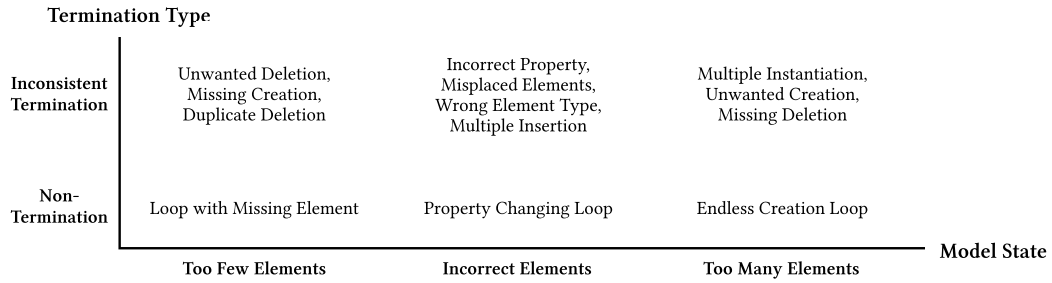

Figure 5.1.: Hybrid failure classification based on both the termination type and the model state. It combines our classification in Table 5.3 with the classification proposed by Klare et al.

diverging properties are used for the classification. Klare et al. classify failures according to how the change propagation terminates, meaning which type of termination, while this thesis classifies them according to the resulting models and the differences in their elements after the termination. We argue that either one of the dimensions alone is not enough. Klare et al. distinguish in the classification of failures between termination that results in an inconsistent state and termination that results in duplications. However, we argue that duplications in models a subset of the class inconsistent state, as there are either multiple elements that should represent the same element or multiple references the represent the same relation. Since termination with duplications is part of the category of termination that results in an inconsistent state, these categories can be combined, leaving only the two categories of non-termination and termination with an inconsistent state. This reduced classification is not very meaningful, as the results of our case study suggest that the majority of failures would fall into the category of inconsistent termination. In our case study, 108 out of 119 failures, which is 90.7%, are failures of inconsistent termination, while only 11 out of 119 failures are failures of non-termination. Consequentially, we argue that the termination type alone is not enough to classify failures. This thesis, on the other hand, only classifies failures according to the model state, which is also not enough on its own. As an example, Table 5.8 shows little information about which types of faults lead to which types of failures. Consequentially, this motivates the need to combine both the termination type and the model state in one classification.

Therefore, we propose a two-dimensional classification that combines the aspects of both. This classification is depicted in Figure 5.1. It contains two dimensions, namely the termination type and the model state. The model state, on the one hand, differentiates between the categories too many elements, too few elements, and incorrect elements. The termination type, on the other hand, differentiates between the two categories of inconsistent termination and non-termination. All classes can be categorized according to these two dimensions. Note that the categories called inconsistent-termination and duplication are now one category. Similarly, the classes of incorrect naming and incorrect non-name properties are now merged into a single class called "incorrect property". Additionally, we identified endless creation as an additional class for non-termination with too many

	Too Few Elements	Incorrect Elements	Too Many Elements
Inconsistent Termination	4	14	84
Non-Termination	0	11	0

Table 5.12.: Distribution of the failures over the hybrid classification that combines the model state (columns) and the termination type (rows).

elements, even though this case did not occur during the case study. These additional classes may need to be confirmed with an additional case study.

We classified all failures encountered in this case study according to this new classification. The distribution is depicted in Table 5.12. It shows that the majority of failures are terminating with inconsistencies and also cause a model state with too many elements. Consequentially, it is essential to provide prevention strategies for failures of the class that combines both of these characteristics.

5.4. Measurements

We use the same metrics as Klare et al. [22] to reason about the validity of our results and our analysis. The first measures how many failures of the encountered failures were successfully categorized:

$$Identified\ Failure\ Ratio = \frac{categorized\ failures}{total\ failures}$$

The second measures how many failures could be resolved through fixing the correlating faults:

$$Resolved\ Failure\ Ratio = \frac{resolved\ failures}{total\ failures}$$

In an ideal case both measurements are $IdentifiedFailureRatio = ResolvedFailureRatio = 1$, as this means all identified failure are resolved. Because of the time constraints of this thesis, we only used the 16 core tests instead of all 39 tests for the last two stages (see Table 4.1). It is essential to mention that these unused test cases produce additional failures that we, therefore, did not encounter in our case study. In total, we encountered 119 failures, 29 faults, and 29 mistakes. For all four stages, we measured $IdentifiedFailureRatio = ResolvedFailureRatio = 1$ since all detected failures were classified and resolved through fixing the correlating faults. We were able to trace the correlating mistake-fault-failure chains for all encountered failures in this case study. All encountered mistakes, faults, and failures could be classified successfully. All encountered faults could be fixed, which results in all correlating failures no longer appearing. The comparison of our classification with previous work indicates that the classification is complete and correct. Moreover, we were also able to validate further the classification proposed by Klare et al. [22], as all non-implementational mistakes, faults, and failures could be classified.

6. Lessons Learned

While the previous chapter analyzed the results of the case study, this chapter discusses the lessons that can be learned from the case study results. Thus, this chapter is meant to assist in avoiding issues when building bidirectional transformations for networks of transformations. First, section 6.1 answers *Research Question 3*, which asks about the prevention of mistakes, by discussing strategies that allow the prevention of mistakes reliably during the construction of the transformations. Second, section 6.2 answers *Research Question 4*, which asks about unpreventable mistakes and faults, by discussing all three encountered unpreventable mistakes that cannot be prevented during the construction of bidirectional transformations and their correlating faults. It also explains what these mistakes have in common and what transformation and network developers should keep in mind. Third, section 6.3 discusses additional challenges that we did not directly encounter during our case study. However, we identified them as potential sources for mistakes when it comes to transformation design. We do not solve these challenges. However, we discuss these challenges to highlight the difficulties that come with them. Last, section 6.4 answers *Research Question 5*, which asks about the effects of redundancy by discussing the different effects of redundancy in the network observed during this case study. More specifically, we discuss how redundancy affects the mistakes, faults, and failures in a network of bidirectional transformations.

6.1. Prevention Strategies

Research Question 3 asks which strategies reliably allow the prevention of mistakes, faults, and failures by construction. It also asks about the knowledge required to do so. In this section, we answer the given question with a particular focus on the network knowledge mistake by proposing prevention strategies for specific types of mistakes. These strategies contain patterns that instruct the transformation developer on how to solve the specific problem. First of all, if we prevent a mistake of a causal chain, it will not lead to a manifestation of a fault and therefore, no failures will be caused. Therefore it is enough to prevent the mistake since it also prevents the fault. We divided the mistakes into three categories: Technical mistakes, transformation knowledge mistakes, and network knowledge mistakes. Technical mistakes can be prevented by carefully implementing a transformation design and utilizing methods that are known from classical software testing, such as unit tests and code reviews. We, therefore, do not discuss the prevention of these mistakes any further. Transformation knowledge mistakes can be prevented by carefully designing the transformation rules, as it only requires knowledge about the domains of the source and target model. Similar methods as for the technical mistakes can be employed. Transformation knowledge mistakes regard more the topic of transformation

construction and less the topic of multi-model consistency preservation. Therefore, we do not discuss the prevention of these mistakes any further. The last category of mistakes are mistakes based on network knowledge. We enable the prevention of these mistakes by introducing three strategies to avoid problems as early as during the transformation construction. All three strategies were derived from the fixes that were used to correct the faults encountered in this case study and from the knowledge we derived on the correlating mistakes. For each strategy, we discuss the underlying problem, how this problem can be avoided with the strategy, what needs to be considered to implement the pattern and what knowledge is required to do so. First, we discuss preventive naming scheme enforcement to avoid failures due to inconsistent naming schemes. Second, we introduce the create-on-rename-pattern, which deals with the edge cases caused by the change propagation of unnamed model elements. Third, we discuss the find-or-create pattern, which enables dealing with the external creation of model elements when checking if a correlating model element already exists.

These three prevention strategies cover the fixes for 19 of the 29 faults. This means one of the strategies matches the fix or is a different fix. The preventive naming scheme enforcement covers four faults that are all naming scheme related. The create-on-rename pattern covers three faults which are related to the propagation of unnamed elements. The find-or-create pattern covers 12 faults, which are all due to not considering the external creation of model elements. Consequentially, 19 of the 29 mistakes could have been prevented with these three strategies alone. More importantly, these strategies prevent all but three network knowledge mistakes. Combined with the previously discussed preventability of technical mistakes and transformation knowledge mistakes, 26 out of 29 mistakes can be reliably prevented during the transformation construction.

6.1.1. Preventive Naming Scheme Enforcement

To prevent failures due to naming scheme inconsistencies during the transformation construction, we propose the pattern *preventive naming scheme enforcement*. Preventive naming scheme enforcement describes always enforcing a naming scheme, even if it would not be explicitly required. As an example, let us assume we want to transform from the metamodel $\mathcal{M}_{source}$ to the metamodel $\mathcal{M}_{target}$, as they share a certain amount of overlapping information. Therefore we are constructing a unidirectional transformation $T_{source \to target} : \mathcal{M}_{source} \mapsto \mathcal{M}_{target}$. Let $S \in \mathcal{M}_{source}$ and $T \in \mathcal{M}_{target}$ be model element types with S mapped to T. When $s_i \in S$ is created, $T_{source \to target}$ is responsible for creating the correlating element $t_i \in T$. Let us consider the mapping for a pair of matched named elements (s, t) with $s \in S$ and $t \in T$.

First, let us assume that elements of type T require a name that starts with a capital letter. As a consequence, the transformation $T_{source \to target}$ is required to enforce $t.name = s.name.toFirstUpper$ to keep the pair (s, t) consistent. Naturally we implement the transformation $T_{source \to target}$ according to this naming scheme. This means it capitalizes the names of these elements. Second, let us consider another case. In this case, both elements of the types T and S require a name with capital first letters. As a consequence, the transformation $T_{source \to target}$ could enforce $t.name = s.name$ to keep the pair (s, t) consistent. When implementing $T_{source \to target}$ there are now different op-

tions. One might be inclined to implement this name mapping as previously stated, with $t.name = s.name$, because the name of s should already be capitalized. For $T_{source \rightarrow target}$ this means when element (s) is renamed, t should be renamed to the new name of s. The problem is, assumptions like these break in networks of bidirectional transformations, as unintended states appear through the side effects in the network. Let us imagine an unintended state leads to the name of (s) being temporarily in lowercase. As a consequence, this would lead to t having the same lowercase name, even though the transformation designer explicitly considered the naming scheme. When following the pattern of preventive naming scheme enforcement, we would instead implement the mapping for this second case between s and t as $t.name = s.name.toFirstUpper$, which means we enforce the capitalization, even though we assume that source name is already capitalized. This way, the target element is renamed as expected (with a capital first letter), even if the source element is not named as expected. It helps to avoid naming scheme inconsistencies due to unexpected or unintended states.

The knowledge required to implement this pattern in a bidirectional transformation is only the domain knowledge of the source and target metamodels and how to transform between them. The reason for this is that each transformation only needs to enforce the naming schemes of the source elements. We illustrate this pattern for names, as they are commonly affected by failures in networks of transformations (see Table 5.2) due to their importance in models and because we encountered this problem in the case study with element names. However, this pattern can be generalized for any property of a model element where the property is transformed from or to another model element with certain constraints.

6.1.2. Create-on-Rename Pattern

During the case study, we identified edge cases, like edge case values of attributes, as a source for mistakes. Furthermore, names are often affected by failures (see Table 5.2). A common failure that affects names and is based on edge cases is the creation of unnamed elements. Unnamed elements are elements where the element type defines a name attribute, but the element itself has no value set for this attribute. We explicitly exclude elements whose type does not define a name attribute, meaning elements that cannot be named. For some models, unnamed elements might be unusual but valid, therefore unexpected. For other models, unnamed elements might invalid and, therefore, unintended. We observed unexpected or unintended model states to be a common cause for failures. Transformations might not be built with unexpected or unintended model state in mind and, therefore, further propagate inconsistencies in the network if that is the case. As an example, consider a model M_1 where elements are always created with a name. When transforming to that model M_1 from another model M_2, where unnamed elements are considered valid, the transformation creates unnamed model elements for the model M_1 that it usually would not contain. This means the model M_1 has an unexpected state. When we now transform from M_1 to any other model, the transformation designer might expect the elements to be always named, as is usually the case. Because the assumption is now broken, this can lead to failures.

We, therefore, propose the *create-on-rename* pattern that prevents these issues by construction. The pattern prescribes only to create target elements for source elements if the source element is named. To be specific, when a source element is renamed, a correlating target element is created if the source element was previously unnamed. When a target element is created, the source model is not changed at all. As an example, Java packages should never be created without a name, as the containing Java model would not be semantically correct and it leads to failures. This means when transforming from any model to Java, packages should be created only when the correlating source element is renamed, which implements the create-on-rename pattern. During our case study, we encountered such an issue with *Fault 9*, where an unnamed PCM repository led to the creation of unnamed Java packages.

This pattern should be used in transformations where elements of the target model have a name attribute that should always be set. If that is the case, the pattern should be applied in the transformation to every transformation rule regarding element creation, where the element types define name attributes. We argue that this pattern is more likely the case when names are used to identify model elements. If that is not the case, for example because unique identifiers are used, unnamed elements in the model might be considered as valid. This pattern can be implemented without any additional knowledge, as all knowledge required for this pattern needs to be already available to design the transformation itself.

6.1.3. Find-or-Create Pattern

The most common failure during this case study was duplicate creation, which accounts for 68.9% of all failures. Consequentially, it is crucial to prevent these failures by avoiding the mistakes that lead to them during the construction of the individual transformations. Most of these failures we caused by the fault of not checking on external creation by other transformations when creating elements. In order to prevent this, we propose the *find-or-create* pattern. We see this pattern as the logical extension of the transformation-internal existence checks proposed Syma [46] for dense networks. The existence checks by Syma are only meant for tracking element creation in a single bidirectional transformation and, therefore, not sufficient to prevent duplicate creation between multiple bidirectional transformations. As an example, let us assume we want to transform from the metamodel $\mathcal{M}_{source}$ to the metamodel $\mathcal{M}_{target}$, as they share a certain amount of overlapping information. Therefore we are constructing a bidirectional transformation $T_{source \to target} : \mathcal{M}_{source} \mapsto \mathcal{M}_{target}$. Let $S \in \mathcal{M}_{source}$ and $t \in \mathcal{M}_{target}$ be model element types with S mapped to T. When $s \in S$ is created, $T_{source \to target}$ is responsible for creating the correlating element $t \in T$. If $T_{source \to target}$ contains the aforementioned fault, it is only going to check if it already previously created an element of type T that matches s. If this is not the case, t is created and matched with s.

This breaks as soon as more than one transformation transforms changes to $\mathcal{M}_{target}$. Lets assume we use $T_{source \to target}$ in a network with second transformation $T_{other \to target} : \mathcal{M}_{other} \mapsto \mathcal{M}_{target}$ for the model $\mathcal{M}_{other}$. Let $O \in \mathcal{M}_{other}$ be a model element type with O mapped to T. When $T_{other \to target}$ is executed before $T_{source \to target}$ it leads to the following events: First, the creation of $o \in O$ leads to the creation of $t_1 \in T$ by $T_{other \to target}$. Second,

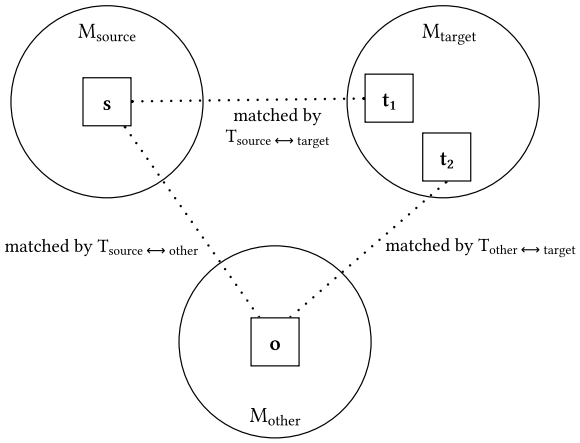

Figure 6.1.: Duplicate creation of two semantically identical elements in model M_{target} due to the missing check on external creation.

s is created and is matched with o. This means s could be created by a transformation between $\mathcal{M}_{other}$ and $\mathcal{M}_{source}$. The transformation $T_{source \to target}$ now checks if it already previously created an element of type T that matches s. Since this is not the case, it creates $t_2 \in T$. However, because of the transitive matchings from t_1 to s, from s to o, and from o to t_2, it is semantically identical to t_1. These matchings are visualized in Figure 6.1. As a result, two duplicated elements of T exist, one matching s and one matching o, while there should be only one that matches both. If t_1 is contained in the model through a single-element containment, it is overwritten by t_2. If t_1 is contained in $\mathcal{M}_{target}$ through a multi-element containment, both duplicate elements are co-existent.

This can be fixed by ensuring that every transformation checks if any other transformation already created a matching element before creating a new one. The difficulty for preventing this during the transformation construction hereby lies in the fact that the transformations do not know of each other's existence, as they were created independently without each other in mind. Luckily, the only network knowledge required to prevent these faults is the fact that the transformation could be used in a network of bidirectional transformations. We identify a pattern that prevents these faults by construction. We call this pattern the find-or-create pattern. This pattern is depicted in Algorithm 1. Note that *matches*() checks if two elements are matched, while *shouldMatch*() checks for two elements if their element types are mapped and, more importantly, if they represent the same concept. This is, for example, the case when an element has exactly the same properties as the element to be created. Moreover, *match*() matches two elements for the transformation and is therefore used to restore a matching, while *create*() creates an instance of an element type.

The knowledge required to implement the find-or-create pattern in a bidirectional transformation is only the knowledge required to design the bidirectional transformation and additionally the knowledge that this transformation might be used in a network of bidirectional transformations, which motivates the problem of external creation. Thus, the main challenge is to implement this pattern for specific metamodels. Especially locating

Algorithm 1 Find-Or-Create Pattern for $T_{source \to target} : \mathcal{M}_{source} \mapsto \mathcal{M}_{target}$

$T \in M_{target}, S \in M_{source}, s \in S$
if $\nexists t \in T : matches(s, t)$ **then**
  if $\exists t \in T : shouldMatch(s, t)$ **then**
    $match(s, t)$
  else
    $t \leftarrow create(T)$

potential matches to check if there is an element that should match is not always straight forward.

When trying to identify potential matches, two problems arise: First, the problem of aggregating all model elements of the desired element type. As other transformations create potential matches, they might not be known to the implementing transformations. If the target model does not offer access to all elements of a particular type, they might need to be located, in the worst case by traversing the whole model structure. Second, the problem of identifying the potential match. Given a set of model elements with the same type as the element to be created, which one represents the element the transformation is about to create? To identify a potential match, it must be semantically identical to the element that is about to be created. At best, unique identifiers can be used. If that is not the case, the model structure, as in containments, or a set of properties can be utilized. We identified four different approaches for identifying potential matches, all of which solve the two problems mentioned above.

locate via direct containment First, *locate via direct containment*. Because containment structures play an important role in many models, it is often a viable solution to utilize them for the identification of matches. When the parent element of the element to be created is known, it is trivial to use the containment, which would have been used to add the newly created element to locate potential matches. If the containment in the target model is a single element containment, the potential match can be directly retrieved. If it is a multiary containment, an identifier, name, or a combination of properties might be used to identify the right target element. As an example, when trying to locate potential matches of a PCM component in a UML model, the containing UML package is known, as it is needed to insert a newly created class. This means one can locate the potentially matching class by name among all the classes contained in the UML package.

locate via correlating containment This is a variation to the previous solution for cases where the parent element of the element to be created is not known. If the parent elements of both the source and the target element are mapped, it is possible to take the parent of the source element and retrieve its correlating counterpart in the target model. This target model parent element can be used to identify a potential match with the containment relation. Again, either directly, if it is a single element containment or with an identifier, if it is a multiary containment. As an example, when trying to find a potentially matching Java interface for a UML interface, one can retrieve the correlating Java package of the containing UML package and then

locate the potentially matching interfaces through the contained compilations units and further by their contained interfaces. An interface is a match when its name equals the name of the UML package.

locate via model traversal If it is not possible to directly or indirectly locate the container of the element to be created, a potential match can be found by traversing the whole source model. Models are often designed in tree-like containment structures like a directed cyclic graph. However, this approach requires one or more properties that can be used to identify a match, such as a unique identifier or name. As an example, this approach can be used when trying to locate a potentially matching UML package for a Java package. As Java packages are, in the model used in this case study, root elements and are not in a containment hierarchy, it is not possible to locate a parent element of the UML package to be created. However, since all elements of the UML model are contained under one root element, this structure can be easily traversed. The names of the traversed UML packages are compared to the Java package namespace in order to locate a matching UML package. This means that the namespace of the Java package must be a concatenation of the names of the UML packages from the root to the potential match.

locate via all instances if all of the previous approaches are not applicable, an alternative is explicitly tracking all instances of a model element. This is usually the case for root model elements. This approach locates potential matches by filtering all instances of the target element type by unique properties such as name or identifier. To access all instances of a model element type, it might be required to maintain a set of instances, for example, by matching the instances with their correlating metaclass. As an example, this approach is required when trying to locate a potentially matching Java package for a correlating UML package. Java packages are individual root elements and, therefore, not directly contained by another element. To access all existing Java packages, they are hence matched with the package metaclass when created. These matchings grants access to all instances, even if the model does not allow it. When iterating the list of all instances, the package namespaces are used to find a potential match.

All of these approaches are only applicable if some conditions are met. All four approaches require the model element type to have some unique properties that allow identifying a potential match. This can be a unique identifier or any combination of properties that are, all together, unique for each element. If there is no such identifier or set of identifiers, it is still possible to use non-unique properties. However, in this case, there is no guarantee that the right element is matched, as multiple elements might satisfy these properties. The first approach, *locate via direct containment*, also requires a containment reference to be known that would be used to insert the element to be created. The second approach, *locate via direct containment*, requires that such containment reference exists and that the parent elements of the source and target elements are matched. The third approach, *locate via model traversal*, requires the model to be traversable. The fourth approach, *locate via all instances*, requires some way to access all instances of a model element type. If the conditions of multiple of these approaches are met by properties of the source and target

metamodels, it is up to the transformation designer to pick what approach to use. We recommend considering these approaches in the order of their listing, as they become less and less clean by a design standpoint in ascending order. Additionally, the computational complexity of the two latter approaches is worse compared to the first two, as the third might require traversing the whole model to exclude the existence of a potential match, and the fourth approach requires checking all instances of a model element type. This makes the latter two approaches less feasible for models with a large number of elements.

When designing a transformation language, one might consider including language constructs that support the find-or-create patterns. Languages that separate the checking of patterns (matching) and the restoration of consistency (actions) should support the find-or-create patterns with the matching syntax. The Reactions language of the Vitruvius is such a language that distinguishes between matchings and actions. We, therefore, propose to extend the Reactions language to support the find-or-create pattern natively. This would reduce the time overhead for the transformation designer during the transformation construction and make the transformation definitions more concise. Additionally, supporting the find-or-create pattern with the transformation language might help the transformation designers to consider the scenario of external model element creation.

6.2. Unpreventable Mistakes

Out of all 29 mistakes in this case study only three cannot be prevented by construction: These mistakes are *Mistake 10* and *18* as well as *Mistake 21*. This section addresses *Research Question 4*, which asks about the existence of mistakes that cannot be prevented during the construction of bidirectional transformations. We answer *Research Question 4* by discussing all three unpreventable mistakes and the faults in which they are manifested, as well as by explaining what all three mistakes have in common. Even when knowing that the transformations are supposed to be used in a network of transformations, preventing these failures would still both require detailed knowledge about the network structure as well as insight in the interaction of the transformations and the change propagation. It is essential to prevent failures in a network during the construction of transformations, as this is the only way to ensure that any network that contains the transformations is able to restore consistency correctly. Since this means we want to prevent failures before assembling the network by combining the transformations, we do not know any details about the network itself. Consequentially, the prevention of any failures by construction can only rely on the knowledge about the two domains of the transformations: The source and the target model. While it is fair to assume the use of the transformation in a network, anything beyond that needs to be considered as unknown. As a result, mistakes like the three previously mentioned ones cannot be prevented during transformation construction, as this would require the developer of a single transformation to know about the others. However, they can be fixed later when assembling the network of bidirectional transformations. In the following we discuss *Mistake 10* and *18* as well as *Mistake 21* in detail. *Mistake 10* and *18* are mistakes that manifested themselves in faults of mismatching root management. *Mistake 21* manifested itself in faults of mismatching naming schemes. In this section, we discuss both types of unpreventable mistakes that we encountered

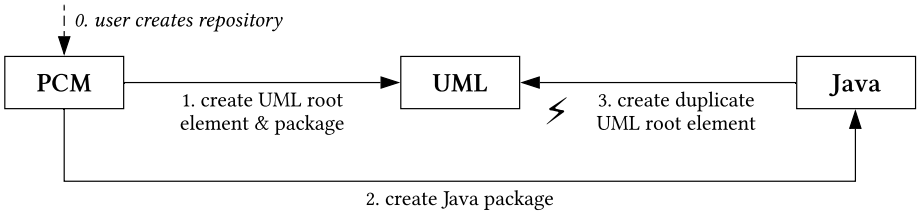

Figure 6.2.: Mismatching root element management that results in a duplicate creation failure. The UML root element is created twice: First by $T_{PCM \to UML}$ and a second time by $T_{Java \to UML}$.

in this thesis. We then explain what these mistakes have in common and what we can takeaway from these mistakes.

6.2.1. Mismatching Root Element Management

Both *Mistake 10* and *18* manifest themselves in a very similar fault, namely a difference in managing UML root elements. The UML metamodel utilizes a root element called *model* that contains all other elements in a tree. Usually a *model* contains packages which then contain types. For clarity, we use the term *root element* instead of *model* to avoid confusing it with the source or target models of a transformation. The problem lies in managing these root elements. Every transformation that transforms into UML models needs to find out whether a UML root element already exists. If this is not the case, it needs to be created to apply any further changes to the UML model. In order to find out whether the model exists, a transformation usually assumes the name and location of the root element. Some transformations might expect it to be at a location correlating to a fixed path. Other transformations might use dynamic names and locations depending on the information contained by the source model of the transformation. Lastly, the transformation could also require the user to specify the name and location of the root element. After the creation of the root element, transformations with source models that also utilize root elements can map these root elements and keep restoring consistency through this. Transformations that cannot map an element to the UML root element need other mechanisms for consistency preservation. One possibility is to manage a global list of all UML root elements known to the transformation. In the worst case, this would mean asking the user for the UML model every time changes are propagated to UML.

Even with user input that is as expected, deviating ways of managing UML root elements can lead to issues during consistency preservation. In our case study, this was caused by the transformations $T_{PCM \to UML}$ and $T_{Java \to UML}$. $T_{PCM \to UML}$ always asks the user for the name and location of the root element. If such a root element does not exist, a new root element is created and used. $T_{Java \to UML}$ remembers a root element by mapping it to a specific tag. This means the transformation first checks if it already created a root element. If such a root element does not exist either, a new one is created which will then be remembered. This difference means that in the case study network, $T_{PCM \to UML}$ might be executed first, which then creates a root element. When $T_{Java \to UML}$ is executed, no root element is mapped to the tag and therefore a second root element is created. This

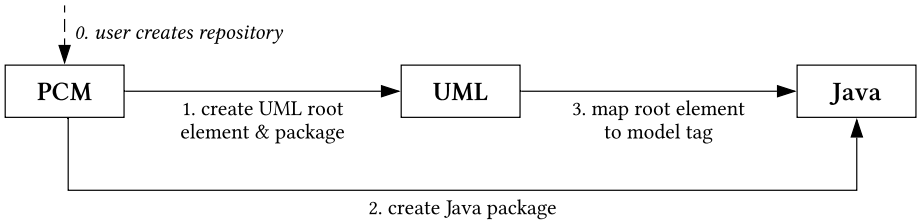

Figure 6.3.: Mismatching root element management that does not result in a duplicate creation failure. The execution $T_{UML \rightarrow Java}$ before $T_{Java \rightarrow UML}$ prevents the duplicate creation failure although the fault exists.

is depicted in Figure 6.2. During this case study, this very fault led only sometimes to a failure depending on the execution order. When the backward transformation $T_{UML \rightarrow Java}$ was executed before the forward transformation $T_{Java \rightarrow UML}$, the mapping to the tag for the newly created root element was created, and the failure did not occur. The execution order where the failure does not occur is illustrated in Figure 6.3. This means $T_{PCM \rightarrow UML}$ and $T_{Java \rightarrow UML}$ are both deeply incompatible even though both transformations on their own are working as intended and are designed under reasonable assumptions. While this problem is easy to fix by adapting all transformations with UML as the target model in a way that they manage root elements, in the same way, it is not possible to prevent this issue during the construction of a transformation on its own without knowing the topology of the network and additionally how each relevant transformation manages root elements.

6.2.2. Mismatching Naming Scheme

Mistake 21 is also a mistake that cannot be prevented during the independent construction of the transformations. The fault correlating to this mistake is the inconsistency of naming schemes regarding PCM repositories. $T_{PCM \leftrightarrow UML}$ maps a repository to UML package while $T_{PCM \leftrightarrow Java}$ maps a repository to Java package. Each pair mapped by these transformations are supposed to have consistent naming. Consistent for these relations does not mean equal, as different naming schemes apply to PCM repositories, UML packages, and Java packages. Both Java and UML packages are generally named with the first letter lowercase, which is enforced by the transformations. The PCM metamodel allows repositories to be named with the first letter being upper- and lowercase. This means no naming scheme is enforced. At first glance, this seems like an issue that can be easily prevented during the transformation construction. Always using a lowercase first letter when transforming from repositories to UML or Java packages and always using an uppercase first letter the other way round is a reasonable expectation for the transformations, as during the construction of a transformation the creator is expected to have knowledge of both the source and target domain. However, this is not enough to avoid failures in a network of transformations.

Fault 21 sometimes leads to failures, depending on the execution order in the network. Figure 6.4 shows how such a failure arises. Initially, a PCM repository with a lowercase

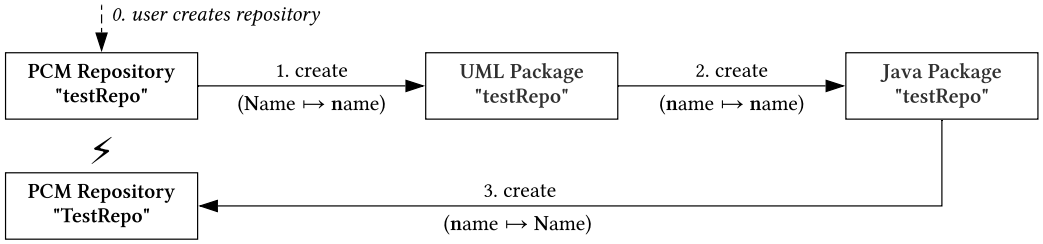

Figure 6.4.: Mismatching naming scheme expectations that result in a duplicate creation failure. Two repositories are created instead of one due to $T_{Java \leftrightarrow PCM}$ expecting one with a name with a lowercase first letter.

name is created by a user. First, this change is propagated by $T_{PCM \leftrightarrow UML}$, which creates a UML package with the same name, as it already is lowercase. For a name with an uppercase first letter, the name would have been converted. Second, this new change is then propagated by $T_{UML \leftrightarrow Java}$, which creates a Java package with the same name as the UML package. Third, $T_{Java \leftrightarrow PCM}$ propagates this change back to the PCM model. Just as the previous transformation did, it checks if a correlating target element already exists. In contrary to the previous transformations is that a correlating target element exists. Because no element is directly mapped, it checks if a PCM repository exists, that matches the name of the Java package. However, because it expects a repository with an uppercase first letter, no matching repository is found. Consequentially, the transformation created a new repository with a first letter uppercase name. As a result, two duplicate repositories exist.

To return to the question of prevention, this fault can not be prevented for the transformations by construction because at least one transformation restricts the naming scheme of a metamodel even further. Consequentially, the knowledge that the transformation is used in a network and that there is transitive change propagation, is not enough. What is required is detailed knowledge about the network topology, what naming schemes are employed by all models in the network and, most importantly, if they are further restricted by transformations. Since during the construction of a transformation, there is no guarantee which metamodels and, therefore, which naming schemes are eventually part of the network, the transformation can only guess whether the naming scheme of a source or target model is enforced as specified or if another transformation further restricts the naming scheme. During the construction of the network, these faults can be fixed by defining what naming scheme can be expected when looking for an already existing element. In this case study *Fault 21* (as depicted in Figure 6.4) could be fixed in the transformation $T_{Java \leftrightarrow PCM}$ by looking for existing repositories with a name that starts with an uppercase or lowercase letter. While this fix works for this specific network, it does not work anymore if we connect a new metamodel $\mathcal{M}$ to this network with a transformation $T_{PCM \leftrightarrow M}$ which enforces a different naming scheme, such as all uppercase. This shows again that the issue cannot be prevented during the transformation creation.

Additionally, some pairs of naming schemes allow only defining a forward transformation but not the correlating backward transformation. An example of such a pair of

naming schemes is first-letter-uppercase and all-upper-case. First letter uppercase can be transformed into all uppercase, but the other way round it is not possible, as it is unclear for every but the first letter whether the letter is supposed to be upper- or lowercase. This suggests that it might be possible to define a partial order for naming schemes, which is not discussed in this thesis as it goes far beyond its scope. This topic of decomposition of relations for multi-model consistency preservation is further discussed in [31].

6.2.3. Similarities and Takeaway

Both of these unpreventable mistakes are rooted in an underlying issue. This issue is the cause of the mistakes not being preventable by during the transformation construction. Both mistakes are rooted in a situation where transformations are forced to decide between a set of options, where there is no inherently correct choice. For the mismatching root element management, it is the choice *how* to manage these root elements. In this case study, we encountered three different options: A fixed name and path, dynamic names and paths, and name and path based on user input. However, this is an open problem, which means there might be more options. For the mismatching naming schemes, the decision is if and how to restrict the naming scheme during the transformation. Similarly to the root element management, there are multiple options. In general, different transformations in a network of bidirectional transformations can now make different choices. If these choices are different, this may lead to failures if the different choices are creating conflicts during the change propagation in the network. However, this cannot be prevented during the transformation construction, as there is no knowledge of how other transformations may choose. Only when the network is being assembled, and it is clear what transformations are part of the network, it is possible to check if all transformations made the same choice or not and if there is a potential for conflicts.

In summary, these problems are not preventable during the construction of the transformation because the transformations are forced to decide between a set of options, where there is no inherently correct choice. Different transformations in a network can now make different choices, and during the transformation construction, it is not clear what option other transformations chose. Even when knowing the transformations may be used in a network, it is not possible to predict the consequences of the transitive change propagation at all. Consequentially, transformation developers should avoid making these choices if possible. If they need to be made, it is essential to consider how other transformations might choose. However, that still does not prevent any faults. Network developers need to be aware of these faults in order to spot and resolve these faults when assembling a network by combining transformations.

6.3. Other Challenges for Multi-Model Consistency Preservation

In this section, we discuss two topics that we identified as especially challenging during this case study. However, these topics are not directly part of the encountered mistakes, faults, and failures. We identified these topics during the development of fixes for unrelated

faults. They are potential sources for mistakes when it comes to transformation design. We do offer any solutions to the problems of these topics in this thesis since this is beyond the scope of the thesis. This thesis is meant to analyze mistakes, faults, and failures that arise during the case study and not any potential mistakes, faults, and failures. However, it is essential to discuss these topics to highlight the difficulties that come with them. First, we discuss the topic of default values of attributes and how they lead to difficulties during the specification of consistency preservation. Second, we discuss how multiple model roots are challenging during the consistency preservation process.

6.3.1. Default Values

Default values can be defined for attributes of model elements. They set the attribute value at the moment the model element is created to a specific pre-defined value. For example, named attributes can have default names. In this case study, instances of named elements of the PCM metamodel are initialized with the default name "aName". Default values are relatively problematic when keeping model elements consistent that both have the same default values, or one of them has no default values at all. Problems arise when two mapped model elements have different default values, or one of the mapped model elements has no default value. In the following, we will discuss both cases to illustrate how, on an exemplary basis, default values can complicate consistency preservation.

However, before we discuss the two problematic cases, let us first look at the unproblematic case where no default names are used. Figure 6.5 depicts this situation. In both models, the names can have two states. Either they have no name, or they have a name. If one of the names is not set, the other name needs to be not set as well. If one of the names is set, the other name needs to be kept consistent, which means it is updated to have the same name. This is the basic case, which is meant to illustrate how consistency preservation gets more challenging when default names are introduced. In the following, we discuss both problematic cases.

First, let us discuss the issues for differentiating default values. Let us consider the example for default names, which means name attributes with default values. When keeping two model elements from different models consistent, the default values of the names of the model elements might differ. This introduces several constraints to consistency preservation. Figure 6.6 depicts the consistency relations when keeping such elements consistent and how they are affected by the potential values of the names. First of all, when both model elements have their respective default names, they might need to be considered consistent. This means when an element of type A with the default name "a" is created in model M_1, a correlating element of type B with the default name "b" needs to be created in model M_2 to restore consistency. Second, non-default names need to be kept consistent. This is the case when names have been explicitly set. When, for example, an element of type A is renamed to a custom name, the correlating element of type B needs to renamed to that custom name as well. Problems arise when one element has a custom name that matches the default name of another element. In this case, a decision has to be made on how to map this name. Should it be mapped to the default name, or should the other element with the default name be explicitly renamed? Figure 6.6 visualizes how default names might need to be mapped to non-default names with the depicted

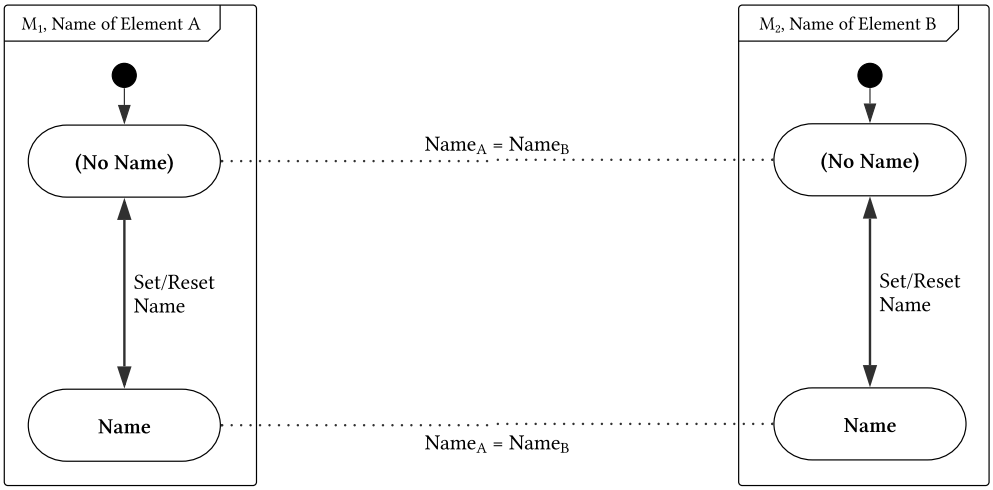

Figure 6.5.: State diagram that depicts consistency preservation between two elements with no default values at all. Note that only the state of one model element is depicted in each model. Dotted lines represent the matches that are made based on the consistency relations for the model element types, while non-dotted lines represent state transition for the attribute values.

consistency relations. Without any default values, the only consistency relation required would be the one between the two non-default names. Thus, this scenario introduces multiple edge cases to the consistency preservation that need to be considered.

Second, let us discuss the issues when one of the mapped model elements has no default value. Let us, again, consider an example with default names. When keeping two model elements from different models consistent, one element might have a default name, while the other one has no default name. This means the latter element is created unnamed. This introduces several constraints to consistency preservation. Figure 6.7 depicts the consistency relations when keeping such elements consistent and how they are affected by the potential values of the names. Again, non-default names need to be kept consistent between the two model elements. However, in this scenario, the default name needs to be kept consistent with the non-existent name. This means when an element of type A with the default name is created in the model M_1, an unnamed element of type B needs to be created in the model M_2 and the other way around. In a network of bidirectional transformations, this has to be considered across all transformations. For example, when looking for the existence of a potential match before creating an element (see Algorithm 1), default names and missing names need to be matched to avoid duplicate creation of elements with mismatching names. Even more problematic is that elements without default names might allow removing the name, which means the element is unnamed again. For default names, this is not always given, as the default value is often meant as a temporary value while the value was not explicitly set yet. This is depicted in Figure 6.7, where it is possible to transition between the two states of Element B, but only possible to transition in one direction between the two states of Element A. This means, when removing the name of an element of type B in M_2, it might be required to rename the

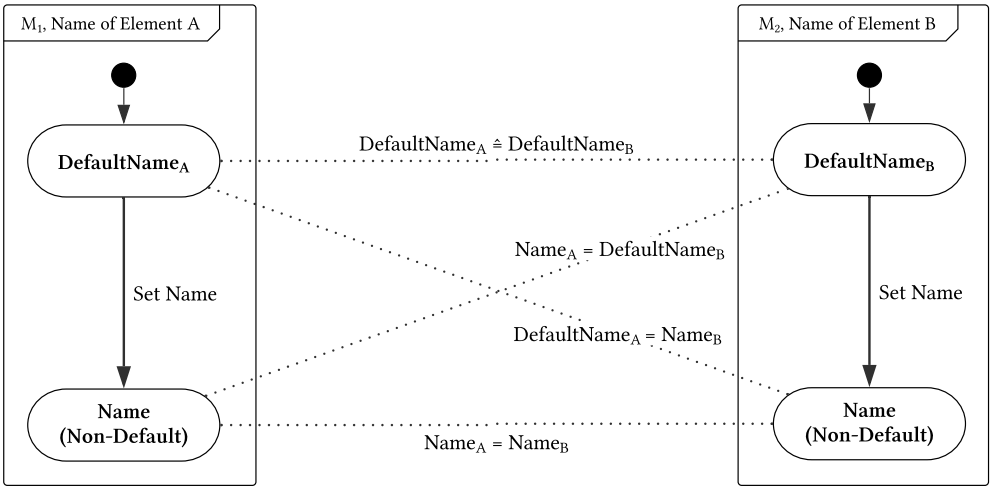

Figure 6.6.: State diagram that depicts consistency preservation between two elements with deviating default values. Note that only the state of one model element is depicted in each model. Dotted lines represent the matches that are made based on the consistency relations for the model element types, while non-dotted lines represent state transition for the attribute values.

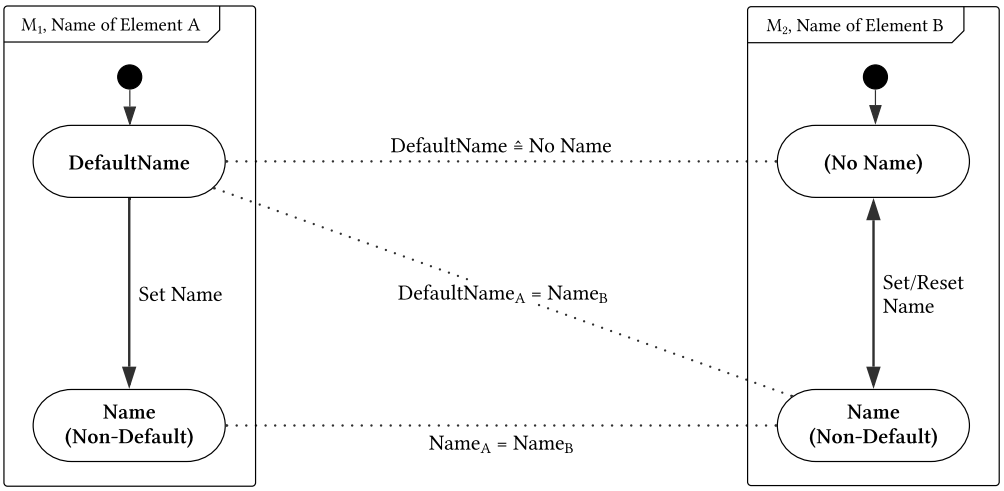

Figure 6.7.: State diagram that depicts consistency preservation between one element with a default value and one without. Note that only the state of one model element is depicted in each model. Dotted lines represent the matches that are made based on the consistency relations for the model element types, while non-dotted lines represent state transition for the attribute values.

correlating element of type A in M_1 to the value that also is the default name to keep the two elements persistent according to the relations in Figure 6.7. Since this is technically not a default value for the attribute, it might be hard to distinguish these two states (default name and explicitly set to the default name). These two states are different because, in a case where the name of an element of type A in M_1 has been explicitly set to a name matching the default value, this name should not be propagated back to M_2, as this would change the element of type B from unnamed to explicitly named. This is the opposite of the change that has been initially made to the model. To summarize, in this scenario, the introduced constraints can be even more complicated.

6.3.2. Multiple Model Roots

Many metamodels are designed in a way that model instances start from a single root element, and the containment structure of the modem instance form a tree. However, the structure of a metamodel can also be completely different. Some metamodels, for example, are built in a way that one model contains multiple root elements. Between the different root elements or children of the different root elements might be some relations, but none regarding containment. Technically, these root elements can be seen as independent models that reference each other. However, this does not affect the problems we are describing in the following. In this case study, we used the metamodel for Java, in which packages and compilation units are root elements. Consequentially, when representing a standard Java program with this metamodel, there are many root elements, as every type is contained in a compilation unit. Additionally, there are usually multiple packages.

This metamodel property of having multiple model roots increases the complexity of consistency preservation. There are multiple reasons for this. During our case study, we encountered the following difficulties regarding multiple root elements: First, when keeping a single-root model with a multi-root model consistent, there needs to be a mechanism to match elements from the single-root model with the elements under the correct root element from the multi-root model. For example, when transforming UML packages to Java packages, the UML packages that are all contained under one root element need to be matched to the individual Java packages that each on their own are root elements. In this case, such a mechanism would compare the Java package namespaces to the names in the package hierarchy of the UML model. If that cannot be automatically resolved, user input might be required. However, user input is generally considered a source for potential inconsistencies, as there is no control over what a user does. Second, when applying the find-or-create pattern (see section 6.1) for model elements that are root elements, three of the approaches for locating a potential match for the element to be created can no longer be used. Both approaches that use containment cannot be used, as root elements are, per definition, not contained by another model element. The approach of model traversal is not viable as well, as we are not trying to locate an element in a containment structure under a known root element. The only approach left is tracking all instances which requires some coordination between the different transformations of a network. Thus, this approach is not ideal when trying to employ the find-or-create pattern during the transformation construction, as the network structure might not be known.

To summarise, multiple root elements introduce additional complexity to the consistency preservation. Because the metamodels are often pre-existing, this problem cannot be avoided. Consequentially, the transformation designer needs to deal with these unique properties.

6.4. Consequences of Redundancy

During this case study, we observed that redundancy in networks of bidirectional transformations has effects on consistency preservation. Some of these effects were already mentioned in previous chapters. We discuss the details of the effects of transitivity in this section to answer *Research Question 5*, which explicitly asks how redundant bidirectional transformations affect the mistakes, faults, and failures in a network. One of the key differences to the case study of previous work is that in our case study, the network of transformations contains redundant paths. As previously mentioned, for each bidirectional transformation in our network (see Figure 4.1), the other two bidirectional transformations form a path between the source and target models of the initial bidirectional transformation. This path and the initial transformation contain a certain amount of redundant consistency preservation mechanisms. As an example from the case study, take the bidirectional transformation $T_{PCM \leftrightarrow Java}$. It contains transformation rules that map PCM operation interfaces to Java interfaces. This transformation ensures that creating an operation interface in a PCM model will lead to the creation of an interface in the correlating Java model. However, when chaining the other two transformations, namely $T_{PCM \leftrightarrow UML}$ and $T_{UML \leftrightarrow Java}$, this chain also maps PCM operation interfaces to Java interfaces, with the differences that it does it transitively via UML interfaces. This means that the path offers redundant transformation rules for the model element pair of PCM operation interfaces and Java interfaces. Essentially, all three transformations form a redundancy-based cycle for the interface concept.

In this case study, redundancy-based cycles were introduced to the network in the third stage with the transformation $T_{PCM \rightarrow Java}$ and then again with adding $T_{Java \rightarrow PCM}$ in the fourth stage. We observed an increase in failures for the last two stages. Additionally, when tracing the faults that caused the failures, we observed an increased number of faults. The number of faults and failures per stage can be seen in Table 4.2. A total of 84.87% of the failures occurred in the last two stages. We argued that redundancy-based cycles increase the number of issues during the change propagation and the severity of the issues (as in failures per fault are increasing). However, this increased number of faults or failures is only a consequence of the increased complexity. Consequentially we argue that an increase in redundancy in a network through redundancy-based cycles also results in increased vulnerability to inconsistencies and therefore requires even more careful transformation design. This highlights the need to employ preventive strategies for networks of bidirectional transformations, such as the three strategies introduced in section 6.1.

We also observed that the types of faults changed with the introduction of redundancy-based cycles. While the faults during the first two stages were only technical faults and faults of missing change propagation, in the last two stages the majority of faults regard the

transformation interaction (see Table 5.5). To be specific, out of the 22 faults in the last two stages, 18 regard the transformation interaction, with 12 out of those 18 faults being faults regarding creation conflicts due to not considering external creation. Consequentially, we argue that redundancy-based cycles in a network lead to an increased number of creation conflicts. However, this can be prevented by implementing the find-or-create pattern during the transformation construction (see section 6.1). It can be argued that this increase in failures is caused by the aspect of transitive change propagation, meaning the actual increase in chained transformations and the propagation of changes through these chained transformations. We, however, argue that the main factor, at least in our case study, is the redundancy between transformations that is caused by redundancy-based cycles, as both the creation conflict faults and the root element management faults are mainly dependent on the fact that multiple transformations in the network have the same target model while their source models are connected through other transformations. Creation conflict faults and the root element management faults account for 14 of the 18 faults encountered in the last two stages.

When trying to increase transitivity in a network, which means increasing the number of chains of transformations or increasing the length of these chains, without increasing redundancy in a network, it will result in a linear network (see Figure 3.4). For this topology, external creation conflicts as we encountered them are not a problem. While there are multiple transformations with the same target model, their source models are not connected through other transformations, as every model is a bottleneck in this topology. This is indicative of our argument that redundancy-based cycles are the main cause of the increase of failures in our case study. Despite these observations, it is essential to acknowledge that both aspects, redundancy, and transitivity, are heavily intertwined. For example, it is generally not possible to increase the redundancy in a network without increasing the transitivity. To summarize, we observed that redundancy, as in redundancy-based cycles, increased the number of faults and failures. We also observed that with the introduction of redundancy-based cycles the type of faults changed towards faults regarding transformation interaction. According to our observations, with the increase in redundancy, consistency preservation gets more complex and the network is more vulnerable to issues during consistency preservation.

7. Threats to Validity

In this chapter, we discuss the threats to the validity of our case study. In general, it is essential to acknowledge that a case study, by definition, can only cover specific scenarios. Consequentially, drawing any conclusion from a case study is threatened by the possibility by the chosen scenarios not being representable for the space the case study is conducted in. In our case, this means that the case study as a whole might not be representative enough for multi-model consistency preservation with networks of bidirectional transformations. However, this case study addresses a particular context (see section 4.1). To recapitulate this context, we assume the metamodels are pre-existing and generally designed for different tasks. We also assume that some or all model transformations might be pre-existing and defined by different experts without each other in mind. Third, we assume the amount of overlapping information might be varying depending on the metamodels. We only claim to be able to draw conclusions on this specific context and under the mentioned assumptions. The remainder of this chapter is structured in the following: First, in section 7.1, we discuss two threats to the internal validity of this case study: The number of test cases and the granularity of test cases. Second, in section 7.2, we discuss three threats to the external validity of this case study: The small size of the network, the choice of metamodels, and overfitting of the classification.

7.1. Internal Validity

Regarding internal validity, a threat is the choice of our test case set. Several factors can be discussed: The number of test cases, the granularity of test cases, and the bias regarding the concepts tested by test cases. First, we discuss the threat regarding the number of test cases. We utilized 39 test cases for the first two stages. It can be argued that more test cases are required to make the results more representative. As previously mentioned, we only used the 16 core test cases for the last two stages (see Table 4.1). The reason for this was that those 16 test cases already produced so many failures, that the time constraints of this thesis made it impossible to consider the remaining test cases for the last two stages. However, in the first two stages of this case study, all 39 test cases were used. The threat to the validity with this lies in the fact that these missing test cases could have changed the results of the case study and therefore influenced the classification. For example, additional mistake-fault-failures chains that give additional information on how different types of mistakes, faults, and failures are connected could have been identified. However, because the distribution of the mistakes, faults, and failures is very distinct, we argue that the distribution over the classes will not change. Moreover, counteract this threat, we compared our classification and analysis with previous work which suggests that it is indeed complete and correct. We plan on using the remaining test cases for the

last two stages as future work. Second, the granularity of the test cases might be a threat, as the test cases are fine-grained and mostly cover single concepts. Largescale test cases could be able to cause different failures. We also plan on doing that as future work. Third, while the tests attempt to cover different concepts of the three metamodels evenly, there is no guarantee that they did not miss any failures. To counteract this, we additionally checked the persisted models of the test cases manually to spot further inconsistencies. This manual inspection was able to detect the failures regarding improper deletion.

7.2. External Validity

Regarding external validity, we identified three threats to the case study: The small size of the network, the choice of metamodels, and overfitting of the classification. First, while the network used in this case study is a dense network of bidirectional transformations, its size is still relatively small, since it contains only three models. We assume we can generalize from this case study on any network of bidirectional transformations. However, we cannot guarantee this, as larger networks might bring unforeseeable challenges and therefore result in other failures, faults, and mistakes. Nevertheless, with this case study, we confirmed conclusions and generalizations made in previous work [22, 46] with a smaller, linear network. Moreover, we completed the classifications from the case study argument-based. This might indicate that it is possible to draw conclusions on larger networks of bidirectional transformations.

Second, another threat is the choice of metamodels. We utilized the same metamodels as the previous case study [22, 46], which allows comparing the results. However, we observed that the specifics of the metamodels and their overlapping information has a direct impact on the issues that arise during consistency preservation. This raises the question of whether the choice of metamodels affects the results of a case study through their structure. Since EMOF is a widespread standard and all metamodels in our case study conform to it, we can at least say that the general structure that a metamodel specifies is limited by the meta-metamodel itself. This, however, still leaves a certain degree of freedom in the metamodel design.

Third, overfitting of the classification. One can argue that our classification might be too specialized for the exact problem found during our case study. As a result, our classification could not apply to mistakes, faults, or failures of future case studies. We counteract this, as previously mentioned, by comparing our classification to the one proposed by Klare et al. [22]. However, while this confirms that we can classify their problems with our classification and the other way round, they used the same metamodels and transformations, which could affect the type of encountered mistakes, faults, and failures. Nevertheless, we argue that the set of possible types of mistakes, faults, and failures is finite as it correlates to the modeling formalism. For example, the classification according to the model state cannot be incomplete, as for EMOF models, the only options are that there are too few, too many or incorrect elements. In order to test this hypothesis, another case study would need to be conducted that uses other metamodels and, therefore, also other transformations.

8. Related Work

This chapter lists previous work that serves as the foundation for this thesis. Moreover, it discusses general related work to multi-model consistency preservation and networks of bidirectional transformations. First, section 8.1 discusses the previous case study regarding networks of transformations on which the case study conducted in this thesis is built upon. Second, section 8.2 lists different consistency preservation approaches. Third, section 8.3 discusses related related work regarding binary transformations. Fourth, section 8.4 section discusses related related work regarding multiary transformations.

8.1. Previous Case Study

This thesis builds on a previous case study by extending its setup. Regarding the previous case study, we refer to two publications. Klare et al. [22] analyze issues in networks of bidirectional transformations. The authors define different levels of consistency definitions, identify failures and mistakes in transformation networks, and discuss avoidance strategies. They conduct a case study with a linear network of bidirectional transformations between three metamodels. Syma [46] explores transitive combinations of binary transformations. The thesis catalogs six failure potentials and conducts a case study to evaluate its findings. Two patterns are proposed to deal with two specific failure potentials. It is also the basis for the previously mentioned work by Klare et al. [22]. These two publications are the foundation for our case study. While they conducted a similar case study with the same metamodels, they use a simpler, linear network without redundant bidirectional transformations. We extend their network of bidirectional transformations to include redundancy between transformations and chains of transformations. These two previous publications also discuss a broad range of problems, such as the change propagation order, while we focus only on failures and mistakes during change propagation in networks of bidirectional transformations. As we are trying to confirm the results of previous work, we compare our results and classification to previous work in detail in section 5.3.

8.2. Consistency Preservation

There are several related publications regarding consistency preservation and consistency restoration. In [20] Klare investigates problems in multi-model consistency preservation. His work discusses the interoperability of independently developed binary transformations, derives patterns for non-intrusive transformation interoperability, and proposes an approach for decomposing consistency relations. Klare identifies, among other problems, the following five challenges for multi-model consistency preservation:

interoperability describes the property of independently developed transformations to be combinable in a black-box manner.

compatibility describes consistency between transformations, meaning the requirement to transformations not to contradict each other.

modularity means transformations should only depend on their source and target metamodels, but not other metamodels or transformations.

comprehensibility describes the property that consistency relations should be as easy to understand as possible.

evolvability means transformations should be designed to be changed without too much effort.

This work by Klare is strongly related to this thesis, as it defines the concepts our case study is built on and identifies the challenges networks of transformations face. In [21], Klare et al. argue that overlapping information is often caused by multiple metamodels representing the same concept and therefore suggests making such duplicated concepts explicit by adding a concept metamodel which has relations to the initial metamodels. The authors call this the *Commonalities* approach. It leverages the hierarchical composition of such concept metamodels to enable multi-model consistency preservation. In our case study, however, we do not use this approach. Instead, duplicated concepts in overlapping information are kept implicit, and consistency is preserved with bidirectional transformations.

Lano et al. [25] propose patterns for the composition of transformations. While we are also trying to find patterns that can prevent failures and mistakes during change propagation in networks of bidirectional transformations, we are focusing primarily on finding the failures and mistakes. Denton et al. [9] discuss the challenges of combining models into multi-models. Model integration and consistency preservation are such challenges. Another contribution they make is an experimental platform for multi-modeling called *NAOMI*. Macedo et al. [27] propose a classification of consistency preservation approaches. They especially discuss concerns regarding inter-model consistency, dedicated multi-model support, and bidirectional transformations. Pepin [31] discusses the decomposition of consistency relations in order to detect redundant information within consistency relations. The decomposition procedure is meant to help in finding incompatibilities in consistency specifications. Meier et al. [28, 29] discuss and compare different consistency preservation approaches. All approaches are based on a Single Underlying Model (SUM), but differ in how they build and evolve such SUMs. Additionally, the authors present guidelines for selecting a SUM construction approach for specific projects. This relates to this thesis, as one of the approaches discussed is Vitruvius, the framework used in this case study. However, it is not possible to conduct this case study with the other approaches, as each approach is designed for specific use cases and therefore makes different assumptions on consistency preservation. As an example, some of the approaches merge the different models into one model, making networks of bidirectional transformations not applicable.

8.3. Binary Transformations

There is extensive research regarding binary transformations [18, 41, 42, 45]. Triple Graph Grammars (TGGs) are a special kind of graph grammar that is used for bidirectional model-to-model transformations [37, 36, 12]. Another concept, introduced by Diskin et al., is delta-lenses which are delta-based bidirectional transformations [10]. Stevens [45] introduces an algebraic representation for bidirectional transformations, and focuses mainly on lenses. As previously mentioned, the transformations in our case study are implemented in the Reactions language [19] as part of the Vitruvius framework [24], which allows to define binary transformations.

8.4. Multiary Transformations

There is an ongoing discussion about the use of multiary versus binary bidirectional transformations [20]. In this thesis, we only study networks of binary bidirectional transformations, as we want to analyze use cases where it is not straightforward to define consistency relations for multiple models, as a transformation designer would require expertise in each of the model domains. While this might be realistic for a few models, it is not feasible when increasing the number of models in a network to allow large-scale consistency preservation. QVT-R enables multiary transformations, but is, according to Macedo et al. [26], underspecified regarding the operationalization of multiary transformation. Macedo et al., however, propose an extension that solves this problem. Similarly, Trollmann et al. [47] extend Triple Graph Grammars to support multiary transformations. Stevens [43, 44] discusses multiary bidirectional transformations and explores how multiary consistency relations can be expressed through binary consistency relations. The author also studies how consistency might be restored in a network of bidirectional transformations. In [44], the author explicitly discusses bidirectional transformations modifying the same model without interfering with one another, which we observed to frequently fail during this case study. This is related to our work, as we base our multi-model approach on networks of bidirectional transformations. Stevens also discusses the problem of the execution order, which is a problem we are trying to abstract from in this thesis.

9. Conclusion and Future Work

In this last chapter, we conclude with a summary of the case study, its results, and other contributions. Finally, we discuss possibilities for future work.

9.1. Conclusion

In this thesis, we conducted a case study on the mistakes, faults, and failures that arise during multi-model consistency preservation with networks of bidirectional transformations and how they can be avoided. We based our case study on three pre-existing metamodels, namely the PCM metamodel, the UML metamodel, and the Java metamodel. We built a network out of three pre-existing bidirectional model transformations that were not designed with each other in mind. They are the pairwise transformations between the aforementioned metamodels. We used a pre-defined set of fine-grained test cases that first created a model instance for each metamodel and then made changes to one of the model instances in the network. Next, they called the consistency preservation mechanism that executed the model transformation one-by-one until the network is stable, meaning no transformation execution makes any further changes. Additionally to the automatic checks by the test cases, we also manually inspected the persisted models to not miss any failures. The failures either occur during the test cases or are caught during the inspection. Each fault is the cause of one or many failures and is manifested in the transformation definitions. It is the manifestation of a mistake made by a transformation developer. During this case study we found 119 failures, 29 faults, and 29 mistakes.

First, we categorized the failures according to the model state, which means the models are either missing elements, have too many elements, or have incorrect elements. Second, we categorized the faults according to their scope, meaning if the fault is technical, transformation-internal, or regarding the transformation interaction. Technical faults are faults where the transformation design is correct, but the transformation is incorrectly implemented. Transformation-internal faults are faults where the transformation design of a single transformation is incorrect. Faults regarding the transformation interaction are faults where the design of multiple transformations leads to faulty interactions between them. Third, we categorized the mistakes according to the knowledge scope, which means what knowledge is required to avoid a mistake. We found three categories: Technical mistakes, transformation knowledge mistakes, and network knowledge mistakes. Technical mistakes are mistakes based on missing technical knowledge. For example, this can be the knowledge of the transformation language. Transformation knowledge mistakes are mistakes where there was either missing or incorrect knowledge regarding the design of a transformation. This can be, for instance, the domain knowledge of the source and target metamodels. Network knowledge mistakes are mistakes that regard knowledge on the

network of bidirectional transformations and on how changes propagate in that network. The most common failure class is the duplicate element creation where two elements that are semantically identical are created in a model. These failures account for 68,9% of the failures in this case study. They are often caused by the fault of not considering that, when creating an element, another transformation might already have created a semantically identical element. This fault class is the most common fault in the case study making out 41.4% of all faults. These faults are manifestations of the most common mistake class, which are mistakes due to not considering transitive consequences at all. These mistakes account for 44.8% of all mistakes.

Our observations confirm that it is often possible to prevent faults in a network during the construction of transformations with minimal knowledge of the network. Even with just the knowledge that the transformation is used in a network of bidirectional transformations, it is possible to prevent 89.7% of all faults that we encountered. We argue that technical mistakes and mistakes regarding only knowledge of transformation can be prevented by careful transformation design and implementation. We identified three strategies that prevent different network knowledge mistakes encountered in this case study during the transformation construction. They prevent the manifestation of these mistakes in faults. Therefore, two-thirds of the encountered faults can be prevented with these three strategies alone. In total, only three mistakes we found cannot be prevented during the transformation construction. For these mistakes, it is not possible to predict transitive consequences at all. They all have one thing in common, which is the underlying cause for not being preventable by construction: Transformations are forced to decide between a set of options where there is no inherently correct choice. Different transformations in a network can now make different choices, and during the transformation construction, it is not clear what option other transformations chose. We compared the results of our case study to a previous case study by Klare et al. [22] and Syma [46]. They explored issues with change propagation in simple, linear networks of bidirectional transformations while the network in our case study is a dense network with redundant paths. This means there may be two or more concatenations of transformations that relate the same metamodels across different other metamodels. Their classification has some differences to ours, as they take a broader look at issues in networks of bidirectional transformations but also exclude some types of mistakes and faults. We can confirm through our results that they propose a complete and correct classification that fulfills its purpose. The other way round, their results also confirm the completeness of our classification due to the similarity of both classifications. We propose a hybrid classification for failures that combines aspects both of our and their classification (see Figure 5.1).

In conclusion, we observed that the most common failure type is failures of duplicate creation. The most common faults are faults of not checking if other model transformations already previously created a matching element when creating a model element. It is the manifestation of not considering that transformation may be used in a network of transformations. Almost all of the observed mistakes, faults, and, therefore also failures can be prevented during the transformation construction. Transformation developers can use the prevention strategies we proposed to prevent mistakes during the transformation construction. Network designers can use the knowledge regarding the choices transformations have to make, to spot unpreventable faults when assembling the network. This thesis

offers the following benefits. First, the three prevention strategies allow transformation developers to systematically avoid a large number of faults during the creation of the transformations by implementing their proposed patterns. Additionally, the classification of mistakes, faults, and failures increases the awareness of developers for those issues and therefore helps to avoid making mistakes. Finally, network developers benefit from the systematic knowledge regarding how choices made by transformations lead to unpreventable faults. This knowledge assists them in spotting and resolving faults when assembling a network by combining transformations.

9.2. Future Work

We discuss future work in the following. We used only a subset of the test cases for the third and fourth stages due to the time constraints of this thesis. Therefore, we plan on using the remaining tests to find further mistakes, faults, and failures. Which we then plan on using to confirm further our classifications and our conclusions on networks of bidirectional transformations. Furthermore, the set of test cases could be extended to cover more scenarios, and therefore might allow finding even more failures. This is intended to be future work. Extending the network with another metamodel is also possible. This, however, requires additional pre-existing transformations which might not be available. Nevertheless, extending the network is a possibility for future work.

In this thesis, we abstracted from the execution order of transformations in the network. Regarding possible future work, one might consider finding an approach to determine an optimal execution order of the different transformations in the network. However, there is no optimal execution order in many cases. Consequentially, it might be required to find a heuristic that gives an approximate solution, for example, by trying to reduce the average number of issues.

Bibliography

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. 4th ed. The Java series. Upper Saddle River, NJ: Addison-Wesley, 2006. ISBN: 0-321-34980-6. DOI: 10.1002/9780470693698.ch1. URL: http://etf.beastweb.org/index.php/site/download/Java_Programming.pdf.

[2] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82 (2009), pp. 3–22. DOI: 10.1016/j.jss.2008.03.066.

[3] Jean Bézivin. "On the unification power of models". In: *Software & Systems Modeling* 4.2 (2005), pp. 171–188. DOI: 10.1007/s10270-005-0079-0. URL: https://doi.org/10.1007/s10270-005-0079-0.

[4] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The unified modeling language reference manual*. Vol. 2. Addison-Wesley Reading, 1999.

[5] Erik Burger. "Model Driven Software Development". Lecture Slides for the Chapter on Model Transformations. Nov. 2017.

[6] Fei Chen. "Change-Driven Consistency Preservation between UML Models and Java Code (*Änderungsgetriebene Konsistenzhaltung zwischen UML-Klassenmodellen und Java-Code*)". Bachelors's Thesis. Karlsruhe: Karlsruhe Institute of Technology (KIT), 2015.

[7] Anthony Cleve et al. "Multidirectional Transformations and Synchronisations." English. In: *Dagstuhl Reports* 8.12 (2019), pp. 1–48. ISSN: 2192-5283. DOI: 10.4230/DagRep.8.12.1.

[8] Steve Cook et al. *Unified Modeling Language (UML) Version 2.5.1*. Standard. Object Management Group (OMG).

[9] Trip Denton et al. "NAOMI – An Experimental Platform for Multi-modeling". In: *Model Driven Engineering Languages and Systems*. Ed. by Krzysztof Czarnecki et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 143–157. ISBN: 978-3-540-87875-9.

[10] Zinovy Diskin et al. "From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case". In: vol. 6981. Oct. 2011, pp. 304–318. DOI: 10.1007/978-3-642-24485-8_22.

[11] Eclipse Foundation. *Model Development Tools (MDT)*. 2020. URL: https://www.eclipse.org/modeling/mdt/ (visited on 02/20/2020).

[12] Holger Giese and Robert Wagner. "Incremental Model Synchronization with Triple Graph Grammars". In: *Model Driven Engineering Languages and Systems*. Ed. by Oscar Nierstrasz et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 543–557. ISBN: 978-3-540-45773-2.

[13] James Gosling and Henry McGilton. "The Java Language Environment". In: *Sun Microsystems Computer Company* 2550 (1995). URL: http://www.oracle.com/technetwork/java/langenv-140151.html.

[14] James Gosling et al. *The Java Language Specification*. Vol. 8. Addison-Wesley Professional, 2015. URL: http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[15] Object Management Group. Version 2.5.1. Oct. 2016. URL: https://www.omg.org/spec/MOF.

[16] Florian Heidenreich. *Jamopp: The java model parser and printer*. Techn. Univ., Fakultät Informatik, 2009.

[17] Florian Heidenreich et al. "Closing the Gap between Modelling and Java". In: *Software Language Engineering*. Ed. by Mark van den Brand, Dragan Gašević, and Jeff Gray. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 374–383.

[18] Zhenjiang Hu et al. "Dagstuhl seminar on bidirectional transformations (BX)". In: *ACM SIGMOD Record* 40.1 (2011), pp. 35–39.

[19] Heiko Klare. "Designing a Change-Driven Language for Model Consistency Repair Routines". Master's Thesis. Karlsruhe: Karlsruhe Institute of Technology (KIT), 2016. DOI: 10.5445/IR/1000080138. URL: http://dx.doi.org/10.5445/IR/1000080138.

[20] Heiko Klare. "Multi-model Consistency Preservation". In: *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS 2018*. Copenhagen, Denmark, Oct. 2018, pp. 156–161. DOI: 10.1145/3270112.3275335.

[21] 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: 978-1-7281-5125-0. DOI: 10.1109/MODELS-C.2019.00058. URL: http://dx.doi.org/10.1109/MODELS-C.2019.00058.

[22] Heiko Klare et al. "A Categorization of Interoperability Issues in Networks of Transformations". In: *Journal of Object Technology* 18.3 (July 2019). Ed. by Anthony Anjorin and Regina Hebig. The 12th International Conference on Model Transformations, 4:1–20. ISSN: 1660-1769. DOI: 10.5381/jot.2019.18.3.a4. URL: http://www.jot.fm/contents/issue_2019_03/article4.html.

[23] Anneke G Kleppe et al. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[24] Max E. Kramer, Erik Burger, and Michael Langhammer. “View-Centric Engineering with Synchronized Heterogeneous Models”. In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. VAO ’13. Montpellier, France: ACM, 2013, 5:1–5:6. ISBN: 978-1-4503-2070-2. DOI: 10.1145/2489861.2489864. URL: http://sdqweb.ipd.kit.edu/publications/pdfs/kramer2013b.pdf.

[25] K. Lano et al. “Correct-by-construction synthesis of model transformations using transformation patterns”. In: *Software & Systems Modeling* 13.2 (May 2014), pp. 873–907. ISSN: 1619-1374. DOI: 10.1007/s10270-012-0291-7. URL: https://doi.org/10.1007/s10270-012-0291-7.

[26] Nuno Macedo, Alcino Cunha, and Hugo Pacheco. “Towards a Framework for Multi-directional Model Transformations”. In: vol. 1133. Mar. 2014.

[27] Nuno Macedo, Tiago Jorge, and Alcino Cunha. “A feature-based classification of model repair approaches”. In: *IEEE Transactions on Software Engineering* 43.7 (2017), pp. 615–640. DOI: 10.1109/tse.2016.2620145.

[28] Christopher Meier Johannes and Werner et al. “Classifying Approaches for Constructing Single Underlying Models”. In: *Model-Driven Engineering and Software Development*. Ed. by Slimane Hammoudi, Luís Ferreira Pires, and Bran Selić. Cham: Springer International Publishing, 2020, pp. 350–375. ISBN: 978-3-030-37873-8. DOI: 10.1007/978-3-030-37873-8_15. URL: http://dx.doi.org/10.1007/978-3-030-37873-8_15.

[29] Johannes Meier et al. “Single Underlying Models for Projectional, Multi-View Environments”. In: *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*. INSTICC. SCiTePress, 2019, pp. 119–130. ISBN: 978-989-758-358-2. DOI: 10.5220/0007396401190130.

[30] David Lorge Parnas. “Software aging”. In: *Proceedings of 16th International Conference on Software Engineering*. IEEE. 1994, pp. 279–287.

[31] Aurélien Pepin. “Decomposition of Relations for Multi-model Consistency Preservation”. Master’s Thesis. Karlsruhe: Karlsruhe Institute of Technology (KIT), 2019. DOI: 10.5445/IR/1000100374. URL: http://dx.doi.org/10.5445/IR/1000100374.

[32] Ralf H. Reussner et al. *Modeling and Simulating Software Architectures – The Palladio Approach*. Cambridge, MA: MIT Press, Oct. 2016. 408 pp. ISBN: 9780262034760. URL: http://mitpress.mit.edu/books/modeling-and-simulating-software-architectures.

[33] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN: 0321245628.

[34] Timur Saglam. “Automatic Integration of Ecore Functionality into Java Code”. Bachelor’s Thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT), 2017, p. 76. DOI: 10.5445/IR/1000070341. URL: http://nbn-resolving.org/urn:nbn:de:swb:90-703419.

[35] Jörg Schäuffele. *E/E Architectural Design and Optimization using PREEvision*. Tech. rep. SAE Technical Paper, Apr. 2016. DOI: 10.4271/2016-01-0016.

[36] Andy Schürr. "Specification of graph translators with triple graph grammars". In: *Graph-Theoretic Concepts in Computer Science*. Ed. by Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 151–163. ISBN: 978-3-540-49183-5.

[37] Andy Schürr and Felix Klar. "15 Years of Triple Graph Grammars". In: *Graph Transformations*. Ed. by Hartmut Ehrig et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 411–425. ISBN: 978-3-540-87405-8.

[38] S. Sendall and W. Kozaczynski. "Model transformation: the heart and soul of model-driven software development". In: *IEEE Software* 20.5 (Sept. 2003), pp. 42–45. ISSN: 1937-4194. DOI: 10.1109/MS.2003.1231150.

[39] Herbert Stachowiak. *Allgemeine Modelltheorie*. 1973.

[40] Thomas Stahl and Markus Völter. *Model Driven Software Development: Technology, Engineering, Management*. Chichester: Wiley, 2006. ISBN: 0-470-02570-0.

[41] Perdita Stevens. "A Landscape of Bidirectional Model Transformations". In: *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*. Ed. by Ralf Lämmel, Joost Visser, and João Saraiva. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 408–424. ISBN: 978-3-540-88643-3. DOI: 10.1007/978-3-540-88643-3_10. URL: https://doi.org/10.1007/978-3-540-88643-3_10.

[42] Perdita Stevens. "Bidirectional model transformations in QVT: semantic issues and open questions". In: *Software & Systems Modeling* 9.1 (2010), p. 7.

[43] Perdita Stevens. "Bidirectional transformations in the large". In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE. 2017, pp. 1–11. DOI: 10.1109/models.2017.8.

[44] Perdita Stevens. "Maintaining consistency in networks of models: bidirectional transformations in the large". In: *Software and Systems Modeling* 19.1 (2020), pp. 39–65. DOI: 10.1007/s10270-019-00736-x. URL: https://doi.org/10.1007/s10270-019-00736-x.

[45] Perdita Stevens. "Towards an Algebraic Theory of Bidirectional Transformations". In: *Graph Transformations*. Ed. by Hartmut Ehrig et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–17. ISBN: 978-3-540-87405-8.

[46] Torsten Syma. "Multi-model Consistency through Transitive Combination of Binary Transformations". Master's Thesis. Karlsruhe: Karlsruhe Institute of Technology (KIT), 2018. DOI: 10.5445/IR/1000104128. URL: http://dx.doi.org/10.5445/IR/1000104128.

[47] Frank Trollmann and Sahin Albayrak. "Extending Model Synchronization Results from Triple Graph Grammars to Multiple Models". In: *Theory and Practice of Model Transformations*. Ed. by Pieter Van Gorp and Gregor Engels. Cham: Springer International Publishing, 2016, pp. 91–106. ISBN: 978-3-319-42064-6.

A. Appendix

PCM Element	UML Element	Java Element
Repository	Package	Package
System	Package	Package
System	Class	Class
System	Class	CompilationUnit
RepositoryComponent	Package	Package
RepositoryComponent	Class	Class
RepositoryComponent	Class	Compilation Unit
RepositoryComponent	Operation	Constructor
OperationInterface	Interface	Interface
OperationInterface	Interface	Compilation Unit
PrimitveDataType	PrimitiveDataType	PrimitiveType
CompositeDataType	Class	Class
CompositeDataType	Class	Compilation Unit
CompositeDataType	Generalization	TypeReference
CollectionDataType	Parameter	OrdinaryParameter
InnerDeclaration	Property	Field
OperationSignature	Operation	InterfaceMethod
OperationSignature	Parameter (Return)	Method
Parameter	Parameter	OrdinaryParameter
RequiredRole	Property	Field
RequiredRole	Parameter	OrdinaryParameter
ProvidedRole	Realization	ConcreteClassifier
AssemblyContext	Property	Field
AssemblyContext	Operation	Constructor
AssemblyContext	-	ClassifierImport
AssemblyContext	-	NewConstructorCall
-	Enumeration	Enumeration
-	Enumeration	Compilation Unit
-	EnumerationLiteral	EnumerationConstant
-	Operation	ClassMethod

Table A.1.: Incomplete list of the element mappings that describe the three transformations $T_{PCM \leftrightarrow UML}$, $T_{UML \leftrightarrow Java}$, and $T_{PCM \leftrightarrow Java}$. Each row represents an implicit n-ary element mapping formed by the transformations. Note that this is just an overview, for the detailed transformations rules we refer to [46, 6].