Multi-model Consistency through Transitive Combination of Binary Transformations

Master's Thesis of

Torsten Syma

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

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

1. June 2018 – 30. November 2018

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, 30. November 2018

. . . . . .

(Torsten Syma)

Abstract

Software systems are usually described through multiple models that address different development concerns. These models can contain shared information, which leads to redundant representations of the same information and dependencies between the models. These representations of shared information have to be kept consistent, for the system description to be correct. The evolution of one model can cause inconsistencies with regards to other models for the same system. Therefore, some mechanism of consistency restoration has to be applied after changes occurred. Manual consistency restoration is error-prone and time-consuming, which is why automated consistency restoration is necessary. Many existing approaches use binary transformations to restore consistency for a pair of models, but systems are generally described through more than two models. To achieve *multi-model consistency preservation* with binary transformations, they have to be combined through transitive execution.

In this thesis, we explore transitive combination of binary transformations and we study what the resulting problems are. We develop a catalog of *six failure potentials* that can manifest in failures with regards to consistency between the models. The knowledge about these failure potentials can inform a transformation developer about possible problems arising from the combination of transformations. One failure potential is a consequence of the transformation network topology and the used domain models. It can only be avoided through topology adaptations. Another failure potential emerges, when two transformations try to enforce conflicting consistency constraints. This can only be repaired through adaptation of the original consistency constraints. Both failure potentials are case-specific and cannot be solved without knowing which transformations will be combined. Furthermore, we develop two transformation implementation patterns to mitigate two other failure potentials. These patterns can be applied by the transformation developer to an individual transformation definition, independent of the combination scenario. For the remaining two failure potentials, no general solution was found yet and further research is necessary.

We evaluate the findings with a case study that involves two independently developed transformations between a component-based software architecture model, a UML class diagram and its Java implementation. All failures revealed by the evaluation could be classified with the identified failure potentials, which gives an initial indicator for the completeness of our failure potential catalog. The proposed patterns prevented all failures of their targeted failure potential, which made up 70% of all observed failures, and shows that the developed implementation patterns are applicable and help to mitigate issues occurring from transitively combining binary transformations.

Zusammenfassung

Softwaresysteme werden häufig durch eine Vielzahl an Modellen beschrieben, von denen jedes unterschiedliche Systemeigenschaften abbildet. Diese Modelle können geteilte Informationen enthalten, was zu redundanten Beschreibungen und Abhängigkeiten zwischen den Modellen führt. Damit die Systembeschreibung korrekt ist, müssen alle geteilten Informationen zueinander konsistent beschrieben sein. Die Weiterentwicklung eines Modells kann zu Inkonsistenzen mit anderen Modellen des gleichen Systems führen. Deshalb ist es wichtig ist einen Mechanismus zur Konsistenzwiederherstellung anzuwenden, nachdem Änderungen erfolgt sind. Manuelle Konsistenzwiederherstellung ist fehleranfällig und zeitaufwändig, weshalb eine automatisierter Konsistenzwiederherstellung notwendig ist. Viele existierende Ansätze nutzen binäre Transformationen, um Konsistenz zwischen zwei Modellen wiederherzustellen, jedoch werden Systeme im Allgemeinen durch mehr als zwei Modelle beschrieben. Um *Konsistenzerhaltung für mehrere Modelle* mit binären Transformationen zu erreichen, müssen diese durch transitive Ausführung kombiniert werden.

In dieser Masterarbeit untersuchen wir die transitive Kombination von binären Transformationen und welche Probleme mit ihr einhergehen. Wir entwickeln einen Katalog aus sechs Fehlerpotentialen, die zu Konsistenzfehlern führen können. Das Wissen über diese Fehlerpotentiale kann den Transformationsentwickler über mögliche Probleme beim Kombinieren von Transformationen informieren. Eines der Fehlerpotentiale entsteht als Folge der Topologie des Transformationsnetzwerks und der benutzten Modelltypen, und kann nur durch Topologieänderungen vermieden werden. Ein weiteres Fehlerpotential entsteht, wenn die kombinierten Transformationen versuchen zueinander widersprüchliche Konsistenzregeln zu erfüllen. Dies kann nur durch Anpassung der Konsistenzregeln behoben werden. Beide Fehlerpotentiale sind fallabhängig und können nicht behoben werden, ohne zu wissen, welche Transformationen kombiniert werden. Zusätzlich wurden zwei *Implementierungsmuster* entworfen, um zwei weitere Fehlerpotentiale zu verhindern. Sie können auf die einzelnen Transformationsdefinitionen angewendet werden, unabhängig davon welche Transformationen letztendlich kombiniert werden. Für die zwei übrigen Fehlerpotentiale wurden noch keine generellen Lösungen gefunden.

Wir evaluieren die Ergebnisse mit einer Fallstudie, bestehend aus zwei voneinander unabhängig entwickelten binären Transformationen zwischen einem komponentenbasierten Softwarearchitekturmodell, einem UML Klassendiagramm und der dazugehörigen Java-Implementierung. Alle gefundenen Fehler konnten einem der Fehlerpotentiale zugewiesen werden, was auf die Vollständigkeit des Fehlerkatalogs hindeutet. Die entwickelten Implementierungsmuster konnten alle Fehler beheben, die dem Fehlerpotential zugeordnet wurden, für das sie entworfen wurden, was 70% aller gefundenen Fehler ausgemacht hat. Dies zeigt, dass die Implementierungsmuster tatsächlich anwendbar sind und Fehler verhindern können.

Contents

List of Figures

List of Tables

1 Introduction

Formalized representations of information are used by many disciplines and for different reasons. In the development of software systems, for example, many artifacts are produced that describe different properties of the developed system, which are important for a specific stake holder. Each artifact can be understood as a model of the system. This includes, for example, the system's requirements, design documentation, implementation and system test descriptions. Because these artifacts describe properties of the same system, some information is shared and duplicated between them. Such information has to be consistently evolved for all artifacts that contain it.

In many cases, manual consistency preservation is used to ensure that all artifacts are consistent. However, manual consistency preservation is error-prone and time-consuming. For one, the person responsible for updating the models might themselves receive little immediate benefit from doing so, therefore, the motivation is low and the task is delayed or might be forgotten. Secondly and more importantly, it is not always clear which documents have to be adapted, because the tracing information is not stored explicitly. Instead, dependencies between elements are often only implicitly documented through similar naming conventions. On top of causing additional effort to find all dependencies in case of a change, this kind of implicit tracing can easily break down if a simple name change is not properly propagated.

In the model-driven software development context, the problems of manual consistency preservation are mitigated by using model transformations to automate the process and by using explicit tracing information.

1.1 Consistency Preservation through Model Transformations

Model transformations can be used for many purposes, such as system analysis, code generation and consistency preservation. Derived models, like a Java class implementation that is derived from its UML design, generally need to be extended with additional information. Otherwise the original design model would have sufficed. Simply generating a new variant of the derived model, in case of design changes, would overwrite the additional information in the derived model. It is therefore necessary to use an *incremental* transformation to avoid losing information. Additionally, if editing the generated model is allowed, then we may need to propagate the changes back to the design document to keep both models consistent. Therefore, the transformation also needs to be *bidirectional*.

Software systems are generally concerned with more than two models, which implies the necessity for multiary transformations or a combination of multiple binary transformations. One solution to the problems caused by redundancies between multiple models is to avoid them by unifying all information regarding the software system and its development

process into a redundancy-free single underlying model (SUM) [1]. However, in practice this is hardly achievable, because of the complexity of combining multiple heterogeneous models. Even if such a model is found, it would most likely be impractical to use in real world scenarios.

By contrast, the Vitruvius project [24, 10] encapsulates the multiple models needed to develop one system in a virtual SUM (VSUM). The VSUM hides the fact that it actually consists of multiple individual models by only allowing the user to edit the models through well-defined views. Internally, the models are kept consistent through explicit consistency preservation mechanisms that automate their synchronization instead of preventing redundancy altogether. For this purpose, the Vitruvius project provides transformation languages to define consistency-preserving transformations between the involved domains.

Instead of defining additional models or multiple binary transformations, a multiary transformation could directly preserve the consistency of all involved models. For example, the Query View Transformation standard [19] provides the declarative transformation language QVT-R, which promises real multi-model transformation capabilities. However, Macedo et. al.[14] showed that the QVT-R standard is still underspecified and proposed extensions to solve these shortcomings. Additionally, the more models are involved, the more complex the transformation development becomes.

Multiary transformations require the transformation developer to knows all involved models in detail. However, in practice a developer only knows a specific set of models, which are relevant to him, in sufficient detail to be able to formalize all dependencies between them. Therefore, a combination of partial specifications is necessary. Binary transformations are the smallest building blocks for a transformation network and we therefore focus on binary transformations.

1.2 Transitive Transformation Combination

In a strictly directed generative transformation pipeline, each transformation takes one input model and outputs a new model, which functions as the input for the next transformation. With regards to the first input and the last output model, all transformation steps in between function as one *combined transformation*. As already stated, we need bidirectional, incremental transformations for consistency preservation. Because of the incrementality, the pipeline then propagates changes instead of whole models. An abstract example of this is depicted in Figure 1.1. We call this process *transitive change propagation*.

In general, the transformation network topology does not have to be linear. Without proper care of the network composer, the combined transformations can even try to enforce conflicting notions of consistency. This complicates the consistency preservation, because the propagation process can produce conflicting changes, potentially leading to information loss or even non-terminating propagation loops. Figure 1.2 shows a concrete example, where a software component of a Palladio Component Model (PCM)[2, 21], the UML design of its implementation, and the actual Java implementation need to be kept consistent. In this small example, multiple problems can manifest depending on the used transformations and consistency definitions.

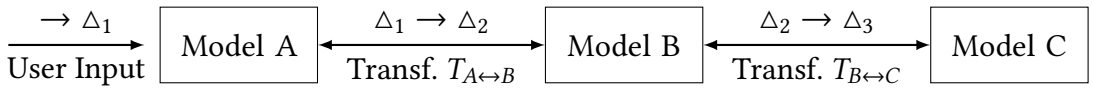


Figure 1.1: Example for transitive change propagation. The user input $\triangle_1$ is transformed and propagated through the transformation network, which ideally restores consistency among the three models.

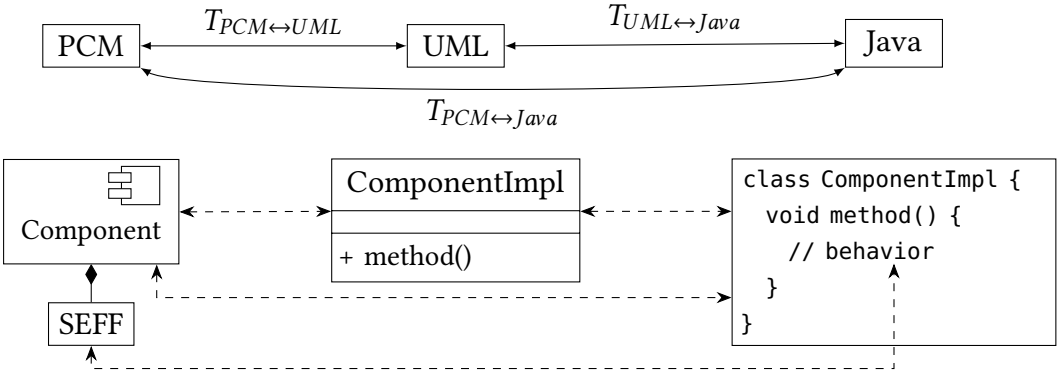


Figure 1.2: A concrete example used to demonstrate transformation combination problems. The dashed arrows represent the dependencies between the elements. PCM refers to the Palladio Component Model. A Service Effect Specification (SEFF) describes the component behavior for a provided service.

- Without the transformation $T_{PCM \leftrightarrow Java}$ the component behavior, described by the Service Effect Specification (SEFF), cannot be synchronized with the method implementation in the component's java implementation class.

- With the transformation $T_{PCM \leftrightarrow Java}$ as part of the transformation network, the network now contains a cycle. On creation of the PCM component, there are two transformation paths that can produce the Java implementation class, potentially leading to a duplication.

- If the implementation actually is duplicated, for example by $T_{PCM \leftrightarrow Java}$, this change can then propagate back to PCM via $T_{UML \leftrightarrow Java}$ and $T_{PCM \leftrightarrow UML}$, creating a duplicate PCM component and ultimately resulting in a loop.

- If the implementation is not duplicated, the transformations may still try to enforce different consistency definitions. Assume for example that $T_{PCM \leftrightarrow Java}$ requires the Java implementation class to share the same name as the PCM component and that $T_{PCM \leftrightarrow UML}$ requires the implementation name to end with the suffix "-Impl". These constraints obviously contradict each other, but are otherwise reasonable definitions.

1.3 Research Goals

In this thesis, we explore the problems of transformation combination through transitive change propagation for the purpose of multi-model consistency preservation. We answer the following research questions:

Q1 How does the application of transitive change propagation for a single bidirectional transformation result in failure?

Q2 What problems emerge from the transitive combination of model transformations, that do not already occur in binary transformations?

Q3 How can specific problems be solved in case-agnostic way?

First, we decompose the definition of consistency on a model-level into sets of consistency constraints that define consistency for the semantic overlap between two model elements, which we call *correspondence types*. Furthermore, we define a *correspondence graph* as a network of elements that are linked by their semantic dependencies, which are described through correspondence types. Such correspondence graphs allow us to observe how changes may propagate through a transformation network based on element-level propagation paths, rather than low-resolution model-level propagation paths. The benefit of these element-level change propagation paths is that they reveal the interaction between the general change propagation process and the case specific consistency constraints in more detail.

We then select sets of properties of the transformation process and explore how they can produce failures, with regards to multi-model consistency, using minimal artificial correspondence graph examples and different change sequences. Based on the observed failures, we develop a catalog of failure potentials to answer questions *Q1* and *Q2*. The knowledge about these failure potentials can then inform a transformation developer about possible problems arising from the combination of transformations.

Furthermore, we develop transformation implementation patterns that mitigate specific failure potentials, but are independent of the combination scenario, as an answer to questions *Q3*. These patterns can then be applied by the transformation developer to an individual transformation definition, which increases the chance that the individual transformation can be successfully combined with others.

The findings are evaluated using a case study that combines the transformations $T_{\text{PCM} \leftrightarrow \text{UML}}$ between the Palladio Component Model and UML class diagrams, and $T_{\text{UML} \leftrightarrow \text{Java}}$ between UML class diagrams and Java. The transformation $T_{\text{PCM} \leftrightarrow \text{UML}}$ was developed for this thesis, whereas the transformation $T_{\text{UML} \leftrightarrow \text{Java}}$ was independently developed by Chen [3].

1.4 Thesis Structure

First, in Chapter 2, we introduce the foundations regarding model transformations and transformation properties, as well as the frameworks and domains that are used in the case study and some of the examples throughout the contribution chapters.

In Chapter 3, we first develop the concept of a correspondence graph and explore how failures can occur as a result of transformation combination. Based on these failures, we then develop the failure potentials catalog to answer questions *Q1* and *Q2*. In Chapter 4, we address question *Q3* and we develop two transformation implementation patterns to prevent element-creation conflicts and to prevent the propagation of deprecated information.

Chapter 5 provides detailed descriptions of the consistency constraints that are implemented by the two transformations $T_{\text{PCM} \leftrightarrow \text{UML}}$ and $T_{\text{UML} \leftrightarrow \text{Java}}$. In Chapter 6, we then combine said transformations and apply them to a realistic software architecture model scenario in order to evaluate the relevance of the identified failure potentials and to clarify the generalizability of the proposed patterns.

Lastly, Chapter 7 gives an overview of related work in the fields of model consistency preservation and transformation combination and we conclude this thesis in Chapter 8 by restating the contributions and evaluation results and by suggesting topics for future work.

2 Foundations

In this chapter, we introduce the foundations necessary for the rest of this thesis. First, we introduce model-driven software development, meta-modeling and consistency between models. Then we introduce model transformations and relevant transformation properties. Lastly, we introduce the frameworks and domain models, which we use in the evaluation case study and in some of the examples throughout the contribution chapters.

2.1 Model-driven Software Developement

Model-driven Software Development (MDSD) is a software development technique that aims to improve productivity and increasing code quality, by treating models as essential artifacts of the development process. Domain specific models and domain specific languages (DSL) abstract from the underlying general purpose models or languages, by encoding domain specific reoccurring complexity and redundancy behind higher-level concepts. This complexity is then re-injected into lower-level models, like program code, by transformations that know how to translate the higher-level concept into lower-level concepts or instructions, without the model developer having to know the implementation details. This frees mental capacity and increases productivity by avoiding reoccurring tasks. It also improves the solution quality, because it forces the application and evolution of a single solution pattern for a reoccurring problem, rather than co-evolving multiple case specific variations that have to be kept in sync. A transformation could also perform complex optimizations in the process, further increasing the quality of the solution.

In order to make models more accessible to machine processing, they have to be formally defined. This is achieved through metamodeling, defining higher-abstraction models that themselves define how lower-abstraction models are defined.

2.2 Meta-modeling

A model is an abstraction of a set of objects, often a compositum and its parts, and their relations with regards to a specific purpose. According to Stachowiak [25], a model can be characterized by three qualities. It reproduces some properties of an original object or system (mapping) while omitting others (reduction). Which properties are mapped and which are reduced, is decided based on the purpose for which the model is constructed (pragmatism). A common example for a model in software development is a UML class diagram. It represents the structure of the software system's implementation, while abstracting the implementation details in order to allow design on a higher abstraction level and to serve as a documentation. This example also shows that the original, in this

case the program code, does not have to exist before the model is created. Instead the model can be a blueprint for the original and prescribe some of the properties the original should possess. And the original can itself be a model of some other object or system. In the example, the code is a model of the program behavior while abstracting from some of the hardware and platform specifics.

A metamodel is a special type of model that formally defines how its instances have to be structured. This formality makes the instances of a metamodel better accessible to machine processing and metamodels are therefore often a focus in MDSD. According to Stahl [26] a metamodel is defined by four artifacts: abstract syntax, concrete syntax, static semantic and dynamic semantic. The *abstract syntax* defines concepts that can be instantiated in the model instances, for example, objects and relations between objects are commonly modeled concepts. The *concrete syntax* defines a formalism how the concept instances of a model are expressed, either textually (e.g. programming languages) or visually (e.g. UML class diagrams). While a metamodel has exactly one abstract syntax, there can exist multiple equivalent concrete syntaxes for different purposes. For example, the visual representation of a UML diagram is far easier for a human to comprehend and to evolve than an encoding of the same model as a XML document, which is easier for a program to process. The *static semantic* of a metamodel defines additional constraints on the well-formedness of a model instance. These constraints have to be statically analyzable, without having to execute the model (if the model is executable). And lastly, the *dynamic semantic* defines the meaning of the modeled elements. In the following, we will consider a model to be correct if it conforms to its abstract syntax and static semantic.

Just like a model is an instance of a metamodel, a metamodel can itself be an instance of another meta-metamodel, which prescribes rules and constraints on the elements of the metamodel. This definition chain can be continued, until a self-describing metamodel is found. Each level in the hierarchy is called a meta-level or meta-layer, with the real object on the lowest layer, sometimes called M0.

2.3 Consistency

A software system can be developed using more than one model and because these models describe different properties of the same system, they often represent some information common to both models. These semantic overlaps should be kept in sync to prevent errors in the development process. This is especially important in MDSD, because the conflicting specifications could cause problems for the code generator. Nuseibeh [17] defines inconsistency as “any situation, in which two descriptions do not obey some relationship that is prescribed to hold between them.” We will call these prescribed relationships *consistency constraints*. For practical purposes, we will define *consistency* as the inverse of inconsistency and claim a set of models is consistent, if all their consistency constraints are satisfied, even though such a definition is probably false, because it is unlikely that a list of consistency constraints is truly complete. It is important to note that a set of models does not have to be consistent (global consistency), even if every possible subset of models is consistent when looked at in isolation (local consistency), as has been proven by Nuseibeh in [16], Appendix A.

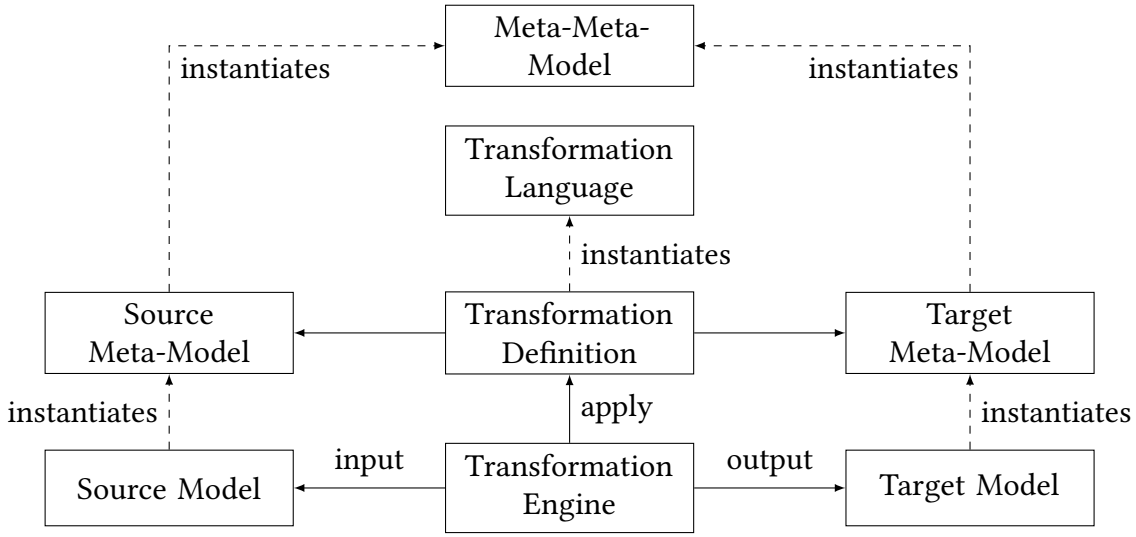


Figure 2.1: Basic model transformation framework example

Model transformations can be used to ensure model consistency, by propagating changes in one model to all related models. In this context, it is possible to define consistency constraints explicitly through a declarative DSL that generates the necessary transformations, like the QVT-R standard [19] or the Commonalities Language [6]. Alternatively, the consistency constraints can be defined implicitly through the effects of an imperative transformation.

2.4 Model Transformations

A model transformation describes how one or more target models can be generated from one or more source models. It is usually defined on the metamodel level and applied to concrete models. Important to note is that transformations do not have to be defined between different metamodels. For example, a name refactoring on a Java class would be a transformation from Java code to Java code.

Mens [15] separated the following terms. A *transformation rule* defines how one or more concepts of the source domain can be transformed into one or more concepts of the target domain. A *transformation definition* is a set of transformation rules. A *transformation* is the application of a transformation definition to a concrete source model.

The execution of a transformation is performed by a *transformation engine* by application of the transformation rules, which are provided by the transformation definition, to the provided source model. Figure 2.1 depicts how models, transformation engine and transformation definition interact in a model transformation framework. Depending on the intended properties of the transformations, the transformation engine can take multiple models, change deltas or additional information like trace models as a transformation's input.

The following subsections describe transformation properties and possible transformation network topologies.

2.4.1 Transformation Properties

In this section we describe the arity, directionality and the incrementality as possible properties of a transformation.

The *arity* of a transformation describes the number of models that transformation operates on. The most common differentiation is made between binary transformations and mutiary (more than two models) transformations, because binary transformations are the minimal example.

A transformation definition has a *directionality* that represents which domains are the input and which are the output. A transformation definition can be *bidirectional*, which indicates that it defines mapping rules for both directions, from the source to the target model and the other way around. Two opposite directed transformations can be similar to a bidirectional transformation when they are combined, but they will always need at least two transformation steps to change both model sets, whereas a bidirectional transformation can change both the source and target model sets in a single step.

A transformation can have varying degrees of *incrementality*, based on how much information it investigates in the source model (source-incremental) and how much of the target model is changed (target-incremental). A transformation that is not incremental takes the complete source model and generates a completely new target model from it. A source-incremental transformation looks only at parts of the source model, usually the parts that have changed, which necessitates some facility to track changes. A target-incremental transformation only changes limited parts of the target model, which necessitates some facility to store the trace information in order to determine which target object was generated from which source object. Both forms of incrementality usually go hand in hand to some degree. Incrementality is especially important in cases where both models are subject to manual changes and/or express different semantics. Consider for example, a code generator that produces Java code from a UML class diagram. A batch transformation would generate the whole Java class anew, discarding potential changes done by a developer, even if only the name of a method changed. In contrast, a highly incremental transformation might only generate one new method or rename the old one, leaving the manual implementation intact.

2.4.2 Transformation Topologies

Transformations can be combined to form transformation networks. Every metamodel represents a node and transformations represent edges that connect these nodes. This is in theory true for transformations with any arity, however, commonly only binary transformations are combined. And even multiary transformation descriptions, defined with specific DSLs like the Commonalities Language [6], might be internally implemented as sets of binary transformation. We differentiate three types of transformation networks topologies that are build from binary transformations.

A *star topology*, with one central metamodel and all other metamodels as leaves, results in the minimal number of transformations necessary. However, every change needs to be propagated across a minimum of two edges, through the central metamodel. This means that the central metamodel has to be capable of expressing every concept of all other

models. Otherwise, some concepts cannot be synchronized between the leaves of the network.

A *tree topology* can reduce the design effort, by grouping metamodels with a high level of semantic overlap under a common inner parent node. That way some of the concepts might not have to pass through the root metamodel. But this still does not guarantee that all semantic overlaps can be synchronized and there are now multiple pass-through metamodels. Therefore changes potentially need to be propagated across a longer transformation path, which leads to an increased risk of encountering incompatibilities between the individual transformations. If a network has a tree topology, where each node has exactly one child and one parent, we will call this a linear network.

The third option is a *mesh topology* that does not pose any particular topology constraints aside from connectedness. As a result, network cycles are allowed. In its extreme form, a mesh topology reflects a fully connected network. This type of topology can ensure that all concepts can be synchronized by introducing direct transformations between all models with dependent information. However, these additional transformations introduce cycles into the network, which introduce ambiguities in transformation execution order and the potential for non-terminating transformation loops.

2.5 Frameworks and Case Study Domains

In this section, we introduce the frameworks and domain models, which we used in the evaluation case study and in some of the examples throughout the contribution chapters.

2.5.1 Meta Object Facility and Eclipse Modeling Framework

The Meta Object Facility (MOF)[18] is a self describing meta-metamodel standard. The Essential Meta Object Facility (EMOF) is a reduced variant of the MOF, containing the meta-metamodel elements necessary to describe object-oriented metamodels. We use the EMOF formalism throughout this thesis as basis for model or metamodel discussions.

The Eclipse Modeling Framework (EMF)[30] is an extensions to the integrated development environment Eclipse, adding support for model-driven development. The `ecore` meta-metamodel is a concrete implementation of the EMOF standard and is provided by the EMF. It functions as the meta-metamodel for all domain metamodels used in the case study and takes the role of meta-metamodel in Figure 2.1.

2.5.2 Vitruvius Project

The Vitruvius project [24, 10] provides a framework for a model-driven development approach and builds on the Orthographic Modeling approach by Atkinson [1]. Vitruvius encapsulates the multiple models needed for the system design in a Virtual Single Underlying Model (VSUM). The VSUM hides the fact that it actually consists of multiple individual models by only allowing the user to edit the models through well defined views. Internally the models are kept consistent through explicit consistency preservation mechanisms that automate their synchronization. For this purpose, the Vitruvius project provides

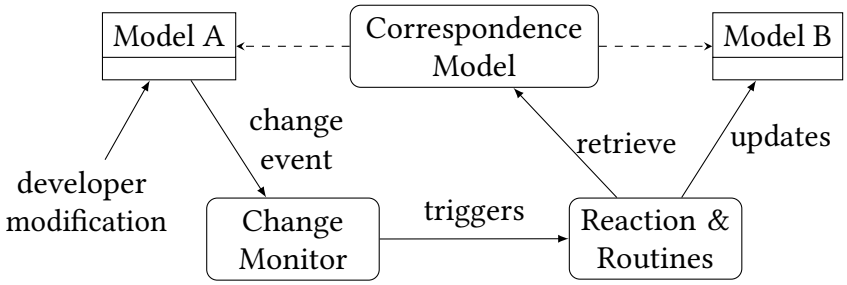


Figure 2.2: Simplified representation of the Vitruvius consistency preservation framework for the Reactions language.

transformation languages to define consistency-preserving transformations between the involved domains.

One of the provided transformation languages is the *Reactions language* [23, 9], a DSL for manually defining incremental model transformations based on atomic changes that occur in the participating models. It is of particular interest for this thesis, because the transformations used in the case study are implemented using the Reactions language.

2.5.3 Reactions Language

The Reactions language [23, 9] is an imperative transformation language provided by the Vitruvius project. It is used to define binary, incremental and change-driven transformations. A transformation definition is composed of multiple rules, so called reactions, that call consistency-restoring routines to ensure consistency between two models. The reactions are triggered and executed in response to changes of the source model. The combination of two opposite facing transformations, implicitly defines a set of consistency constraints between both models.

Figure 2.2 shows how these reactions fit into the Vitruvius consistency preservation framework. The framework monitors both models in order to detect changes performed by the system developer. When a change is detected, it is tracked and, if necessary, decomposed into atomic change operations. The framework then checks the defined reactions to find those that trigger for the observed change and executes them, thereby restoring consistency with the second model. In order to find the correct elements to modify in the target model, the reactions access the *correspondence model* of the Vitruvius framework and add or retrieve the tracing information necessary. For that purpose, binary tuples, called correspondences, are stored in the correspondence model. Correspondences in the Vitruvius framework do not have inherent semantic and they have to be explicitly generated and deleted by the transformation implementation, putting the responsibility in the hands of the transformation developer, whereas other DSLs might automatically produce object traces based on the transformation rules. However, because the correspondences serve as the trace model for a transformation and the transformation enforces a specific set of consistency constraints, the correspondences usually reflect the semantic dependencies between model elements. In the following contribution chapters, we re-define and use the word “correspondence” as an explicit and formalized relation between model

elements based on consistency constraints. But because the used transformations are implemented using the Reactions language, the formalized “correpsondence”-relation is then implemented through the Vitruvius correspondences.

2.5.4 Case Study Domains

For the evaluation case study, we use two transformations between the three domains that we describe in the following.

The Palladio Component Model (PCM)[2, 21] is used to design and analyze component-based software systems. This thesis only uses a part of the PCM model, namely the repository model. It contains data type definitions, contractual interfaces and component definitions that can then be used to compose a software system architecture. Additionally, the PCM supports the modeling of abstracted component behavior. The components defined in the repository can be used in an assembly model to compose the architecture of a software system. Furthermore, it can model the target environment (servers, computation power, network latency, etc.) and how the system components are supposed to by deployed across the servers.

The Unified Modeling Language is a standard for system design. This thesis only uses UML class diagrams and for the transformations and we rely on the UML implementation provided by the Eclipse platform (UML2)[29, 20].

Lastly, we use the Java metamodel provided by the Java Model Printer and Parser (JaMoPP)[8, 7]. The general purpose programming language Java is not based on the EMOF standard and the `ecore` implementation thereof. Therefore, our transformations based on the EMOF metamodels cannot directly edit Java implementation files. The JaMoPP provides an `ecore`-compatible metamodel of the Java language. The source files are parsed and printed, and the Java elements can then be accessed and edited as if they were just another model.

3 Failure Potentials of Transitive Change Propagation

The goal of this thesis is to explore how we can use transformation combination and transitive change propagation to achieve multi-model consistency preservation. Specifically, we focus on the combination of binary transformations, because they represent the minimal building blocks of a transformation network, yet they still pose significant problems in practical scenarios.

An incremental binary model transformations can be used to synchronize both its source and its target mode, so that they conform to the consistency constraints that define consistency between them. In the context of this thesis, **model synchronization** is the process of applying a consistency-restoring transformation to a set of models. If all models in the transformation network can be evolved, then it is important, that the transformations are target-incremental instead of generative, so that information that is unique to the target model can be preserved. Model synchronization is a form of model transformation, with the focus on inter-model consistency instead of model generation, but we often use these terms synonymously. It is different from transitive change propagation, in that transitive change propagation is the process, by which we attempt to simulate a multi-model transformation through transitive execution of binary transformations, whereas model synchronization is the application of said simulated transformation. Other works sometimes associate the term "synchronization" with concurrent model modification or immediate model updates. We do not make this distinction. In general we assume that user changes can accumulate in a single model before the synchronization process is started and the synchronization terminates before the user can directly edit any of the models in the transformation network again.

We can combine two binary transformations T_1 and T_2, if the target model of T_1 functions as the source model of T_2, to form a transitive transformation $T_2 \circ T_1$, by transitively executing one after the other. Figure 3.1 shows an exemplary combination of two binary transformations, one between PCM and UML and one between UML and Java. In doing so, we hope to extend the synchronization of information from either pair of models to now also synchronize PCM models with Java models. The combination of transformations forms a transformation network, where each domain represents a network node, and each transformation represents an edge of the network. For Figure 3.1, this results in PCM, UML, and Java representing network nodes and T_1, and T_2 are the edges between them.

Now we describe what we mean by transitive change propagation. Assume a single binary and directed transformation, and both source and target model are consistent with each other. When a user modifies the source model, then the target model may no longer by consistent to it. The delta between the original and the new source model state can be described as a change. Depending on the granularity with which changes are defined, the

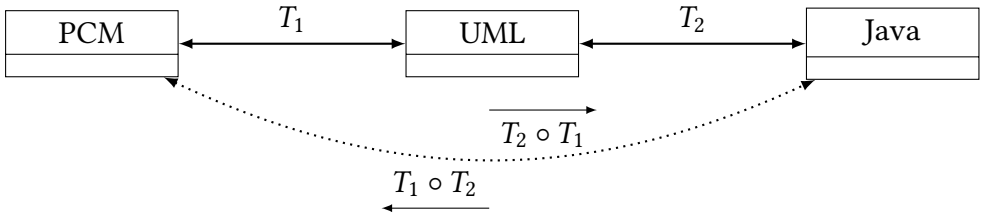


Figure 3.1: This example combines two binary transformations, T_1 between PCM and UML and T_2 between UML and Java, to realize a transitive transformation between PCM and Java.

delta can be described as a single complex change or a list of smaller changes, down to a list of atomic changes. After the change occurred, the transformation has to be executed in order to restore consistency. If the models were inconsistent before the transformation, then the transformation modifies the target model and produce a new delta, described through a new change. We call this **change propagation**.

Now we return to the example transformation network in Figure 3.1. A change to a PCM model has to be propagated to the UML model, where a new change occurs. Then the same mechanism starts again to propagate the change from the UML model to the Java model. We effectively propagate the change from PCM to Java through transitive transformation execution, and we therefore call this process **transitive change propagation**. If we want to allow the user to modify other models as well, then we need bidirectional transformations, so that changes can also be propagated from Java to PCM. In general transformation networks, the network topology may have branches and, as a result, a single change may produce multiple propagated changes. We try to achieve multi-model consistency preservation for all domains in the network through transitive change propagation.

The intention of this chapter is to identify and define failure potentials, which can manifest in inconsistency, loss of information, or transformation failure during the model synchronization process, that emerge from the interplay between transitive change propagation and the consistency constraints that are implemented by the involved transformations, even if the individual binary transformations may function as intended in the normal non-transitive execution setting. We speak of failure potentials and not of plain failure causes, because they do not always have to end in synchronization failures. Table 3.1 gives an overview of the failure potentials, which we identify in this chapter, and relate them to the factors that enable their emergence, as well as their possible failure manifestations. With the exception of incompatible consistency constraints, we assume that these potentials can be mitigated or even eliminated through combinations of

- better transformation implementation,
- more sophisticated transformation support structures (like trace models based on semantics),
- adaptations to the transformation network topology,
- transformation engine behavior (like change consolidation),

- or through deliberate user disambiguation, if none of the above prove sufficient.

But in order to think about solutions, it is necessary to identify the problems first.

In Section 3.1, we define correspondences as a possible trace model and explain their relation to model consistency. Then in Section 3.2, we develop some assumptions regarding correspondences and change propagation, which are necessary for the following sections. And from Section3.3 onward, we explore how properties of the transitive change propagation process can lead to synchronization failures, using abstract or concrete examples, and identify the emergent failure potential in the specific scenario. A short overview of possible causes is shown in Table 3.1. For all examples, we assume incremental, delta-based, binary transformations with access to a global trace model, as opposed to a transformation local trace model. We use explicitly defined consistency constraints to argue about transformation behavior, instead of providing concrete transform rule definitions and arguing about alternate implementations of implicit constraints. This allows us to stay independent of the employed transformation language. Furthermore, we assume metamodel definitions based on the EMOF formalism (Section 2.5.1).

3.1 Correspondence and Correspondence Type

First, we define the term "correspondence". Then we explain how correspondences are represented and used for argumentative reasoning about transformation rule behavior.

Model instances of different metamodels can represent similar concepts through different sets of elements and relations. For example, a UML class diagram is an abstraction of a Java implementation, or a component model (like a PCM model) can represent the composition structure of some of the implemented classes. In both cases, the model pairs share a semantic overlap that has to be synchronized after modifications in order to stay consistent. The semantic overlap of two models is composed of multiple semantic overlaps between concrete elements of those models. For example, a *uml::Interface uI* instance that abstracts from a *java::Interface jI*, shares a semantic overlap with said *jI*. The rules that have to hold between two overlapping elements can be expressed through consistency constraints. If all consistency constraints of overlapping elements are satisfied, then we consider the elements to be consistent. By extension, a pair of models is consistent if all the consistency constraints of their contained elements are satisfied. The consistency constraints have to be developed by a domain expert and their enforcement has to be implemented by the transformation developer.

We propose to define the consistency constraints between metamodel classes instead of instantiated elements, so that they can be defined independent of the concrete models. Then we can construct a set of consistency constraints for each metamodel pairing. Each set contains multiple consistency constraints that describe under which condition elements are considered to share semantic overlap (mapping constraints) and what predicates have to hold for those elements to be considered consistent (feature constraints). Examples for such constraints can be seen in Tables 3.2 and 3.3. It is important to note that a single metatmodel class can be referenced by multiple mapping constraints if the semantic interpretation of that class's instances is dependent on the context of that instance and therefore instances of said metamodel class can have different semantic overlaps. Therefore, we define a

Failure Potential	Cause	Possible Effects
Change Conflicts (Sections 3.3.1, 3.4)	postponed change application, change confluence (+ topology)	information loss, element duplication + inconsistency, propagation loop
Deprecated Change Events (Section 3.3.2)	immediate change application + multiple changes	information loss, propagation loop
Visibility of Unresolved Changes (Section 3.3.2)	immediate change application + multiple changes	information loss, inaccurate failure warnings
Unsynchronizable States (Section 3.5)	change granularity + consistency constraints	information loss, inconsistency
Concept Bottlenecks (Section 3.6)	network topology + metamodel definition (+ consistency constraints)	inconsistency between distant models
Incompatible Consistency Constraints (Section 3.7)	constraint contradictions, structurally different mappings	Anything, because then global consistency is undefined

Table 3.1: Overview of the failure potentials, which we discuss, and their more fundamental causes. Some failure potentials only manifest through the interaction of multiple factors, which is written as a "+" between the causes.

separate **correspondence type** for each possible semantic overlap and decompose the complete set of consistency constraints between a pair of metamodels into subsets, which are represented by correspondence types.

Def. **Correspondence Type**:
A correspondence type $corrT$ describes a possible semantic overlap between elements of A and B. It is defined by

- $R_{corrT} \subseteq A \times B \mid A \in M_A, B \in M_B$, a relation between two metamodel classes A, B of the metamodels M_A, M_B,
- $P_{corrT} = \{p1, .., pn\}$, a set of consistency constraints that define consistency for the tuples in R_{corrT}.

We sometimes represent a correspondence type through the classes that are involved in its definition if it is the only correspondence type between those two classes.

- $corrT \Leftrightarrow A \sim B$

By evaluating the constraints of all correspondence types for two concrete models, we can check whether the models are consistent with each other. But we ideally want to use **correspondence instances** as a trace model that tracks, which elements have to be synchronized with each other. The instantiation of a correspondence type can be required by a mapping constraint of some other correspondence in order to restore consistency. We therefore have to infer when to create correspondence, based on the mapping constraints in other correspondence type definitions.

Def. **Correspondence**:
A correspondence (instance) $corr$ of the correspondence type $corrT$ is a tuple (a, b) in R_{corrT} and it demonstrates that the elements a and b share the semantic overlap described by $corrT$.
We represented the participation of two elements a, b in $corr$ as:

- $a \sim_{corrT} b \quad (\Leftrightarrow (a, b) \in R_{corrT} \mid a \in A, b \in B)$

A correspondence type is instantiated by the consistency restoring transformation, when a mapping constraint pm of another context correspondence $c2 = (d \sim_{corrT2} e)$ requires the existence of that instance in order to achieve consistency $c2.pm(d, e) \Rightarrow a \sim_{corrT} b \mid a \in A, b \in B$.

Tables 3.2 and 3.3 show two correspondence type definitions, one between a uml::Interface and a java::Interface, and one between a uml::Operation and a java::Method. The uml::Interface~java::Interface correspondence type has a mapping constraint that requires the instantiation of the uml::Operation~java::Method correspondence type.

In later sections, we often use exemplary correspondence instances to discuss specific problem cases, without providing the full correspondence type definition or showing the full context that required the instantiation of the correspondence. Instead, we only

UJ-Interface	uI : uml::Interface ~ jI : java::Interface
feature constraints	
same name,	uI.name == jI.name
corresponding super interfaces	uG.general ~$_{\text{UJ-Interface}}$ jT.classifier (bijective), with uG ∈ (uI.generalizations), jT ∈ (jI.extends)
mapping constraints	
method mapping	uM ~$_{\text{UJ-Method}}$ jM (bijective), with uM ∈ (uml::Operation ∩ uI.ownedOperations), jM ∈ (java::Method ∩ jI.members)

Table 3.2: Exemplary correspondence type definition for the semantic overlap between a uml::Interface and a java::Interface.

UJ-Method	uM : uml::Operation ~ jM : java::Method
feature constraints	
same name,	uM.name == jM.name
mapping constraints	
ordinary parameter	uP ~$_{\text{UJ-In-Parameter}}$ jP (bijective), with uP ∈ (uM.ownedParameters.filter(param.direction == IN), jP ∈ (jM.parameters)
return parameter	uM.returnParameter ~$_{\text{UJ-Return-Parameter}}$ jM

Table 3.3: Exemplary correspondence type definition for the semantic overlap between a uml::Operation and a java::Method.

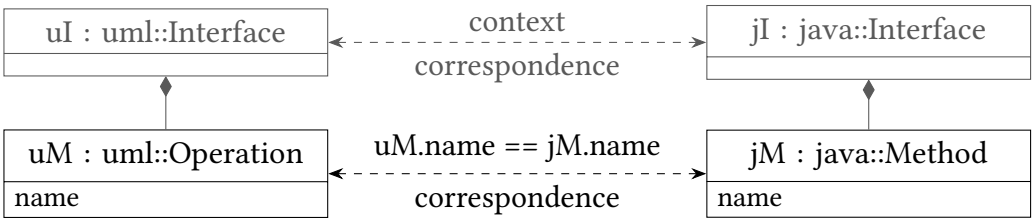


Figure 3.2: Correspondence instantiation example for a *uml::Operation* and a *java::Method*, as defined in Table 3.3. The correspondence type of the context is defined in Table 3.2.

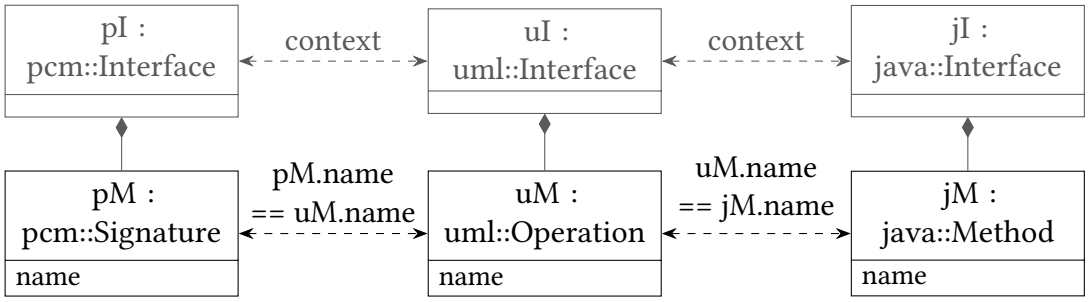


Figure 3.3: This is an extension of the example in Figure 3.2. It now also includes the similarly correspondences between a *pcm::Signature* and the *uml::Operation*.

show the relevant context for the problem case we discuss and instead of a complete correspondence type definition, we provide only the constraints, which are relevant to the argumentation, next to the dashed correspondence arrow. Mapping constraints are often relatively verbose and are therefore omitted. Instead, relevant mapping constraints are implied by the context of the corresponding elements and by the fact that we show a valid instantiation of the correspondences. An example of this can be seen in Figure 3.2, where correspondence instances for both correspondence types defined by Tables 3.2 and 3.3 are depicted. The correspondences are shown as dashed bi-directional arrows. Elements are assigned role-labels *uM* and *jM*, so that they can be referenced in the constraint, which are annotated to the correspondence arrows. The signature mapping constraint of the Interface correspondence *uI~jI* is only implied by the fact that the Operation and Method actually participate in an established correspondence, as a result of their context.

To further simplify the discussed figures, aspects that introduce unnecessary complexity are often withheld. For example in Figure 3.2, it is not mentioned that an interface has a name or that methods can have parameters. For correspondence types that require more complex and lengthy consistency constraint definitions, the correspondence relation is labeled and the consistency constraints are defined in the figure description or the surrounding text.

In the following sections, we often have to discuss how transformation rules and the transformation engine may behave, when multiple correspondences and their consistency constraints are relevant. We therefore often shown correspondences together and show how they combine to form a correspondence graph.

Def. **correspondence graph**:
A correspondence graph *cg* is formed by the maximal set C_{max} of all instantiated correspondences, so that *cg* is still connected.

- $nodes = \{e_1, e2 \mid (e_1, e_2) \in C_{max}\}$
- $edges = C_{max}$

Such correspondence graphs allow us to observe how changes may propagate through a transformation network based on element-level propagation paths, rather than low-resolution model-level propagation paths. The benefit of these element-level change propagation paths is that they reveal the interaction between the general change propagation process and the case specific consistency constraints in more detail.

An example of a correspondence graph can be seen in Figure 3.3, which now extends the exemplary correspondence instantiation of Figure 3.2 to include a correspondence with a similar correspondence type definition between a *pcm::Signature* and the *uml::Operation*. The element *uM* now participates in two correspondences, one correspondence with *pM* (*pM*~*uM*) and one correspondence with *jM* (*uM*~*jM*). If we now interpret all elements as nodes and all correspondences as edges between those nodes, then we can find two connected graphs, each containing 3 elements and 2 edges. Neither graph can be extended to include another element or correspondence, without then being disconnected. Consequently, Figure 3.3 contains two correspondence graphs; the afore mentioned *pM*~*uM*~*jM* graph and its context graph *pI*~*uI*~*jI*.

In this section we propose correspondence types as a way of defining possible types of semantic overlap between elements and as a means to systematically decompose and group consistency constraints. We propose correspondence instances as a trace model, because they directly link the elements that need to be synchronized with each other. Lastly, we introduce correspondence graphs, because they reveal interactions between the general change propagation process and the case specific consistency constraints, which is relevant for the following examples, where we try to identify failure potentials.

3.2 General Observations about Transitive Change Propagation

This section discusses general observations about transitive change propagation. These observations are not supposed to derive problems emerging from transitive change propagation just yet, but instead they should explain some properties and assumptions that are necessary for the following sections.

First we argue, that it is sufficient to examine consistency constraints and their implementing transformation rules, based on a set of corresponding elements and their relevant context, instead of examining the combination of complete transformation definitions. Building on that, we show that linear transformation networks can still exhibit cyclic change propagation paths, when examined at the correspondence graph and transformation rule level. Therefore linear transformation networks can be expected to exhibit the similar problems as a cyclic transformation networks.

Next, we show that limiting the direction, in which changes may be propagated through a transformation network, also limits the consistency constraints that can be expressed. Because of that, we assume changes to be able to propagate along any transformation edge in the network, irrespective of the previous path. And lastly, we argue that multiple changes can accumulate across the models of a transformation network, and therefore change-driven transformations and the transformation engines need to be able to handle batches of changes.

3.2.1 Argument for Context-Based Transformation Examination

As a whole, a transformation definition for practical application is very complex, because it consists of many transformation rules, that may even interact with each other. A transfor-

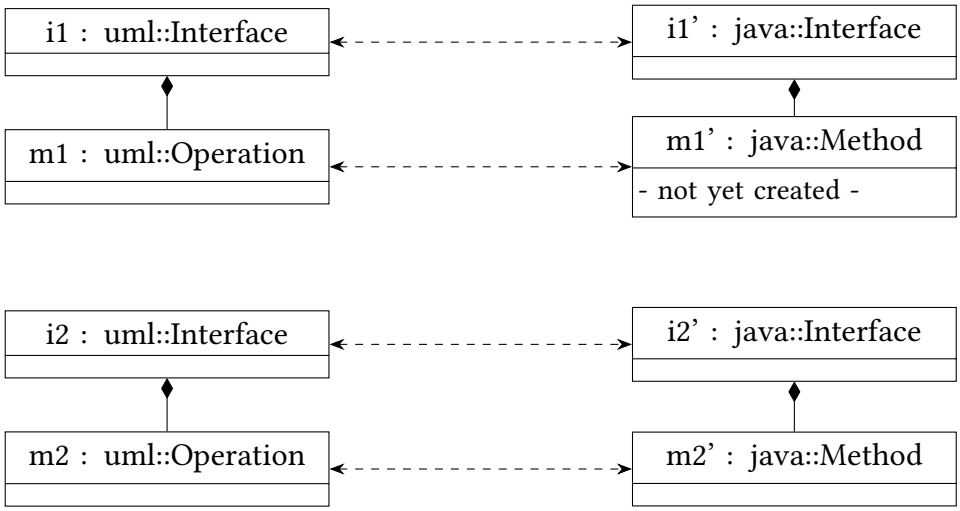


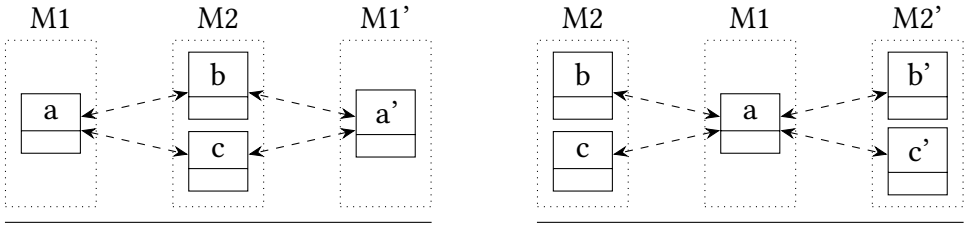
Figure 3.4: Double instantiation of the correspondence type described in Figure 3.2.

mation rule has to synchronize elements that share some semantic overlap and semantics are often derived from the context of the element. This context includes the upward containment hierarchy of the element, its features (attributes and references), and information of the trace model. In practical examples of source incremental transformations, such a context is usually limited to a subset of elements that is smaller than the complete model.

Take the example in Figure 3.4, two uml::Interfaces *i1* and *i2*, each containing a method *m1* and *m2* respectively, correspond to two java::Interfaces *i1'* and *i2'*. The transformation is supposed to synchronize the corresponding interface instances, which includes synchronizing the methods defined by each interface. Upon creation of *m1*, the executed transformation rule needs to check whether *m1* is to be synchronized and where the element to synchronize it with should be. By retrieving the context of *m1*, which is *i1*, and noticing that *i1* corresponds to *i1'*, it becomes clear that a corresponding method *m1'* has to be created, initialized and added to *i1'*. If *i2* were a parent interface of *i1*, that would would result in the inclusion of *i2* in the context of *i1*, and then it would be relevant for the synchronization of *i1*. But it still would not influence the synchronization of *m1*, so long as neither *m1* and *i2* nor *m1* and *i2'* share any implicit or explicit consistency constraints. So we can find a cut-off, that limits the context that is relevant for the transformation rule.

The existence of *i2* and *m2* along with their corresponding elements, and whether *i1* and *i2* share the same containing model, is irrelevant for the synchronization of *m1*. For the purposes of applying this transformation rule, each interface-instance could be its own model, as long as the trace model still reveals which *uml::Interface* should be synchronized with which *java::Interface*. The same would be true for feature changes, e.g. a name change of *i1*. The only element that shares a consistency constraint with *i1.name* is *i1'*.

Because of the observation in this section, we discuss the effects and difficulties of transitive change propagation mostly independent of the concrete transformation network topology and rather base conceptual observations on correspondence graphs.



(a) correspondence cycle (b) correspondence tree

Figure 3.5: Trivial abstract correspondence cycle example in a linear network.

3.2.2 Cyclic Correspondence Graphs in Linear Transformation Networks

Given the observation, that we can define and analyze a single transformation rule based on the minimal necessary semantic context of the changed object, we can then extend this idea to networks of multiple semantically overlapping and therefore corresponding contexts, which we call correspondence graphs. It is relatively easy to construct an example, where one element shares some semantic overlap with, and therefore corresponds to, two separate elements in another model. It is then obvious, that the combination of two such correspondence graphs can form either a tree or a cycle, even though the transformation network only forms a line. A trivial, though on the surface useless, example of this would be the synchronization of a model with a copy of itself, by propagation through another model. In figure 3.5, an instance of this abstract example can be seen. And because changes are propagated along correspondences, even such linear transformation networks can exhibit the problems one would intuitively associate with branching or cycle containing networks. But in practical scenarios, one would expect the frequency, with which particular correspondence graph patterns occur, to differ depending on the transformation network topology. For example, in a network that combines specialization relations with predominantly natural one-to-one concept mappings, one would expect relatively few cyclic correspondences.

A concrete example of a correspondence graph cycle can be seen in Figure 3.6. A *pcm::Signature* and a *java::Method* correspond not only to the *uml::Operation*, but also to a *uml::Parameter*, because return types are represented structurally different in the UML domain.

Technically it is even possible to create a cyclic correspondence graph in a single binary transformation, if in each model two elements correspond with both other elements, building a bipartite correspondence graph.

3.2.3 Argument for Undirected Transitive Change Propagation

In the later discussions, we often look at single bidirectional transformations combined with transitive change propagation, because some failure potentials can already manifest in this simplified scenario. And we can expect that any unsolved problem in this minimal scenario is bound to be at minimum equally problematic in general transformation networks. Some of the problems may be solvable by a sufficiently abstract transformation language that then generates the appropriate checks to avoid the specific problem. In that case, the

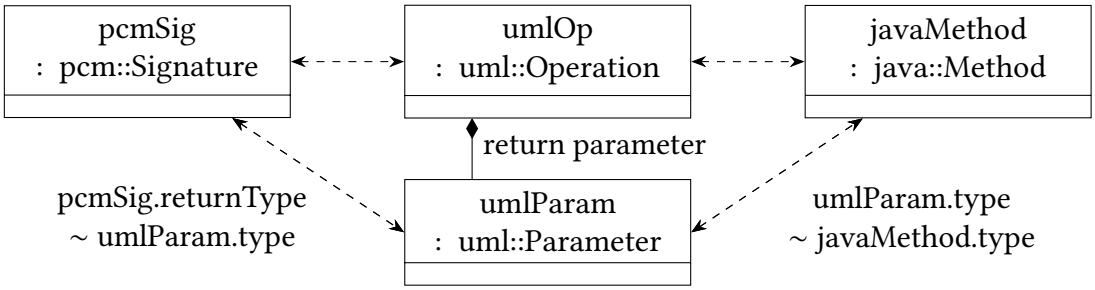


Figure 3.6: Concrete correspondence cycle example in (PCM ↔ UML ↔ Java)-network.

transformation language developer would still have to know about the failure potentials and find general solutions. Also, that would only fix a single transformation definition, because the language can only generate expected checks for changes produced by the transformation itself, not necessarily for changes introduced by other independently developed transformations. We therefore do not differentiate between the transformation developer and the transformation language developer.

In this section, we argue that it is necessary to allow undirected transitive change propagation, even if that introduces cycles to the propagation graph, and as a result bidirectional transformations are minimal cyclic networks. By first looking at strictly directed change propagation, we see that some sets of consistency constraints cannot be restored by it. Therefore, we have to allow transformations to modify their source model, which necessitates the backpropagation of changes, and leads to the necessity for undirected transitive change propagation in general.

We first look at strictly directed change propagation, where changes cannot be propagated backward along already traversed transformations. Such a limitation could be enforced by a transformation engine. Stevens [27] discusses the possibility of using an authority set, meaning a set of models in a transformation network that may not be changed through the synchronization process. By setting the source model of a change to be an authoritative model, it is possible to disallow the backpropagation of changes. With a tiered notion of authority, it would be possible to impose an intended direction of information flow onto a transformation network. In this scenario, the propagation path through a cycle-free transformation network is always defined and any once synchronized pair of models stays consistent if the initial transformation achieved consistency. The introduction of cycles would lead to ambiguity in transformation execution order and confluence of information (see Section 3.4), but the propagation graph would still be acyclic.

However, strictly directed change propagation limits the set of possible consistency constraints that can be enforced, because some inter-model constraints might implicitly impose intra-model constraints that can only be restored through changes in the source model. To demonstrate this, examine the example in Figure 3.7 and assume that the correspondence graph has already been instantiated. The explicit constraints (*comp-compPkg*) and (*comp-compImpl*) imply an implicit naming constraint between *compPkg* and *compImpl*. If now a change *c1 = set(compPkg.name, newname)* occurs, for example through user input, and the transformation from UML to PCM cannot modify *compImpl*, then the

transformation also cannot restore consistency. Furthermore, because the propagated change *c2 = set(comp.name, newname')* cannot be propagated back, the models stay inconsistent. In the example above, it might arguably be better to disallow changes to the UML elements in that context, and only allow modifications through a restricted view of the underlying models, which then modifies the PCM element. But such an approach might not be appropriate for all possible sets of constraints and if the elements are created as a result of other transformations, then the initialization should still terminate in a consistent state.

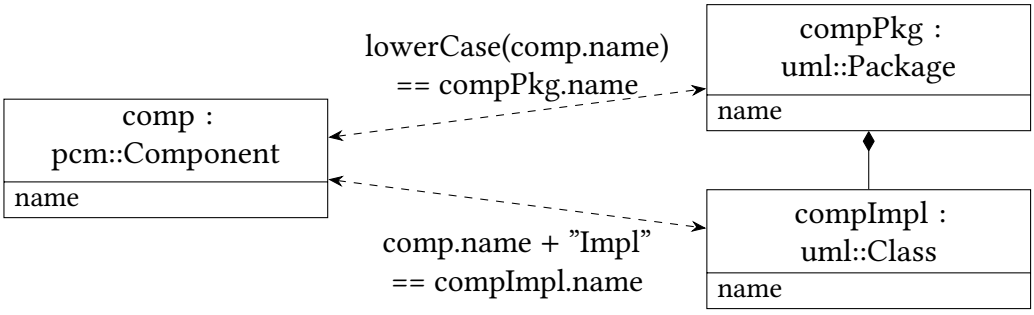


Figure 3.7: Reduced correspondence graph for a *pcm::Component* and its corresponding *uml::Package* and *uml::Class*.

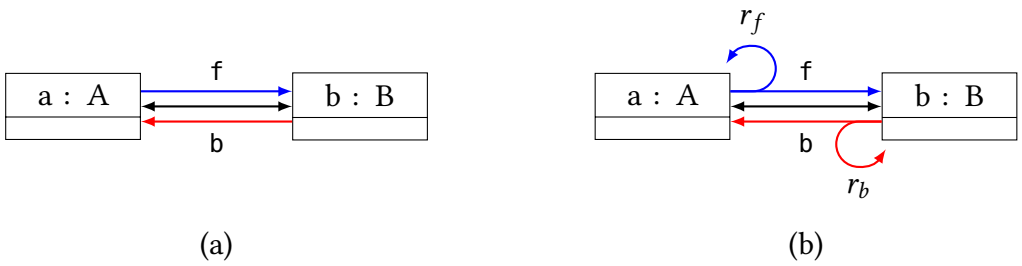


Figure 3.8: Possible decomposition of a bidirectional transformation.

As we just saw, it is necessary to allow a binary transformation to modify both models. This can be done in different ways, depending on the directionalization of the transformation definition. Any bidirectional transformation needs to be operationalized into different application directions for forward and backward execution. This is a necessity to enforce the roles of the source model as the updated version, and the target model as the model that needs to be updated. Without this directionalization, any transformation could trivially restore consistency by discarding any new changes and returning to some previous consistent state.

Figure 3.8 shows two possible ways to directionalize a binary transformation. Based on transformation language and implementation, a transformation definition might only implement strictly directed transformation rules (Fig. 3.8.a) that can only modify the target model. In that case, it would be necessary to perform a backward transformation after any forward transformation in order to change the source model. This could be

done automatically, because we know that it might be necessary, or through *undirected transitive change propagation*, where changes in the target model trigger the backward transformation. In the example above, *c2 = set(comp.name, newname')* would propagate back to UML and result in a third change *c3 = set(compImpl.name, newname")*. At this point, we would already have the risk of a cyclic propagation loop through alternating forward and backward transformations and the rule implementation would have to perform checks in order to avoid this loop.

Alternatively, the transformation language could produce transformation rules that can alter both the source and the target model. The transformation could be conceptually split into a preparatory refactoring step and the usual forward transformation (Fig. 3.8.b). The refactoring could be used to brings the source model to a state, for which the forward transformation is enough to restore binary consistency. This would make it possible to restore the problematic consistency constraints in Figure 3.7 with a single transformation execution, and allow us to avoid undirected change propagation for a single bidirectional transformation.

However, if we now move up one level and look at a minimal transformation network of two transformations, like the one in Figure 3.9, we can see that the changes that result from a refactoring step may need to be propagated back through the propagation graph, because they might break constraints between other model pairs. Therefore, it is now necessary to use undirected transitive change propagation, in order to propagate and resolve the refactoring changes.

Going forward, we assume that any change can be propagated independent of its previous propagation path. As a consequence, any bidirectional transformation is automatically a minimal possible cyclic transformation network of two opposite directed transformations and has to be implemented in a way that avoids transformation loops.

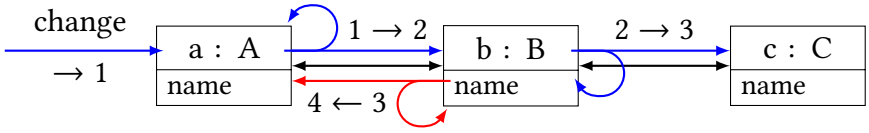


Figure 3.9: Example for the necessity of undirected transitive change propagation.

3.2.4 Accumulated Change Assumption

A change-driven transformation engine and transformation language process change events to trigger transformation rules. The intuitive assumption, when implementing a transformation rule, is that there is exactly one change that needs to be synchronized. But in reality there could be multiple changes, potentially even multiple changes to the same element.

The number of changes that accumulate before a model synchronization step is started partially depends on the timing, when the synchronization is started, and on the change granularity. For example, the user could choose to synchronize the model after every single change or only after a arbitrary number of changes. In the second case, we could artificially reduce the number of changes that need to be processed per synchronization

phase to one, by either automatically synchronizing, or by grouping small changes into one large change. But then the question is, what constitutes a change.

The smallest possible changes would be atomic changes, like for example:

- **create** a single uninitialized element
- **set** the value of a single-valued feature
- **unset** the value of a single-valued feature
- **insert** a single element into a list-feature
- **remove** a single element from a list-feature
- **delete** a single element

But some atomic changes cannot occur alone. For example, a *create* usually has to be followed by an *insert* into the model containment hierarchy, and a *delete* also has to *remove* and *unset* any references to the removed element.

The largest possible changes could be composed of any number of atomic changes, up to the creation of a fully initialized model. At some point the scope of the change becomes so large that the incremental nature of the transformations is lost, and it is unlikely that the change can be processed by a single or small set transformation rules. Therefore, the change might have to first be decomposed, before it can be processed.

For the sake of argument, we assume for a moment that each synchronization is started with only a single change, regardless of the change granularity. Now we have to look at the transformation behavior, because each transformation may be succeeded by multiple others. A single transformation rule can produce multiple atomic changes in the target model. For example in Figure 3.7, a composed (*create* + *insert*)-change for a *pcm::Component* can result in two *create* changes for the *uml::Package* and the *uml::Class* along with the necessary *set* changes to initialize both new elements. Again, depending on the change granularity, one could still argue that this constitutes a single new composed change, that has to be handled by the subsequent transformations.

However, independent of change granularity, the number of un-synchronized changes in the network can still multiply and distribute across multiple models, as well as accumulate in a single model, based on the network structure. Figure 3.10 shows, how the number of changes can multiply or accumulate at branching network paths, depending on the propagation direction.

Therefore we always have to assume that there can be multiple queued changes, independent of synchronization interval and change granularity.

Going forward, we assume that for any change propagation, there can be multiple queued changes. Also, in order to avoid having to repeatedly explain what constitutes a single composite change, we assume atomic change granularity for the rest of this thesis.

3.2.5 Summary of Assumptions

The following is a small overview of the assumptions we just argued for and which we assume to hold for the rest of this chapter.

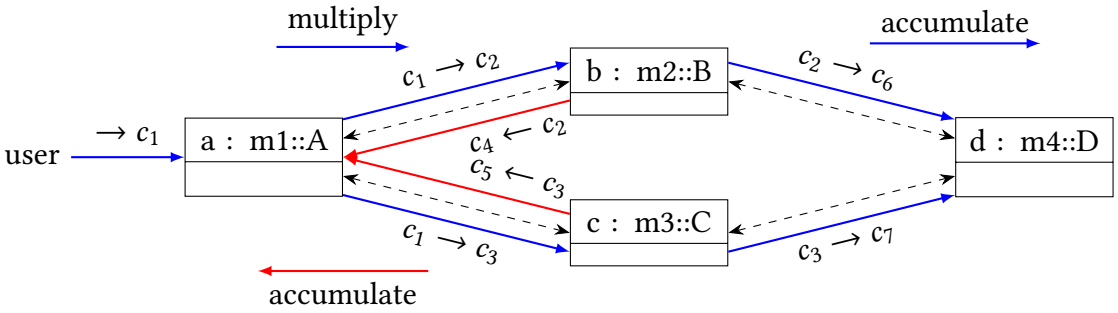


Figure 3.10: Change multiplication and accumulation example.

- Transformation rules can be validly discussed based on correspondences and the context of their participants.
- Because correspondences and their contexts are sufficient to express transformation rules, changes can be assumed to propagate along correspondence graphs.
- Cyclic change propagation problems can emerge in linear transformation networks, because such networks can contain cyclic correspondence graphs.
- Undirected change propagation is necessary to express all consistency constraints.
- Bidirectional transformations form minimal cyclic transformation networks as a consequence of undirected change propagation.
- We have to assume multiple queued changes, as results of possibly branching correspondence/transformation graphs.
- And we assume atomic changes for simpler discussion in the following sections.

3.3 Change Application and Change Resolution

A change-driven model transformation engine processes change events and triggers transformation rules depending on the processed event. We differentiate between the *change* as "that which affects the model" and the *change event* as "that which describes the effect of a change". The application of a single change to the model under modification does not have to coincide with the time, at which the respective change event is processed by the transformation engine. To demonstrate this, we first differentiate two terms.

- To **apply** a change means to modify the model affected by the change, so that the new state is reached and the effects of the change are visible when retrieving information from that model.
- To **resolve** a change (event) means that the transformation engine processes the change event and triggers transformation rules in order to propagate the change through the transformation network.

Now, assuming a batch of queued changes, the intuitive strategy is to apply and resolve each change one at a time. We call this strategy **apply-on-resolve**. Alternatively, it would

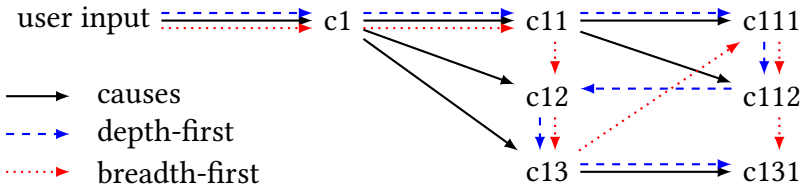


Figure 3.11: Abstract example for depth-first versus breadth-first change resolution.

be possible to first apply all changes together and then resolve the change events. We call this strategy **batch-apply**. The latter is operationally easier, because the executed transformations can simply modify the affected models directly, thereby automatically applying all changes, and the transformation engine only has to monitor the affected models and record the occurring change events. In contract, with apply-on-resolve, the engine would have to record the changes on a temporary model, or roll them back, before re-applying them individually.

3.3.1 Apply-on-Resolve

First consider the apply-on-resolve strategy. It ensures the intuitive assumption of the transformation developer that the information of the change event is in sync with the state of the source model. However, because the executed transformations can trigger new changes, without knowing about the already queued changes or their respective new changes, it is possible for old and new changes to be in conflict with each other. A **change conflict** occurs whenever a change negates the effects of another, or when element creation changes are unintentionally duplicated. The former can be intended, for example when the user overwrites a prior feature assignment. The latter cannot be intentional, because two element creations do not cancel or interfere with each other, unless both elements were intended to be the same and the duplication results in an inconsistent correspondence graph. These change conflicts can occur in the source model or in the target model, depending on the resolution strategy.

Figure 3.11 illustrates the difference in change resolution order between a depth-first and a breadth-first resolution approach, with the help of an abstract change propagation scenario. We can track all changes by their causal relation: parent change c_x caused (a transformation to produce) the child change c_y. If we interpret the tuples in that relation as edges of a graph, we obtain a cycle-free tree structure. We define the **resolve-depth-first** strategy as first resolving all child-changes before resolving the siblings. In contrast, the **resolve-breadth-first** strategy resolves siblings before children. Using resolve-depth-first there is a chance that changes are propagated back to the source model before all queued changes of the source model are resolved. Resolve-breadth-first instead runs the risk of producing conflicting or duplicate changes to the target model, because later transformations do not yet see the changes created by the earlier executed transformations.

Take for example an established instance of the correspondence type in Figure 3.12 and two unresolved name changes on the same model element a. Figure 3.13 shows how the changes are resolved when using apply-on-resolve and different resolution orders. First

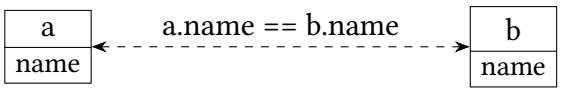


Figure 3.12: Correspondence instance example for two named elements.

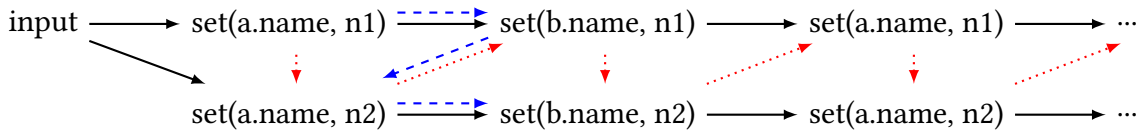


Figure 3.13: A causal relation tree example for two name changes and apply-on-resolve. It shows how new changes are caused by the resolution of other changes (black arrow) depending on the used resolution order: depth-first (blue, dashed) or breadth-first (red, dotted).
This is not the same as a change propagation graph, because changes may well be propagated along cyclic paths through the transformation network, but each change can only be caused by a singe other change (or user input), which results in the acyclic tree structure.

examine the depth-first resolution. The first change on *a.name* is propagated to *b.name* and not propagated back, since the correspondence is already synchronized according to the specification that both names have to be the same. Next, the second name change is resolved in a similar manner, so that both *a.name* and *b.name* are set to *n2*. The second change effectively overwrites the first change. This might have been the intended outcome of a user overwriting an earlier decision or an unintended side effect in some transformation rule, but at least it does not cause further problems.

Now examine the breadth-first resolution. The resolution of the first change on *a.name* creates a new name change (*c3 = set(b.name, n1)*) and adds it to the queue. The resolution of the second change then creates a new name change (*c4 = set(b.name, n2)*) and also adds it to the queue. Now we are faced with the same situation as before, but simply propagating the equivalent changes in the other direction. The propagation is stuck in a **propagation loop**.

Even this simple example fails when using breadth-first resolution. To demonstrate the problem with apply-on-resolve and depth-first resolution, consider the correspondence graph defined in Figure 3.14. Assume the correspondences are not yet initialized and both elements *a* and *c* are created together, but their changes *create(a)* and *create(c)* are not yet resolved. On resolving *create(a)*, *b* is created, which then resolves to create an element *c'*, because the transformation can not yet know that the creation of *c* has been queued but not applied. Afterwards *create(c)* is applied and now two elements exist, which are supposed to be the same. Using breadth-first resolution in this scenario could result in different outcomes depending on the concrete transformation implementation:

- If the resolution of *create(c)* tries to create *b'*, because it does not yet see the *b* created by the resolution of *create(a)*, then there is duplication of *b*.

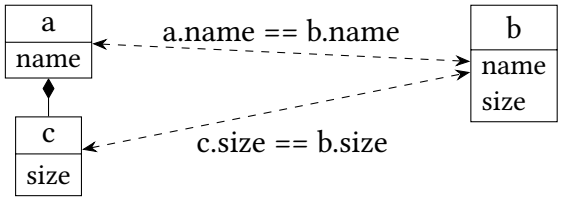


Figure 3.14: Extended Fig. 3.12 to demonstrate element duplication scenarios.

- Otherwise, if the transformation rule triggered by *create(c)* checks if the context requires the synchronization of *c*, namely that *a* participates in an *a*~*b* correspondence, then the creation of *b'* can be avoided.
  - But maybe *b* and *c* are not synchronized, ...
  - unless the resolution of *create(b)* detects the existing *c*.
- And without the depicted containment, the context clue to avoid a duplication of *b'* could either not be detected, or the context would have to be defined much broader.

Both depth-first and breadth-first resolution fail in one of the scenarios because the executed transformations can not see queued changes and therefore run the danger of generating conflicting changes. Since the transformations can not intervene in those cases, the transformation engine has to deploy a mechanism for consolidating conflicting changes. This may work for **set-feature conflicts** where a conflict is detectable by identifying that two value changes that affect the same feature are not equivalent. However, it is unclear how to detect **element-creation conflicts** when much of the semantic meaning of an individual element is derived from the constraints enforced through transformations, but not inherent to the element's model.

3.3.2 Batch-Apply

The batch-apply strategy differs in that all queued changes are applied before the change events are resolved. As a result, transformations triggered by a change event early in the event queue can see the applied effects of later changes. Because of that, transformation rules can be formulated in a way that avoids creating unintended conflicting changes. The possibility, but also the responsibility, to prevent such conflicts lies with the transformation developer. However, now using the batch-apply strategy, the developer faces the problem that the information about a change, provided by a change event, may be deprecated, if the subsequent application of a later change invalidated the change associated with the change event in question. This forces the developer to differentiate between valid and **deprecated change events** based on the information available from the source model.

Take the example in Figure 3.12 and again assume two consecutive name changes *c1 = set(a.name, n1)* and *c2 = set(a.name, n2)*. Both changes are applied and *c2* overwrites *c1* so that *a.name* is set to *n2* before the change events are resolved. Since *c1* is the earlier change, it is resolved first, but the change event claims that *a.name* has been set to *n1* even though the model shows *a.name == n2*. The developer can now either rely on the change events or on the information present in the models.

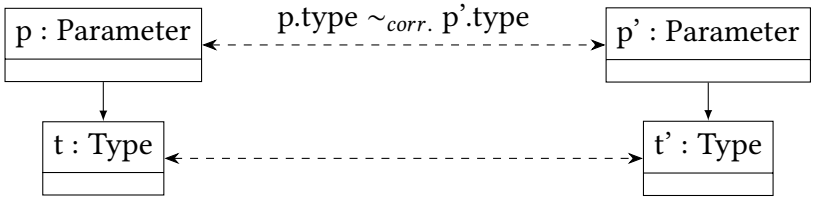


Figure 3.15: Simple parameter and type correspondence example.

Examine first the implications of relying purely on change event information. Because the assumption is that the change is up-to-date, as if apply-on-resolve had been used, the results are similar. A breadth-first resolution leads to the same propagation loop. A depth-first resolution terminates and correctly synchronizes to *a.name = n2 = b.name*, however in the process there are multiple unnecessary transformation steps, because the resolution of *c1* first synchronizes to *a.name = n1 = b.name*, overwriting the initially applied change *c2*, before resolving *c2* and reaching the target state.

By instead using an incremental state-based transformation rule, where the scope is determined by the event (which elements and/or which of their features), but the propagated information are retrieved from the model, the number of transformation steps can be reduced and the propagation loop can be prevented. The resolution of both change events try to propagate the same information and the resolution of *c2* does not generate follow-up changes. One drawback is that this only works if the transformation can be implemented state-based, as the change sequence can not be retrieved from the model, only from the change events, but we assume that most practical examples can be implemented in such a way.

If changes that have not yet been resolved, but are visible to transformations, are propagated using state based transformation rules, then a new problem can emerge, where the retrievable context cannot yet be correctly propagated, because correspondences of referenced elements might not exist yet. The correspondence type example in Figure 3.15 can demonstrate how this manifests. Assume the following batch of changes *c1 = create(p)*, *c2 = create(t)*, *c3 = set(p.type, t)*. All three changes are simultaneously applied and then resolved in the named order. The resolution of *c1* sees *p* and that *p.type* is set to reference *t* and therefore the transformation rule creates *p'* and attempts to set *p'.type* to *t'*, but *t'* does not yet exist. At this point, depending on the implementation, the transformation might notify the user that an error has occurred, because an element can not be synchronized, or it might even abort the propagation. But if the transformation engine continues to resolve the other changes, creates *t'* and sets *p'.type = t'*, then the intended final consistent state can be reached. So there is an error potential in the **visibility of unresolved changes**, that can at least in such simple cases be healed by later changes.

Such temporarily unsynchronizable constraints are in most cases a result of change event resolution order. If for example the changes establishing the necessary correspondences were executed first (in the above example resolve *create(t)* first), then such reference-resolution failures can be prevented. However, in the general case where cyclic reference-dependencies (in the simplest case bidirectional references) can occur, this is not solvable through propagation re-ordering. Instead the propagation of previously not synchronizable

information has to be triggered again, after the prerequisite changes are resolved. This can be realized through additional change events for the elements and features in question. But because partial information are propagated, the transformation developer has to be careful that the partial information is not propagated back to the source model and overwrites the still pending changes.

3.3.3 Summary

Apply-on-resolve is an appealing strategy, because it provides the transformation developer with consistent information across models and change events. However, because pending changes and their effects are not visible during the execution of earlier transformations, this can result in unintended **conflicting changes** which may lead to propagation loops or element duplications, without giving the transformation developer a chance to intervene through adequate checks. Therefore, the transformation engine would need an intervention mechanism to consolidate conflicting changes itself, but a general solution seems unlikely, as some of an elements semantics come from the constraints implemented by the transformations.

In contrast, **batch-apply** provides the transformation developer with the necessary information to intervene by making later changes visible. The downsides are potentially **deprecated change events** and **visible but unresolved changes**, which need to be handled by the transformation developer. But unless a generic transformation engine solution is found to manage conflicting changes, batch-apply is the operationally realistic strategy, even with the increased transformation development effort. Another observation is that depth-first resolution strategy seems to produce less disruptive or at maximum as bad failures as breadth-first resolution in both discussed scenarios, apply-on-resolve and batch-apply.

Going forward, we assume batch-apply unless otherwise specified.

3.4 Confluence Problems

Bi-directional transformations need to derive different operational transformation rules for forward and backward execution, so that changes can be propagated from the source to the target model, and the other way around, depending on where the pending changes occurred. As we argued in section 3.2.3, we need undirected change propagation to express all consistency constraints. But this also leads to potential propagation cycles along the forward and backward transformation directions. And because we have a chance of propagation path cycles, multiple transformed versions of an initial change can reach the same model.

Figure 3.16 depicts the possible directionalization options for bidirectional transformation (strictly directed or with refactoring) along a single correspondence. Without regards for the underlying transformation implementation that should prevent looping or unnecessary transformation steps, a change to element a, introduced by the user, can be propagated along any number of paths. It could be resolved after a single forward transformation(path=(f)), an additional backward step (path=(f, b)) or any

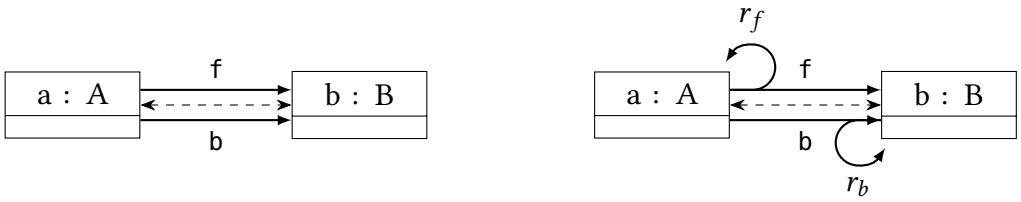


Figure 3.16: Possible decomposition of a bidirectional transformation (copy of Figure 3.8).

longer path of alternating forward and backward steps (path=(f, b, f, b, f)). In the second example the path could also contain repeated interspersed refactoring steps (path=$(r_f, r_f, f, b, f, r_b, b, f)$). In order to prevent conflicting changes, both forward and backward transformation need to check, if the information of a change is already present in the target model, before propagating it. Otherwise even simple resolution paths like p1=(user->create(a), forward->create(b), backward->create(a)) lead to a confluence of information, where two *create(a)*-changes reach the initial model and produce an element duplication.

In our context, **confluence** always refers to a scenario where changes are propagated to a model or element along multiple paths through the transformation network. The basis for transitive change propagation is the idea that a transformation propagates a change from its source to its target model. A change propagation path can then be defined by a tuple of the transformations that propagated the change to a specific model.

Def. **confluence**:
Two paths through a graph structure are confluent, if they reach the same node along different edges. Let:

- graph $G = (N, E)$, nodes N, edges $E \subseteq N \times N$
- path $p = (t_1, ..., t_i, t_{i+1}, ..., t_n)$, with $i \in [1, n], t_i.end = t_{i+1}.start, t_i \in E$
  $p.start = t_1.start$, $p.end = t_n.end$

then two paths p_1, p_2 are confluent, if

- $(p_1 \neq p_2) \wedge (p_1.start = p_2.start) \wedge (p_1.end = p_2.end)$

Def. **transformation path confluence (model level)**:
If we define a change propagation path as a tuple of transformations and the graph structure as follows:

- graph G = transformation network TN
- nodes N = models M of the transformation network TN
- edges E = transformations $T \subseteq \{t = (m_1, m_2) \mid m_1, m_2 \in M\}$

then we can use the confluence definition above to determine if two such transformation paths are confluent.

Because correspondences are defined in such a way, that they link the elements that need to be synchronized, a change to one element of a correspondence has to be propagated to

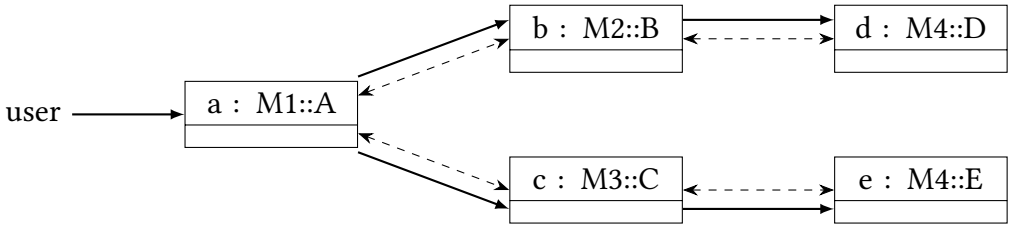


Figure 3.17: An example correspondence graph to show that in general transformation path confluence ≠ correspondence path confluence.

the other element. We can therefore also track the propagation path of a change based on the traversed correspondences.

Def. **correspondence path confluence (element level)**:
A similar path confluence definition can be formulated for paths through a correspondence graph:

- graph G = correspondence graph
- nodes N = instantiated elements
- edges E = correspondences between the elements

Using correspondence paths, we get a higher resolution of the propagation path, on an element level. The example in Figure 3.17, demonstrates how the correspondence propagation paths of a change can be confluence free, while the transformation propagation paths are confluent, because the elements d and e are part of the same model. For a change introduced to model $M1$, the transformation paths $tp_1 = ((M1, M2), (M2, M4))$ and $tp_2 = ((M1, M3), (M3, M4))$ are confluent, while the correspondence paths $cp_1 = ((a, b), (b, d))$ and $cp_2 = ((a, c), (c, e))$ are not confluent. Assuming the change really only needed the modification of the corresponding elements, then the transformation propagation path confluence might falsely indicate a failure potential, even though on an element level, the propagated changes operate on different parts of the target model.

Whenever a confluence of information occurs, it leads to a possibility for conflicting changes to be generated, and as observed in previous sections, such conflicts can cause propagation loops or loss of information by overwriting each other. Depending on the path taken, the confluent changes might also contain different information, because different metamodels might use different default values for their elements. As a result, even if a convergent terminal state is reached, it is not guaranteed that a different equally valid propagation path would lead to the same state.

Figure 3.18 shows an example of a cyclic correspondence graph and one exemplary change propagation path. The depicted resolution steps results in an element duplication of c. Even if the duplication of c is avoided, the value initialization for $c.y$ my be indeterminate, because it depends on the change resolution order. The default value of one metamodel is propagated first and subsequently overwritten by the default value of the other (y_A or y_D could be default values of the metamodels or of the transformation definitions). There are multiple alternative propagation paths that produce similar confluence

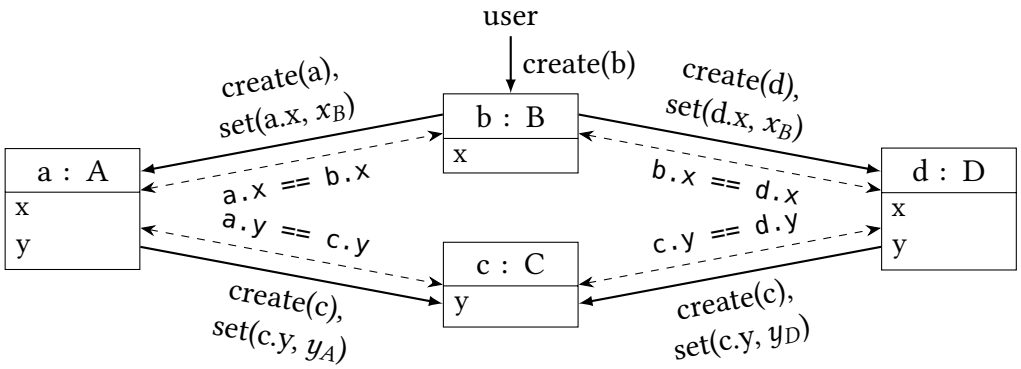


Figure 3.18: Confluence example with element duplication and inconsistent value initialization.

problems for different elements. Assume for example using depth-first resolution, then the transformation engine might resolve the *b-a* correspondence first, which produces a path *(b-a-c-d-b)* back to *b* and either a duplicate *b'* is created or $b.x == x_B$ is overwritten by x_D. Set-value-change confluence is less problematic then create-changes when used with batch-apply, because it often converges to some value, even if the concrete value is dependent on resolve order. Create-changes and set-value-changes that produce element creations in other models have a much higher risk of producing propagation loops, if the transformation developer does not address them. A counter example, where a set-value change can produce a propagation loop, would be a string mapping rule that appends some suffix with each transformation step.

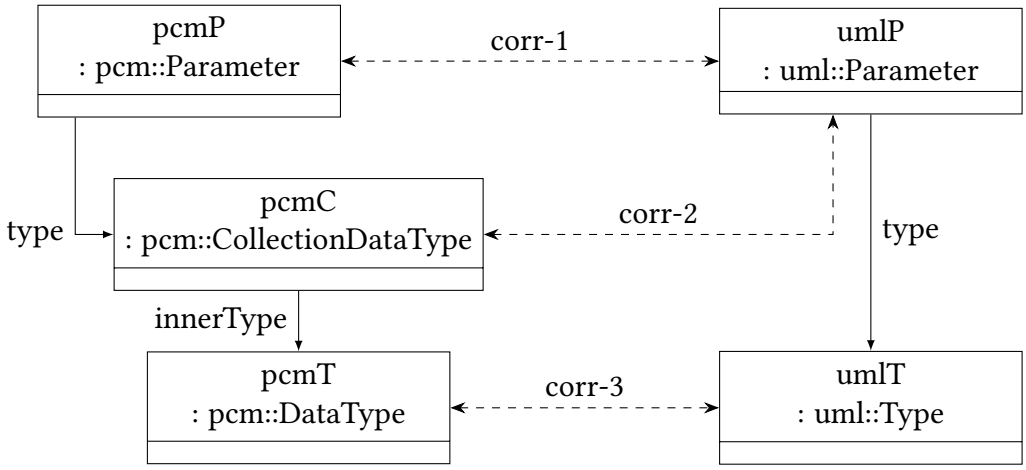
3.5 Change Granularity and Unsynchronizable States

Consistency constraints can disallow a subset of model states, by declaring them inconsistent, independent of the state of the corresponding model, even if they are conform to the metamodel definition. Such a constraint may be in place, because both involved metamodels represent the same concept differently, and some configurations of model *A* cannot be represented by model *B*. We call such model states **unsynchronizable states**.

Assume a consistency-restoring transformation, that is executed after a change *c* transitions model *A* to an unsynchronizable state. Per definition, the transformation cannot bring *B* to a consistent state with *A*, without changing *A*. There are now three possible operationalizations for the transformation:

1. revert the change *c* that produced the unsynchronizable state,
2. perform complementary changes on *A*, until it can be synchronized,
3. or leave the models inconsistent.

Option 1) leads to a loss of information in model *A* and either directly overwrites a user input, which is irritating to the user, or it overwrites a change initiated by a previous transformation, in which case the overwrite has to be propagated back and potentially causes



* corr-1 = (pcmP.type $\sim_{corr-3}$ umlP.type $\wedge$ umlP.multiplicity==(1..1))
$\vee$ (pcmP.type $\in$ pcm::CollectionDataType $\wedge$ pcmP.type $\sim_{corr-2}$ umlP)
* corr-2 = pcmC.type $\sim_{corr-3}$ umlP.type $\wedge$ umlP.multiplicity==(0..*)

Figure 3.19: An example of a valid instantiation for a *pcm::Parameter~uml::Parameter* correspondence with a type reference to a *pcm::CollectionDataType*.

problems for other transformations. Option 2) runs the risk of selecting an unintended target state, if there are multiple possibilities. And option 3) breaks the assumption that both models are consistent afterwards. None of these options is clearly preferable to the others. The concrete transformation implementation has to be decided on a case by case basis, to find the most appropriate solution in the given context.

Furthermore, unsynchronizable states can be necessary in order to transition between two synchronizable states. The example in Figure 3.19 shows a correspondence graph in which an element can have two synchronizable value-configurations and two unsynchronizable states. More specifically, a *uml::Parameter* can have the two multiplicity configurations, (1..1) and (0..*) (written as <lower-bound>..<upper-bound> where "*" stands for "unlimited"), that conform to the consistency constraints. In order to transition from one configuration to the other, either (0..1) or (1..*) has to be passed, both of which cannot be resolved to a consistent state of the PCM-context.

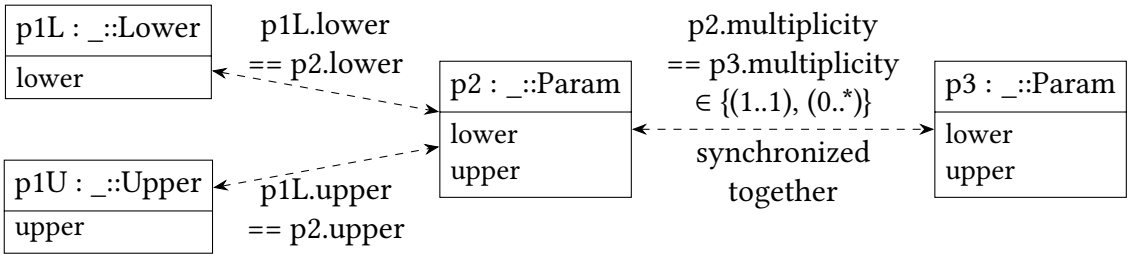


Figure 3.20: This is an abstract example to demonstrate problematic interaction between change granularity and unsynchronizable states.

Problems concerning unsynchronizable states are in part a consequence of the change granularity, with which changes are detected and processed. By accumulating changes to a model and postponing the synchronization until a synchronizable state has been reached, the problem can be avoided for a single binary transformation. But how atomic changes are bundled, so that they form a synchronizable and resolvable change, cannot be generally solved by the transformation engine, because the consistency constraints define what constitutes a synchronizable state for a specific transformation definition. Therefore the ideal change granularity may differ between sets of constraints and context. Figure 3.20 shows an abstract example scenario, again concerned with parameter multiplicity. One transformation requires a complex change that changes both the lower and upper multiplicity boundaries together, the other transformation(s) do not have the same constraints and therefore are fine with independent changes to those features. Also, by combining independently developed transformation definitions, it cannot generally be assume that other transformations know not to produce such constraint specific unsynchronizable states. This can be seen in the example above, where changes to *p1L* and *p1U* can be propagated independently, and therefore trigger two independent changes of *p2*, even though the following propagation to *p3* might be better able to process a bundled change. Even if both changes to *p1L* and *p1U* translate to a synchronizable state of *p2* (e.g. p2.multiplicity=(1..1) afterwards), if the transformation $p2 \longrightarrow p3$ cannot detect that the second change is coming, then it might overwrite the first change, which then also necessitates the overwrite of the second change. So as a consequence of transitional unsynchronizable states, a synchronizable state may be rejected.

3.6 Concept Bottlenecks

Metamodels are often developed with different concerns and therefore different concepts in mind. Most metamodel pairs share some semantic overlap, but also model some concepts unknown to the respective other. A model transformation can at best keep the overlapping concepts consistent. By transitively combining transformations, the set of concepts, which can by synchronized along a single path, is reduced to the set of concepts common to all models along that path. This implies that a concept shared between the end-points cannot be propagated or synchronized along said path, if a single model along the way cannot express that concept. The model that cannot express the shared concept functions as a bottleneck for the information that can by synchronized. We therefore call this a **concept bottleneck**. Now if there exists no alternate path in the transformation network, along which the shared concept can be synchronized, then the end-point models can desynchronize if such a concept is instantiated or modified. While this does not produce transformation failures for a single binary transformation, because each transformation can still correctly synchronize the models to a locally consistent state, the set of all models in the transformation network can still be globally inconsistent. Therefore a transformation network, with concept bottlenecks and no alternate paths, potentially fails at multi-model consistency preservation. This effect is linked to Nuseibeh's observation ([16], Appendix A) that local consistency does not equal global consistency.

Figure 3.21 shows an example of a concept bottleneck. Even though model A and C share the same concept y and it is possible to formulate a constraint that would enforce consistency, changes to $a.y$ can not be propagated without a direct transformation from A to C, because B does not have a concept to represent y. If now a user inputs different values for $a.y$ and $c.y$, then models are globally inconsistent and the synchronization process cannot fix or even detect this circumstance.

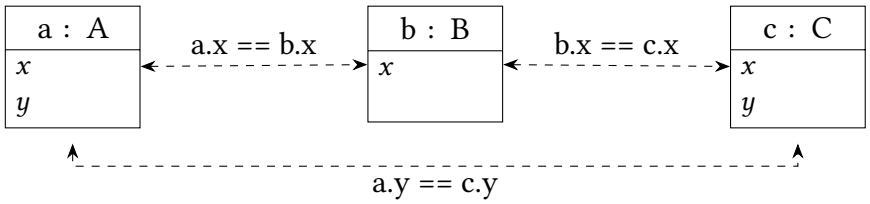


Figure 3.21: Concept bottleneck example

One option, for solving the concept bottleneck in the example, is to adapt the metamodel of B so that the concept in question can be expressed. However, with metamodels that are used for practical applications, this approach is hardly possible, because the pass-through metamodels run the danger of becoming overly complex and the number of metamodels that have to be adapted grows with path length. Additionally, when standardized metamodels are used, it might not even be possible or allowed to change the metamodels.

Another option is to add additional transformation definitions to the transformation network, so that concept bottlenecks can be bypass. But the additional transformations are likely to introduce new cyclic dependencies, thereby increasing the number of failure potentials associated with cycles, which we discuss in section 3.4.

3.7 Incompatible Consistency Constraints

The previous sections were concerned with interoperability issues. This section discusses incompatibility between transformations, each of which implement a sets of consistency constraints. While interoperability issues emerge through transitive change propagation, transformation incompatibility exists when the combination of the underlying constraints either does not result in the intended global consistency or some of the constraints directly contradict themselves.

One example for incompatible consistency constraints are structurally different mapping assumptions. Take the two transformations PCM ↔ UML and the other UML ↔ Java. Now assume that PCM ↔ UML maps between externally defined *pcm::PrimitiveDataTypes* and *uml::PrimitiveType* instances and the other transformation uses manually defined *uml::PrimitiveType* and maps them to the Java standard primitive types. Were the transformations combined as they are, it would not be possible to synchronize the type of a method parameter to the same semantic primitive type across all three models, because there are no rules implemented, that specify when predefined and manually created *uml::PrimitiveType* instances are semantically equivalent. In fact, it is hard to argue how that would work, if the method parameter can only be assigned one type.

Secondly, incompatible consistency constraints can lead to contradictions when combined. Figure 3.22 shows an example of this. A *pcm::Component* is mapped to its implementation class, but one transformation appends a suffix “Impl” to make this relationship clear to the user, whereas the other transformation omits the suffix. The constraints (p.name == u.name) and (u.name == j.name) together imply (p.name == j.name), which directly contradicts the constraint (p.name + “Impl” == j.name).

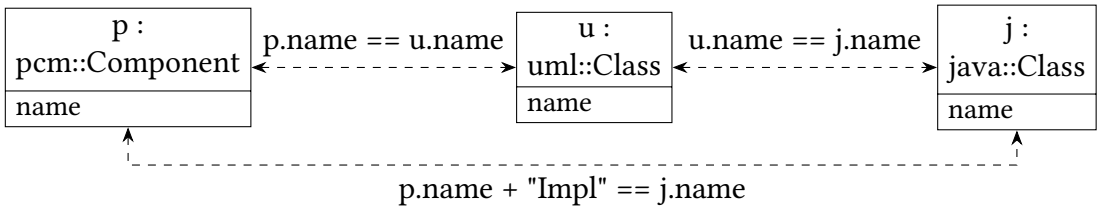


Figure 3.22: Consistency constraint contradiction example.

The category of incompatible consistency constraints is relatively weakly defined to capture most problems stemming from the consistency specifications of the transformation network, because these problems cannot be fixed by a smarter implementation of each independent transformation. This broad classification runs the danger of ignoring or grouping more specific problem causes. Future work might reveal, if this category should be split into sub-classes, and how to formulate consistency constraints in order to increase the chances for compatible, yet (mostly) independent developed transformations.

3.8 Indeterminate Change Order in Distant Models

This section is mainly argumentative, because non of the examined models and transformations explicitly relied on specific change sequences. As a result we cannot provide an example where the preservation of the change sequence is relevant.

But we assume that if a transformation’s output relies on the order in which changes occur and are processed, then it is important to preserve the relevant sequence or even reorganize pending changes accordingly. And we can show, that in general it is not possible to ensure this in a transformation network with independently developed transformations.

First, if we assume a single transformation definition, then we can either rely on an informed user to provide the information in the required order, or we can limit the editor in such a way, that the necessary change order is enforced, perhaps through an input dialog. Now assume a single transformation that produces multiple changes in the target model. The order of the resulting changes, directly relies on the implementation of the transformation. Therefore, if the resulting target changes need to be ordered, this can be accomplished by a proper implementation. If multiple transformation rules need to trigger, then the resulting target change order is dependent on the rule execution order. Furthermore, if a transformation relies on incremental transformation rules that only synchronize a limited context, mostly independent of each other, then the rule execution order might not be explicitly defined, but rather accidental based on some arbitrary

ordering of the definitions. Then the target changes would only be partially ordered, which could pose a thread to the proper execution of the following transformation.

Now, even if we assume that the execution of all transformation rules are ordered for a transformation definition, we still cannot assure that the resulting changes occur in the intended order, because other transformations in the same transformation network might have modified the target model as well. The transformation engine would have to know in which order to execute the transformations, so that the concatenation of their changes still preserves the requirement. Furthermore, if transitive change propagation is also considered and changes can be propagated back to the source model, then the problem is suddenly even more complex, as it is unclear if the backpropagation of a target change should be resolved before or after the forward propagation of the next transformation.

Even if the change order can be controlled on a per-transformation basis, the transformation execution order may depend on the network topology and engine implementation. As a result, a specific intended change sequence cannot be guaranteed for the general case.

3.9 Summary of Proposed Failure Potentials

We have identified six failure potentials of transitive change propagation. with the goal of multi-model consistency preservation. An overview is provided by Table 3.1 at the beginning of this chapter, and now we want to summarize the failure potentials in more detail.

A **change conflict** occurs when a later change invalidates the effects of an earlier one. Examples include: two *set* changes that affect the same element's feature with different values, a *remove* that reverts an element *insert* into a list-feature, or the collision of two *create* changes that were supposed to create one and the same element. Change conflicts can be produced by the user, in which case we would not classify them as failures, through postponed change application, as is the case for the apply-on-resolve strategy, or as a consequence of change confluence.

A **change event is deprecated** if the described effect of the change does no longer match the model state that is present at the time when the event is resolved and the subsequent transformation rule is executed. This can occur as a consequence of immediate change application (batch-apply) and the accumulation of multiple changes to the same model, when for example a second change directly overwrites the first one, and both are applied before the resolution of the first change. A transformation that relies on the information provided by the change event might then propagate a false model state.

The effects of still **unresolved changes** can be **visible** to transformation rules if multiple changes accumulate at one model and all changes are immediately applied. Because the effects can be retrieved from the source model state at the time of transformation execution, a state-based transformation rule may attempt to synchronize a reference in the target model, for which the corresponding referenced element has not yet been created.

An **unsynchronizable states** is a source model states, for which there is no consistent target model state, without modification of the source model. Such states are a consequence of the consistency constraints between two models, and they can become problematic, if the change granularity, with which changes are applied or propagated, does not support

the circumvention of such states. Take for example two atomic source model changes that would together transition the source model to a synchronizable state, but when applied alone, either change results in an unsynchronizable state and potential information loss.

Concept bottlenecks are a consequence of the transformation network topology and the information that can be represented by the involved metamodels. If a concept cannot be represented in a domain, then the instances of that concept cannot be synchronized along a transformation path through said domain, which may lead to global inconsistency.

And lastly, **incompatible consistency constraints** are constraints that either lead to a direct contradiction when combined or imply structurally different mappings, which cannot be implemented together. We assume this to be the only failure potential that is unsolvable if we do not change the consistency constraints, which means that it is not solvable through generic adaptations to the synchronization process or transformation implementation patterns.

We do not include **indeterminate change order** (Section 3.8) in this list, because we cannot provide an example of a failure scenario. But it is theoretically possible to produce a failure, if the outcome of an involved transformation relies on a particular change order.

4 Proposed Transformation Implementation Patterns

The last chapter discussed the properties and failure potentials that emerge from transitive change propagation and transformation combination with regards to multi-model consistency preservation. In this chapter we now propose patterns for transformation implementation that mitigate some of the explored failure potentials in a generalizable way.

Not all failure potentials can be can be prevented from manifesting a synchronization failure through implementation patterns. Incompatible consistency constraints and concept bottlenecks are problems of the transformation network specification. Incompatible consistency constraints can only be fixed if we change the consistency constraints. Concept bottlenecks are a result of the chosen metamodels and transformation network topology. If a concept cannot be expressed in a pass-through model, then no binary transformation implementations can synchronize the concept through said model. Change conflicts that occur as a consequence of delayed change application (Section 3.3.1) also cannot be avoided through better transformation implementation. Because the transformation rules cannot see the queued changes during their execution, they cannot detect whether a target model change would conflict with later changes, and consequently cannot avoid producing such conflicts. However, where transformations can detect the chance of a conflict or the conflict itself, there is a possibility for intervention.

One goal should therefore be to configure the change propagation process and the transformation engine in such a way that we can detect as many failure potentials as possible, at least so long as the engine cannot prevent the failure potential. To that end we proposed in Section 3.3.2 the immediate application of all accumulated changes to a model, before resolving individual change events. This allows the transformations to detect the effects of later changes and to adapt the output accordingly. But the immediate change application results in new failure potentials, one of them being the chance of deprecated change events, where the delta described by the change no longer represents the information present in the underlying model.

In the following we explain, how we can detect whether a change events is valid or deprecated, using change event *validity checks*. This is followed by a discussion about how to avoid element-creation change conflicts, based on possible element retrieval strategies, which we can use as *existence checks* to determine whether a target element already exists.

4.1 Dealing with Deprecated Change Events

Deprecated change events are change events, whose described effect no longer represents the underlying model state, because later changes modified the same feature of the same element before the event was resolved. They are a direct consequence of feature change conflicts, where the same feature of the same element is modified multiple times. Therefore, the occurrence of deprecated change events is unavoidable as long as feature change conflicts can occur. However, feature changes conflicts can be necessary if the user may perform multiple subsequent changes before the synchronization is started and metamodels provide different default values. Furthermore, feature changes conflicts can also occur as a consequence of change confluence in propagation process or suboptimal transformation implementation.

The example we use throughout this section consist of two subsequent, opposite replace changes that affect the feature f of an element e: *c1 = replace(e.f, x → y), c2 = replace(e.f, y → x)*. If both are immediately applied, before either is resolved, then the model state is $e.f = x$ when *c1* is being processed. Therefore the claim of *c1* that the new value for $e.f$ is y no longer is true, and *c1* is considered deprecated. If the triggered transformation rule relies on the event information, then it might propagate deprecated information.

4.1.1 Context-local State-Based Transformations

One simple way to avoid propagating deprecated information is to only propagate the current model state, ignoring the value information provided by the events. We can derive the local context that has been modified by a change from the change event, and we therefore know which elements might have to be synchronized. Take again the example change *c1 = replace(e.f, x → y)*. We know that *c1* modified the feature $e.f$. Therefore the triggered transformation has to restore all consistency constraints that reference $e.f$ and we can retrieve the current value from the model, which provides us with the information $e.f = x$, because of the effect of *c2*. The resolution of *c1* propagates the information $e.f = x$, event though the event description of *c1* claims $e.f = y$. We thereby avoid propagating deprecated information.

After the local context of an element and feature has been synchronized as the result of a change event, any subsequent change event that affects the same element and feature, whose change was already applied, becomes redundant, because its effects have already been propagated. But from a transformation rule's point of view, the redundant change events are indistinguishable from events that still need to be resolved. This possibly results in multiple propagations of the same information and unnecessary computation overhead. Another downside is that we cannot retrieve information about previous model states from the current state. As a result, cleanup routines may be ambiguous or computationally more expensive, because they rely on the detection of broken constraints and heuristically determining the appropriate action.

This approach is often at least partially necessary, irrespective of deprecated change events, because consistency constraints may require the examination of more context information than is provided by the individual change event's description.

4.1.2 Validity Check for Change Events

When a change event is deprecated and its effect description no longer matches the current state of the model that is affected by said change, then we can compare the change event description to the model state in order to detect that the change event is deprecated. The simplest example is that if a change event *c = set(e.f, z)* claims that value z has been set for feature f of element e, but the model state reveals that the value of $e.f$ is set to something other than z, then c is deprecated. We call such a check an *event validity check*, and how to perform the check is dependent of the change event. Changes that remove an element from a list have a different semantic and therefore also require a different check routine.

We can use event validity checks in order to detect deprecated change events and filter them out if necessary. Whenever we filter out deprecated events, we assume that at least one valid change event remains, because somehow the feature must have been set. This quasi enforces state-based propagation, thereby preventing propagation of deprecated information, because it ensures that the change event description is at least compatible with the present model state, and it reduces the number of repeated propagations of the same information in so far, as discarded change events no longer trigger transformations. We can get the benefit of accurate change events, which we would normally have using apply-on-resolve, and we can still detect the effects of yet unresolved change events, thereby having a chance to prevent the propagation of conflicting changes. Additionally, if a transformation relies on change application order, this order can still be reconstructed from the change events.

Depending on the used change types, there are different ways their events can become deprecated, and how deprecated changes have to be handled can depend on the way change events are generated. For some changes it is unclear what the appropriate response to a deprecated event is. Take following example: element e has a single-valued feature f that is set to value x, and is now overwritten to hold the value y. The transformation engine could generate different change events: *set(e.f, y)*, *unset(e.f, x)*, *replace(e.f, $x \rightarrow y$)* as a combinations thereof, or all of these change events. The *set* event provides the new information that needs to be propagate. The *unset* event provides information about the previous state, which may be helpful for cleanup. The *replace* provides both of the above, but it can be partially invalid as a result of the combination of information, because the old value may have been overwritten, but the new value is no longer up-to-date. For example in the change sequence *c1 = replace(e.f, $x \rightarrow y$)*, *c2 = replace(e.f, $y \rightarrow z$)*, *c1* correctly states that x has been overwritten and may require cleanup, but y no longer provides the correct current model state. We could therefore treat *c1* as if it represented a valid *unset(e.f, x)* and a deprecated *set(e.f, y)*. The value y was only temporarily set and it is arguable, whether the transitional state may require cleanup. We propose preserving events with information about past transitional states, to keep all possible intervention options available to the transformation developer.

In Table 4.1, we provide a list of predicates to check whether or not a change event is valid, for a selected set of possible change-event definitions. Event validity checks for *create* and *delete* events are unnecessary, because the absolute nature of existence versus non-existence leaves little room for such events to become deprecated in the first place. A *delete* change cannot be deprecated, because any element that could replace the previously

change event c	`isDeprecated(c)`
element existance changes	
`create(element)`	N/A
`delte(element)`	N/A
single valued feature changes	
`set(element.feature, newValue)`	`(element.feature != newValue)`
`unset(element.feature, oldValue)`	`(element.feature == oldValue)`
`replace(element.feature, oldValue, newValue)`	`((element.feature != newValue) ∨ (newValue == oldValue))`
list feature changes	
`insert(element.feature, newValue)`	`!(element.feature.contains(newValue))`
`remove(element.feature, oldValue)`	`(element.feature.contains(oldValue))`

Table 4.1: Change event validity checks.

existing one, is still a new and distinct element. For a *create* change, we could argue that it is deprecated by a subsequent delete, however, then we should no longer be able to reference the created element in the first place, and therefore the changes to that element would all have to have been discarded. We therefore assume that event validity checks for *create* and *delete* events are unnecessary.

4.1.3 Change Consolidation by the Transformation Engine

Change consolidation is the process of consolidating changes or change events that effect each other in order to reduce the number of individual changes. It cannot be implemented by the transformation, because each transformation (rule) can only see the one change event by which it has been triggered. As a result, this pattern has to be implemented by the transformation engine, as a preparatory step before the transformation rules are actually executed.

Using the introductory example change sequence *c1 = replace(e.f, x → y), c2 = replace(e.f, y → x)* again, we can reduce both changes based on their effects because they both operate on the same element. From *c1* and *c2*, we can derive *replace(e.f, x → y → x) = replace(e.f, x → x)* and replacing x through x effectively does nothing, so the event is unnecessary. Effectively no change occurred between the model state before *c1* and after *c2*. If both transformations triggered by the replace events would be processed free of intended side effects and we do not require cleanup for the transitional value y in other models, then we can safely discard both changes. However, we don't know if that is generally the case.

Now assume a slightly different change sequence, where the changes are not the inverse of each other: *replace(e.f, x → y), replace(e.f, y → z).* We can reduce the sequence to a single change *replace(e.f, x → z).* This again discards the information that y was temporarily set, which would be preserved with event validity checks, but in contrast to the purely

state-based transformation in Section 4.1.1, we at least preserve the information about the original model state.

Change consolidation prevents the propagation of deprecated information, and it maximally reduces the number of events that have to be processed and with it the number of repeated propagations of the same context.

4.1.4 Summary regarding Deprecated Change Events

We propose using event validity checks to discard events that describe a deprecated model state (e.g. *set(e.f, deprecatedValue*). As a result only the current model state is propagated and the number of repeated propagations of the same context is potentially reduced. We further propose preserving events with information about past transitional states (e.g. *remove(e.f, transitionalValue*), to keep all possible intervention options available to the transformation developer.

Well developed change consolidation strategies may be advantageous if transitional states prove unproblematic, and it is therefore a question for further research.

4.2 Avoiding Element-Creation Conflicts

As we discussed in Section 3.4, in a transformation network with transitive change propagation, we can have confluence of information along multiple paths. Therefore the information that would have to be propagated by a transformation may already be present in the target model, either through some other propagation path or because the currently processed source change was the result of a previous change in the target model, and is now being evaluated in the backpropagation direction. If the triggered transformation does not first check whether the target model already contains the information, then it might produce conflicting changes in the target model.

Intended feature changes by the user for single-valued features are unproblematic, as long as the transformations are compatible and produce the same value along multiple paths, because a second change that reaches the same element and sets the same value does not modify the model state and therefore no further changes are generated. As a result, the propagation process can converge to the same value and terminate. However, if an element is instantiated through user input, the transformation tries to propagate the metamodel default-values, and that default-value finds a propagation path back to the user input, then it may unintentionally overwritten an intentional user input. The propagation may still converge and terminate, but the intended input is still lost. Such feature change conflicts are hard to detect, because overwrites can be necessary to restore consistency after a user change, and a single transformation cannot differentiate whether the incoming change represents an intended new value or a conflicting value propagated along another path.

By contrast, conflicting changes that create (and insert) a new element in the target model cannot converge, because any new element is a unique instance. It either replaces the previous element in a single-valued feature or it results in an additional (unintended) entry in a list feature. Both scenarios generate new changes, even if the replaced element

is equal to the new one, and may leading to a propagation loops if subsequent element creations also replace and duplicate. However, it is usually not intentional to replace one element through a new and unique but semantically equivalent instance, or to fill a list with semantically equivalent instances (duplicates). We interpret such occurrences as element-creation change conflicts. And we expect the transformation that would produce the element-creation change conflicts to identify if the target element already exists and if it exists, then the transformation should adapt its feature values in order to achieve model consistency, instead of replacing or duplicating it. Because the existence of an element is absolute and cannot be deprecated (only its feature values can be deprecated) it is easier to detect if a transformation might produce an element-creation change conflict than to detect if a transformation might produce an feature change conflict.

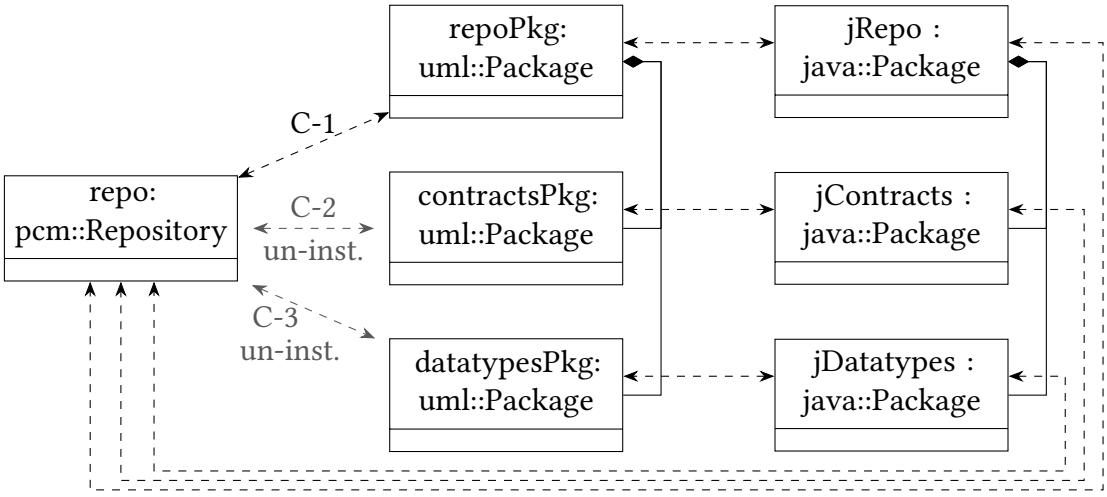
We focused on a solution to prevent creation change conflicts, because they can more easily produce serious propagation failures, while also being better detectable than feature change conflicts.

4.2.1 Element Existence Check

The simplest example for an element creation that leads to a creation loop is when a bidirectional transformation creates a new element in the target model that corresponds to the element created by the user, and the transformation then does not check if the user element already exists on the change backpropagation. Now a duplicate element exists on the original input side and the ping-pong continues. The solution to this scenario is obvious; the transformation has to check in either direction whether the corresponding element already exists. We call this straight forward pattern **existence check**.

In the simple bidirectional scenario above, this can easily be accomplished by checking the trace model to determine whether or not the correspondence to the target element exists. In general, however, the element may have been created in the target model without the instantiation of the direct correspondence linking the source and the target elements, for example if the target element was created along another transformation path. In that case, we have to explore additional sources of information in order to find the target element. This includes correspondences of the source model context, the target model context and global trace information (correspondence paths through domains foreign to the transformation that performs the check). If we can retrieve the element, then it exists, and we can instantiate the direct correspondence for faster and unambiguous subsequent retrievals, followed by the necessary synchronization. Only if it cannot be retrieved, then the transformation has to create the target element itself.

In the following subsections we discuss how best to design the element retrieval, based on generic considerations, and we rank-order the different retrieval strategies, according to their reliability to find the intended element. We use the example shown in Figure 4.1 to explore how the different retrieval strategies might work, so that a transformation rule, which is triggered by the change that created the *pcm::Repository repo*, can detect the existence of the *uml::Package contractsPkg* in order to avoid creating a duplicate. Assume that the same change first propagates from PCM to Java and then further to UML. All elements are already instantiated as a result, but the correspondences between *contractsPkg* and *repo*, and between *contractsPkg* and *repo* are still missing.



* C-1: repo.name == repoPkg.name
* C-2: contractsPkg.name == "contracts"
* C-3: datatypesPkg.name == "datatypes"

Figure 4.1: This is an extended correspondence graph for a *pcm::Repository* mapping and its corresponding *uml::Packages* and *java::Packages*. The example includes a direct transformation from PCM to Java, and possible correspondences that would be relevant, which were not part of the case study.

4.2.1.1 Direct Element Retrieval via Correspondence

In most cases, we can use trace information to navigate between two domains and, in our case with correspondences, we should be able to directly retrieve the target element if the correspondence exists. So a first retrieval attempt could be trying to directly retrieve the element from the correspondence model. In many cases it is enough to filter the existent correspondences by the type of the searched element, but if that is not enough, then we can annotate the correspondences with their exact correspondence type. This allows the exact retrieval of the target elements based on the correspondence type definition, if the correspondence exists and the element cannot participate in multiple instances of the same correspondence type. The set of *contractsPkg*-candidates in the example would be empty, because the direct correspondence to the *contractsPkg* does not yet exist, but in the fully instantiated correspondence graph we could retrieve it.

However, as we just discussed, changes can potentially reach the target model along other paths, and may have already instantiated the target element (*contractPkg*) without instantiating the correspondence type. In that case we cannot retrieve the target element through direct correspondences, but at the same time we cannot yet conclude that the element doesn't already exist.

4.2.1.2 Context-based Element Retrieval

We can instead try to heuristically retrieve the element from the context, by limiting the number of elements that could potentially be the correct corresponding element. If the

element has a container in the target model, which we can retrieve, then we can navigate the context and filter the candidates based on their feature values.

In general, a transformation only has to create an element if its existence is necessary in order to restore a consistency constraint. Then the consistency constraint definition has to prescribe where the element should be inserted in the model, which is the target context. Now that we have the context, we can then see if any of the elements in that context already fulfills the consistency constraint, but the correspondence is just not instantiated yet. The element that fulfills the consistency constraint is the target element. However, depending on the correspondence graph structure, the target element may not yet be fully/correctly initialized, because some values may only be set along a specific correspondence path. In that case, we cannot use the consistency constraint in its full definition. Instead, if we can identify a feature that has to be unique among the candidates and it matches the consistency constraint, then we can, with relative confidence, uniquely identify the target element if it exists.

In our example we can navigate to the main *uml::Package repoPkg* for the *pcm::Repository*, and search its nested-packages-feature for possible candidates. The name of the nested packages can be used as a unique and therefore identifying feature, and the consistency constraints require the contracts package to be called "contracts". So if there exists a *uml::Package* nested within the *repoPkg*, which is named "contracts", then we can be relatively sure, that it is the correct target element. But without the constraint on the name, we might have also found the *datatypesPkg*, or any other sub-package of *repoPkg* that is not depicted in the correspondence graph, as a potential *contractsPkg*-candidates.

4.2.1.3 Indirect Element Retrieval via Correspondence Graph

If we have no direct way of retrieving the target's context, we might instead be able to transitively resolve the instantiated correspondences and retrieve the target element that way. This assumes, that if the target is created by another transformation path, then it might participate in a correspondence types defined between another set of metamodels. And if the transformation graph has a cycle and the semantic interpretation of the element stays the same, then the representations should be at least structurally similar along both paths. Otherwise we could never achieve network-wide consistency, because the consistency constraints would contradict each other. In the example, we would retrieve *{jRepo, jContracts, jDatatypes, repoPkg, contractsPkg, datatypesPkg, repo}*, restricted to the target metamodel and target element type, this still leaves *{repoPkg, contractsPkg, datatypesPkg}*.

Here, differently than with the retrieval from direct correspondences, we cannot rely on the correspondence types. For one, the correspondence types defined for other binary transformations are supposed to be independently developed and therefore unknown to the transformation rule, which we try to apply. Secondly, even if the correspondence types where known, we would have to find a deterministic way of classifying the correspondence path and match it to a direct correspondence type, without prior knowledge about which transformations will be combined. But with a set of candidates, we can again try to constrain them based on features.

Def. **retrieveOther(s : Element, corr : CorrepondenceType)**:

```
1. candidateSet ⟵ directRetrieval(s, corr)
2. if(candidateSet.isEmpty) then :
3.   contextCandidates ⟵ contextRetrieval(s, corr)
4.   indirectCandidates ⟵ indirectRetrieval(s, corr)
5.   candidateSet ⟵ intersect(contextCandidates, indirectCandidates)
6.   if (candidateSet.isEmpty) then :
7.     candidateSet ⟵ contextCandidates
8.   if (candidateSet.isEmpty) then :
9.     candidateSet ⟵ indirectCandidates
10. if (candidateSet.size == 1) return candidateSet.first
11. else if (candidateSet.size == 0) return null
12. else return userDisambiguate(candidateSet)
```

Table 4.2: Combined element-retrieval pseudo-code for the Existence Check pattern.

4.2.1.4 Combination of Retrieval Methods

Now that we have described three ways of retrieving a target element for a specific correspondence type, we combine them in order to improve the reliability. The direct retrieval via the correspondence instance of the searched correspondence type, retrieves exactly the one target element we search, if the correspondence exists. This is enough of a check, if we are only looking at a single binary transformation, because then both the forward and the backward transformation would instantiate the correspondence as soon as all relevant elements exist. But with transformation networks, there are other paths along which a change can propagate and therefore the correspondence does not have to be instantiated. Direct retrieval is precise, because it only returns one element, but it may be insufficient to show the absence.

Context retrieval tries to search the context of where the target element would be if it existed. The idea is that in order for an element duplication to occur, we have to create and insert the duplicate into the target model. So if we have a place where we would insert it, then we can check if a matching element already exists. But in some cases, there may be ambiguity as to which element is the target element. Through the consistency constraints and sometimes because the metamodel definition, inappropriate candidates can be filtered out. We cannot generally avoid the possibility of multiple candidates, as that is often consistency constraint specific. However, if the target element exists, then it is contained in the set of candidates, and if we can find a key feature specification for the constraint, then we can identify the correct candidate.

Similarly, indirect element retrieval, by transitively navigating the existing correspondences, can produce a set of candidates. But unlike with the context retrieval, it does not guarantee that only elements in the context are found, because other transformations

could impose other constraints and produce similar elements, that are still not the intended target. Also, depending on consistency constraints of the other transformations, the target element may not be reachable along any other path of the correspondence graph than through the missing direct correspondence. The advantage of this method is that it can avoid user interaction, if the context of the searched element cannot be determined any other way.

Because of the advantages and problems we just outlined, we suggest to combine these retrieval methods as shown in the pseudo code example in Table 4.2. The direct retrieval is precise and should therefore be considered first. If it does not find the element, we should attempt the context retrieval, and we can try to limit the number of found candidates by calculating the intersection between the set of candidates found via context retrieval and the set found via indirect retrieval. If this intersection turns out to be empty, then we can check the context retrieval set on its own. Whether the indirect retrieval set should be considered as a standalone last option is questionable, because it is theoretically possible to find elements completely outside the context of the target element. If multiple possible candidates remain, at any step along this elimination process, then we should defer the disambiguation between the elements to the user, in order to avoid incorrect correspondences.

5 Correspondence Type Definitions between PCM, UML and Java

The case study that is used for the evaluation of this thesis uses two binary and bidirectional transformations:

- The transformation $T_{\text{PCM}\leftrightarrow\text{UML}}$ between the Palladio Component Model (PCM) domain and the Unified Modeling Language (UML) domain.
- The transformation $T_{\text{UML}\leftrightarrow\text{Java}}$ between the UML domain and the Java domain, which is modeled by the JaMoPP metamodel.

Both transformations are implemented using the Vitruvius framework and the Vitruvius Reactions Language. The domains and framework are explained in the foundations-chapter, Section 2.5.

The transformation $T_{\text{PCM}\leftrightarrow\text{UML}}$ was developed in the context of this thesis, based on the consistency mappings described by Langhammer ([12], p.68-77). For the case study, it is combined with the transformation $T_{\text{UML}\leftrightarrow\text{Java}}$, which was already implemented by Chen [3], in order to evaluate the behavior and synchronization failures of this minimal transformation network.

The PCM domain was limited to PCM Repository models, which represent repositories for component based software development. This limitation excludes PCM models for system assembly, component deployment and behavior analysis. And the UML domain was limited to UML class diagrams, explicitly excluding all other diagram types. Similarly, the synchronization scope of the JaMoPP metamodel was limited to relatively basic java constructs, excluding for example nested classifiers and generic class and method definitions.

This chapter provides a detailed list of the correspondence type definitions and the underlying consistency constraints, split up according to the transformations that implement these consistency constraints. We introduce correspondence types in Section 3.1, as formalizations for possible types of semantic overlap between elements of different domains. Table 5.1 shows how we define correspondence types. The consistency constraints associated with each correspondence type specify what constitutes consistency for elements that participate in correspondences of that type, thereby also defining how elements have to be mapped by the transformations.

This list of the correspondence type definitions has a dual purpose. By explicitly defining each implemented correspondence type, we invite the reader to decide for themselves, whether or not the used notion of consistency is appropriate with regard to the used metamodels. Secondly, the correspondence types will be the main unit of measure for counting errors in the evaluation. In order for the reader to follow the argumentation for

<Correspondence-Type-Name>	<role-labels and metamodel types of elements that can participate in correspondences of this type>
<abbreviated description>	<consistency constraint specification>
⋮	⋮
<abbreviated description>	<consistency constraint specification>

Table 5.1: The syntax we use to define correspondence types. Each correspondence type is assigned a unique name, and the participating classes are given role labels, by which they can be referenced in the consistency constraint specifications.

or against an error classification, we need to reference the correspondence type that could not be properly synchronized.

5.1 PCM↔UML Correspondence Type Definitions

In this section, we lists the correspondence type definitions relevant to the transformation $T_{\text{PCM}\leftrightarrow\text{UML}}$. The correspondence types are grouped by the concepts that are represented through the elements in the PCM domain, so that we can discuss the structural differences of how the concept is represented and why a concept may need multiple correspondence types to be represented in the opposite domain.

5.1.1 Concept: Component Repository

A component repository for a component based software system architecture contains datatype definitions, architectural interfaces, and component definitions, which a software architect can then use to compose and analyze a software system. These artifacts can be stored in different packages of a code repository, which lead to the correspondence types defined in Tables 5.2, 5.3, and 5.4.

5.1.2 Concept: Data Types

In order to define contractual interfaces between components, the types of information, that can be passed through an interface's signatures, have to be defined. To that end, a *pcm::Repository* provides datatype definitions, which need to be resolvable to *uml::Types* with the same meaning.

5.1.2.1 Concept: Primitive Type

The *pcm::PrimitiveDataType* represent minimal chunks of meaning and the number of primitive types is limited by the metamodel. As a result it is not necessary for a user to re-

PU-RepositoryPkg	pRepo : pcm::Repository ~ uRepoPkg : uml::Package
same name	lowerCase(pRepo.name) = uRepoPkg.name
contracts package	pRepo ~$_{\text{PU-ContractsPkg}}$ uContractsPkg (injective), with uContractsPkg ∈ uRepoPkg.nestedPackages
datatypes package	pRepo ~$_{\text{PU-DatatypesPkg}}$ uDatatypesPkg (injective), with uDatatypesPkg ∈ uRepoPkg.nestedPackages
component mapping	pComp ~$_{\text{PU-ComponentPkg}}$ uComponentPkg (injective), with pComp : pcm::RepositoryComponent ∈ pRepo.components, uCompPkg : uml::Package ∈ uRepoPkg.nestedPackages

Table 5.2: Correspondence type definition for the semantic overlap between a *pcm::Repository* and the *uml::Package* that represents the main package of the repository.

PU-ContractsPkg	pRepo : pcm::Repository ~ uContractsPkg : uml::Package
package name	uContractsPkg.name = "contracts"
interface mapping	pI ~$_{\text{PU-Interface}}$ uI (bijective), with pI : pcm::OperationInterface ∈ pRepo.interfaces, uI : uml::Interface ∈ uContractsPkg.ownedElements

Table 5.3: Correspondence type definition for the semantic overlap between a *pcm::Repository* and the *uml::Package* that represents the contract package of the repository.

PU-DatatypesPkg	pRepo : pcm::Repository ~ uDatatypesPkg : uml::Package
package name	uDatatypesPkg.name = "datatypes"
primitive type mapping	pPT ~$_{\text{PU-PrimitiveType}}$ uPT, with pPT ∈ PCM standard primitive types, uPT ∈ UML standard primitive types
composite type mapping	pCD ~$_{\text{PU-CompositeType}}$ uCD (bijective), with pCD : pcm::CompositeDataType ∈ pRepo.datatypes, uCD : uml::Class ∈ uDatatypesPkg.ownedElements

Table 5.4: Correspondence type definition for the semantic overlap between a *pcm::Repository* and the *uml::Package* that represents the datatype package of the repository.

PU-PrimitiveType	pT : pcm::PrimitiveDataType ~ uT : uml::PrimitiveType
no constraints!	Externally defined types are final and only mapped, not synchronized.

Table 5.5: Correspondence type definition for the semantic overlap between a *pcm::PrimitiveDataType* and a corresponding *uml::PrimitiveType*. The PU-PrimitiveType does not impose constraints, because at some point minimal concepts cannot be further broken down, but instead have to be mapped. Therefore, these correspondences function as tuples in the mapping relation.

PU-CompositeType	pT : pcm::CompositeDataType ~ uT : uml::Class
same name	pT.name = uT.name
attribute mapping	pAtt $\sim_{\text{PU-Attribute}}$ uCD (bijective), with pAtt : pcm::InnerDeclaration ∈ pT.innerDeclarations, uAtt : uml::Property ∈ uT.ownedAttributes
parent type mapping	pParent $\sim_{\text{PU-CompositeType}}$ uParent (bijective), with pParent : pcm::CompositeDataType ∈ pT.parentTypes, uParent : uml::Class ∈ uT.generalizations.general

Table 5.6: Correspondence type definition for the semantic overlap between a *pcm::CompositeDataType* and its implementing *uml::Class*.

define the same primitive types for every repository. Instead, the user can reuse predefined *pcm::PrimitiveDataType* instances, that are provided through an externally defined primitive datatype repository. Similarly, there are reusable predefined primitive type models for UML, albeit the UML metamodel does not limit the number of unique primitive types. We chose to use two such externally provided models with primitive type definitions, and only map the elements to each other, instead of using manually created instances. Because the prescribed primitive type mapping is relevant as soon as a *pcm::Repository* is instantiated, and the consistency constraints that define the mapping are predicated on the existence of an element in the containment hierarchy, the constraints are enforced by the *PU-DatatypesPkg* correspondence type. The only function of correspondences defined by Table 5.5 is to track tuples in the the mapping relation.

5.1.2.2 Concept: Composite Type and Attribute

Composite data types can be created through composition of other data types. Such composite data types can be represented in both domains, through *pcm::CompositeDataType* and through *uml::Class* (Table 5.6). Internally each attribute of a composite data type needs to be mapped between the domains. The correspondence type for that semantic overlap is defined in Table 5.7.

PU-Attribute	pAtt : pcm::InnerDeclaration ~ uAtt : uml::Property
same name	pAtt.name = uAtt.name
type correspondence	((pAtt.datatype $\sim_{\text{PU-PrimitiveType}}$ uAtt.type $\vee$ pAtt.datatype $\sim_{\text{PU-CompositeType}}$ uAtt.type) $\wedge$ uAtt.multiplicity=(1..1)) $\veebar$ (pAtt.datatype $\in$ pcm::CollectionDataType $\wedge$ pAtt.datatype $\sim_{\text{PU-CollTypeProp}}$ uP)

Table 5.7: Correspondence type definition for the semantic overlap between a *pcm::InnerDeclaration* and a *uml::Property*. Multiplicity is written as <lower-bound>..<upper-bound> where "*" stands for "unlimited".

PU-CollTypeParam	pCD : pcm::CollectionDataType ~ uP : uml::Parameter
collection multiplicity	uP.multiplicity=(0..*)
type correspondence	pCD.innerType $\sim_{\text{PU-PrimitiveType}}$ uP.type $\vee$ pCD.innerType $\sim_{\text{PU-CompositeType}}$ uP.type

Table 5.8: Correspondence type definition for the semantic overlap between a *pcm::CollectionDataType* and a *uml::Parameter*, whose multiplicity and type together represent the same collection type. Multiplicity is written as <lower-bound>..<upper-bound> where "*" stands for "unlimited".

5.1.2.3 Concept: Collection Type

Collection data types represent collections with arbitrary multiplicity of instances of an inner type. In the PCM, such types are represented by a *pcm::CollectionDataType*, that intentionally does not provide a more detailed specification, in order to have a high level of abstraction from the actual implementation on the software architecture level. In a UML domain, such a type can be expressed through the instantiation of a generic type, to form a concrete type, that can then be referenced by *uml::Parameters* or *uml::Properties* (in the role of attributes). Alternatively, collection typed parameters and attributes can be expressed through their multiplicity. In that case, a parameter or attribute shares a semantic overlap with a *pcm::CollectionDataType*, as defined by the correspondence types in Tables 5.8 and 5.9. We chose the latter option, for compatibility to the existing UML-Java transformation and to preserve the abstraction from references to concrete collection implementations.

5.1.3 Concept: Interface

Components communicate via contractual interfaces and the signatures therein. These concepts can be relatively similarly be expressed in both the PCM and the UML, which leads to relatively natural one-to-one mappings.

PU-CollTypeProp	pCD : pcm::CollectionDataType ~ uP : uml::Property
collection multiplicity	uP.multiplicity=(0..*)
type correspondence	pCD.innerType $\sim_{\text{PU-PrimitiveType}}$ uP.type $\vee$ pCD.innerType $\sim_{\text{PU-CompositeType}}$ uP.type

Table 5.9: Correspondence type definition for the semantic overlap between a *pcm::CollectionDataType* and a *uml::Property*, whose multiplicity and type together represent the same collection type. Multiplicity is written as <lower-bound>..<upper-bound> where "*" stands for "unlimited".

PU-Interface	pI : pcm::OperationInterface ~ uI : uml::Interface
same name	pI.name = uI.name
signature mapping	pM $\sim_{\text{PU-Signature}}$ uM (bijective), with pM : pcm::OperationSignature $\in$ pI.signatures, uM : uml::Operation $\in$ uI.ownedOperations
parent interface mapping	pSupI $\sim_{\text{PU-Interface}}$ uSupI (bijective), with pSupI : pcm::OperationInterface $\in$ pI.parentInterfaces, uSupI : uml::Interface $\in$ uI.generalizations.general

Table 5.10: Correspondence type definition for the semantic overlap between a *pcm::OperationInterface* and a *uml::Interface*, which are relevant on the software architecture level.

The correspondence type for an interface, which is relevant on the software architecture level, is defined between a *pcm::OperationInterface* and a *uml::Interface* and is provided by Table 5.10.

5.1.3.1 Concept: Signature and Parameter

The information flow through an interface is defined by the method signatures of the interface and the data types referenced by the parameters of those signatures. Signatures can be expressed directly in both PCM and UML and the correspondence type for that semantic overlap is defined by Table 5.11. But the way return values are represented is different in the two domains.

Where a *pcm::OperationSignature* has a feature for its return type, a *uml::Operation* instead contains return parameters, in addition to its ordinary parameters. Therefore, a *pcm::OperationSignature* shares a semantic overlap with a *uml::Operation*, defined by Table 5.11, and that operation's return parameter, defined by Table 5.12 The roles of a *uml::Parameter* is differentiated by its "direction". All regular *uml::Parameters* (those with "IN", "IN_OUT", or "OUT" directions) are bijectively mapped to *pcm::Parameters*, and their directions are mapped to the *pcm::Parameters*' modifiers. This correspondence type is defined by Table 5.13.

PU-Signature	pM : pcm::OperationSignature ~ uM : uml::Operation
same name	pM.name = uM.name
regular parameter mapping	pP $\sim_{\text{PU-RegularParam}}$ uP (bijective), with pP : pcm::Parameter ∈ pM.parameters, uP : uml::Parameter ∈ uI.ownedParameters ∧ uP.direction ≠ RETURN
return parameter mapping	pP $\sim_{\text{PU-ReturnParam}}$ uP (injective), with pP : pcm::Parameter ∈ pM.parameters, uP : uml::Parameter ∈ uI.ownedParameters ∧ uP.direction = RETURN

Table 5.11: Correspondence type definition for the semantic overlap between a *pcm::OperationSignature* and a *uml::Operation*.

PU-ReturnParam	pM : pcm::OperationSignature ~ uP : uml::Parameter
parameter name	uP.name = "returnParam"
parameter direction	uP.direction = RETURN
type correspondence	((pM.returnType$\sim_{\text{PU-PrimitiveType}}$ uP.type ∨ pM.returnType$\sim_{\text{PU-CompositeType}}$ uP.type) ∧ uP.multiplicity=(1..1)) ⩒ (pM.returnType∈ pcm::CollectionDataType ∧ pM.returnType$\sim_{\text{PU-CollTypeParam}}$ uP)

Table 5.12: Correspondence type definition for the semantic overlap between a *pcm::OperationSignature* and the return *uml::Parameter* of its corresponding *uml::Operation*.

PU-RegularParam	pP : pcm::Parameter ~ uP : uml::Parameter
same name	pP.name = uP.name
parameter direction	uP.direction ≠ RETURN ∧ (pP.modifier, uP.direction) ∈ pcm-uml modifier mapping
type correspondence	((pP.datatype $\sim_{\text{PU-PrimitiveType}}$ uP.type ∨ pP.datatype $\sim_{\text{PU-CompositeType}}$ uP.type) ∧ uP.multiplicity=(1..1)) ⩒ (pP.datatype ∈ pcm::CollectionDataType ∧ pP.datatype $\sim_{\text{PU-CollTypeParam}}$ uP)

Table 5.13: Correspondence type definition for the semantic overlap between a *pcm::Parameter* and a *uml::Parameter*.

PU-ComponentPkg	pComp : pcm::RepositoryComponent ~ uPkg : uml::Package
same name	firstToLowerCase(pComp.name) = uPkg.name
implementation mapping	pComp ~$_{\text{PU-ComponentImpl}}$ uImpl (injective), with uImpl : uml::Class ∈ uPkg.nestedPackages

Table 5.14: Correspondence type definition for the semantic overlap between a *pcm::RepositoryComponent* and a component *uml::Package*.

PU-ComponentImpl	pComp : pcm::RepositoryComponent ~ uImpl : uml::Class
implementation name	pComp.name + "Impl" = uImpl.name
implementation is final	uImpl.isFinal
constructor mapping	pComp ~$_{\text{PU-ComponentConstr}}$ uConstructor (injective), with uConstructor : uml::Operation ∈ uImpl.ownedOperations
required role mapping	pRequired ~$_{\text{PU-RequiredAtt}}$ uAtt (injective), with pRequired : pcm::RequiredRole ∈ pComp.requiredRoles, uAtt : uml::Property ∈ uImpl.ownedAttributes
provided role mapping	pProvided ~$_{\text{PU-Provided}}$ uIR (injective), with pProvided : pcm::ProvidedRole ∈ pComp.providedRoles, uIR : uml::Realization ∈ uImpl.interfaceRealizations
assembly context mapping	(pComp *instanceOf* pcm::ComposedProvidingRequiringEntity) ⇒ pAC ~$_{\text{PU-ACProp}}$ uProp (injective), with pAC : pcm::AssemblyContext ∈ pComp.assemblyContexts, uAtt : uml::Property ∈ uImpl.ownedAttributes

Table 5.15: Correspondence type definition for the semantic overlap between a *pcm::RepositoryComponent* and an implementation *uml::Class*.

5.1.4 Concept: Component

A software component is an encapsulated unit of software that designed for blackbox reuse and composition. As part of the blackbox principle, a component implementation is encapsulated in its own component package, together with the rest of the supporting code that is required for the component implementation. And a component needs a constructor that provides a signature for proper initialization of the component. Because a constructor is not explicitly modeled in the UML class diagram, we instead use a *uml::Operation*, and we have to enforce that the name of the operation matches the *uml::Class* that represents the implementation through consistency constraints. Therefore, a *pcm::RepositoryComponent* shares a semantic overlap with a *uml::Package* (Table 5.14), a *uml::Class* (Table 5.15), and a *uml::Operation* (Table 5.16).

PU-ComponentConstr	pComp : pcm::RepositoryComponent ~ uConstr : uml::Operation
implementation name	pComp.name + "Impl" = uConstr.name
required role mapping	pRequired $\sim_{\text{PU-RequiredParam}}$ uParam (injective), with pRequired : pcm::RequiredRole ∈ pComp.requiredRoles, uParam : uml::Parameter ∈ uConstr.ownedParameters
assembly context mapping	(pComp *instanceOf* pcm::ComposedProvidingRequiringEntity) ⇒ pAC $\sim_{\text{PU-ACParam}}$ uParam (injective), with pAC : pcm::AssemblyContext ∈ pComp.assemblyContexts, uParam : uml::Parameter ∈ uConstr.ownedParameters

Table 5.16: Correspondence type definition for the semantic overlap between a *pcm::RepositoryComponent* and the *uml::Operation* constructor of the component implementation.

PU-Provided	pProvided : pcm::ProvidedRole ~ uIR : uml::Realization
contract correspondence	pProvided.interface $\sim_{\text{PU-Interface}}$ uIR.contract

Table 5.17: Correspondence type definition for the semantic overlap between a *pcm::ProvidedRole* of a component and the interface *uml::Realization* of the component implementation.

5.1.4.1 Concept: Provided- and Required-Role

The ability to compose software systems of multiple components is ensured through contractual interfaces between components, that specify what services are provided by a component and what services are required for the component to function. The providing role, a component engages in, is realized through the implementation of the provided interface. Therefore, a *pcm::ProvidedRole* shares a semantic overlap with a *uml::Realization*, as defined in Table 5.17.

A *pcm::RequiredRole* of a component *Comp1* represents the need for a service, provided by another component *Comp2*. The type of service that is required is specified through the required interface reference of the *pcm::RequiredRole*. In order for *C1* to properly delegate the calls to this service to the correct deployed component during execution, the implementation of *C1* needs a field to store the reference to *c2* and a constructor parameter to set said field upon initialization. Therefore, a *pcm::RequiredRole* shares a semantic overlap with a *uml::Property*, defined in Table 5.18, and a *uml::Parameter*, defined in Table 5.19.

PU-RequiredProp	pRequired : pcm::RequiredRole ~ uProp : uml::Property
same name	pRequired.name = uProp.name
single-valued	uParam.multiplicity = (1..1)
contract correspondence	pRequired.interface $\sim_{\text{PU-Interface}}$ uProp.type

Table 5.18: Correspondence type definition for the semantic overlap between a *pcm::RequiredRole* of a component and the *uml::Property*, which stores the component, whose services are required.

PU-RequiredParam	pRequired : pcm::RequiredRole ~ uParam : uml::Parameter
same name	pRequired.name = uParam.name
parameter direction	uParam.direction = IN
single-valued	uParam.multiplicity = (1..1)
contract correspondence	pRequired.interface $\sim_{\text{PU-Interface}}$ uParam.type

Table 5.19: Correspondence type definition for the semantic overlap between a *pcm::RequiredRole* of a component and the constructor *uml::Parameter* required for the component initialization.

PU-ACProp	pAC : pcm::AssemblyContext ~ uProp : uml::Property
same name	pAC.name = uProp.name
single-valued	uProp.multiplicity = (1..1)
component correspondence	pAC.innerComponent $\sim_{\text{PU-ComponentImpl}}$ uProp.type

Table 5.20: Correspondence type definition for the semantic overlap between a *pcm::AssemblyContext* and a *uml::Property*.

5.1.4.2 Concept: Composed Components and Assembly Context

Components can in turn be composed of components. In PCM, this is realized through *pcm::CompositeComponent* and *pcm::SubSystem*, which are both concrete subclasses of *pcm::ComposedProvidingRequiringEntity* and *pcm::RepositoryComponent*. Subclasses of *pcm::ComposedProvidingRequiringEntity* can contain *pcm::AssemblyContext* elements that represent instantiated internal components. Therefore, each *pcm::AssemblyContext* is mapped to a *uml::Property* with a type reference to the implementation of the internal component, as defined in Table 5.20.

The provided and required roles of the components represented by assembly contexts are linked via *pcm::AssemblyConnectors* among each other and delegated outwards to the containing composed component's provided and required roles via *pcm::ProvidedDelegateConnectors* and *pcm::RequiredDelegateConnectors* respectively. These connections define how the internal components need to be passed to each other on initialized, and how they are assigned to their respective assembly context properties. This information could be transformed to code of the composite component's constructor, but UML class diagrams cannot express method behavior. As a result, PCM connectors are not synchronized and therefore are not subject to mappings or constraints.

5.1.5 Unmapped Concepts

UML class diagrams do not support the specification of method behavior, but some PCM elements do contain abstract behavior descriptions. This disparity of expressible concepts leaves some aspects of a component repository without a mapped correspondence. Similarly UML can express concepts that are not required for the representation of a component repository.

The following PCM elements are not mapped to UML representations, because they represent method or constructor behavior:

- Service Effect Specification (SEFFs) – A SEFF allow the component developer to abstractly define the behavior of services provided by a component.
- *pcm::AssemblyConnector* – These define how provided and required roles of *pcm::AssemblyContexts* are delegated between contexts.
- *pcm::ProvidedDelegateConnectors* – These define how provided roles of *pcm::AssemblyContexts* are delegated to the provided roles of the containing component.

- *pcm::RequiredDelegateConnectors* – These define how required roles of *pcm::AssemblyContexts* are delegated to the required roles of the containing component.

5.2 UML↔Java Correspondence Type Definitions

In this section, we lists the correspondence type definitions relevant to the transformation $T_{\text{UML}\leftrightarrow\text{Java}}$.

Because UML class diagrams are a tool for modeling object oriented systems and Java is a object oriented programming language, many of the concepts can be mapped quite naturally, but for limited structural variations. One difference between UML and Java is that a UML class diagram can represent a complete system in one model with a clear hierarchical containment structure between all relevant elements, whereas a Java code project consists of many separate files, which can be interpreted as separate models, and cross-references are solved through a prescribed package folder structures and string comparisons. The JaMoPP provides an EMOF compatible Java metamodel that allows clear navigation of single Java files, represented by *java::CompilationUnit*, but navigation across file boundaries are sometimes difficult. Therefore, we will use simplified expressions in such cases, instead of correctly reflecting the underlying metamodel or file system structure. This mainly impacts descriptions of *java::Package* and *java::TypeReference*.

Throughout the evaluation process, $T_{\text{UML}\leftrightarrow\text{Java}}$ was adapted multiple times in order to fix already documented synchronization failures and to reveal additional failures that were inhibited by more fundamental ones. An example for the necessity of such adaptations is that if an interface creation leads to a propagation loop, then we cannot study the synchronization of the inner signatures without first fixing the cause of the propagation loop. We provide the consistency constraints and correspondence types as they are defined after the adaptations of the evaluation.

5.2.1 Concept: Package

A *uml::Package* is mapped to *java::Package* and the semantic overlap is defined by the correspondence type in Table 5.21. In the Java metamodel, the nesting structure of packages is not represented through containment references, but rather through a list of namespaces. The UML models' containment references allow to derive an equivalent list. Apart from this difference, all contained elements are bijectively mapped, which includes enumerations, interfaces, classes and sub-packages.

5.2.2 Concept: Classifier

Both the UML and the Java metamodel define concepts for enumerations, interfaces and classes. These classifiers are directly contained in a package in a UML model. In the Java metamodel, each classifier $C1$ is either contained in a *java::CompilationUnit* with a name matching $C1$'s name, or within the containment hierarchy of another classifier $C2$, if $C1$ is a nested classifier. For simplicity, we do not map nested classifiers. Therefore a UML classifier maps to a CompilationUnit and the mapped Java classifier. And because a Java

UJ-Package	uPkg : uml::Package ~ jPkg : java::Package
same name	uPkg.name = jPkg.name
set namespaces	uPkg.namespace = jPkg.namespaces
nested package mapping	uNested ~$_{\text{UJ-Package}}$ jNested (bijective), with uNested : uml::Package ∈ uPkg.nestedPackages, jNested : java::Package ∈ jPkg.subpackages
class compilation-unit mapping	uC ~$_{\text{UJ-ClassCU}}$ jCU (bijective), with uC : uml::Class ∈ uPkg.ownedElements, jCU : java::CompilationUnit ∈ jPkg.compilationUnits
interface compilation-unit mapping	uI ~$_{\text{UJ-InterfaceCU}}$ jCU (bijective), with uI : uml::Interface ∈ uPkg.ownedElements, jCU : java::CompilationUnit ∈ jPkg.compilationUnits
enum compilation-unit mapping	uE ~$_{\text{UJ-EnumCU}}$ jCU (bijective), with uE : uml::Enumeration ∈ uPkg.ownedElements, jCU : java::CompilationUnit ∈ jPkg.compilationUnits

Table 5.21: Correspondence type definition for the semantic overlap between a *uml::Package* and a *java::Package*.

UJ-EnumCU	uE : uml::Enumeration ~ jCU : java::CompilationUnit
fully qualified name	uE.qualifiedName + ".java" = jCU.name
enum mapping	uE ~$_{\text{UJ-Enum}}$ jE (bijective), with jE : java::Enumeration ∈ jCU.classifiers

Table 5.22: Correspondence type definition for the semantic overlap between a *uml::Enumeration* and a *java::CompilationUnit* for the corresponding *java::Enumeration*.

classifier and its CompilationUnit share an implicit consistency constraint on their names, the uml classifier also shares a name constraint with the *java::CompilationUnit*, which we make explicit.

5.2.2.1 Concept: Enumeration

A *uml::Enumeration* is mapped to a *java::Enumeration* and the containing *java::CompilationUnit* and the semantic overlap are defined by the correspondence types in Tables 5.22 and 5.23. All enumeration literals are bijectively mapped and only overlap in their names, which is defined in Tables 5.24.

UJ-Enum	uE : uml::Enumeration ~ jE : java::Enumeration
same name	uE.name = jE.name
enum constant mapping	uEL $\sim_{\text{UJ-EnumConst}}$ jEC (bijective), with uEL : uml::EnumerationLiteral ∈ uE.literals, jEC : java::EnumerationConstant ∈ jE.constants

Table 5.23: Correspondence type definition for the semantic overlap between a *uml::Enumeration* and a *java::Enumeration*.

UJ-EnumLiteral	uEL : uml::EnumerationLiteral ~ jEC : java::EnumerationConstant
same name	uEL.name = jEC.name

Table 5.24: Correspondence type definition for the semantic overlap between a *uml::EnumerationLiteral* and a *java::EnumerationConstant*.

5.2.2.2 Concept: Interface

A *uml::Interface* is mapped to a *java::Interface* and the containing *java::CompilationUnit* and the semantic overlap are defined by the correspondence types in Tables 5.25 and 5.26. For interfaces both the super-interfaces and the signatures have to be mapped bijectively.

5.2.2.3 Concept: Class

A *uml::Class* is mapped to a *java::Class* and the containing *java::CompilationUnit* and the semantic overlap are defined by the correspondence types in Tables 5.27 and 5.28.

Classes can inherit/extend from super-classes. In UML multi-inheritance is allowed, whereas Java only allows single inheritance for classes. As a result, the bijective mapping of a *java::Class*' super-class also constrains the corresponding *uml::Class* to single inheritance. But in both domains, classes can realize multiple interfaces, which leads to a natural mapping

UJ-InterfaceCU	uI : uml::Interface ~ jCU : java::CompilationUnit
fully qualified name	uI.qualifiedName + ".java" = jCU.name
interface mapping	uI $\sim_{\text{UJ-Interface}}$ jI (bijective), with jI : java::Interface ∈ jCU.classifiers

Table 5.25: Correspondence type definition for the semantic overlap between a *uml::Interface* and a *java::CompilationUnit* for the corresponding *java::Interface*.

UJ-Interface	uI : uml::Interface ~ jI : java::Interface
same name	uI.name = jI.name
super interface reference mapping	uSuperRef ~$_{\text{UJ-SuperInterfaceRef}}$ jSuperRef (bijective), with uSuperRef : uml::Generalization ∈ uI.generalizations, jSuperRef : java::TypeReference ∈ jI.extends
method mapping	uM ~$_{\text{UJ-InterfaceMethod}}$ jM (bijective), with uM : uml::Operation ∈ uI.ownedOperations, jM : java::InterfaceMethod ∈ jI.members

Table 5.26: Correspondence type definition for the semantic overlap between a *uml::Interface* and a *java::Interface*.

UJ-ClassCU	uC : uml::Class ~ jCU : java::CompilationUnit
fully qualified name	uC.qualifiedName + ".java" = jCU.name
class mapping	uC ~$_{\text{UJ-Class}}$ jC (bijective), with jC : java::Class ∈ jCU.classifiers

Table 5.27: Correspondence type definition for the semantic overlap between a *uml::Class* and a *java::CompilationUnit* for the corresponding *java::Class*.

A second structural difference for classes is that Java explicitly models *java::Constructors*. We therefore have to differentiate contained *uml::Operations* by their name, and map them accordingly.

5.2.3 Concept: Inheritance and Realization

In both domains, the super-interface and super-class references are represented through separate elements in both models, *uml::Generalization* and *java::TypeReference* respectively, instead of a simple reference in the interfaces. Additionally, depending on the context a *uml::Generalization*/*java::TypeReference* has to point to an interface or a class in order to be conform to the metamodel. Therefore we define two correspondence types between a *uml::Generalization* and a *java::TypeReference* for the specific contexts (Tables 5.29 and 5.30).

The fact that a class realizes a specific interface is modeled similarly. The correspondence type for that is defined in Table 5.31, between a *uml::Realization* and a *java::TypeReference*.

5.2.4 Concept: Method

UML class diagrams do not represent method behavior, therefore, they only need to specify the method signature, which is done using *uml::Operation*. And *uml::Operations* are used for interface signatures and class methods alike. In Java these concepts are differentiated:

- A *java::InterfaceMethod* has to be public, but does not need to model implementation.

UJ-Class	uC : uml::Class ~ jC : java::Class
same name	uC.name = jC.name
synchronized modifiers	(uC.isFinal = jC.isFinal) ∧ (uC.isAbstract = jC.isAbstract) ∧ (uC.isAbstract ≠ uC.isFinal)
super class reference mapping	uCRef ~$_{\text{UJ-SuperClassRef}}$ jC.extends (bijective), with uCRef : uml::Generalization ∈ uC.generalizations
implemented interface reference mapping	uIRef ~$_{\text{UJ-ImplementsRef}}$ jIRef (bijective), with uIRef : uml::Realization ∈ uC.interfaceRealizations, jIRef : java::TypeReference ∈ jC.implements
constructor mapping	uM ~$_{\text{UJ-Constructor}}$ jM (bijective), with uM : uml::Operation ∈ uC.ownedOperations ∧ uM.name = uC.name, jM : java::Constructor ∈ jC.members
method mapping	uM ~$_{\text{UJ-ClassMethod}}$ jM (bijective), with uM : uml::Operation ∈ uC.ownedOperations ∧ uM.name ≠ uC.name, jM : java::ClassMethod ∈ jC.members
attribute mapping	uAtt ~$_{\text{UJ-Attribute}}$ jAtt (bijective), with uAtt : uml::Property ∈ uC.ownedAttributes, jAtt : java::Field ∈ jC.members

Table 5.28: Correspondence type definition for the semantic overlap between a *uml::Class* and a *java::Class*.

UJ-SuperInterfaceRef	uSuperRef : uml::Generalization ~ jSuperRef : java::TypeReference
super interfaces correspond	uSuperRef.general ~$_{\text{UJ-Interface}}$ jSuperRef.classifier

Table 5.29: Correspondence type definition for the semantic overlap between a *uml::Generalization* and a *java::TypeReference* that both represent a reference to a super interface.

UJ-SuperClassRef	uCRef : uml::Generalization ~ jCRef : java::TypeReference
super classes correspond	uCRef.general ~$_{\text{UJ-Class}}$ jCRef.classifier

Table 5.30: Correspondence type definition for the semantic overlap between a *uml::Generalization* and a *java::TypeReference* that both represent a reference to a super class.

UJ-ImplementsRef	uIRef : uml::Realization ~ jIRef : java::TypeReference
implemented interfaces correspond	uIRef.contract $\sim_{\text{UJ-Interface}}$ jIRef.classifier

Table 5.31: Correspondence type definition for the semantic overlap between a *uml::Realization* and a *java::TypeReference* that both represent a reference to an implemented interface.

UJ-InterfaceMethod	uM : uml::Operation ~ jM : java::InterfaceMethod
same name	uI.name = jI.name
interface methods are public	uM.isPublic ∧ jM.isPublic
ordinary parameter mapping	uParam $\sim_{\text{UJ-OrdinaryParam}}$ jParam (bijective), with uParam : uml::Parameter ∈ uM.ownedParameters ∧ uParam.direction = IN, jParam : java::OrdinaryParameter ∈ jM.parameters
return parameter mapping	uReturnParam $\sim_{\text{UJ-ReturnParam}}$ jM (bijective), with uReturnParam : uml::Parameter ∈ uI.returnParam

Table 5.32: Correspondence type definition for the semantic overlap between a *uml::Operation* and a *java::InterfaceMethod*.

- A *java::ClassMethod* can have different visibility modifiers, be abstract or final, and has to be able to represent some implementation.
- A *java::Constructor* is similar to a *java::ClassMethod*, but has no name of its own, because it automatically takes that of the containing *java::Class*.

We therefore define three separate correspondence types, each defining the semantic overlap between a *uml::Operation* and one of the above mentioned *java::Method*-sub-types, in Tables 5.32, 5.33 and 5.34.

Ignoring the differences, each of these three correspondence types uses the same mapping constraints for its method parameters and return type. Because *uml::Operations* represent the return type through a return parameter, whereas *java::Methods* have a reference "type" that represents that methods return type. Therefore, there exists a semantic overlap between the return *uml::Parameter* and the *java::Methods*. Additionally Java does not support output parameters with reference passing, as might be suggested by *uml::Parameter* with a direction modifier of "OUT" or "IN_OUT", and therefore this also limits the *uml::Parameter*'s direction to the enumeration value "IN".

UJ-ClassMethod	uM : uml::Operation ~ jM : java::ClassMethod
same name	uM.name = jM.name
matching visibility	uM.visibility matches jM.visibility
synchronized modifiers	(uC.isFinal = jC.isFinal) ∧ (uC.isAbstract = jC.isAbstract) ∧ (uC.isAbstract ≠ uC.isFinal)
ordinary parameter mapping	uParam $\sim_{\text{UJ-OrdinaryParam}}$ jParam (bijective), with uParam : uml::Parameter ∈ uM.ownedParameters ∧ uParam.direction = IN, jParam : java::OrdinaryParameter ∈ jM.parameters
return parameter mapping	uReturnParam $\sim_{\text{UJ-ReturnParam}}$ jM (bijective), with uReturnParam : uml::Parameter ∈ uI.returnParam

Table 5.33: Correspondence type definition for the semantic overlap between a *uml::Operation* and a *java::ClassMethod*.

UJ-Constructor	uM : uml::Operation ~ jM : java::Constructor
matching visibility	uM.visibility matches jM.visibility
ordinary parameter mapping	uParam $\sim_{\text{UJ-OrdinaryParam}}$ jParam (bijective), with uParam : uml::Parameter ∈ uM.ownedParameters ∧ uParam.direction = IN, jParam : java::OrdinaryParameter ∈ jM.parameters
return parameter mapping	uReturnParam $\sim_{\text{UJ-ReturnParam}}$ jM (bijective), with uReturnParam : uml::Parameter ∈ uI.returnParam

Table 5.34: Correspondence type definition for the semantic overlap between a *uml::Operation* and a *java::Constructor*.

5.2.5 Concept: Typed Element

Method parameters, methods themselves and class attributes are typed elements and their types need to be synchronized between UML and Java. In all three cases, what constitutes a consistent type assignment between corresponding elements is relatively similar, we therefore discuss the commonalities on the example of the attribute correspondence type defined in Table 5.35, between a *uml::Property uAtt* and a *java:Field jAtt*.

If *jAtt.type* references a classifier (enumeration, interface or class), then *jAtt.type* and *uAtt.type* are consistent if they participate in the respective correspondence.

Alternatively, if either element references a primitive type, like a *java::Integer* for example, then they are consistent if the primitive types match according to the implemented mapping. We do not model this primitive type mapping through correspondences, because in the Java metamodel primitive types are metamodel classes, which would imply a correspondence across modeling levels. Instead we define a mapping between the Java primitive type metamodel classes and *uml::PrimitiveType* instances that are defined in an externally model for standard UML primitive types, the same model, which we used in the PCM to UML transformation, for compatibility. Then *jAtt.type* and *uAtt.type* are consistent if the metamodel class of *jAtt.type* matches the *uml::PrimitiveType* instance of *uAtt.type*.

Because a *java:Field* need a type reference to produce a syntactically correct output in the actual .java-file, the *jAtt.type* should default to the classifier "java.lang.Object" if *uAtt.type* is unset.

In addition to the type reference, the *uml::Property uAtt* also has a multiplicity. To represent this in Java, the corresponding *java:Field*'s *jAtt* has to reference a classifier that implements or extends "java.lang.util.Collection" with a TypeArgument *T*, and now *uAtt.type* has to match *T* instead of *jAtt.type*. Technically, Java primitive type now have to be wrapped by their respective wrapper class, for example a primitive "int" now has to be wrapped as "java.lang.Integer" to function as a TypeArgument, but because this mapping is predefined for the Java domain, there is no transformation ambiguity, and we omitted this aspect in the correspondence type definitions.

The correspondence type UJ-OrdinaryParam between a *uml::Parameter* and a *java::OrdinaryParameter*, which is defined in Table 5.36, uses the same consistency constraints for the type references as in the attribute example. The correspondence type between a return *uml::Parameter* and a *java::Method*, which is defined in Table 5.37, uses similar consistency constraints, but "void" may be a more appropriate default type for a *java::Method*.

UJ-Attribute	uAtt : uml::Property ~ jAtt : java::Field
same name	uAtt.name = jAtt.name
same final value	(uAtt.isFinal = jAtt.isFinal)
types correspond	(uAtt.multiplicity=(1..1) ∧ *match*(uAtt.type, jAtt.type)) ∨ (uAtt.multiplicity=(0..*) ∧ jAtt.type *implements* Collection<T> ∧ *match*(uAtt.type, T)) with *match*(uT, jT) := (uT = NULL ∧ jT = OBJECT) ∨ ((uT, jT) ∈ primitive type mapping) ∨ (uT $\sim_{\text{UJ-Enum}}$ jT) ∨ (uT $\sim_{\text{UJ-Interface}}$ jT) ∨ (uT $\sim_{\text{UJ-Class}}$ jT)

Table 5.35: Correspondence type definition for the semantic overlap between a *uml::Property* and a *java::Field*.

UJ-OrdinaryParam	uP : uml::Parameter ~ jP : java::OrdinaryParameter
same name	uI.name = jI.name
ordinary parameter direction	uParam.direction = IN
types correspond	(uP.multiplicity=(1..1) ∧ *match*(uP.type, jP.type)) ∨ (uP.multiplicity=(0..*) ∧ jP.type *implements* Collection<T> ∧ *match*(uP.type, T)) with *match*(uT, jT) := (uT = NULL ∧ jT = OBJECT) ∨ ((uT, jT) ∈ primitive type mapping) ∨ (uT $\sim_{\text{UJ-Enum}}$ jT) ∨ (uT $\sim_{\text{UJ-Interface}}$ jT) ∨ (uT $\sim_{\text{UJ-Class}}$ jT)

Table 5.36: Correspondence type definition for the semantic overlap between a *uml::Parameter* and a *java::OrdinaryParameter*.

UJ-ReturnParam	uP : uml::Parameter ~ jM : java::Method
return parameter direction	uP.direction = RETURN
types correspond	(uP.multiplicity=(1..1) ∧ *match*(uP.type, jM.type, VOID)) ∨ (uP.multiplicity=(0..*) ∧ jM.type *implements* Collection<T> ∧ *match*(uP.type, T, OBJECT)) with *match*(uT, jT, jDefault) := (uT = NULL ∧ jT = jDefault) ∨ ((uT, jT) ∈ primitive type mapping) ∨ (uT $\sim_{\text{UJ-Enum}}$ jT) ∨ (uT $\sim_{\text{UJ-Interface}}$ jT) ∨ (uT $\sim_{\text{UJ-Class}}$ jT)

Table 5.37: Correspondence type definition for the semantic overlap between a return *uml::Parameter* and a *java::Method*.

6 Evaluation

The first goal of this thesis is the identification of failure potentials that can lead to model synchronization failures with regards to multi-model consistency. In Chapter 3 we identified six such failure potentials that can lead to synchronization failures and we provided at least abstract scenarios per failure potential, in which a synchronization failure manifests, as proof of concept.

In this evaluation, we want to answer the following questions regarding the identified failure potentials:

- How prevalent are the identified failure potentials relative to each other?
  - measured as the percentage of total failures
- How severe is the impact of a failure potential when a failure manifests?
  - relative severity estimate based on failure expression type (propagation loop / information loss / inconsistency)
- Can we explain all observed failures through the identified failure potentials?
- How distinct are the identified failure potentials with regard to failure expression and classification?
  - the number of different failure expression for a specific failure potential
  - the number of failures that required classification with more than one failure potential

Failure potentials can be accounted for by different means and their manifestation can be dependent on the occurrence of specific changes. Therefore we have to expect that not all failure potentials resulting in actual failures and instead remain undetected. *Concept bottlenecks* can be solved through the introduction of additional transformations into the transformation network. *Incompatible consistency constraints* are a specification problems regarding proper consistency in the designed system. When the system is correctly designed, there should not be any contradicting consistency constraints or structurally incompatible mappings (Section 3.7). Without contradicting consistency constraints, *feature change conflicts* should automatically converge, because then all propagation paths produce the same intended feature value. *Element-creation change conflicts* can be prevented with the *existence check* pattern, described in Section 4.2.1. Without change conflicts from transformations, *deprecated change events* only occur as a result of contradictory user input before a synchronization step or because models are initialized with different default-values based on the metamodels, and even those can be filtered out with the *validity check* pattern described in Section 4.1.2. This leaves *unsynchronizable states* and the *visibility of unresolved changes* as the only unaddressed failure potentials.

However, in the general scenario, we have to assume a faulty system specification. The transformation network can have concept bottlenecks, the implemented consistency constraints may contain unidentified contradictions and the implemented consistency constraints may not match the intended global consistency constraints. The question is, how many failure potentials remain in the system, which are not accounted for by the transformation network topology or the transformation implementation, and how severe is their impact on the correct operation of the system.

If we assume transformation implementations without trivial implementation errors, then not all failure potentials are unaddressed. Additionally, the models used for this evaluation do not test all possible metamodel concepts or possible model modification sequences. As a result, we cannot claim that the observed failures reveal all possible failure potentials (instances) of that particular system. We therefore discuss the prevalence and the impact of the failure potential types relative to each other, based on the number of observed failures that can be traced back to the individual failure potential type.

Furthermore, a model can contain multiple instances of the same concept, for example multiple *pcm::BasicComponents* or multiple *java:Classes*, and for each instance the same transformation rules apply with the same unsolved problems. This effectively inflates the number of observed failures per unaddressed failure potential by the number of concept instances present in the case study model. Therefore, we trace the failure expression back to the correspondence type that was incorrectly synchronized and the circumstances that produced the failure. Repeated failures that stem from the same circumstances are counted as one.

The second goal was the development of solutions to prevent the manifestation of the identified failure potentials. In Chapter 4, we developed the *element existence check* pattern to prevent element-creation change conflicts. And we developed the *event validity check* pattern to prevent information loss from unintentional overwrites and to prevent propagation loops after deprecated change events occur. In this evaluation, we want to answer the following questions regarding the identified patterns:

- To what degree can element-creation related failures be solved through *element existence check*?
  - percentage of element-creation related failures prevented
- Is it necessary to use element retrieval methods other than the direct correspondence retrieval to prevent element-creation related failures? (e.g. context based retrieval or indirect correspondence retrieval)
- To what degree can deprecated change event related failures be solved by *event validity check*?
  - percentage of deprecated change event related failures prevented

Concept bottlenecks are only solvable through additional transformations, not through implementation adaptation, and are therefore outside the scope of this thesis. Incompatible consistency constraints are solvable though the refinement of the intended consistency specification for the target system, but this has to be handled problem specific and is

therefore also not a part we can evaluate. We did not find generic approaches to prevent unsynchronizable states, or synchronization problems produced by the visibility of unresolved changes.

6.1 Methodology

We introduced the domains used in the case study for this evaluation in Chapter 2.5. The transformation network used consists of three domains, PCM, UML and Java, and two bidirectional, delta-based, transformation definitions, $T_{PCM \leftrightarrow UML}$ between PCM and UML, and $T_{UML \leftrightarrow Java}$ between UML and Java. The consistency constraints that are implemented by these transformations are defined in Chapter 5. Both transformations are implemented using the Reactions language of the Vitruvius framework. The Vitruvius framework's transformation engine uses immediate change application (batch-apply), it provides the possibility for transitive change propagation and uses depth-first change event resolution, and it provides a trace model, the correspondence model, that can be globally accessed from either transformation.

It is important to note that the transformation $T_{PCM \leftrightarrow UML}$ was developed in the context of this thesis, with transitive change propagation in mind, even in the simple binary case without transformation combination. Therefore, it already used existence checks and event validity checks before the evaluation. In contrast, the $T_{UML \leftrightarrow Java}$ was developed independently outside of this theses and it was implemented with non-transitive change propagation in mind.

6.1.1 Models used for the Evaluation

The artificial software system that is used as a case study is the "media store repository model" of the media store system [22, 28], which is a case study for the Palladio component model [21]. We will call this the *PCM media store model*, because it specifies a repository of component definitions in the PCM domain. To represent the PCM media store model in both other domains, a *UML and a Java media store model*, which are consistent to the PCM media store model, are manually created. Each of these three models functions as a solution model to compare the model synchronization's output models against after a specific evaluation scenario.

In the Java solution model, we did not model information that could not be propagated along the network's transformations because of known concept bottlenecks. This pertains to method implementations in Java that would match the abstract component behavior description provided by PCM Service Effect Specifications, ,and the *java::Field* initialization of a *java::Class* in its constructor, which can be derived from the *pcm::RequiredRoles* and *pcm::Connectors* (Assembly- and Delegate-Connectors) of a corresponding *pcm::RepositoryComponent*.

6.1.2 Evaluation Scenarios

The main evaluation scenarios that provide most observations for this evaluation, are three simulated model-creation scenarios, each one started from another domain. As an example, for the evaluation scenario started from the PCM domain:

- The creation of the solution PCM media store model is simulated and the model synchronization process is started.
- This then produces synchronization failures that hinder the model synchronization from terminating or the model synchronization terminates and we regard the end states of the involved and generated models as output.
  - If the model synchronization does not terminate properly, then we investigate the failure cause, document it and fix it.
  - The scenario is started again with the fixed system, until the model synchronization properly terminates, so that we can evaluate the output.
- When the model synchronization terminates, the output models are compared to the respective solution media store models (PCM, UML, Java). The differences are traced back to their cause and documented.
- If a failure inhibits the synchronization of other concepts, for example if an interface is not created and therefore the synchronization of its signatures is omitted, then that failure cause is fixed and the scenario is started again.

In addition to the media store case study scenarios, we also used hand-crafted minimal evaluation scenarios to provoke failures in the model synchronization of concepts that were not represented in the case study model. This includes for example the synchronization of references to *pcm::CollectionDataTypes*, with different inner types, and the corresponding elements in UML and Java. The evaluation process for these scenarios follows the pattern described above for the case study scenarios.

6.1.3 Failure Expressions and Failure Counting

The model synchronization process through transformation combination and transitive change propagation has the goal of achieving multi-model consistency preservation between all involved models. The following list shows the possible failure expressions:

- Change *propagation loops*, which preventing the termination of the model synchronization
  - Creation loops, alternating value loops, and diverging loops
- *Inconsistency* between the output models after the synchronization terminates
  - local inconsistency, between two domains connected by a transformation
  - global inconsistency between all models of the network
- *Information loss* with regards to the user input, which is represented by the solution models.

We included false warnings about unsynchronizable element references that were automatically repaired by subsequent changes as counted failure expressions, because we recognize that the inconsistency might persist or information might be lost, given another change sequence or transformation implementation.

Global inconsistency can stem from concept bottlenecks or from incompatible structural mappings (incompatible consistency constraints). In both cases, the failure is not related to the synchronization behavior, but rather to the system specification. We list the known potentials for global consistency failures to occur in the respective subsection (Sections 6.2.5, 6.2.6), but we do not include them in the statistical analysis of this evaluation.

Because we want to evaluate the relative impact of the identified failure potentials, we need to assign a severity factor to the possible failure expressions, with regards to how problematic they are to the goal of multi-model consistency preservation. Propagation loops prevent the whole model synchronization from terminating, which leaves all newly introduced information unsynchronized across the network. By comparison, both inconsistency and information loss allow the model synchronization to terminate and at least some parts of the involved models can be correctly synchronized. Therefore propagation loops more severely impact the overall consistency and we assign it the highest severity factor. Inconsistency and information loss equally impact only a subset of elements in the models. It is possible to claim that inconsistency is more problematic, because it can be harder to detect if a user only works on one model in the network, however that claim is speculative. We decide to assign both failure expression types the same severity.

- severity(propagation loop) = 2
- severity(inconsistency) = severity(information loss) = 1

We acknowledge that this severity assignment is arbitrary, as there is no explanation why a propagation loop would be exactly twice as problematic as an instance of inconsistency or information loss. The exact scaling is not as important, because we only want to rank the failure potentials relative to each other based on their average failure severity. We take this as a relative measure of impact on the model synchronization.

As already stated in the introduction to this chapter, a model can contain multiple instances of the same concept, for example multiple *pcm::BasicComponents* or multiple *java:Classes*, and for each instance the same transformation rules apply with the same unsolved problems. This effectively inflates the number of observed failures per unaddressed failure potential by the number of affected concept instances present in the case study model. Therefore, we trace the failure expression back to the correspondence type that was incorrectly synchronized and the circumstances that produced the failure. Repeated failures that stem from the same circumstances are counted as one. For example if the transformation $T_{PCM \leftrightarrow UML}$ created an unintended element duplication, and therefore inconsistency, for every *pcm::BasicComponents* in the case study model, and there are 14 *pcm::BasicComponents* instances, we still only count one failure.

Furthermore, if multiple models contain additional, missing or false information compared to the solution models, and all those differences can be traced back to one root cause, then we only count it as one failure. Take for example three corresponding named elements, whose names are supposed to be the same, and the user renames one of the

elements. Now if at any point in the propagation process the new name is discarded and the models synchronize back to the old name (or no name), then we would observe three differences when comparing the models to the solution models, one per named element. But all three differences are a consequence of one failure and therefore we count it as such. This effectively normalizes the failure count for the number of models in the network.

The classification "failure f is caused by failure potential fp" is performed manually, and we will justify the classifications in the discussion subsections specific to the respective failure potential.

6.2 Results and Discussion

The results of the evaluation scenarios revealed 27 failures that resulted from the transformation combination and transitive change propagation, in addition to 6 concept bottlenecks and 1 structurally different mapping, which were known before the evaluation, and did not count to the failure list. The revealed failures, along with their failure potential classifications, are listed in Tables 6.1 and 6.2. For the rest of this chapter "F-<Number>" references one of the failures listed in the overview tables.

In the following subsections, we explain the failures and their classification decisions, and we answer the questions established in the introduction, with respect to the individual failure potential types. The questions regarding the existence check and validity check pattern are discussed alongside the change conflicts and deprecated change events respectively. Lastly, we compare the failure potentials to each other and give an overview of the findings.

6.2.1 Change Conflicts and Existence Checks

We found 17 failures (F1-F17) that were the result of change conflicts. All 17 of these failures were a consequence of element-creation change conflicts, associated with the correspondence types defined between UML and Java, because essentially the instantiation of any correspondence type between UML and Java led to one such failure. This can be explained by the fact that the transformation $T_{UML \leftrightarrow Java}$ was initially developed for directed, non-transitive change propagation, where it was not necessary to check if an element already exists. As a result any element creation by $T_{UML \leftrightarrow Java}$ led to further element creations when the new element creation event is propagated backwards. By contrast $T_{PCM \leftrightarrow UML}$ was only involved in F17, because it was implemented for transitive change propagation and employed existence checks from the start.

6.2.1.1 Failure Explanations

In the cases of F1-F13, the creation-change conflicts resulted in creation loops in the same manner. For example F1, the creation of an *uml::Package* triggers the creation of a *java::Package*, and the creation of *java::Package* triggers the creation of a *uml::Package*. In both directions, $T_{UML \leftrightarrow Java}$ does not check if the corresponding package of the other domain already exists.

Failure Expression	F-ID	Failed Synchronization for Correspondence	Change Conflict	Deprecated Change Event	Visibility of Unresolved Changes	Unsynchronizable States	Incompatible Consistency Constraints	Implementation Errors
Creation Loop	1	UJ-Package	✓					
	2	UJ-Enum	✓					
	3	UJ-EnumLiteral	✓					
	4	UJ-Interface	✓					
	5	UJ-SuperInterfaceRef	✓					
	6	UJ-InterfaceMethod	✓					
	7	UJ-Class	✓					
	8	UJ-SuperClassRef	✓					
	9	UJ-ClassMethod	✓					
	10	UJ-Constructor	✓					
	11	UJ-ImplementsRef	✓					
	12	UJ-Attribute	✓					
	13	UJ-OrdinaryParam	✓					
Element Duplication	14	UJ-EnumCU	✓					
	15	UJ-InterfaceCU	✓					
	16	UJ-ClassCU	✓					
	17	PU-ReturnParam + UJ-ReturnParam	✓					

Table 6.1: Synchronization failure classification overview (part 1)

Failure Expression	F-ID	Failed Synchronization for Correspondence (and broken constraint)	Change Conflict	Deprecated Change Event	Visibility of Unresolved Changes	Unsynchronizable States	Incompatible Consistency Constraints	Implementation Errors
Diverging Loop	18	UJ-ClassCU + UJ-Class (same names)					✓	
Alternating Value Loop	19	UJ-ClassMethod (matching visibility)	(✓)	✓				✓
	20	UJ-Constructor (matching visibility)	(✓)	✓				✓
User Input Overwritten	21	PU-CollTypeProp + UJ-Attribute (types correspond)				✓		
	22	PU-CollTypeParam + UJ-ReturnParam (types correspond)				✓		
	23	PU-CollTypeParam + UJ-OrdinaryParam (types correspond)				✓		
(temporarily) Unsynchronized Reference	24	UJ-ImplementsRef (impl. interfaces corr.)			✓			
	25	UJ-Attribute (types correspond)			✓			
	26	UJ-ReturnParam (types correspond)			✓			
	27	UJ-OrdinaryParam (types correspond)			✓			

Table 6.2: Synchronization failure classification overview (part 2)

In the cases of F14-F16, the creation-change conflicts only resulted in element duplication of the involved *java::CompilationUnit*. For example in the case of F14, the creation of an *uml::Enum* triggers the creation of a *java::CompilationUnit* and a *java::Enum*, but only the creation of a *java::Enum* inside a *java::CompilationUnit* triggers the creation of a *uml::Enum*. Then the combination of both transformations leads to the following behavior: the creation of a *java::Enum jE* inside a *java::CompilationUnit jECU* triggers the creation of a *uml::Enum uE*, which then triggers the creation of a duplicate *java::CompilationUnit jECU'* on the change backpropagation.

For F17 the creation-change conflicts also resulted in an element duplication of the return parameter of a *uml::Operation uO*. A *pcm::OperationSignature* is mapped to a *java::Method* via a *uml::Operation*, and if either is established, both sides attempt to create the return parameter of *uO*. However, both sides neglected to check if the parameter already exists, which leads to the duplication.

Necessary feature change conflicts occurred for model default values that were immediately overwritten by the values intended by the solution model, but non of these feature change conflicts directly led to synchronization failures. For failures F19-F20 a feature change conflicts occurred as a result of an implementation error in the propagation of visibility modifiers. These feature change conflicts were unnecessary and could have been avoided by a better transformation implementation, but they occurred and consequently one of the change events was deprecated (more details in Section 6.2.2). However, we did not classify F18-F19 as feature change conflicts, because the failure did not stem from the overwritten information (because the overwrite was intentional with regards to the implementation), but rather from the propagation of the overwritten and therefore deprecated information.

6.2.1.2 Change Conflict Evaluation

We found 17 failures (F1-F17) that were the result of change conflicts, of a total of 27 observed synchronization failures. This makes change conflicts the most prevalent failure potential with 63% detected failures. It is important to note, that all these failures were caused by element-creation change conflicts. This confirms the argumentation in Section 4.2, which states that element-creation change conflicts are more problematic then feature change conflicts. We observed 2 possible failure expression types of the element-creation conflicts. F1-F13 are propagation loops (severity=2) whereas F14-17 are element duplications, which are inconsistencies, but allow the termination of the synchronization (severity=1). This results in a weighted severity score of $severity$(change conflict) = $(13 * 2 + 4 * 1)/17 = 1.76$.

6.2.1.3 Existence Check Pattern Evaluation

All 17 element-creation change conflicts could be prevented by the existence check pattern.

In a scenario, where the transformation $T_{UML \leftrightarrow Java}$ is used in isolation but with transitive change propagation, the simplest version of the existence check, only using direct correspondences for the element retrieval, is enough to prevent F1-F16. But when combined with $T_{PCM \leftrightarrow UML}$ some classes and packages now participate in more complex correspon-

F-ID	Failed Synchronization for Correspondence	Required Existence Check Variant		
		direct	context	indirect
1	UJ-Package		✓	
2	UJ-Enum	✓		
3	UJ-EnumLiteral	✓		
4	UJ-Interface		✓	
5	UJ-SuperInterfaceRef	✓		
6	UJ-InterfaceMethod	✓		
7	UJ-Class		✓	
8	UJ-SuperClassRef	✓		
9	UJ-ClassMethod	✓		
10	UJ-Constructor		✓	
11	UJ-ImplementsRef	✓		
12	UJ-Attribute		✓	
13	UJ-OrdinaryParam		✓	
14	UJ-EnumCU	✓		
15	UJ-InterfaceCU	✓		
16	UJ-ClassCU	✓		
17	PU-ReturnParam + UJ-ReturnParam		✓	

Table 6.3: Required existence check variants for the observed failures

dence graphs and the correspondences can be instantiated along multiple paths. Take the example correspondence graph in Figure 4.1 for a *pcm::Repository* and its corresponding *uml::packages* and *java::Packages*, which we used to explain the existence check pattern in Section 4.2.1, and assume the *java::Packages* for the repository-package and the contracts-package are inserted by the user. Now the propagation of repository-package creation can result in the creation of a contracts-package, which should be the same as the user inserted one. To detect that the user already inserted the package, $T_{UML \leftrightarrow Java}$ has to extend its existence check pattern for the *UJ-Package* to include the context retrieval attempt.

The necessity for the context retrieval can be shown in a similar manner for some of the other correspondence types. Table 6.3 gives an overview of the existence check pattern variants that was necessary to prevent the observed failures. The existence check variant using indirect element retrieval, via transitive correspondence resolution, was never necessary.

The existence check patterns could successfully prevent all observed element-creation change conflicts, using direct (correspondence) and context-based element retrieval, without having to use user disambiguation or indirect correspondence retrieval.

6.2.2 Deprecated Change Events and Event Validity Checks

We only found the two failures, F19 and F20, related to deprecated change events. In both cases, the deprecated change events are created by a suboptimal implementation of the transformation rules that synchronize the visibility modifiers between UML and Java elements. Both failures are quite similar, therefore we explain only F19, which impacts the correspondence type *UJ-ClassMethod* defined in Table 5.33.

Every time the visibility of a *uml::Operation* is changed, the visibility modifier of the corresponding *java::ClassMethod* is first removed and then the new modifier is set. These two steps are performed despite the fact that the correct modifier might already be set. This temporarily introduces an incorrect state and then immediately overwrites said incorrect state with the intended consistent state, producing two new change events, of which the first is deprecated, even if the model was consistent. We therefore classify F19 and F20 as partially caused by an implementation error of the application direction $UML \rightarrow Java$ of the transformation $T_{UML \leftrightarrow Java}$.

Now in the other direction, we have two change events for the *java::ClassMethod*'s visibility, where the first event is deprecated. But the specific transformation rules that propagate visibility changes from Java to UML do not use event validity checks or state-based transformation. As a result, first the deprecated value is propagated, overwriting the intended *uml::Operation* visibility, and immediately afterwards the intended value is restored and the loop begins anew. We therefore also classify F19 and F20 as failures resulting from deprecated change events.

Fixing either the implementation error or introducing event validity checks, both options solve the alternating value loops of the failures F19 and F20.

6.2.2.1 Deprecated Change Event Evaluation

We found 2 failures (F19-F20) that were partially the result of deprecated change events, of a total of 27 observed synchronization failures, which equals only 7.4% of the detected failures. Both F19 and F20 are propagation loops (severity=2), which translates to $severity$(deprecated change events) = 2.

It is slightly surprising that we observed so few synchronization failures resulting from deprecated change events, event though, every element that is created by a transformation is subsequently initialized with new feature values, which may conflict with the metamodel default values, thereby causing the default value change events to be deprecated. In addition to that, multiple user inputs also produce deprecated change events. We suggest three possible reasons for this sparsity:

- $T_{PCM \leftrightarrow UML}$ was designed for transitive change propagation and was implemented with event validity checks from the beginning, and the validity checks may have prevent some failures.
- $T_{UML \leftrightarrow Java}$ did not include event validity checks, but it mostly relied on state based transformation rules, which we suggest might be another option to prevent such failures (see Section 4.1.1).
- Or deprecated change event might not be very problematic in general.

6.2.2.2 Event Validity Check Evaluation

We could solve 2 of 2 (100%) observed failures related to deprecated change events by using event validity checks. However, it is difficult to claim the effectiveness of event validity checks, based on the fact that two synchronization failures, with the same cause, could be prevented through such checks, especially if the failures also could have been avoided through a better transformation implementation. The stronger claim for the effectiveness of event validity checks might be that so few failures emerge, because $T_{PCM \leftrightarrow UML}$ already implemented event validity checks for most processed change events.

Also, we had to use a slightly altered check formulation than the one we provided in Table 4.1, to account for semantic equivalence between visibility modifiers instances of the same metamodel class in the Java metamodel. In Java, the possible visibility modifiers are metaclasses in the Java metamodel, for example *java::Public*, and instances are added as annotations to the element, for which the visibility modifier is active. As a consequence, checking if the exact removed modifier instance is still contained in the set of modifiers, does not reveal if another instance of the same class is present/active. Therefore the default validity check is not sufficient and we had to use a formulation based on metamodel specific semantic:

```
isDeprecated(remove(e.modifiers, m)) :=
    (e.modifiers.findFirst(m2 | m2 instanceOf typeOf(m)) != null)
```

6.2.3 Visibility of Unresolved Changes

We found 4 temporary failures (F24-F27) that occurred because the effects of unresolved changes were visible to transformation rules before the context could be properly synchronized. Of the total of 27 observed synchronization failures, these failures make up 14.8%. As we will explain in a moment, F24-F27 did still terminate with consistent models and no loss of information, but could have resulted in information loss under different circumstances. Therefore it is questionable, if we should assign them a severity of "0" or "1". We decide on $severity$(visibility of unresolved changes) = 0, because no actual failure persisted after the transformation terminated.

Visibility of unresolved changes, as described in Section 3.3.2, refers to the fact that if we immediately apply all changes before resolving the individual change events, then their effects may be visible in the affected model. It is beneficial, because it allows the transformation rules to take into account what changes are already queued and avoid contradicting these changes. It is problematic, because transformation rules can see elements that do not yet have a corresponding counterpart in other models, and when the transformation rules try to find that counterpart, they may fail.

Similarly to the example in the Section 3.3.2, we observed element references that could not be synchronized because the element was not yet created for the following correspondences:

- F24: A *uml::Class uC* can implement a *uml::Interface uI*, this is expressed through a *uml::Generalization* that references said interface. If the corresponding *java::Interface* has not yet been created, because the creation change of *uI* has not yet been resolved, then the transformation cannot synchronize the information present in the model.

- F25-F27: Attributes, parameters and, in Java, methods can all reference a type. If the referenced type's correspondence has not been initialized before the synchronization of the referencing element, then the failure emerges.

In each case, the transformations gave a warning that a reference could not be properly synchronized, but continued the change propagation process. By the time the change event that described the set-feature change for the unsynchronized reference, the change that created the corresponding element had already been resolved, so that the reference could then be correctly synchronized. The set-feature change event effectively "healed" the previous synchronization failure.

In Table 6.2, we listed the failure expression of F24-F27 as "(temporarily) Unsynchronized Reference", to emphasize that the output models were consistent after the synchronization terminated, which makes the classification as failures questionable. We chose to report these temporary failures, because they might turn into real failures. If the correct value that could initially not be synchronized were to be overwritten by the backpropagation of the incorrect value, the temporary failures might turn into concrete information loss failures.

Failures F24-27 could be avoided, but not solved, through incremental insertion of the solution model, which ensured that the creation-changes of the referenced elements were propagated first.

6.2.4 Unsynchronizable States

We found 3 failures (F21-F23) that occurred because of unsynchronizable states, which makes up 11.1% of all observed failures. In these failures, information inserted into the test model was lost (severity=1), which translates to $severity$(unsynchronizable states) = 1.

These three failures are related to the mapping of *pcm::CollectionDataTypes* across the different domains. The defined consistency constraints require the UML representation to have a multiplicity of exactly one element (1..1) or arbitrary many elements (0..*). The implementation of the transformations do not discard multiplicity changes that result in an unsynchronizable state, but rather keep the current type interpretation for the corresponding typed elements. However, if a new type is set in any of the involved models, then the corresponding typed elements are brought to a consistent state. For UML this means to propagate the new type to the other models, defaulting to a normal/non-collection interpretation if the multiplicity was previously unsynchronizable. For the other domains it is then not visible that the unsynchronizable multiplicity is intended and in an effort to create consistency, the multiplicity is changed to (1..1).

So if a collection type interpretation is to be set on the UML model, then the user input can be overwritten in an attempt to restore consistency, ultimately loosing information, even if later changes might have brought the model to a consistent state without discarding that information.

6.2.5 Incompatible Consistency Constraints

We observed F18 as a consequence of contradicting consistency constraints, which is a part of incompatible consistency constraints.

Concretely, for F18, the constraints of the correspondence type *UJ-Class* originally prescribe for a *uml::Class uC* and the corresponding *java::CompilationUnit jCU* that their names had to be the same, however, the *jCU*'s name has to be fully qualified including the file extension, e.g. "package.ClassName.java". Yet Java also requires the contained *java::Classifier* to have the same name as the unqualified name of the *jCU*, which the original constraints also said was equal to the name of *uC*. This intra-transformation constraint contradiction on its own only produced an incorrect Java model state. The propagation loop was a result of the interaction with the transformation $T_{PCM \leftrightarrow UML}$. A *pcm::RepositoryComponent pRC* requires that its corresponding implementation *uml::Class uC* has the same name, but for an appended implementation suffix "Impl", e.g. "ComponentNameImpl". The PCM-UML constraint required a different suffix then the UML-Java constraint, which we classify as a contradiction.

When the *pRC*'s name is propagated to *uC* the "Impl" is appended and then further propagated to *jCU*. For *jCU* the name is changed to be qualified and the qualified name is propagated back, through *uC*, to *pRC*. And from *pRC* the name is again propagated to *uC* and again "Impl" is appended. At that point the name is no longer fully qualified, the file extension is missing by Java metamodel definition, which is why the name is again changed and the propagation cycle continues. The string that represents the name is continually grown, which is why we describe the failure expression as a diverging loop.

We observed one failure (F18) as a result of incompatible consistency constraints, which makes up 3.7% of all observed failures. That failure was a propagation loop (severity=2), which translates to $severity(\text{deprecated change events}) = 2$.

In addition to the observed failure, there was an additional case of incompatible consistency constraints, in that the structural mapping of primitive data types between PCM and UML was not compatible to the primitive type mapping between UML and Java. The transformation $T_{PCM \leftrightarrow UML}$ prescribed the use of externally defined *pcm::PrimitiveDataType* and *uml::PrimitiveType*. The transformation $T_{UML \leftrightarrow Java}$ used to map manually defined *uml::PrimitiveTypes* to the metamodel-predefined Java types. This was a specification problem of the case study system, and we changed the implementation and constraints to use predefined primitive types for both transformations. We did not count this in the evaluation statistic, because it was known prior to the evaluation and no consistent solution could be derived from this mapping disparity, against which to compare the transformation outputs.

6.2.6 Concept Bottlenecks

Concept bottlenecks are a consequence of the transformation network topology and the metamodels of the involved domains. They can be circumvented by additional transformations and are therefore a specification problem of the designed system. By our definition of the case study, concept bottlenecks are the consequence of shared concepts between the PCM and the Java domain, which can not be expressed in the UML domain (at least not in class diagrams). The following is a list of non-shared concepts that were known prior to the evaluation.

- PCM Service Effect Specifications represent an abstraction of the implementation of *java::ClassMethods*, which realize a method of a contractual interface, which is provided by a component's implementation.
- A *pcm::RequiredRoles* maps to a *uml::Property* and a *uml::Parameter* in the component's constructor, which both share a relation that cannot be expressed in UML, but that relation can be expressed in Java through to the initialization of the attribute in the component's constructor.
  - This similarly applies to *pcm::AssemblyConnector*
  - and *pcm::RequiredDelegateConnector*.
- A *pcm::ProvidedDelegateConnector* can be represented through the implementation of *java::ClassMethods* that realize a provided constractual interface by delegating the interface call to an assembled inner component (attribute of a component's implementation).
- A *pcm::CollectionDatatypes* should always map to the same Java concrete implementation type of java.lang.util.Collection, which cannot be expressed with the chosen consistency specification, because the multiplicity and (inner-) type configuration of the typed-multiplicity-elements (*uml::Property* or *uml::Parameter*) is equal for all possible collection implementations. Changes to the selected collection type in PCM or Java do not result in changes for the UML elements, and therefore cannot be synchronized with the respective other domain.

We did not include concept bottlenecks in the statistical part of the evaluation, because they cannot be solved without an additional transformation, which was not in the scope of this thesis.

6.2.7 Completeness of the Failure Potential Catalogue

In Chapter 3, we identified six failure potentials. One of the questions for this evaluation is, whether or not this catalog of failure potentials is complete, meaning that the identified failure potentials are sufficient to capture all model synchronization failures. In Tables 6.1 and 6.2, we list the observed failures and which failure potential we identified as the cause (or more specifically, which failure potential was not accounted for by the transformation implementation, resulting in the observed failure). We found, that each of the observed failures could be traced back to an overlooked failure potential, which implies that the proposed catalog is reasonably complete.

6.2.8 Distinctness of the Failure Potentials

The failure potentials can be distinct with regards to the failure classification, or with regards to the failure expressions that have been classified with a specific failure potential. The more varied the failure expressions of one failure potential, the more likely it is that the failure potential stretches too broad of a category. In contrast to that, if we have many failures that are classified with the same set of failure potentials, then that set of failure

potentials might be overly specific or they might have an common, unidentified underlying cause.

We observed exactly one failure expression for each failure potential, except for change conflicts where we saw element duplication and creation loops. But both element duplication and creation loops are closely related, because element duplication easily leads to a creation loops if the existence checks are missing in both transformation directions. Therefore, the failure potentials are distinct with regards to the failure expressions. However, we only observe failures for element-creation change conflicts. Feature change conflicts would likely result in different failure expressions, and it might be advantageous to explicitly differentiate these change conflicts into two distinct subcategories.

We were able to classify each observed failure with exactly one failure potential, except for F19-F20, which suggests that all failure potentials except change conflicts and deprecated change events are sufficiently distinct. F19-F20 occurred as a result of problematic transformation implementation that produced an unnecessary feature change conflict and subsequent deprecated change event, and the propagation of deprecated change events led to the propagation loop. Because deprecated change event cannot occur without feature change conflicts, there is an obvious overlap between the two. But by the same argumentation for why we classified F19-F20 as a result of deprecated change events, there is utility in the separation of both failure potentials. For one, both potentials represent different opportunities for the transformation developer to intervene, one is preventive and one is reactive. And secondly, feature change conflicts are necessary if information has to be overwritten, yet we still have to deal with the deprecated change events of such overwrites. We therefore regard (feature) change conflicts as separate failure potentials from deprecated change events.

6.3 Summary

The results of the evaluation scenarios revealed 27 failures that resulted from the transformation combination and transitive change propagation. In addition to the above mentioned runtime failure, the combination of both used transformations had 6 concept bottlenecks and 1 structurally different mapping (*incompatible consistency constraint*), which are specification problems we knew before the evaluation and did not count to the runtime failure list. Concept bottlenecks can only be solved through additional transformations, which was not part of this thesis. The structural mapping was adapted to be compatible between both transformations before the evaluation. Table 6.4 shows an overview, comparing the failure statistics of the individual failure potentials.

The most prevalent failure potentials were *(element-creation) change conflicts* with 17 of all runtime failures (63%), all of which could be solved by the *element existence check pattern*. Similarly all (only 2 of the 27) *deprecated change event* failures could be fixed by the proposed *event validity check*. With these two simple patterns alone 70% of the runtime failures could be prevented, which were also the most severe failures, because 15 of the combined 19 failures were propagation loops that hinder the model synchronization process from terminating.

FailurePotential	Failures #	Failures %	Expression Types #Loops	Expression Types #Inconsist.	Expression Types # Info.Loss	Severity avg.
Runtime						
Change Conflict	17	63.0%	13	4	0	1.76
Deprecated Change Event	2	7.4%	2	0	0	2
Visibility of Unresolved Changes	4	14.8%	0	0	(4)	0 or 1
Unsynchronizable States	3	11.1%	0	0	3	1
Incompatible Consistency Constraints	1	3.7%	1	0	0	2
	27		16	4	3(7)	1.44 (1.59)
System Specification						
Concept Bottlenecks	6					
Incompatible Consistency Constraints	1					

Table 6.4: Overview of failure statistics with regards to failure potentials. The number of failure expressions from "visibility of unresolved changes" is in brackets, because the failure was only temporary, which is also why the severity is 0 or 1 depending on interpretation.

We also found one consistency constraint contradiction (*incompatible consistency constraint*) in the original consistency specification of the transformation $T_{UML \leftrightarrow Java}$. And we found 4 temporary failures where a transformation rule could not be correctly applied as a consequence of the *visibility of unresolved changes*, but all four failures were automatically restored through subsequent change events. Whether or not these failures are always unproblematic is unclear. Lastly, we found 3 failures from *unsynchronizable states*, equaling only 11% of all runtime failures, where a user input to a UML model could be overwritten if the synchronization was started before a synchronizable state was reached.

Multi-model consistency preservation could be achieved for this limited case study, with the exception of the information affected by the concept bottlenecks. And all observed failures could be classified as "caused by" exactly one failure potential, which ensures some level of validity and completeness for the developed failure potential catalog.

6.4 Threats to Validity

- Pre-existing knowledge about Java and the transformation $T_{UML \leftrightarrow Java}$ may have influenced the development of the transformation $T_{PCM \leftrightarrow UML}$. Consequently, it may have reduced the number of incompatible consistency constraints.
- The transformation network used for this evaluation only had a linear topology, without transformation network cycles. As a result, the number of possible correspondence graph cycles is drastically reduced (only one cycle in the case study), which probably influenced the failure potential frequency, especially reducing the number of change conflicts. But we observed a few branching correspondence graphs, which implies that the findings are at least likely to be generalizable to case studies with branching transformation networks.
- The evaluation scenarios focused primarily on model creation and initialization, not on the modification of existent elements. This limits the number of explored change sequences and reduces the likelihood of finding all failures.
- The number of failures we found for each failure potential, except for change conflicts, is quite low.

7 Related Work

The goal of this thesis was to explore multi-model consistency and how we can achieve it through the combination of transformations. One aspect of that is model consistency in general, and most approaches regarding model synchronization use some form of model transformation to automate the process. Therefore, model transformation is the most closely related branch of research, in particular binary transformations and transformation combination.

Consistency in a general sense has been formulated by its complement, inconsistency. Nuseibeh et al. [17] describe inconsistency is a situation where two descriptions do not adhere to a prescribed relation, and they describe a framework for managing inconsistencies in [16]. In this framework, inconsistencies are detected by consistency rules. These take a comparable role to the predicate based consistency constraints that we used. The Object Constraint Language (OCL) allows the formal definition of invariants and pre-/post-conditions. Lano et al. [13] use such OCL constructs to generate bidirectional transformations and enforce these constraints.

Many different works have developed the field of binary transformations. Triple Graph Grammars (TGGs) are a formalism based on graph-pattern replacement rules, but it has a limited expressiveness. Diskin et al. [5, 4] develop a separate formalism for delta-based bidirectional transformations, called delta-lenses. The transformations we used in this thesis are implemented using the Reactions Language [9, 23], which is an imperative transformation language and less formal then the afore mentioned formalisms. However, it is turing-complete, because it allows transformation rules to be implemented using Java, which makes it maximally expressive.

The QVT-R standard [19] and the approach by Lano et al. [13] can be used to declaratively define bidirectional transformations. These declarative mapping rules are similar to the correspondence type definitions, which we used to specify the consistency constraints our transformation should implement. It might be possible to improve our approach by generating transformation rules from consistency type definitions. Kramer [11] developed a catalog of automatic attribute mapping inversions, which may be useful in that regard.

Xiong et al. develop a system that can synchronize concurrent model updates [32]. This is different from the scenarios studied in this thesis, because we do not explicitly examine concurrent model updates. However, because of the transitive change propagation of individual changes, unresolved changes can be queued for multiple models, which is a similar situation, as if the changes had occurred in different models to begin with. A second difference is the fact that [32] uses state-based transformations that produce whole new updated copies of the modified models, whereas the transformations we use are incremental and merely updated the existing models.

On the topic of multiary transformations, QVT-R is probably most known, because the standard claimed multiary transformation capabilities from the start. However, Macedo

et al. [14] show that QVT-R is underspecified with regards to the operationalization for multiary transformation, and they propose an extensions to solve this problem. Another approach is the Graph Diagram Grammar formalism by Trollmann [31], which is an extension of the TGG formalism to multiple models.

Transformation combination in the general case is still an open problem. While, Diskin's delta-lens formalism works for the asymmetric case [4] and allows multi-view modeling through concatenation of the lenses, the symmetric case, where neither model can be completely derived from the other, cannot be extended in the same way. Stevens [27] discusses the properties of combined bidirectional transformations on a theoretical basis, with regards to resolvability and confluence, while limiting the transformation direction through authority models. Her conclusion regarding consistency restoration through transformation networks was "mostly negative", because, in general, a network might converge to different consistent states depending on the resolution path (if a change is resolvable at all). In Section 3.2.3, we argue that the inclusion of strict authority instances, which are models that may not be changed through the consistency restoration, might be too strict of a limitation, because it limits the consistency constraints that can be resolved. The effect of an authority model might instead be achieved through a view that restricts the user's edit-ability of the network models, so that the effects would not have to be propagated to the authority models. However, the confluence examples we explored still suggest that the convergent state can be non-deterministic.

8 Conclusion and Future Work

In this chapter, we conclude the thesis by first summarizing the contributions and then giving a short overview of ideas for future work.

8.1 Conclusion

This thesis explored how binary, bidirectional, delta-based transformations can be combined through transitive change propagation in order to achieve multi-model consistency preservation. We developed a catalog of six failure potentials that reflect how the interaction between the consistency constraints of different transformations and transitive change propagation can produce conflicting model changes, lose change information or fail to restore global consistency. Furthermore, we propose two transformation implementation patterns to mitigate the effects of change conflicts. Lastly, the evaluation revealed the relevance of the identified failure potentials and the effectiveness of the proposed implementation patters.

We introduced transitive change propagation and how it can be used to combine transformations to form a transformation network, through transitive application of the transformations to the output model-changes of prior transformations. Additionally, we defined model synchronization as the application of consistency-restoring transformations, where consistency between a set of models is defined through consistency constraints. The combination of multiple binary consistency-restoring transformations through transitive change propagation then extends the model synchronization to multi-model consistency preservation. We decomposed the definition of consistency on a model level into sets of consistency constraints that define consistency for the semantic dependencies between two model elements, which we called *correspondence types*. Furthermore, we defined a *correspondence graph* as a network of elements that are linked by their semantic dependencies, which are described through correspondence types. We then used correspondence graphs to investigate how changes may propagate through a transformation network, because they provide a more detailed introspection into the propagation process based on element-level propagation paths, rather than low-resolution, model-level propagation paths.

We revealed a number of emergent failures of the model synchronization process by studying how sets of properties of the transformation engine and the transitive change propagation behave, when they are applied to a minimal correspondence graph example and a specific change sequence. Based on the observed failures, we developed a catalog of *six failure potentials* with regards to multi-model consistency preservation. One failure potential is a consequence of the transformation network topology and the involved domains. If two domains share a concept and the domains are indirectly connected by

two transformations, but the pass-through domain cannot express this concept, then the consistency of this concept's representations cannot be ensured. This failure potential can only be avoided through additional transformations. Another failure potential is a consequence of conflicting consistency definitions of the employed transformations. It can only be avoided by correctly defining the consistency constraints between models, so that the combination of the constraints matches the intended global consistency. These are both case specific problems of the combination of binary transformations that cannot be solved without knowing which transformations will be combined. Furthermore, we developed two transformation implementation patterns to mitigate two other failure potentials. One pattern prevents element-creation conflicts where elements might be duplicated as a consequence of cyclic change propagation paths and the other mitigates the effects of feature change conflicts, after they occurred, by preventing the propagation of deprecated information. These patterns can be applied by the transformation developer to an individual transformation definition, independent of the combination scenario. For the remaining two failure potentials, no general solution was found yet and further research is necessary.

We provided detailed and semi-formal definitions of the consistency constraints that are implemented by the two transformations, which we used in the following evaluation, between the Palladio Component Model, UML class diagrams and Java. We grouped the consistency constraints into correspondence type definitions to describe how the metamodel concepts are mapped between the involved domains.

In the evaluation, we investigated the relevance of the identified failure potentials using a realistic use case and a transformation network of two independently developed transformations. This revealed that all observed failures could be classified with exactly one failure potential. Most observed failures were a consequence of transitive change propagation (70%), all of which could be fixed through implementation of the proposed patterns. To determine the impact of a specific failure potential type, we classified all observed failures also by effect. A slight majority of the observed failures were propagation loops (59%), which stopped the change propagation from terminating. Through the implementation of the proposed patterns and one fixed consistency constraint contradiction, all propagation loops were prevented, leaving only information loss failures in specific test cases. However, further studies are necessary, especially using more than two transformations and a cyclic transformation network, to see how transformation cycles impact the failure distribution and to verify the generalizability of the developed patterns.

In summary, we identified a catalog of six failure potentials that can inform a transformation developer about possible problems when a transformation is used with transitive change propagation or combined with other transformations. Furthermore, we developed two transformation implementation patterns to prevent and mitigate the manifestation of two of the failure potentials. Because all failures of the case study could be classified with the identified failure potentials and the implementation patterns prevented all of the failures they were designed to prevent, we are reasonably confident that the findings can be applied to further scenarios.

8.2 Future Work

Based on our findings, we suggest the following three topics for future research. Most importantly, additional case studies are necessary to verify the generalizability of the findings of this thesis. The compatibility of transformations can be analyzed based on consistency constraint definitions. And additional strategies to mitigate the effects of failure potentials can be developed.

In future work, additional case studies with different domains and especially different transformation network topologies should be evaluated to verify the completeness and relevance of the failure potential catalog. In the initial case study, we only evaluated the findings with a cycle free transformation network. We showed that such a network can still exhibit similar properties as cyclic networks, because of cyclic correspondence graphs, but the case study only contained one type of cyclic correspondence graph. A cycle-containing transformation network would drastically increase the possibility of confluence problems and would likely change the distribution of failure potentials. The investigation of additional domains would help to verify the general applicability of the developed patterns.

Another topic of interest is the evaluation of transformation compatibility based on consistency constraints. Currently, the compatibility of transformations has to be manually evaluated through in-depth study of the implemented consistency constraints or through testing-based evaluation. Both approaches are error-prone and work-intensive. It would be preferable to find an automatic process for evaluating the compatibility of transformations. We introduced correspondence types to define the types of semantic overlaps that can exist between a pair of metamodel classes and we defined consistency for instances of the participating classes through consistency constraints. Transformations then have to implement the specified consistency constraints. When a set of transformations is combined to a transformation network, we may be able to derive potential correspondence graphs from the correspondence type definitions of the transformations. We can then collect the consistency constraints that apply to the correspondence types in the correspondence graph and analyze if the set of consistency constraints is satisfiable or if it contains contradicting constraints. If this proves successful, we could provide the transformation developer with valuable information about where a transformation may have to be adapted in order to achieve a correctly operating transformation network.

Unintentional change conflicts can occur if confluent paths propagate divergent information or through problematic transformation implementation. In this thesis, we proposed a transformation implementation pattern to detect and prevent element creation conflicts, but we currently have no pattern for the detection of feature change conflicts. One possible topic for future work could therefore be the development of a data structure that allows the transformation engine or the transformations to detect when a feature change conflict might lead to an unintended loss of information. If we record all changes that occur in the process of transitive change propagation, we can detect when a change conflict has occurred by comparing which features of which elements have been affected twice. However, some change conflicts are necessary, for example when a user performs multiple modifications on the same element or when a metamodel provided default value has to be overwritten in the initialization process. For a single transformation that has

no information about the change history both the intentional and unintentional feature change conflicts are indistinguishable and therefore not avoidable.

The following description is an initial design idea for a structure that enables the differentiation between intentional and unintentional feature changes. We can track all changes by their causal relation: parent change c_x caused (a transformation to produce) the child change c_y. If we interpret the tuples in that relation as edges of a graph, we obtain a cycle-free tree structure. We can then assign each change in the tree a priority based on its position in the tree structure. At any point in the tree, we assign the priorities by the following two rules: a parent change always has higher priority than all its child changes, and a newer child change and all its children have priority over all its siblings. The further a change was propagated, the lower its priority is and it cannot invalidate a parent change that has caused it. And newer changes have priority over older ones, which allows default values to be overwritten by later initialization changes. We can then extend the priority of a change to the element features affected by it. Now, if a change tries to modify a feature that has higher priority than the change itself, a problematic change conflict is detected, because overwriting the higher priority value would break the constraint that required the present value or it would overwrite a user input. And we can further differentiate, what the cause of the change conflict might be. If the priority of the target feature belongs to a parent change, then we can assume that a transformation loop has occurred. Otherwise we can assume that a more relevant change is already applied and should not be overwritten.

We suggest these three topics for future research, because we think they promise valuable insight while also being reasonably well realizable. Of course additional topics are also conceivable.

Bibliography

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

[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. URL: http://dx.doi.org/10.1016/j.jss.2008.03.066.

[3] Fei Chen. "Änderungsgetriebene Konsistenzhaltung zwischen UML-Klassenmodellen und Java-Code". Bachelor's thesis. 76128 Karlsruhe, Germany: Karlsruhe Institute of Technology, 2017.

[4] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. "From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case". In: *Journal of Object Technology* 10 (2011), 6:1–25. ISSN: 1660-1769. DOI: 10.5381/jot.2011.10.1.a6. URL: http://www.jot.fm/contents/issue_2011_01/article6.html.

[5] Zinovy Diskin et al. "From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case". English. In: *Model Driven Engineering Languages and Systems*. Ed. by Jon Whittle, Tony Clark, and Thomas Kühne. Vol. 6981. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 304–318. ISBN: 978-3-642-24484-1. DOI: 10.1007/978-3-642-24485-8_22. URL: http://dx.doi.org/10.1007/978-3-642-24485-8_22.

[6] Joshua Gleitze. "A Declarative Language for Preserving Consistency of Multiple Models". Bachelor's thesis. 76128 Karlsruhe, Germany: Karlsruhe Institute of Technology, 2017.

[7] 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. Vol. 5969. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 374–383. ISBN: 978-3-642-12106-7. DOI: 10.1007/978-3-642-12107-4_25. URL: http://dx.doi.org/10.1007/978-3-642-12107-4_25.

[8] Florian Heidenreich et al. *Jamopp: The java model parser and printer*. Tech. rep. 2009.

[9] 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.

[10] 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://doi.acm.org/10.1145/2489861.2489864.

[11] Max E. Kramer and Kirill Rakhman. "Automated Inversion of Attribute Mappings in Bidirectional Model Transformations". In: *Proceedings of the 5th International Workshop on Bidirectional Transformations (Bx 2016)*. Ed. by Anthony Anjorin and Jeremy Gibbons. Vol. 1571. CEUR Workshop Proceedings. Eindhoven, The Netherlands: CEUR-WS.org, 2016, pp. 61–76. URL: http://nbn-resolving.org/urn:nbn:de:0074-1571-4.

[12] Michael Langhammer. "Automated Coevolution of Source Code and Software Architecture Models". PhD thesis. 76128 Karlsruhe, Germany: Karlsruhe Institute of Technology, 2017.

[13] K. Lano and S. Yassipour-Tehrani. "Verified bidirectional transformations by construction". English. In: *CEUR Workshop Proceedings* 1693 (Oct. 2016), pp. 28–37. ISSN: 1613-0073.

[14] Nuno Macedo, Alcino Cunha, and Hugo Pacheco. "Towards a framework for multidirectional model transformations". In: *3rd International Workshop on Bidirectional Transformations - BX*. Vol. 1133. CEUR-WS.org. Athens, Greece: CEUR-WS.org, Mar. 2014. URL: http://ceur-ws.org/Vol-1133/paper-11.pdf.

[15] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. "A Taxonomy of Model Transformations". In: *Language Engineering for Model-Driven Software Development*. Ed. by Jean Bezivin and Reiko Heckel. Dagstuhl Seminar Proceedings 04101. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. URL: http://drops.dagstuhl.de/opus/volltexte/2005/11.

[16] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. "Making inconsistency respectable in software development". In: *Journal of Systems and Software* 58.2 (2001), pp. 171–180. ISSN: 0164-1212. DOI: https://doi.org/10.1016/S0164-1212(01)00036-X. URL: http://www.sciencedirect.com/science/article/pii/S016412120100036X.

[17] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. "A framework for expressing the relationships between multiple views in requirements specification". In: *Software Engineering, IEEE Transactions on* 20.10 (1994), pp. 760–773. ISSN: 0098-5589. DOI: 10.1109/32.328995.

[18] Object Management Group (OMG). *Meta Object Facility Specification Version 2.5.1*. Last accessed on 2018-05-29. URL: https://www.omg.org/spec/MOF/.

[19] Object Management Group (OMG). *MOF Query/View/Transformation Specification Version 1.3*. Last accessed on 2018-05-29. URL: https://www.omg.org/spec/QVT/.

[20] Object Management Group (OMG). *Unified Modeling Language Specification Version 2.5.1*. Last accessed on 2018-05-29. URL: https://www.omg.org/spec/UML/2.5.1/.

[21] 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.

[22] Software Design and Quality Group (SDQ), Karlsruhe Institute of Technology (KIT). *Media Store case study system*. Last accessed on 2018-05-29. URL: https://sdqweb.ipd.kit.edu/wiki/Media_Store.

[23] Software Design and Quality Group (SDQ), Karlsruhe Institute of Technology (KIT). *The Reactions Language – Vitruv Wiki*. Last accessed on 2018-05-29. URL: https://github.com/vitruv-tools/Vitruv/wiki/The-Reactions-Language.

[24] Software Design and Quality Group (SDQ), Karlsruhe Institute of Technology (KIT). *VIew-cenTRic engineering Using a VIrtual Underlying Single model(VITRUVIUS)*. Last accessed on 2018-05-29. URL: https://sdqweb.ipd.kit.edu/wiki/Vitruvius.

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

[26] Thomas Stahl et al. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Heidelberg: dpunkt-Verlag, 2007. ISBN: 978-3-89864-881-3.

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

[28] Misha Strittmatter and Amine Kechaou. *The Media Store 3 Case Study System*. Tech. rep. 2016,1. Faculty of Informatics, Karlsruhe Institute of Technology, Feb. 2016. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/3792054.

[29] The Eclipse Foundation. *Eclipse Modeling – MDT*. Last accessed on 2018-05-29. URL: http://www.eclipse.org/modeling/mdt/?project=uml2.

[30] The Eclipse Foundation. *Eclipse Modeling Framework (EMF)*. Last accessed on 2018-05-29. URL: http://www.eclipse.org/modeling/emf/.

[31] 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.

[32] Yingfei Xiong et al. "Synchronizing concurrent model updates based on bidirectional transformation". In: *Software & Systems Modeling* 12.1 (Feb. 2013), pp. 89–104. ISSN: 1619-1374. DOI: 10.1007/s10270-010-0187-3. URL: https://doi.org/10.1007/s10270-010-0187-3.