Building Transformation Networks for Consistent Evolution of Multiple Models

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

vorgelegte

Dissertation

von

Heiko Klare

aus Höxter

Tag der mündlichen Prüfung: XX. Monat XXXX
Erster Gutachter: Prof. Dr. Ralf H. Reussner
Zweiter Gutachter: Prof. Dr. Colin Atkinson

Abstract

Zusammenfassung

Contents Overview

Contents

List of Figures

List of Tables

Part I.

Prologue [35 p.]

1. Introduction [15 p.]

1.1. From DocSym

Model-driven Software Development proposes the usage of models as primary artifacts of the development process [SV06]. Those models describe different system properties for the interests of specific stakeholders, known as *multi-view modelling*, or at different abstraction levels, representing refinements. In both cases, the models describe the same system and are thus not disjoint but contain redundant or dependent information. Developers must be aware of those dependencies to ensure that models are modified consistently.

In large software systems, a single developer cannot know about all dependencies [PRV08], in the following referred to as *consistency relations*, which inevitably leads to inconsistencies. Therefore, automated mechanisms that preserve consistency according to those consistency relations are necessary. For that purpose, incremental, bidirectional model transformations are commonly used. However, most research considers *binary transformations*, restricted to pairs of models, and does not explicitly consider consistency between more than two models [Ste17], which we refer to as *multi-model consistency*. Model transformations can either be specified imperatively or declaratively. They differ in who operalizationalizes the preservation of constraints that have to hold, in the first case the transformation developer and in the second case an automated mechanism of the transformation language. This is why we do not explicitly distinguish these approaches, as all problems apply to both approaches and only different roles have to deal with them.

Although it is possible to combine binary transformations by transitively executing them, it is yet unclear what problems may arise from that, especially if each transformation is developed independently and treated as a black box. We will exemplify this on the simple example in Figure 1.1, in which consistency relations define a mapping of a component in an Architecture Description Language (ADL) to a class in object-oriented design, which is again represented by an implemented class in Java code. The name of the class is defined to be the component name with an “Impl” suffix (cf. [Lan17]). When all these relations are expressed in transformations, it is, for example, possible that both transformations from ADL to Java, once over UML (R_1 and R_2) and once directly (R_3), create a Java class after creating an ADL component. We refer to that as an *interoperability problem*. The transformation specification would have to avoid an overwrite and therefore have to consider dependencies between transformations, using, for example, a shared trace model. In general, an interoperability problem is an unexpected behavior of transformations, which only occurs if they are executed transitively, but not if each is executed on its own.

Additionally, it is easy to see that combining binary transformations leads to trade-off decisions. The ternary relation can either be expressed by three binary transformations between all pairs of metamodels or by two binary transformations with the third being the combination of the two others. The first option leads to redundancies in the specifications, as each pair of transformations has to have an equal semantics than the third. For example, the ADL to Java transformation for R_3 must be equal to the combination of the transformations ADL to UML (R_1) and UML to Java (R_2). Consequently, those transformations may be incompatible if not correctly defined, e.g., by leaving out the suffix addition in the transformation for R_3. An alternative is to omit the transformation for R_3 by transitively executing the two others. However, in this case, modularity is reduced, because it is not possible to use only Java and the ADL to develop a specific system and omit the UML. We refer to this as *specification trade-offs*.

Instead of developing approaches to express multiary consistency relations, there are reasons to adhere to binary transformations, and to research their combinability. As stated by Stevens [Ste17], it is hard enough to think about binary relations. Additionally, each domain expert, who specifies transformations, will usually only have knowledge about the relations between two or at most a rather limited set of metamodels. We therefore plan to make the following contributions to research on multi-model consistency preservation:

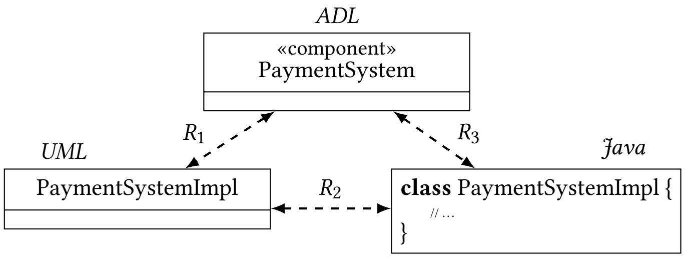


Figure 1.1.: Example models with binary consistency relations

Transformation interoperability. When several binary transformations are developed independently, they must be combinable in a black box manner, introduced as the *interoperability problem*. We will therefore identify problems that can arise from that combination and develop a catalog of patters that can be followed by the transformation developer or language to achieve *non-intrusive* interoperability of binary transformations.

Decomposition of consistency relations. The usage of binary transformations for multi-model consistency preservation leads to *specification trade-offs* regarding essential challenges. We will provide a classification of those challenges and investigate the influence of the way in which transformations are specified on them. We will especially investigate how consistency relations can be decomposed into independent subsets, as this allows a partial optimization regarding those challenges.

Make common concepts explicit. Metamodels often represent the same concepts in different ways. As another contribution to reduce *specification trade-offs*, we propose an approach to make these common concepts explicit to improve comprehensibility of transformations and to improve their modular reuse.

Throughout this paper, we use a simplified notation for metamodels and heir consistency relations to ease their illustration. We consider metamodels to be sets of elements and consistency relations to be sets of symmetric, binary relations between those elements. To ease the representation of combinations of consistency relations, we define the concatenation operation for two consistency relations R_1 and R_2 as:

$$R_1 \otimes R_2 \coloneqq \{(x, y) \mid \exists\, t : x\, R_1\, t\, R_2\, y\}.$$

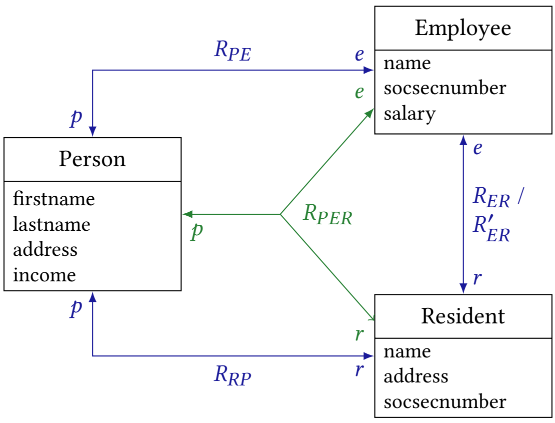
This is the subset of the transitive closure of two relations that contains only the relations transitively defined over R_1 and R_2. It can be also expressed as the natural join of R_1 and R_2 with an additional projection that removes the common elements of both relations. The operator is commutative since the relations are assumed symmetric.

1.2. From SoSym MPM4CPS

The scale of modern software systems and their embedding into cyber-physical systems leads to a high and even increasing complexity of systems to be built. To handle that complexity, different roles operate on appropriate extracts and abstractions of the system under construction described by different models or views. Such a fragmentation of information across different models is common at a high level, i.e., mechanical, electrical and software engineers usually use different models and associated tools to describe a system in their domain. Additionally, different models can be used on a low level by engineers from the same domain, such as software engineers using different models for architecture specification, behavior development and deployment. For example, the development of Electronic Control Units (ECUs) software in automotives comprises different tools or standards for specifying the system and software architecture, such as SysML [Obj19] or AUTOSAR [Sch15], for defining the behavior, such as MATLAB/Simulink [Mat] or ASCET [ETA], and for defining the deployment on multi-core hardware architectures, such as Amalthea [ITE; Wol+15]. Since all these models describe the same system, they usually share an overlap of information in terms of dependencies or redundancies, which can lead to inconsistencies if overlapping information is not modified properly in all models. Recent research investigated such dependencies between ASCET and SysML [GHN10], as well as Amalthea and how to resolve them [Maz+17; Maz16].

Incremental model transformations are a common approach to resolve such inconsistencies by enabling developers to explicitly specify how inconsistencies can be resolved (semi-)automatically. Especially bidirectional model transformations [Ste10], which specify the relations between two metamodels and routines how consistency of their instances can be restored, are well suited and well researched. Relating more than two metamodels can either be achieved by defining a multi-directional transformation between all of them or by specifying bidirectional transformations between pairs of them in a modular way and combine them to a network that is able to check and preserve consistency between several models. Figure 1.2 exemplifies these different possibilities at the example of relation of transformations between three simple metamodels for persons, employees and residents. We use an informal notion of consistency, defined more precisely later on, which requires that if any person, employee or resident is contained in a model, there must also be the other two elements with the same names, addresses, incomes and social security numbers. are equal. This relation can either be expressed as a ternary relation, denoted as R_{PER}, or as three binary relations R_{PE}, R_{PR}, R_{ER}. In such a simple scenario a single developer may be able to define all these relations. However, in a more complex scenarios, like the relations between the previously mentioned SysML, Amalthea and ASCET metamodels, there may not be a single person having the knowledge about all these dependencies [PRV08], but there may be different domain experts knowing about relations between subsets of the metamodels [Kla18]. Additionally, it is difficult to think about complex multiary relations [Ste17]. In consequence, building networks of bidirectional transformations provides several benefits over building multi-directional transformations.

Such a network of bidirectional transformations may contain cycles of transformations. Figure 1.2 exemplifies why it may be unavoidable to have such cycles. There is no pair of binary relations, such that it is equivalent to the ternary relation R_{PER}, because each pair of metamodels shares unique information that is not represented in the third one. An essential issue with such cycles is that they impose the possibility of defining contradictory constraints, such that the relations cannot be fulfilled at the same time. In such a case, the relations are considered *incompatible*. Consider the three binary relations R_{PE}, R_{PR}, R'_{ER} in Figure 1.2. These relation cannot always be fulfilled, because R'_{ER} requires the resident name to be lowercase, whereas the other relations relate the names as they are and thus allow the lowercase



$$
\begin{aligned}
R_{PER} = \{\langle p, e, r\rangle \mid\ & p.firstname + \text{"␣"} + p.lastname = e.name = r.name \\
& \wedge p.address = r.address \wedge p.income = e.salary \\
& \wedge e.socsecnumber = r.socsecnumber\}
\end{aligned}
$$

$$
\begin{aligned}
R_{PE} &= \{\langle p, e\rangle \mid p.firstname + \text{"␣"} + p.lastname = e.name \\
&\quad \wedge p.income = e.salary\} \\
R_{PR} &= \{\langle p, r\rangle \mid p.firstname + \text{"␣"} + p.lastname = r.name \\
&\quad \wedge p.address = r.address\} \\
R_{ER} &= \{\langle e, r\rangle \mid e.name = r.name \\
&\quad \wedge e.socsecnumber = r.socsecnumber\} \\
R'_{ER} &= \{\langle e, r\rangle \mid e.name.toLower = r.name \\
&\quad \wedge e.socsecnumber = r.socsecnumber\}
\end{aligned}
$$

Figure 1.2.: Three simple metamodels for persons, employees and residents, one ternary relation R_{PRE} between them and three binary relations R_{PE}, R_{PR}, R_{ER} for each pair of them, with R'_{ER} as an alternative for R_{ER}.

names. In consequence, for a resident with a non-lowercase name it is not possible to find a consistent person and employee. However, in a transformation network, compatibility of the relations defined by the transformations is a necessary requirement for their repair routines to properly restore consistency [Kla+19].

In this article, we consider the relations defined by bidirectional transformations. We clarify the notion of *compatibility* of these relations and develop an approach to prove compatibility of relations in a given network of transformations. To achieve this, we formally define a notion of consistency, based on fine-grained consistency relations, as well as compatibility. Building on this formalism, we are able to derive an inductive, formal approach for proving compatibility of relations by identifying those that are redundant. The essential idea is that if consistency relations have a specific kind of tree structure, we are able to show that they are inherently compatible. Furthermore, we show that adding redundant relations to such a tree preserves compatibility. In consequence, reducing an arbitrary network of relations to a tree by removing redundant relations proves compatibility. Finally, we present an operationalized approach based on that formal approach for QVT-R to prove compatibility of a network of QVT-R relations. That approach transforms QVT-R relations into first-oder logical formulae and finds redundant relations by applying an SMT solver. More detailed, we make the following contributions:

Compatibility Formalization (C1): We formalize a notion of consistency and precisely define *compatibility* of relations in a network of transformation.

Formal Approach (C2): We define a formal, inductive approach for proving compatibility of relations based on a notion of redundancy and relation trees.

Operationalized Approach (C3): We propose an approach that applies the formalism to QVT-R and show how a translation to logical formulae and the usage of SMT solver can be used to prove compatibility.

Applicability Evaluation (C4): While correctness of the approach is given by construction and proven on the formalism, we apply the approach to case studies to show applicability of the approach.

It is, in general, not possible to prove that transformations are incompatible if the language used to describe consistency relations has sufficient expressiveness and is thus undecidable, such as QVT-R. On the other hand, it is possible to prove that transformations are compatible. Our approach is designed to operate conservatively, thus in cases it claims compatibility, the transformations actually are compatible. However, there may be cases in which relations are compatible but the approach is not able to prove that.

The main benefit that our approach imposes is that it enables domain experts to define transformations independently and to automatically detect their compatibility both during development as well as afterwards when combining them to a network. This relieves them from the necessity to align the transformations with each other a priori and ensuring compatibility manually.

Discuss different benefits / scenarios more clearly, especially second benefit / application scenario: User define relations and wants on-the-fly feedback on compatibility to other existing relations, rather then a posteriori checking

Research Goal 1. A transformation developer shall know about all necessary properties of a transformation network to achieve its correct execution and he or she shall be provided with techniques to guarantee these properties by construction whenever possible and otherwise techniques that allow him or her to check them.

1.3. Research Questions

Properties of Transformation Networks

RQ 1. Which issues can occur when independently developed Bidirectional Transformations (BX) are combined to a network?

- *RQ 1.1.* Which failures can occur, when BX are combined to a network?
- *RQ 1.2.* What mistakes can be made that lead to failures?
- *RQ 1.3.* How can these mistakes be categorized regarding conceptual levels in the specification process for BX?

RQ 2. How are properties of transformation networks affected by the topology?

- *RQ 2.1.* Which properties are relevant when defining networks of BX?
- *RQ 2.2.* Which topologies of network exist and how do they affect that properties?

Evaluation

Keine dezidierte Evaluation, lediglich Argumentation

Building Correct Transformation Networks

RQ 3. How can transformations be analyzed regarding contradictions in specified constraints?

- *RQ 3.1.* What is an appropriate formalism for describing transformations that can be analyzed regarding potential contradictions?
- *RQ 3.2.* Which kinds of contradictions can be detected by analyzing transformations following a specific formalism?

RQ 4. How can interoperability of independently developed BX be achieved by construction?

- *RQ 4.1.* Which kinds of mistakes can be avoided by construction of the individual BX?
- *RQ 4.2.* How can we prove that those mistakes and only those mistakes can be avoided by construction?
- *RQ 4.3.* How can each of these mistakes be avoided by a transformation developer during independent development of a single BX?

RQ 5. What is an appropriate strategy for orchestrating independently developed BX to perform a fixed-point iteration?

- *RQ 5.1.* Which strategies for orchestrating a network of BX exists and what are their properties?
- *RQ 5.2.* How should those properties be weighted and which of the strategies should be chosen for orchestration?

Evaluation

Goal (Functionality): The analysis can be used to find contradictions in specifications
Question: Does the analysis find contradictions if they exist?
Metric: Recall: Ratio of true positives to true positives + false negatives
Question: Does the analysis find contradictions although they do not exist?

Metric: Precision: Ratio of true positives to true+false positives
Question: Does the analysis find non-contradictions although they exist?
Metric: Ratio of false negatives to false+true negatives

Goal (Functionality): The techniques to avoid mistakes by construction actually avoid interoperabililty issues
Question: Are the identified failures that can occur complete?
Metric: Ratio of number of identified failures to total number of failures
Question: Are the relations of identified mistakes to identified failures correct?
Metric: Ratio of failures resolved by fixing the identified mistake to all failures
Question: Does the application of avoidance techniques lead to interoperable transformations?
Metric: Ratio of changes that are propagated correctly to those that are not propagated correctly

Goal (Applicability): The techniques can be applied independently to single transformations
Question: Are there cases in which information about other transformations are necessary to solve issues?
Metric: Ratio of number of fixes that require information about other transformation to total number of fixes with user interactions
Ratio of number of fixes that require information about other transformation to total number of fixes without user interactions

Improving Non-Functional Properties of Transformation Networks

RQ 6. How can a topology of transformation be build that optimizes non-functional properties of transformation networks?

RQ 6.1. How can transformation contradictions be avoided by language design?

  RQ 6.2. How can modularity be achieved in a way such that an arbitrary set of metamodels for which consistency is specified can be used in an actual project?

RQ 7. How should a language specific for multi-model consistency be defined that supports a non-functional property-optimizing topology definition?

  RQ 7.1. What are the design decision for such a language?

Evaluation

Goal (Functionality): Concept and language can achieve consistency between several models
Question: How many model changes in a case study can be properly kept consistent?
Metric: Ratio of successfull test cases

Goal (Practicality): The assumption of defining a tree of Commonalities is achievable in practice
Question: Is the definition of cross-tree relations necessary in a case study?
Metric: Number of cross-tree relations in a case study compared to number of relations

Goal (Practicality/Benefit): A specific language improves conciseness of consistency specifications
Question: How much more concise is the specification for a case study compared to a definition with direct transformations?
Metric: Number of SLOC with Commonalities compared to number of SLOC with Reactions for same case study

Diskussion: Erreichen der Modularität auch evaluieren? Ist per Konstruktion gegeben, könnte man aber natürlich auch noch auswerten (bringt aber nichts).

2. Foundations and Notation [20 p.]

2.1. Notation and Assumptions

In this section, we introduce basic terminology, notations and assumptions that we make throughout the article. We give an overview of the used notation and elements in Table 2.1.

2.1.1. Notation

We usually denote variables representing sets of any kinds of elements in blackboard bold font $\mathbb{S}$, those representing tuples of any kinds of elements in gothic font $\mathfrak{T}$ and write the elements of a tuple in angle brackets $\langle a, b, \ldots \rangle$.

We define the following operators for concise expressions on tuples, which basically allow to treat tuples as sets wherever necessary: For a tuple $\mathfrak{t} = \langle t_1, \ldots, t_n \rangle$, we say that:

$$t' \in \mathfrak{t} :\Leftrightarrow \exists\, i \in \{1, \ldots, n\} : t' = t_i$$

For two tuples $\mathfrak{t}1, \mathfrak{t}2$, we define:

$$\mathfrak{t}1 \cap \mathfrak{t}2 := \{t \mid t \in \mathfrak{t}1 \wedge t \in \mathfrak{t}2\}$$

Note that the intersection of tuples is not a tuple but a set, because we are only interested in getting the elements contained in both tuples but do not need to match their order.

Notations	
$\mathbb{S} = \{a, b, \ldots\}$	Notation for a set $\mathbb{S}$ of elements
$\mathfrak{T} = \langle a, b, \ldots \rangle$	Notation for a tuple $\mathfrak{T}$ of elements
Properties and Classes	
P	Property (attribute or reference)
$I_P = \{p_1, p_2, \ldots\}$	Property values of a property P
$C = \langle P_1, \ldots, P_n \rangle$	Class
$I_C = \{o = \langle p_1, \ldots, p_n \rangle \mid p_i \in I_{P_i}\}$	Instances (objects) of a class C
$o \in I_C$	Object of a class C
(Meta-)Models	
$M = \{C_1, \ldots, C_m\}$	Metamodel
$I_M = \{m \mid m \subseteq \bigcup_{C \in M} I_C\}$	Instances of a metamodel
$\mathbb{M} = \{M_1, \ldots, M_k\}$	Set of metamodels
$I_{\mathbb{M}} = \{\{m_1, \ldots, m_k\} \mid m_i \in I_{M_i}\}$	Instances of a metamodel set $\mathbb{M}$
$m \in I_M$	Model of metamodel M
$\mathbb{m} \in I_{\mathbb{M}}$	Model set of a metamodel set $\mathbb{M}$

Table 2.1.: Notations and elements

Furthermore, we use variables of uppercase letters for all elements at the metamodel level, such as M for a metamodel or C for a class, whereas we use lowercase letters for all elements at the model level, such as m for a model and o for an object. If not further specified, we use the same indices on related elements on the metamodel and the model level, such as model m_1 being an instance of metamodel M_1.

2.1.2. Elements

In general, we consider metamodels as a composition of meta-classes, which in turn are composed of properties representing attributes or references. Models instantiate metamodel and are composed of objects, which are instances of meta-classes and in turn consist of property values, which instantiate properties.

We denote *properties*, which are the information a meta-class consists of, such as attributes or references, as P and the *property values* as instances of a property as $I_P = \{p_1, p_2, \ldots\}$ of property P. We do not need to further differentiate different types of properties into attributes and references, like it is done in other formalizations, such as the OCL standard [Obj14b, A.1] or the thesis of Kramer [Kra17, p. 2.3.2].

We denote *meta-classes*, in the following shortly called *classes*, as tuples of properties $C = \langle P_1, \ldots, P_n \rangle$. Instances of a *class* are *objects*, each being a tuple of instances of the properties of the class and we denote all instances of a class C as $I_C = \{o = \langle p_1, \ldots, p_n \rangle \mid p_i \in I_{P_i}\}$.

We denote a metamodel $M = \{C_1, \ldots, C_m\}$ as a finite set of classes. The instances of a metamodel are sets of objects $I_M = \{m \mid m \subseteq \bigcup_{C \in M} I_C\}$. Each instance, denoted as a *model*, is a finite set of objects that instantiate the classes in the metamodel. For a set of metamodels $\mathbb{M} = \{M_1, \ldots, M_k\}$, we denote the set that contains all sets of instances of that metamodels as $I_{\mathbb{M}} = \{\{m_1, \ldots, m_k\} \mid m_i \in I_{M_i}\}$.

2.1.3. Assumptions

We assume models to be finite, so for each model m, we assume that $|m| < \infty$. Additionally, our formalism assumes objects to be unique within a model m. This is already implicitly covered by the definition of I_M for the instances of a metamodel M. In practice, it is usually allowed to have the same object, i.e., an element with the same type, attribute and reference values, multiple times within the same model. This is, however, only a matter of identity, which we assume, without loss of generality, to be represented within the objects, whereas identity in practice is given by different objects being placed at specific places in memory.

Finally, we assume consistency to be defined in terms of multiple bidirectional transformation, even if more than two metamodels are related. We already discussed in chapter 1 why this is a reasonable approach in comparison to defining multi-directional transformations based on complexity and available knowledge, as argued in existing work [Ste17; Kla18].

Discuss valid models, why we do not consider them and prove that invariants + consistency relations can express any consistency relation. A single relation can consider constraints on the single models, but combination (even within one transformation) can easily contradict constraints of a model.

2.2. Types of Consistency

Discuss which types of consistency may exist (from Ellwangen retreat), how far they can be checked and what the mismatch is between pragmatic types of consistency and their verifyability.

2.3. Transformational Consistency of Multiple Models

Compare multidirectional transformations with networks of transformations. Refer for other consistency approaches to related work.

2.4. Assumptions / Limitations

For ease of understanding, we restrict us to binary consistency preservation rules, although one could define the same for multiary ones.

Part II.

Classifying Transformation Networks [40 p.]

3. Topologies and Properties [15 p.]

3.1. Assumptions and Terminology

We shortly clarify our assumptions and introduce a terminology for consistency that we use to explain our classification. We assume that consistency of more than two types of models is specified using networks of BX rather than multidirectional approaches for two reasons: First, it is easier to think about binary than about n-ary relations [Ste17]. Second, a domain expert usually only knows about consistency relations within a subset of all model types used to develop a system, so modularizing transformations is inevitable. It was also the result of a Dagstuhl seminar that "it seems likely that networks of bidirectional transformations suffice for specifying multidirectional transformations" [Cle+19, p. 7]. Finally, we investigate of a subset of problems that can actually occur, as in a concrete scenario n-ary relations may exist that cannot be expressed by sets of binary relations. Although we limit our considerations to the assumed scenarios, most of our findings could also be extended to a modularization into smaller n-ary relations rather than binary relations.

Definition 1 (Model)**.** A model $M = \{e_1, e_2, \ldots\}$ is a finite set of not further defined elements, such as objects, attribute and reference values.

The exact representation of the model contents is not relevant for our work, which is why we use this lightweight definition. It allows us to transfer the insights to arbitrary models, such as models that are conform to the Essential Meta Object Facility (EMOF) [Obj16b].

Definition 2 (Model Type)**.** A model type $\mathcal{M} = \{M_1, M_2, \dots\}$ is the (usually but not necessarily infinite) set of all models $M_1, M_2, \dots$ that are instances of $\mathcal{M}$.

In the following, let a model M_i be always an instance of model type $\mathcal{M}_i$. This definition constitutes an *extensional description* of models and does not explicitly consider actual instantiation relations between classes and objects, attributes and their values etc., other than containment in the respective model type. We also use the term *metamodel* when referring to an abstract syntax of classes, attributes and associations, as defined in the OCL standard [Obj14b, A.1]. A metamodel constitutes an *intensional description* of models, from which the model type could be derived by enumerating all valid instances, i.e., all models with arbitrary instantiations of classes, their attributes and associations.

Definition 3 (Consistency Specification)**.** A *consistency specification* CS for model types $\mathcal{M}_1, \dots, \mathcal{M}_n$ is a relation $CS \subset \mathcal{M}_1 \times \dots \times \mathcal{M}_n$ between models that are consistent. We denote a binary consistency specification for model types $\mathcal{M}_i$ and $\mathcal{M}_j$ as $CS_{i,j}$.

Enumerating consistent instances to define consistency is comparable to [Ste17]. If there are no restrictions on when models are consistent, CS contains all tuples of models. We denote restrictions for models to be in CS as *consistency constraints.* It would, in theory, also be possible to define CS on an infinite number of model types. However, for ease of understanding and because of missing practical examples, we decided to fix the number of model types in a consistency specification.

We primarily consider binary consistency specifications, which are the binary relations that define consistency pairs of models, and also binary specifications for consistency preservation, which are functions that restore consistency between two models after one of them was modified. In the following, we introduce such consistency preservation specifications. Each consistency preservation specification concerns modifications in instances of two model types. However, instead of defining such a function on two model types, we define it on an arbitrary number of model types, but restrict modifications to instances of two of them. In consequence, a set of binary consistency preservation specifications for an arbitrary number of model types can be defined, whose signatures of input and output are all equal. This leads to a rather verbose definition of consistency preservation specifications, but eases the composition of such functions between more than two model types. If the function only considered the two involved model types, the composition definition would have to properly consider matching function signatures, whereas our definition allows the composition of all functions with each other. A consistency preservation specification expects and returns a tuple of pairs, each representing a change by containing an original and a modified model. The original models in a tuple are always consistent, but a specification may update the modified models.

Definition 4 (Consistency Preservation Specification)**.** For a binary consistency specification $CS_{i,j}$, a *consistency preservation specification* $CPS_{CS_{i,j}}$ is a partial function defined if $(M_i, M_j) \in CS_{i,j}$ that maps a tuple of model pairs, each containing an original model $M_k \in \mathcal{M}_k$ and a modified model $M'_k \in \mathcal{M}_k$, to a new tuple of model pairs:

$$\begin{aligned}
CPS_{CS_{i,j}} : \big((\mathcal{M}_1, \mathcal{M}_1), \ldots, (\mathcal{M}_n, \mathcal{M}_n)\big) \rightarrow \big((\mathcal{M}_1, \mathcal{M}_1), \ldots, (\mathcal{M}_n, \mathcal{M}_n)\big),\\
\big((M_1, M'_1), \ldots, (M_i, M'_i), \ldots, (M_j, M'_j), \ldots, (M_n, M'_n)\big)\\
\mapsto \begin{cases} \big((M_1, M'_1), \ldots, (M_i, M''_i), \ldots, (M_j, M''_j), \ldots, (M_n, M'_n)\big) & (M_i, M_j) \in CS_{i,j} \\ \mathit{undefined} & \mathit{otherwise} \end{cases}
\end{aligned}$$

so that

$$(M''_i, M''_j) \in CS_{i,j}$$

Remark: A specification that always maps to empty models would be valid regarding our definition. It is up to the developer to provide reasonable specifications.

We are interested in consistency preservation specifications that can be executed in arbitrary order, so that they finally terminate in a consistent state regarding all consistency specifications, comparable to a fixed-point iteration. Therefore, it is essential for all specifications to be hippocratic [Ste10], so that no changes are performed when models are already consistent. Let $\mathcal{CPS}$ be a set of preservation specifications for consistency specifications

$\mathcal{CS}$. We denote the set of consistent model tuples regarding $\mathcal{CS}$ as $\mathfrak{M}_{\mathcal{CS}} = \{(M_1, \dots, M_n) \mid \forall\, CS_{i,j} \in \mathcal{CS} : (M_i, M_j) \in CS_{i,j}\}$. We want to achieve that:

$$\begin{aligned}
&\forall (M_1, \dots, M_n) \in \mathfrak{M}_{\mathcal{CS}} : \forall\, M'_1 \in \mathcal{M}_1, \dots, M'_n \in \mathcal{M}_n : \exists\, CPS_1, \dots, CPS_k \in \mathcal{CPS} : \\
&\quad CPS_1 \circ \dots \circ CPS_k\big((M_1, M'_1), \dots, (M_n, M'_n)\big) = \big((M_1, M''_1), \dots, (M_n, M''_n)\big) \\
&\qquad \wedge \forall\, CS_{i,j} \in \mathcal{CS} : (M''_i, M''_j) \in CS_{i,j}
\end{aligned}$$

This means that there is always a sequence of consistency preservation specification applications, potentially with multiple applications of the same specification, that ensures that the modified models in all tuples are consistent after applying it.

Declarative transformation languages are usually well suited to define consistency specifications according to Definition 3, from which a consistency preservation specification is derived. Imperative transformation languages can be used to define consistency preservation specifications according to Definition 4.

3.2. Properties of Transformation Networks

3.2.1. Binary Transformation Interoperability

Multi-model consistency preservation can be a achieved by combining binary transformations to graphs, with the transformations being executed transitively. Since all binary transformations are developed independently of each other, it is necessary that they interoperate properly in a *non-intrusive* way, thus without the necessity for the developer to understand and modify them, which we refer to as *black-box combination.*

Even under the assumption that, in contrast to our introductory motivation, all specifications are free of contradictions, it is easy to see that problems arise when combining binary transformations by transitively executing them. For example, consider the relations in Figure 1.1. If a component is added to the ADL, causing a UML class creation due to R_1, which in turn causes a Java class creation due to R_2, the transformation for relation R_3

does not know that an appropriate class was already created, if the transformations are treated as black boxes. Consequently, the transformation will create the same class again, which may override the existing one, depending on the implementation and execution order. A simple solution for this example would be to have all transformations use a common trace model and check for existing elements before creating them in a transformation. Nevertheless, independently developed transformations will usually not assume that other transformations may already have created corresponding elements. Additionally, the trace model must allow the transformation engine to retrieve transitive traces. However, it is unclear if transitive resolution of traces can always be performed, as it can depend on whether the transitive trace belongs the considered consistency relation or another.

As can be seen in the example, especially the correct handling of trace information in interdependent transformations has to be researched. This applies not only to element creations, but also other change types, such as attribute or reference changes, especially if they are multi-valued. In our thesis, we will therefore apply transitively executed binary transformations in different case studies to identify these and potential further problems. We then want to come up with a catalog of such problems together with solution patterns for them. For example, to avoid duplicate element creations, a simple pattern could be to always check for already existing traces for that consistency relation in the transformations. In consequence, the integration of those patterns into a transformation language or the application of them as a transformation developer is supposed to achieve black-box combinability of the transformations.

3.2.2. Challenges and Topology Trade-Offs

Even if independently developed transformations are interoperable, as discussed before, higher-level challenges remain when considering the way in which transformations express relations or occur if compatibility between the transformations is not given. We present already identified challenges in the following.

Compatibility Transformations preserve consistency, but also have to be consistent among themselves, i.e., they have to adhere to the same consistency relations, referred to as *compatibility*. If, in our example in Figure 1.1, the relation R_3 between ADL and Java is realized in addition to the transitive relation $R_1 \otimes R_2$ over UML, both relations must contain the same name attribute mapping. If the ADL to Java transformation adds an “Impl” suffix [Lan17], whereas the transformations over UML omit that, they are *incompatible*. This can lead to propagation cycles due to alternating values, and to results depending on the transformation execution order. While *compatibility* concerns problems due to the realization of contradictory consistency relations, *interoperability* concerns potentially unexpected behavior although all transformations follow to the same consistency relations.

Modularity The development of different systems requires the usage of different metamodels to describe them. Therefore, transformations should be modular, so that an arbitrary selection of metamodels and transformations between them can be used within a concrete project. If, in our example, transformations are specified transitively across UML, it is impossible to use only the ADL and Java in a concrete project, omitting the UML.

Comprehensibility Maintainability of artifacts, including transformations, depends on their comprehensibility. Comprehensibility can be seen as the number of transformations to consider if a specific consistency relation shall be understood. Realizing consistency relations in transitive transformations can, for example, reduce comprehensibility, as a developer has to consider a set of transformations to understand a single consistency relation.

Evolvability Whenever new transformations shall be defined, e.g. because a new metamodel shall be used and transformations for it have to be defined, the effort should be as low as possible. This concerns the number of transformations to define and how many metamodels are involved in the transformations for one consistency relation if it is expressed transitively.

We will focus on compatibility and modularity, as they are crucial for the applicability of transformations. Defining binary transformations for a set of metamodels leads to a trade-off solution regarding those challenges, as they cannot be solved simultaneously. The intuitive way to define transformations is to express each relation between two metamodels in one transformation, which leads to a dense graph of transformations. In the extreme case, if all pairs of metamodels have non-empty consistency relations, the graph is complete, as shown in Figure 3.1a. In that case, modularity is high

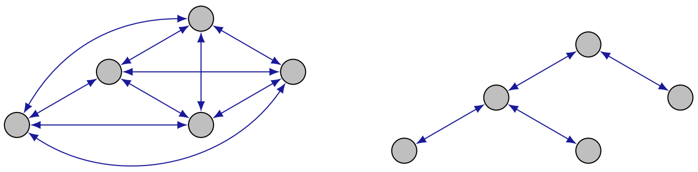
Challenge	Dense Graph	Tree
Compatibility	-	++
Modularity	++	-
Comprehensibility	+	-
Evolvability	-	+

Table 3.1.: Challenge fulfillment by transformation topology

because each metamodel can be excluded without any drawback, but relations are likely to be incompatible, as, in the worst case, each relation is specified over $(n-1)!$ transformation paths if n metamodels are involved. While comprehensibility is high, as each relation is explicitly expressed, adding a metamodel requires to define up to $n-1$ transformations, implying high evolution effort.

Another extreme case is to have each consistency relation only defined over a single path in the transformations graph, which results in a tree of transformations, as shown in Figure 3.1b. In that case, compatibility of transformations is inherently given, as each relation is only defined once, but modularity is reduced, as only metamodels being leaves can be omitted. Comprehensibility is low, as each relation may be defined in a path of up to $n-1$ transformations, but evolvability is rather good, as each relation must only be defined once. We summarize that impact of the topology in Table 3.1.

A tree topology has the drawback that it is not always applicable. It requires that the transformation developers find a subset of all consistency relations between the metamodels that induces a tree and whose transitive closure



(a) Dense graph **(b)** Tree

Figure 3.1.: Extreme examples for transformation topologies

contains all other relations. In general, such a subset cannot be found. Of three metamodels, there must always be one containing the overlapping information of the two others, as only then the transitive closure of two consistency relations subsumes the third. In the example in Figure 3.2, it is necessary that two relations are contained in the transitive closure of the others to get a tree that covers all relations.

In practice, the used topology will potentially be a mixture between those extremes. The natural way to foster the independent development of transformations would be to define one transformation for each consistency relation, resulting in a dense graph with a high potential for incompatible transformations. To deal with that, mechanisms that analyze compatibility between transformations could be researched. Nevertheless, high expressiveness of transformation languages allows only conservative approximations. In our thesis, we will therefore investigate approaches that result in tree-like specifications that directly imply compatibility between the transformations, but with increased *applicability* and *modularity*.

3.3. Topologies of Transformation Networks

Map properties to topologies, also from doc sym and more precisely

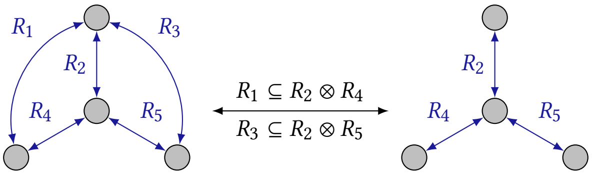

R_1
R_3
R_2
R_4
R_5
$R_1 \subseteq R_2 \otimes R_4$
$R_3 \subseteq R_2 \otimes R_5$
R_2
R_4
R_5

Figure 3.2.: Reducing a consistency relation graph to a tree

4. Consistency Specification Levels [15 p.]

Models that contain concern-specific extracts of a system are a means to deal with the increasing complexity in today's software development. A common approach for preserving consistency between such models are incremental BX, which keep two types of models consistent. Usually, more than two types of models are used in development processes. Keeping them consistent can be achieved by combining BX to networks, which has not been focused in research yet [Ste17]. When such networks contain cycles, information can be propagated across different paths during transformation execution, which may lead to problems on confluence.

Consider the simple consistency relations exemplified in Figure 4.1. A company uses three software systems to manage (1) personnel data, (2) tasks and their assignment to employees, and (3) schedules for work times of employees and the deadlines of tasks. The domain models contain dependent information, especially the data about employees and their relations to tasks, but none of them contains a superset of information of another, which requires to define consistency between all pairs of them. If three domain experts define those binary constraints independently, they can easily contradict. For example, imagine a direct mapping of employee *name* representations between the task management and scheduling system, a concatenation of *firstname* and *lastname* between personnel data and task management system and a comma-separated concatenation of *lastname* and *firstname* between personnel data and scheduling system. These constraints are obviously incompatible, as they cannot be fulfilled at the same time.

While such a problem may be trivially solvable in this simple scenario, it gets difficult in systems with more and larger metamodels, where each domain expert only knows about the relation between two of them, but not

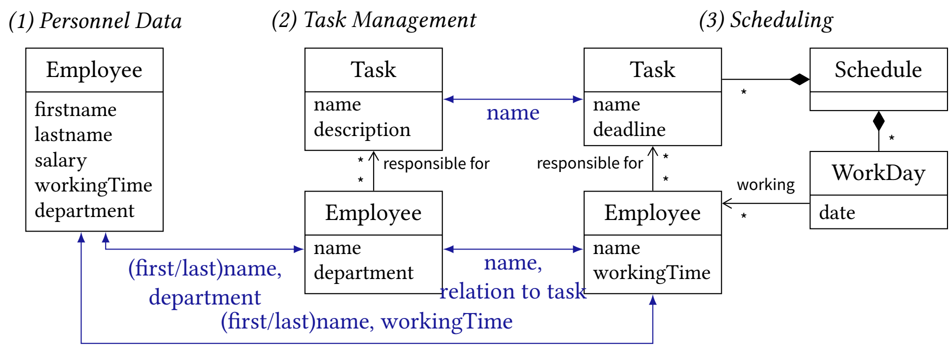


Figure 4.1.: Exemplary Consistency Relations (←→) between Three Simple Metamodels

about the others. In consequence, each BX has to be constructed in such a way that it can be combined with other, independently developed BX in a *black-box* manner later on. Issues that arise from such a combination of independently developed BX have not been investigated yet. In consequence, potential failures, causal mistakes and techniques to avoid them by design are not systematically known.

Our research goal is to identify and categorize issues that can arise from the combination of independently developed BX to networks and how those issues can be avoided by construction. Our main contributions in this paper are:

Classification of consistency specification levels (C1): We identify different conceptual levels at which consistency for a set of model types can be defined.

Categorization of interoperability issues (C2): We identify potential failures and mistakes in transformation networks and relate them to the specification levels.

Issue avoidance strategies (C3): We discuss avoidance strategies for mistakes at the different levels and their degree of independence from the concrete scenario.

Appropriateness evaluation (C4): We show completeness and appropriateness of our categorization by applying it to independently developed transformations.

We want to achieve a development process in which BX are specified as partial descriptions of consistency, which can be combined to a network on demand, so that their repeated execution in arbitrary order leads to a consistent state after changes. Our contributions help to achieve that by forming systematic knowledge on interoperability issues that have to be considered and solved.

4.1. The Consistency Specification Process

Move Specification Levels to top level and remove this somehow

The process of specifying consistency between $n > 2$ types of models using a network of BX can be separated into different conceptual levels. We distinguish three such levels: At the *global level*, we describe the (n-ary) relations between all involved model types. At the *modularization level*, we split these global relations into modular, binary relations. Finally, at the *operationalization level*, we define preservation of consistency according to the modular relations. That classification forms our contribution **C1**.

All of these levels have to be considered during the consistency specification process. A developer specifies consistency on one of these levels, depending on the abstraction level that the transformation language provides, and the transformation engine finally derives an operationalization from that. Although a developer does not specify consistency on multiple levels, he or she has to think about the levels on and above the one consistency is specified on. For example, to define an operationalization, the developer must be aware of the modular consistency relations. The benefit of clearly separating these levels is that they have different potentials for mistakes, faults, and resulting failures. Consequently, avoiding a specific kind of mistake, which is related to one of the identified levels, completely prevents a specific category of failures. We exemplify these levels in Figure 4.2 and explain them in more detail in the following.

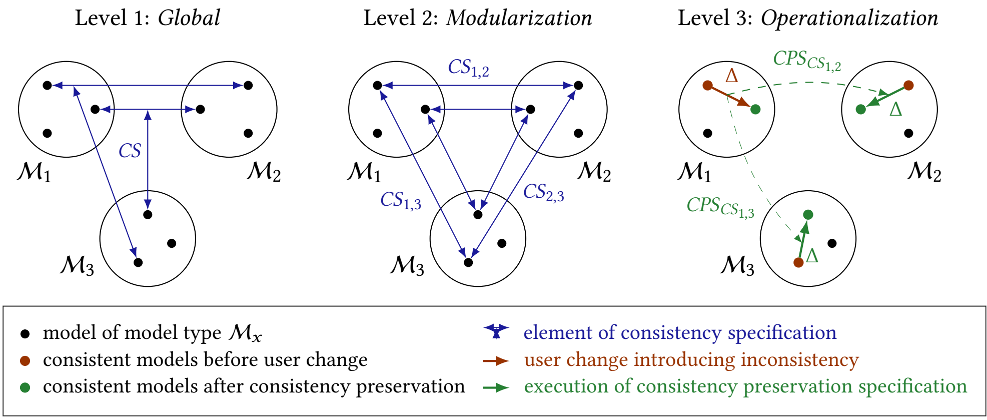

Level 1: *Global*
Level 2: *Modularization*
Level 3: *Operationalization*
CS
$CS_{1,2}$
$CS_{1,3}$
$CS_{2,3}$
$CPS_{CS_{1,2}}$
$CPS_{CS_{1,3}}$
Δ
$\mathcal{M}_1$
$\mathcal{M}_2$
$\mathcal{M}_3$
● model of model type $\mathcal{M}_x$
● consistent models before user change
● consistent models after consistency preservation
element of consistency specification
user change introducing inconsistency
execution of consistency preservation specification

Figure 4.2.: Examples for Abstraction Levels in the Consistency Specification Process

4.1.1. Consistency Specification Levels

Level 1 (*Global*):

At the most abstract level, we consider the knowledge about all actual consistency relations between the involved model types. This knowledge can be represented by an n-ary relation between all model types, containing all tuples of consistent instances of the n model types according to a consistency specification (Definition 3). We refer to this as a *global* consistency specification.

Level 2 (*Modularization*):

At the second level, the global knowledge of the first level is separated into partial, binary consistency relations that, in combination, represent the overall knowledge about consistency in the system. These relations should not contain any contradictions. We do not necessarily need to describe relations between all pairs of model types, since some may not share information that may become inconsistent, or some may be represented transitively across other relations. This knowledge can be represented by up to $\frac{n*(n-1)}{2}$ binary relations, each containing all pairs of instances of two of the model types that are consistent. This corresponds to a set of binary consistency specifications according to Definition 3. We refer to these as *modular* consistency specifications.

Remark: Although in theory not all kinds of n-ary relations can be separated into binary relations [Ste17], we assume that all consistency relations considered in an automated consistency preservation process can be expressed by binary relations. We shortly discussed why this is a reasonable assumption in section 3.1.

Level 3 (*Operationalization*):

At this level, the consistency preservation is operationalized in terms of binary consistency preservation specifications according to Definition 4. As discussed in section 3.1, we consider a set of consistency preservation specifications that can be composed to restore consistency. In contrast to a single BX, an operationalization in networks of BX has to deal with confluence of information. This can lead to problems, such as overwrites or duplications of information, whenever a change can be propagated across at least two paths in the network of BX to the same model. We have seen an example, in which such multiple transformation paths cannot be avoided, in Figure 4.1.

4.1.2. Selecting the Specification Level

A transformation language finally derives a consistency preservation specification from a specification on any of the levels and executes it. Imperative transformation languages expect specifications at the operationalization level, whereas rather declarative, usually bidirectional transformation languages expect specification at the modularization level. Specifications at the global level are rather unusual, but could for example be expressed with multidirectional QVT-R [MCP14], or the Commonalities language [Gle17]. A specification must finally be free of mistakes that can be made on any of those levels. The responsibility depends on the abstraction level the transformation language provides, as the developer is responsible for avoiding mistakes at or above the level at which he or she specifies consistency, whereas the transformation language is responsible for those below.

Specifications must especially be correct regarding all higher levels. This means that an operationalization in consistency preservation specifications must preserve consistency according to the underlying modular consistency specifications. So after changing a consistent set of models, the consistency preservation has to return another set of models that is consistent again, as shown in Figure 4.2. Additionally, modular consistency specifications must be correct regarding the global specification in the sense that it must contain the same sets of models as the global specification. Finally, the global consistency specification has to be correct regarding some, usually informal, notion of consistency for the considered model types. Since this can usually not be validated, we assume a global specification to be correct. This conforms to the notion of *correctness* already defined for BX [Ste10], but is used for the extension to networks of BX here.

5. Network Specification Processes [10 p.]

Part III.

Building Correct Transformation Networks

[110 p.]

6. Correctness in Transformation Networks [20 p.]

Restructure into formalization of correctness (containing compatibility, interoperability etc.) and formal proof of proving correctness. Or maybe move that to prevention section?

6.1. A Basic Notion of Consistency

Intensional / extensional consistency

When we consider a set of models, we would intuitively say that it is consistent if it fulfills some kind of constraints. Defining these constraints to derive or check whether a given set of models is consistent constitutes an *intensional specification* of consistency, because the set that contains all consistent models is intensionally represented by these constraints and can be derived from it. On the contrary, one can also enumerate the consistent sets of models, thus a set of models is considered consistent if it is contained in that enumeration. This constitutes an *extensional specification* of consistency.

Equivalence intensional / extensional specifications

Finally, Both kinds of specifications are equivalent. For each intensional specification the extensional one can be derived by enumerating all models that fulfill the constraints. An extensional specification could also be transferred to an intensional one by defining constraints that are fulfilled by exactly the enumerated instances, in the worst case by constraints that explicitly encode the consistent models. However, for us it will only be relevant that an intensional specification can be transformed into an extensional one.

Add reference

Extensional specifications for theoretical considerations

A developer, who specifies consistency, usually wants to use an intensional specification, like it is also supported by transformation languages such as QVT-R. This is due to the fact that he cannot explicitly enumerate all consistent models but only define constraints that allow to derive them. However, from a theoretical perspective we prefer extensional specifications, because they allow to apply basic set theory. Due to the fact that each intensional specification can be transformed into an extensional one, we can make theoretical statements about extensional specifications that also hold for intensional ones.

Extensional specifications are relations

An extensional specification of consistency enumerates all sets of models that are considered consistent to each other, i.e., it specifies a relation between the models. Since it is easier if the considered models are identifiable with an index, we will consider tuples rather than sets of models throughout the thesis.

Definition 5 (Model-Level Consistency Relation)**.** Given a tuple of metamodels $\mathfrak{M} = \langle M_1, \dots, M_n \rangle$, a *model-level consistency relation CR* is a relation for instances of the metamodels $CR \subseteq I_{\mathfrak{M}} = I_{M_1} \times \dots \times I_{M_n}$.

We consider a tuple of models $\langle m_1, \dots, m_n \rangle \in I_{\mathfrak{M}}$ *consistent to CR* if and only if $\langle m_1, \dots, m_n \rangle \in CR$. Otherwise, we call $\langle m_1, \dots, m_n \rangle$ *inconsistent to CR*.

Start with coarse-grained model-level relations

Given a tuple of models, we consider that tuple of models consistent if it is contained in the consistency relation. This conforms to definition such as the one proposed by **??**. We explicitly denote this kind of consistency relation as *model-level*, because we will later need a more fine-grained notion of consistency relations at the level of metaclasses and need to distinguish between the two.

Modular notions of consistency

If a single relation describes consistency between all relevant models, consistency is directly defined by means of model tuples being in that relations. We call such a relation a *monolithic relation*. However, if we have a *modular* notion of consistency, i.e., a relation does only define consistency between some of the relevant models and the global notion of consistency is a defined by a combination of several such relations, we need an explicit definition for that notion. Actually, we focus on binary relations as a modular representation of consistency, but this definition could also be generalized to relations or arbitrary arity.

Definition 6 (Consistency)**.** Let $\mathfrak{M} = \langle M_1, \dots, M_n \rangle$ be metamodels and let $CR \subseteq I_{M_i} \times I_{M_j}$ be a binary model-level consistency relation for any two metamodels $M_i, M_j \in \{M_1, \dots, M_n\}$. For a given tuple of models $\mathfrak{m} \in I_{\mathfrak{M}} = I_{M_1} \times \dots \times I_{M_n}$, we say that this model set is *consistent to CR* if and only if the instances of M_i and M_j are in that relation:

$$\begin{aligned}&\mathfrak{m} \textit{ consistent to } CR :\Leftrightarrow\\ &\quad \exists\, m_i \in I_{M_i}, m_j \in I_{M_j} : m_i \in \mathfrak{m} \land m_j \in \mathfrak{m} \land \langle m_i, m_j \rangle \in CR\end{aligned}$$

For a set of binary model-level consistency relations $\mathbb{CR}$ for metamodels $M_1, \dots, M_n$, we say that a given tuple of models $\mathfrak{m} \in I_{\mathfrak{M}}$ is *consistent to* $\mathbb{CR}$ if and only if the it is consistent to each consistency relation in that set:

$$\begin{aligned}&\mathfrak{m} \textit{ consistent to } \mathbb{CR} :\Leftrightarrow\\ &\quad \forall CR \in \mathbb{CR} : \mathfrak{m} \textit{ consistent to } \mathbb{CR}\end{aligned}$$

Examplary binary relations

The definition states that given a set of model-level consistency relations the models must be consistent to all of these relations to consider them consistent to the set. Consider, for example, the relations $CR_1 = \{\langle m_1, m_2\rangle\}$, $CR_2 = \{\langle m_2, m_3\rangle\}$, $CR_3 = \{\langle m_1, m_3\rangle\}$ with $m_i \in I_{M_i}$ for metamodels M_i. The then model tuple $\langle m_1, m_2, m_3\rangle$ is consistent to these relations, because it is consistent to each of the binary relations. These consistency relations are equivalent to a monolithic relations $CR = \{\langle m_1, m_2, m_3\rangle\}$, because a model tuple $\mathfrak{m}$ is consistent to CR exactly when it is consistent to $\{CR_1, CR_2, CR_3\}$.

Modular and monolithic specification are not equivalent

Although in that exemplary case the binary relations are equivalent to a monolithic relations, such an equivalence is not always given. In general, two interesting insights come along with that definition of consistency based on modular relations. First, expressiveness of defining consistency modularly by a set of relations is not equal to defining one monolithic relations between all models. Second, a modular definition of consistency can easily contain contradictions, which may lead to an empty set of consistent models.

Binary relations reduce expressiveness

It is easy to see that a combination of binary relations is not able to express the same consistency relations as one monolithic relations. For example, the monolithic relation $CR = \{\langle m_1, m_2, m_3'\rangle, \langle m_1, m_2', m_3\rangle, \langle m_1', m_2, m_3\rangle\}$ cannot be expressed by binary relations. The binary relations necessarily need to contain $\langle m_1, m_2\rangle$, because $\langle m_1, m_2, m_3'\rangle \in CR$ and $\langle m_2, m_3\rangle$, because $\langle m_1', m_2, m_3\rangle \in CR$. However, this would mean that $\langle m_1, m_2, m_3\rangle$ is considered consistent to the binary relations although it not consistent to the modular relation CR. Thus, using sets of binary relations in contrast to a monolithic relation reduces expressiveness. Stevens [Ste17] discusses this property as *binary-definable* in detail and proposed restrictions to binary relations that may be sufficient and still practical for expressing consistency, such as a notion of *binary-implemented* relations. However, we reasonably assume that relations need to be specified modularly anyway, thus we have to accept that this restrictions in expressiveness exists.

Binary relations may be contradictory

Additionally, it is easy to define multiple binary relations of which each can be fulfilled by certain models, but for which no tuple of models exists that is consistent to all of them. Consider the relations $CR_1 = \{\langle m_1, m_2 \rangle\}$, $CR_2 = \{\langle m_2, m_3 \rangle\}$, $CR_3 = \{m_1, m_3'\}$. Although for each of these relations a consistent set of models exists, which is exactly the one defined in each relation, no tuple of models exists that fulfills their combination. This example already demonstrates the worst case, in which no consistent models exist for a set of relations. In other cases, it may be possible that only for some models that are consistent according to one or some of the relations no model tuple exists that is consistent for all models. Consider the relations $CR_1 = \{\langle m_1, m_2 \rangle\}$, $CR_2 = \{\langle m_2, m_3 \rangle, \langle m_2, m_3 \rangle'\}$, $CR_3 = \{m_1, m_3'\}$. In this case, the tuple $\langle m_1, m_2, m_3' \rangle$ would be considered consistent to the relations, but although $\langle m_2, m_3 \rangle \in CR_2$ there exists not $m_1 \in I_{M_1}$ so that $\langle m_1, m_2, m_3 \rangle$ is consistent to all these relations.

Sets of relations can forbid models

It is easy to see that one monolithic relation may be equally represented by an arbitrary number of sets of binary relations by simply adding model pairs to these binary relations that are never consistent to the other relations, like we have seen for the pair $\langle m_2, m_3 \rangle$ in the previous example. This means that the combination of relations can lead to the situation that some models are actually forbidden (like m_3 in the example before) due to the combination of consistency relations. We will later discuss how far this behavior is or should be expected.

Preserving consistency

While the previous discussion only considered when models are considered consistent, it is of especial interest to ensure that consistency of models is preserved. Informally speaking, this means that some mechanism must ensure that after a change to a tuple of models that is consistent according to some relations the models are updated in a way that the resulting tuple of models is consistent again. We call such a mechanism a *consistency preservation rule.*

Monolithic and modular consistency preservation

Like consistency relations, such consistency preservation rules can be realized in an either monolithic or modular way. A monolithic consistency preservation rules takes a tuple of models that is consistent to a consistency relation and a change to these models and returns another tuple of models that is consistent again. In a modular specification of consistency preservation rules, a set of such rules is given which are able to preserve consistency of a subset of the given models according to modular consistency relations. In our case, we consider such rules for two models, each of them restoring consistency according to a binary consistency relation. We will later discuss if and how an execution order of such consistency preservation rules can be determined.

6.2. Notions of Correctness

Before we consider consistency preservation rules and their interaction in detail, we first discuss different notions of *correctness* for them. This is important because there are different correctness dimension, for which we have to make clear which of them are important for us.

Keeping multiple models consistent by means of transformations imposes either a single multidirectional transformation or a combination of several bi- or multidirectional transformations. Each of these transformations is able to take a consistent tuple of models and a change to them and to return a new consistent tuple of models.

6.3. Notions of Correctness

Correctness implicitly covered by definitions

Stevens [Ste10] proposes an explicit notion of *correctness* for transformations. This is based on the fact that her definition of a transformation does only specify that for two given model, which may be inconsistent because one was modified, an update of the other model is returned. The requirement that the originally modified model and the one returned by the transformation have to conform to some consistency relations is specified externally as a notion of *correctness*. We directly relate a consistency preservation rule that restores consistency to an according consistency relation, thus a consistency preservation rule that follows our definition is correct by construction in terms of the correctness definition by Stevens. The same applies to the consistency preservation application function, which we consider *correct* if it fulfills its definition, as that definition already covers all requirements to that function.

Different notions of correctness

In general, correctness can be considered in two ways: First, an artifact may be correct if it simply follows its definition. While for consistency relations, changes and the generic model-level consistency preservation rule generalization function correctness can be canonically achieved, this is not that simple for a consistency preservation rule and the consistency preservation application function, as they have to fulfill some constraints with respect to consistency relations they rely on. Second, an artifact may be correct if it fulfills some, maybe only implicitly known specification. For example, a consistency relation between UML and Java may only be considered correct if it fulfills some "natural" notion of consistency, as people know how elements have to be related because they represent similar things, such as classes, or because a standard like the UML [Obj17] prescribes that.

Correctness regarding global specifications

In this work we do not consider the latter correctness notion with respect to external, maybe not formally specified artifacts, which is part of separate research on validation. However, when considering consistency of multiple models it may be standing to reason that a modular specification of consistency and its preservation has to be correct with regards to some global, monolithic specification. More precisely, there may be a multiary relation putting several metamodels into relation, which the developers at least implicitly know, and a set of binary relations somehow has to respect that multiary relation, i.e., be *correct* with respect to that relation. The same can be imagines for consistency preservation. One may define a multidirectional transformation for a multiary relation, taking a tuple of changes to consistent models and retuning a new tuple of changes, which applied to the models delivers a consistent set of models again. In fact, this would be a realization of the behavior of the consistency preservation application function without relying on modular model-level consistency preservation rules.

Drawbacks of global specification notions

Considering consistency this way has two drawbacks:

Validation Artifacts: The artifacts to check correctness against, i.e., the global, multiary consistency relation as well as an appropriate multidirectional transformation, do usually not exist. If they existed, they could figures/correctness/formal/correctness_notions.png

Figure 6.1.: Different notions of correctness for transformations and networks

directly be used to preserve consistency. Thus is impossible to validate a set of consistency relations and preservation rules against such a global specification.

Modular Knowledge: This notion of correctness requires that the developers have some global knowledge that represents a monolithic, multiary consistency relation and their preservation rules. Usually, this will not be the case, so there is even no implicit notion of the necessary artifacts to validate the modular specifications against, not to be mention an explicit representation.

Add an image for that relation

Overall Goals:

- Define correctness of a transformation network: termination in consistent state

Central Insights:

- Correctness is not the problem, optimality is the problem

- We can only check dynamically whether a consistent state was reached due to Halting Problem. We cannot guarantee to always find a consistent state

Allgemeine Definition Transformationsnetzwerke:

- Definition Transformation aus Relation und Wiederherstellungsroutinen; Routinen nehmen n Modelle und n Deltasequenzen (eine pro Modell) und liefern n Deltasequenzen zurück.
- Im Allgemeinen könnte eine Transformation beliebige dieser Deltasequenzen modifizieren. Wir verlangen jedoch, dass eine Transformation nur Deltas anhängt, also die Sequenzen länger werden
- Genauer beschränken wir auch, welche Sequenzen eine Transformation sehen und ändern darf, genau gesagt darf sie die Sequenzen von zwei Modellen sehen und eine davon verlängern.
- Hier kommt bereits der Unterschied zu bisherigen Transformationen, denn die sehen nur Deltas an einem Modell und erzeugen Deltas an dem anderen. Das ist bei uns schon gänzlich anders. Bidirektionale Transformationen unterstützen das im Übrigen auch nicht, sondern sind nur Spezifikationen, aus denen sich Wiederherstellungsroutinen für beide Richtungen ableiten lassen (siehe Stevens 2010)
- Relationen in erster Instanz auf Modellebene (also bzgl. ganzer Modelle, nicht einzelner Modellelemente) definieren
- Direkt als multidirektionale Transformation definieren, also beliebig viele geändert Ein- und Ausgabemodelle (oder jeweils nur eins?)
- Korrektheit einer Transformation (nach Stevens) definieren!
- Versuchen den Konkatenationsoperator zu definieren ohne dass er alle Metamodelle referenzieren muss (also Transformation wählt aus einer großen Eingabemenge relevanten Modelle aus, ändert relevante und dann fügt der Operator sie in die große Menge ein)
- Definition Transformationsnetzwerk als Tupel aus Metamodellen, Transformationen und einer Ausführungsfunktion.

- Die Ausführungsfunktion führt für eine gegebene Änderung eine Auswahl der Transformationen nacheinander aus.
- Korrektheit eines Netzwerkes definieren: Die Ausführungsfunktion erzeugt eine Transformationssequenz, die angewendet auf eine Änderung für alle Änderungen ein korrektes Ergebnisse produziert, d.h. die Modelle sind konsistent bzgl. allen Konsistenzrelationen.

Das folgende drückt eine Definition entsprechend der Annahme der modularen Spezifikation aus. Wir stellen es später in Bezug zu globalen Konsistenzrelationen und anderen Spezifikationsmöglichkeiten, Stichwort MX

6.4. A Monolithic Notion of Consistency

Having one consistency relation for an arbitrary number of metamodels constitutes a *monolithic* notion of consistency, as it describes consistency for an arbitrary number of metamodels as a whole. Accordingly, a multidirectional transformation [Cle+19] could be defined that takes a consistent tuple of models and any change to them and then returns a consistent tuple of models again, which then reflects the performed change.

Define correctness based on a networks adhering to the behavior of a multidirectional transformation

6.5. A Modular Notion of Consistency

In the following, we only consider binary consistency relations. Having several consistency relations to define how several metamodels are related, we need to define a notion of consistency based on several consistency relations.

We define the following ingredients necessary to handle a network of transformations:

Consistency Preservation Rules: We need binary rules that restore consistency after a model of a pair was modified. These are the binary, modular transformations of the networks

Orchestration Function: We need a function that is able to decide in which order transformation have to be executed after a change.

Application Function: We need a function that is able to apply the rules in the order delivered by the orchestration function.

We explicitly distinguish the orchestration and the application to be able to make more fine-grained statements about the responsibilities for the orchestration and its actual execution. The process is depicted in Figure 6.2. Given models to are consistent according to some consistency relations and changes to them, the orchestration delivers an order of consistency preservation rules (CPRs) for that networks, which is used to parametrize the application function that executes these rules in the given order. The result is, in the best case, a tuple of models that is consistent to the relations again. However, we will see that this is not always possible. Thus, we will especially discuss relevant properties of the artifacts, such as correctness and optimality that reflect how and when this process can be executed successfully.

First of all, we start with a definition of consistency in a network of transformations.

Definition 7 (Consistency). Let $\mathfrak{M} = \langle M_1, \ldots, M_n \rangle$ be metamodels and let $CR \subseteq I_{M_i} \times I_{M_j}$ be a binary model-level consistency relation for any two metamodels $M_i, M_j \in \{M_1, \ldots, M_n\}$. For a given tuple of models $\mathfrak{m} \in I_{\mathfrak{M}} = I_{M_1} \times \cdots \times I_{M_n}$, we say that this model set is *consistent to CR* if and only if the instances of M_i and M_j are in that relation:

$$\begin{aligned}&\mathfrak{m} \textit{ consistent to } CR :\Leftrightarrow \\ &\quad \exists\, m_i \in I_{M_i}, m_j \in I_{M_j} : m_i \in \mathfrak{m} \wedge m_j \in \mathfrak{m} \wedge \langle m_i, m_j \rangle \in CR\end{aligned}$$

figures/correctness/formal/execution_process.png

Figure 6.2.: Execution process in network and artifacts

For a set of binary model-level consistency relations $\mathbb{CR}$ for metamodels $M_1, \ldots, M_n$, we say that a given tuple of models $\mathfrak{m} \in I_{\mathfrak{M}}$ is *consistent to* $\mathbb{CR}$ if and only if the it is consistent to each consistency relation in that set:

$$\begin{aligned}
&\mathfrak{m} \ \textit{consistent to} \ \mathbb{CR} :\Leftrightarrow \\
&\quad \forall CR \in \mathbb{CR} : \mathfrak{m} \ \textit{consistent to} \ \mathbb{CR}
\end{aligned}$$

Definition 8 (Change)**.** Given a metamodel M, a change δ_M is a function that takes an instance of that metamodel and returns another instances:

$$\delta_M : I_M \longrightarrow I_M$$

It encodes any kind of change, which may be just an element addition, or removal, an attribute change and so on, or any composition of changes. We denote the identity change, i.e., the change that always returns the input model, as δ_{id}:

$$\delta_{id}(x) = x$$

For us, it does not matter how the function behaves in cases, in which the encoded change cannot be applied, e.g., because the changed or removed element does not exist. The function may do nothing for those models, i.e. return the identical model, or even be undefined for those model, i.e., be partial.

Check whether this behavior is correct.

We denote the universe of all changes in M, i.e. all subsets of $I_M \times I_M$ that are functional, as

$$\Delta_M = \{\delta_M \mid \delta_M \subseteq I_M \times I_M \land (\langle m_1, m_2\rangle, \langle m_1, m_3\rangle \in \delta_M \Rightarrow m_2 = m_3)\}$$

For a given metamodel tuple $\mathfrak{M} = \langle M_1, \ldots, M_n\rangle$, we denote the set of all tuples of changes in the instances tuples of $\mathfrak{M}$, i.e., in $I_\mathfrak{M}$, as $\Delta_\mathfrak{M}$:

$$\Delta_\mathfrak{M} = \{\delta_\mathfrak{M} = \langle \delta_{M_1}, \ldots, \delta_{M_n}\rangle \mid \forall\, i \in \{1, \ldots, n\} : \delta_{M_i} \in \Delta_{M_i}\}$$

Definition 9 (Model-Level Consistency Preservation Rule)**.** Given two metamodels M_1, M_2 and a binary model-level consistency relation between them $CR \subseteq I_{M_1} \times I_{M_2}$. A *model-level consistency preservation rule* is a function:

$$\textsc{Cpr}_{CR} : (I_{M_1}, I_{M_2}, \delta_{M_1}) \longrightarrow \delta_{M_2}$$

It that takes two consistent models and a change in the first one and returns a change in the second one. We call a model-level consistency preservation rule *correct* w.r.t. CR if the resulting models when applying the input and output change are consistent to CR again:

$$\begin{aligned}
&\forall\, m_1 \in I_{M_1}, m_2 \in I_{M_2} : \langle m_1, m_2 \rangle \in CR \Rightarrow \\
&\quad \forall\, \delta_{M_1} \in \Delta_{M_1} : \exists\, \delta_{M_2} \in \Delta_{M_2} : \\
&\quad \textsc{Cpr}_{CR}(m_1, m_2, \delta_{M_1}) = \delta_{M_2} \wedge \langle \delta_{M_1}(m_1), \delta_{M_2}(m_2) \rangle \in CR
\end{aligned}$$

This is equivalent to the definition of *consistency restorers* in [Ste10], except that their definition is state based, thus only considering two inconsistent models states and deriving a new state of one of the models, whereas our definition is delta based, considering the changes between two models, which gives us the possibility to also take into account how the inconsistency was created and later the possibility to consider the composition of changes.

A model-level consistency preservation rule is defined to restore consistency after a modification in a left model of the underlying model-level consistency relation by creating a change for the right model. To consider consistency preservation rules that preserve consistency in the other direction, we regard the inverse of the consistency relation as well, denoted as $CR^T = \{\langle m_1, m_2 \rangle \mid \langle m_2, m_1 \rangle \in CR\}$.

A model-level consistency relation together with two consistency restorers, or model-level consistency preservation rules in our terminology, forms a *bidirectional transformation*.

Definition 10 (Bidirectional Transformation)**.** Let CR be a model-level consistency relation, and $\textsc{Cpr}_{CR}$ and $\textsc{Cpr}_{CR^T}$ two model-level consistency preservation rules to restore consistency according to that relation in both directions, i.e., after changes in any of the models. A bidirectional transformation is a triple $\langle CR, \textsc{Cpr}_{CR}, \textsc{Cpr}_{CR^T} \rangle$.

The definition could also be given for an arbitrary number of metamodels, but we restrict ourselves to binary specifications, as explained in

ref

.

Definition 11 (Model-Level Consistency Preservation Rule Generalization Function)**.** Let $\textsc{Cpr}_{CR}$ be a model-level consistency preservation rule for metamodels M_i, M_k. Let $\mathfrak{M} = \langle M_1, \dots, M_i, \dots, M_k, \dots, M_n \rangle$ be a tuple of metamodels containing M_i and M_j. A model-level consistency preservation rule generalization function $\textsc{Gen}_{\textsc{Cpr}_{CR}}$ is a function:

$$\textsc{Gen}_{\textsc{Cpr}_{CR}} : (I_{\mathfrak{M}}, \Delta_{\mathfrak{M}}) \to (I_{\mathfrak{M}}, \delta_{\mathfrak{M}})$$

It generalizes $\textsc{Cpr}_{CR}$ such that it can be applied to changes in $\mathfrak{M}$ instead of M_i, i.e. it applies the change delivered by $\textsc{Cpr}_{CR}$ for the relevant models to the given change tuple:

$$\textsc{Gen}_{\textsc{Cpr}_{CR}}(\mathfrak{M}, \delta_{\mathfrak{M}} = \langle \delta_{M_1}, \dots, \delta_{M_i}, \dots, \delta_{M_k}, \dots, \delta_{M_n} \rangle) = \\ (\mathfrak{m}, \langle \delta_{M_1}, \dots, \delta_{M_k}, \dots, \textsc{Cpr}_{CR}(m_i, m_k, \delta_{M_i}) \circ \delta_{M_j}, \dots, \delta_{M_n} \rangle)$$

This function is universally defined and must not be defined individually for a specific model-level consistency preservation rule.

To execute transformations in a network, we need some instance that decides which transformations are executed in which order. We call this an *orchestration function* as it is responsible for orchestrating the execution of transformations.

Definition 12 (Consistency Preservation Orchestration Function)**.** Let $\mathbb{CPR}$ be a set of consistency preservation rules for a set of consistency relations $\mathbb{CR}$ on metamodels $\mathfrak{M} = \langle M_1, \dots, M_n \rangle$. A consistency preservation orchestration function $\textsc{Orc}_{\mathbb{CPR}}$ for these rules is a function:

$$\textsc{Orc}_{\mathbb{CPR}} : (I_{\mathfrak{M}}, \Delta_{\mathfrak{M}}) \to \mathbb{CPR}^{<\mathbb{N}}$$

$\mathbb{CPR}^{<\mathbb{N}}$ denotes all finite sequences of consistency preservation rules in $\mathbb{CPR}$, i.e., $\mathbb{CPR}^{<\mathbb{N}} = \emptyset \cup \mathbb{CPR}^1 \cup \mathbb{CPR}^2 \cup \dots$.

According to this definition, the orchestration functions returns any sequence of model-level consistency preservation rules. This especially includes that rules may occur more than once in that sequence and that applying that sequence to a models and changes to them does not guarantee figures/correctness/formal/divergence_example.png

Figure 6.3.: Example for divergence

that the resulting models are consistent. This degree of freedom is on purpose, because unfortunately it is not always possible to find an orchestration of transformations that results in a consistent state.

It is obvious that we can define consistency preservation rules for which the orchestration function cannot find an execution order that returns a consistent tuple of models after certain changes. Consider the example in Figure 6.3. There exists no execution order for any input value that terminates. The transformations will always increase the value, although the defined relations could be fulfilled for the input value, but the transformations never find that solution.

Although we will discuss restrictions to relations and transformations that reduce the chance that no solution can be found, it will not be possible to ensure that such a solution can always be found. This is due to the reason that transformations can perform arbitrary changes given that transformations are Turing complete, which should not be restricted, because it is unclear which restrictions could be made without forbidding scenarios that should actually we supported. Thus, we assume that transformations are Turing complete.

We explicitly allow the orchestration function to return a sequence that will, if applied to models and changes to them, not deliver a consistent tuple of models. As discusses, this is supposed to reflect cases in which no such sequence can be calculated. However, it may be useful to have some notion of *optimality* that ensures that if a sequence that delivers a consistent result exists, the orchestration function is supposed to find it. Formally, this notion looks as follows.

Definition 13 (Optimal Consistency Preservation Orchestration Function)**.** Let $\mathbb{CPR}$ be a set of consistency preservation rules for a set of consistency relations $\mathbb{CR}$ on metamodels $\mathfrak{M} = \langle M_1, \dots, M_n \rangle$. We say that an orchestration function $\textsc{Orc}_{\mathbb{CPR}}$ for these rules is *optimal* if it always returns a sequence that delivers a consistent set of models if possible, i.e.,

$$\begin{aligned}
&\forall\, \mathfrak{m} \in I_{\mathfrak{M}} : \forall\, \delta_{\mathfrak{M}} = \langle \delta_{M_1}, \dots, \delta_{M_n} \rangle \in \Delta_{\mathfrak{M}} : \mathfrak{m} \textit{ consistent to } \mathbb{CR} \Rightarrow \\
&\quad \big[\big(\exists\, \textsc{Cpr}_1, \dots, \textsc{Cpr}_m \in \mathbb{CPR} : \exists\, \delta'_{\mathfrak{M}} = \langle \delta'_{M_1}, \dots, \delta'_{M_n} \rangle \in \Delta_{\mathfrak{M}} : \\
&\qquad \textsc{Gen}_{\textsc{Cpr}_1} \circ \dots \circ \textsc{Gen}_{\textsc{Cpr}_m}(\mathfrak{m}, \delta_{\mathfrak{M}}) = (\mathfrak{m}, \delta'_{\mathfrak{M}}) \\
&\qquad \wedge \langle \delta'_{M_1}(m_1), \dots, \delta'_{M_n}(m_n) \rangle \textit{ consistent to } \mathbb{CR} \big) \\
&\quad \Rightarrow \big(\exists\, \textsc{Cpr}'_1, \dots, \textsc{Cpr}'_m \in \mathbb{CPR} : \exists\, \delta''_{\mathfrak{M}} = \langle \delta''_{M_1}, \dots, \delta''_{M_n} \rangle \in \Delta_{\mathfrak{M}} : \\
&\qquad \textsc{Orc}_{\mathbb{CPR}}(\mathfrak{m}, \delta_{\mathfrak{M}}) = \langle \textsc{Cpr}'_1, \dots, \textsc{Cpr}'_m \rangle \\
&\qquad \wedge \textsc{Gen}_{\textsc{Cpr}'_1} \circ \dots \circ \textsc{Gen}_{\textsc{Cpr}'_m}(\mathfrak{m}, \delta_{\mathfrak{M}}) = (\mathfrak{m}, \delta''_{\mathfrak{M}}) \\
&\qquad \wedge \langle \delta''_{M_1}(m_1), \dots, \delta''_{M_n}(m_n) \rangle \textit{ consistent to } \mathbb{CR} \big) \big]
\end{aligned}$$

Unfortunately, optimality is a property that we cannot request from an orchestration function. Optimality would mean that the orchestration function can decide whether there is sequence of transformations that leads to consistent models and thus terminate. Due to Turing-completeness of the network this would mean that the orchestration function can decide whether a Turing machine halts, which is proven impossible. Thus, our only goal can be to achieve optimality as far as possible in terms of reducing the degree of conservativeness, i.e., reduce the cases in which no sequence is found although it exists.

We can define a measure for the optimality of an orchestration function:

$$\textit{Optimality}_{\textsc{Orc}_{\mathbb{CPR}}} = \frac{\textit{\# of model / delta pairs for which the function finds an order}}{\textit{\# of model / delta pairs for which an order that termin}}$$

In fact, both these numbers usually infinite, an there is an infinite number of possible models and deltas. However, it does finally not matter for us what the actually value is, but only how to improve that value.

We have to map that value to compatibility, which reduces the number of potential false orders.

The orchestration function does only give us an order and it is intuitively clear how to perform consistency preservation based on given rules and an orchestration function. However, we need to make this process explicit, for which we define an *application function* that is able to perform consistency preservation based on given model-level consistency preservation rules, and orchestration function and the actual models and changes.

Finally, either the orchestration function or an application function must also be able to reflect the cases in which no execution order of model-level consistency preservation rules can be found that restores consistency. From a theoretical perspective it does not make a difference whether the orchestration or the application function makes that decision. Finally, the orchestration function could also directly be encoded into the application function from a theoretical perspective. However, from a practical perspective we may want to be able to find an execution order although there is no order that results in a consistent state, to be able to find out where the problem is to restore consistency.

Property for orchestration: if no consistent solution found than there is no further transformation that could be executed without alternation or because no transforamtion is applicable (undefined for current state, e.g. because of monotony)

Since we do not make any statements about resulting in consistent states yet, we simply foresee this case in the definition of the application case to be able to return $\perp$ indicating that no solution is found for the given models and changes. So we first give a basic definition for such a function without explicitly specifying in which cases the function is expected to return a result other than $\perp$.

Definition 14 (Consistency Preservation Application Function)**.**

Define for transformations instead?

Let $\mathbb{CPR}$ be a set of consistency preservation rules for a set of consistency relations $\mathbb{CR}$ on metamodels $\mathfrak{M} = \langle M_1, \dots, M_n \rangle$ and $\textsc{Orc}_{\mathbb{CPR}}$ an orchestration function for the consistency preservation rules. A consistency preservation application function $\textsc{App}_{\mathbb{CPR}}$ for these rules is a partial function:

$$\textsc{App}_{\textsc{Orc}_{\mathbb{CPR}}} : (I_{\mathfrak{M}}, \Delta_{\mathfrak{M}}) \rightarrow I_{\mathfrak{M}} \cup \{\bot\}$$

The function takes a consistent tuple of models and a tuple of changes that was performed on them and returns a changed tuple of models by acquiring changes from the consistency preservation rules in $\mathbb{CPR}$. It is partial, because it is allowed to return $\bot$ especially for inconsistent input models but potentially also in other cases. It has to fulfill the following conditions:

$$\begin{aligned}
&\forall\, \mathfrak{m} \in I_{\mathfrak{M}} : \forall\, \delta_{\mathfrak{M}} = \langle \delta_{M_1}, \dots, \delta_{M_n} \rangle \in \Delta_{\mathfrak{M}} : \mathfrak{m} \textit{ consistent to } \mathbb{CR} \Rightarrow \\
&\quad \big(\textsc{App}_{\mathbb{CPR}}(\mathfrak{m}, \delta_{\mathfrak{M}}) = \bot \\
&\quad \vee \exists\, \mathfrak{m}' \in I_{\mathfrak{M}} : \textsc{App}_{\mathbb{CPR}}(\mathfrak{m}, \delta_{\mathfrak{M}}) = \mathfrak{m}' \wedge \\
&\qquad \exists\, \textsc{Cpr}_1, \dots, \textsc{Cpr}_m \in \mathbb{CPR} : \exists\, \delta'_{\mathfrak{M}} = \langle \delta'_{M_1}, \dots, \delta'_{M_n} \rangle \in \Delta_{\mathfrak{M}} : \\
&\qquad \textsc{Orc}_{\mathbb{CPR}}(\mathfrak{m}, \delta_{\mathfrak{M}}) = \langle \textsc{Cpr}_1, \dots, \textsc{Cpr}_m \rangle \\
&\qquad \wedge\, \textsc{Gen}_{\textsc{Cpr}_1} \circ \dots \circ \textsc{Gen}_{\textsc{Cpr}_m}(\mathfrak{m}, \delta_{\mathfrak{M}}) = (\mathfrak{m}, \delta'_{\mathfrak{M}}) \\
&\qquad \wedge\, \langle \delta'_{M_1}(m_1), \dots, \delta'_{M_n}(m_n) \rangle = \mathfrak{m}' \big)
\end{aligned}$$

We say that $\textsc{App}_{\textsc{Orc}_{\mathbb{CPR}}}$ is *correct* if its result is either $\bot$ or consistent to CR:

$$\begin{aligned}
&\textsc{App}_{\mathbb{CPR}} \textit{ is correct } :\Leftrightarrow \\
&\quad \forall\, \mathfrak{m} \in I_{\mathfrak{M}} : \forall\, \delta_{\mathfrak{M}} = \langle \delta_{M_1}, \dots, \delta_{M_n} \rangle \in \Delta_{\mathfrak{M}} : \mathfrak{m} \textit{ consistent to } \mathbb{CR} \Rightarrow \\
&\quad \textsc{App}_{\mathbb{CPR}}(\mathfrak{m}, \delta_{\mathfrak{M}}) = \bot \vee \textsc{App}_{\mathbb{CPR}}(\mathfrak{m}, \delta_{\mathfrak{M}}) \textit{ consistent to } \mathbb{CR}
\end{aligned}$$

Achieving a correct application function

The definition of the application function basically ensures that the function either returns $\bot$ or executes the model-level consistency preservation rules given by the orchestration function to retrieve a changes tuple of models. It is considered *correct* if it ensures that its result is either $\bot$ or a consistent model tuple by executing the model-level consistency preservation rules given by the orchestration function. In consequence, the application function can be realized by simply executing the result of the orchestration function and check whether the resulting model tuple is consistent or not and return an appropriate result. Such a realization is generic and does not depend on the actual consistency preservation rules and orchestration function but represents a generic behavior. Additionally, this gives an implementation of that function the ability to present a faulty result to the user, which eases finding out why no consistent state was reached.

Correctness is not crucial

Finally, correctness is not crucial, because correctness can easily be achieved by performing any execution of transformations and just ensuring that we terminate at some point in time and then decide whether the resulting models are consistent or not and appropriately deliver the result.

How to define an orchestration function that is as optimal as possible?

The remaining difficulty is how to define an orchestration function that fulfills the definition, i.e., to find a finite sequence of transformations, and also one that improves optimality, as an *optimal* function can never be given. Although the definition of the orchestration function proposes a closed description of that function, in practice such a function will not have a closed form but will be realized as an algorithm that dynamically decides which transformation to execute next. Therefore the arising problem is that the length of the sequence to execute is not known a priori. Therefore, we need some abortion criterion. When a consistent result is found, this criterion is easy. But since we do not know whether a sequence exist, we need an abortion criterion that is reasonable and does not cut off the process although a consistent solution could be found, thus reducing optimality. A simple realization for that algorithm to deliver a finite sequence of transformations would be to define a fixed termination criterion, such as a specific number of transformation executions. However, there is no upper bound for the number of executed transformations necessary to achieve consistency. Still, a fixed number (even 0) could be defined for the number of executed transformations to fulfill the definition. Hence, optimality would be 0 then as a consistent result is never reached. We therefore discuss in the following how to define an appropriate orchestration function and how to optimize it.

Overall Goal: Find correct orchestration function that improves optimality.

There are two ways to improve optimality of the orchestration function:

1. Optimize the orchestration function, i.e., find a good order (probably this is not possible), at least find an order that helps the developer to find problems
2. Optimize the input, i.e., define requirements to the transformations and their relations representing the input to optimize optimality

We need an example for that

Both goes hand in hand, because restrictions to the input can never lead to an orchestration function that always terminates without leading to unsupported relevant cases.

This conform to two approaches:

1. Dynamic decision about selected transformation and abortion criteria
2. Constructive restrictions that ensure that appropriate order is (easily) found

Application function can be generically defined, orchestration maybe not? We actually want to ensure that both are generic and none of them has to be defined for a specific project.

Now it is obvious that the consistency preservation rules can actually do anything to achieve consistency, including returning always the same set of models that is consistent, although that may not be expected. We will discuss later which reasonable assumptions can be made to the behavior to on the hand not restrict the possibilities of the transformation developer and on the other hand be able to ensure some properties of the transformations and their execution.

First restriction: Input delta of APP only contains changes to one model -> no synchronisation

Second restriction: Input delta is not rejected

Third restriction: Generated deltas are monotone

From a theoretical perspective, it is always possible to a specify consistency relations according to the definition, as it is just a subset of elements. It is also always possible to define a consistency preservation rule for a consistency relation according to its definition, as can simply return any any element of the relation.

This is not true: the source model may not be in the relation, then its not possible, at least with the current definition. With a synchronizing transformation, any modification can be made to both models, then its fine.

The generalization function is generic, so it can always be applied. Finally, the consistency preservation application function is an artifact that cannot be easily specified according to the definition for a given set of consistency preservation rules. It is always possible to have a set of consistency preservation rules for which no application function can be defined that returns a consistent result for at least one input model and change tuple, as there is not sequence of consistency preservation rules that achieves that.

Example!

Even worse, the problem to define such a function is Turing-complete, which makes it impossible to decide whether such a function exists.

Show Turing-completeness

Consequence: From a theoretical perspective, this function is the crucial part!

Essential problem: One transformation may restore consistency between A and B and another between A and C. If then a transformation restores consistency between B and C, the resulting B' and C' may not be consistent A anymore.

Alternative to an app function: Define a *well-definedness* property for a set of transformations, requiring that they can be executed in any order to always terminate consistently. However, this is a very strict requirement, which can usually not be fulfilled, so we do not further investigate that.

Give simple example why that does not work.

Best-behaved app function: Whenever there is a sequence of CPRs, the app function finds item. This is still not possible due to Turing-completeness. The function would need to decide whether the network terminates or not.

Only achievable app function is a best-effort (i.e. conservative) function: A function that either returns a consistent set of models or that does return bottom. Not making a statement about how often a correct result is returned in comparison to how often it is possible.

This approach is conservative. The question is then, how high the degree of conservativeness is. In the worst case, a function that always returns bottom would fulfill the definition, but that is not what we want. We want to reduce conservativeness.

Goal: Find a solution in as many cases as possible, abort in the others (conservatively). There are two subgoals to achieve that: 1. Function must be correct, i.e. always terminate (no endless sequence of CPR) and terminate in a consistent state 2. Function must be as less conservative as possible

It is clear that we cannot give a closed function for APP that just by a given change returns a sequence of CPR that results in a consistent state. APP has to be calculated dynamically during execution. Therefore we consider it as an algorithm in the following.

6.5.1. Achieving a Correct Application Function

These problem cannot occur if a function fulfills the definition, because it always finds a sequence. So the question is how to fulfill the definition.

It is easy to achieve that the APP function only terminates in a consistent state, because knowing the relations allows to check whether all relations are fulfilled.

Need to define that a transformation may not be able to process a specific change? Then there could be inconsistent terminiation because a transformation cannot be executed anymore.

Problems due to which the function does not terminate: Alternation and Divergence

Alternation: Run through same state twice Divergence: Always produce new states without reaching a consistent state

Two possibilities to avoid problems: 1. Make assumptions to transformations that avoid them 2. Detect them dynamically and abort

6.5.1.1. Avoiding Alternation / Divergence

Making assumptions that avoid them is rather hard, as we will show in the following.

Idea: Require monotony to avoid alternation

We would have to relax the definition of transformation to be monotone, because if a transformation is monotone, it may only append information, but this is not always possible, as can be seen in the following example. A monotone transformation must be able to return bottom if it cannot make further changes to restore consistency to the relation.

Definition 15 (Monotone Transformation)**.** Transformation gets models M and deltas D and produces new deltas D'. Taking the union of the original models M and the new models D'(M), then D(M) must be a subset of that, because other elements would have been added and removed afterwards or elements would have been changes once by D and again in a different way by D'.

Generally, monotony could also mean that only the same complete model state is not passed twice.

Why dont we do that?

This would mean that each transformation only appends changes, i.e., if an element was added/removed, the transformation may not do the inverse. The same applies to attribute/reference changes: if an attribute/reference was already changes it may not be changed again. This way, it is by design figures/correctness/formal/monotony_counterexample.png

Figure 6.4.: Counterexample for monotony

impossible to pass through the same state again. Actually, if a monotone transformation returns bottom, the network has to terminate with a failure. However, this is hard restriction to transformations. It leads to the fact that in some networks that actually have a simple solution no solution is found at all. This can be easily seen at the example in Figure 6.4. In the example adding "aa" to the left model, any execution order of the transformations leads to the situation that a previous change must be revoked to result in a consistent state. However, it is possible to derive a consistent state for that input change.

One could now argue that there are binary relations in the example, which may never be fulfilled at all. We will later discuss how far relations that cannot be fulfilled should be restricted. However, in general, this is wanted behavior, because in general it may be necessary that transformations produce intermediate states that are not yet consistent with each other. Otherwise this would means that each transformation is always able to directly deliver a state that is consistent to all other relations, which is especially not possible, because other transformations may add further information to the models. More precisely, a relation may consider <a model consistent to all other models that contain any additional information not affected by the transformation. For example, a UML class model may be considered consistent to all Java models with any implementation of the specified methods, thus to an infinite number of models. Now saying that it should not be

allowed that the transformation selects one with an empty implementation because that is not consistent to another relations induced by another transformation, such as the relationship to a component model, does not make any sense. Thus having those relation elements that may be considered locally consistent but will never occur in a globally consistent tuple of models does not make sense. In the example, we can see that such an inconsistent intermediate state is passed through and afterwards a consistent tuple of models is reached if not requiring monotony. In consequence, requiring monotony from transformations is a too strict requirement, because it is necessary to run through states that may be changed later on.

Theorem 1. An application function for monotone transformations either returns a consistent model or produce a sequence of CPRs returning delta that return models of always growing size (i.e. it diverges).

Divergence cannot be avoided There are rather equal network, one that terminates after a long time and one that never terminates. Consider the example. The relations are defined in a way such that for any allocation for any of them a consistent tuple of models can be found. However, the transformations are not able to find it because they make "bad" choices from a set of choices that are conflicting. This can be seen in the example in Figure 6.5.

Thus, systematically avoiding divergence is not possible.

Detecting Alternation / Divergence In consequence, we propose to dynamically deal with alternation / divergence. To detect alternations, the execution can simply track if a state way already processed. Apart from spatial problems, this does always work. Finding divergence is not that easy, because it is generally not possible to define an upper bound for the number of executions of a single transformation. This is due to the reason that, again, this conflicts with the Halting problem. We can see this at the simple example in Figure 6.6.

Depending on the value X, the transformations have to be executed X times to result in a consistent state. This value can be arbitrarily chosen, thus an figures/correctness/formal/divergence_example.png

Figure 6.5.: Example for divergence

figures/correctness/formal/no_upper_bound_example.png

Figure 6.6.: Example for no upper bound

arbitrary number of executions may be necessary to terminate in a consistent state.

From an engineering perspective, this is still unwanted behavior. We claim that a transformation network that takes thousands of executions of the same transformation to find a consistent state works not as expected and if running into a failure would expose severe problems to find the reasons for that failures. Thus, we propose to simply abort the execution after some time to be sure not to run in an endless loop.

Finally, this problem is comparable to ordinary programming, because there the same situations regarding alternation and divergence can occur that result in non-termination of a program. As we all know, it is impossible to systematically avoid that, but just possible to carefully develop the program and apply best practices to avoid such situations.

In the following, we propose measures to reduce the number of cases in which problematic cases can occur. In a case study, we will see that using such measures already resolves most of the problems that can occur. Additionally, we propose an orchestration strategy that improves the possibility to find errors in case something goes wrong.

Central insight: Alternation / Divergence cannot be avoided systematically (like in ordinary programming), if not restricting transformations in a way that may not be reasonable.

6.5.2. Reducing Conservativeness of the Application Function

Goal: Find a solution in as many cases as possible, abort in the others (conservatively). There are two approaches to achieve that: 1. Reduce the number of cases in which there is no solution by adding assumptions to the relations and transformations (restrict input of app function) 2. Improve the ability to find a solution if it exists (improve capabilities of app function) Secondary goal: In cases, in which no solution is found, support the user in understanding why no solution was found.

Regarding 1: Reduce problematic cases

1. reduce cases in which there is no such solution 1.1. On relation level: Only sets, so analysis possible. Ensure that relations are defined in a way such that they do not allow a locally correct set of CPRs that has no APP solution. If there is a pair of models (or elements of a fine-grained relation) in a relation, a CPR may return it. But if there is no consistent tuple of models containing these two, it does not make any sense to consider these elements (even worse, if we have monotony, adding these elements makes

the network unsolvable). For that reason, we need compatibility. Avoids both alternation and divergence 1.2. On transformation level: Hard to perform analyses Require monotony to avoid alternation Give some example why divergence cannot easily be avoided, thus terminate at some point 2. find the solution in as many cases as possible -> reasonable orchestration strategy Focus on engineering solution

Thus, there arise two questions: - Although theoretically easy, how to practically define a CPR that is synchronizing? - How to define an APP function and which requirements does that impose?

Two levels of correctness:

1. Local correctness: a consistency relation is correct to the global relation and the CPR is to the relation, i.e. given two models and changes in them, the transformation can produce a change that restores consistency regarding the global consistency relation of these two models (i.e. there are some other models with which these two models would be consistent regarding the global specification) –> a network is locally correct, if this property is fulfilled
2. Global correctness: the binary relations together are equal to the global one and the execution function is able to find consistency models after a change to initially consistent models –> network is globally correct, if this property is fulfilled

Potentiell ist lokale Korrektheit (zumindest einer CPR zu ihrer CR per Konstruktion) herstellbar – das war auch das Ergebnis bisheriger Studien –, eventuell auch von einer CR zu einer globalen CR, obwohl die ja eigentlich meist nicht existiert, daher nehmen wir das als gegeben an. Dann zeigen, dass die globale Beziehung der Relationen nicht äquivalent ist zu den einzelnen lokalen, daher kommt hier zusätzliche Komplexität rein (Kompatibilitätsbegriff). Final muss noch die Ausführungsfunktion korrekt sein, hier aber Problem der Turing-Vollständigkeit. Daher Einschränkungen an Transformationen finden bzw. ingenieurmäßige Ausführungsreihenfolge festlegen, die möglichst oft richtige Lösungen findet und sonst konservativ mit einem Fehler terminiert.

On top of ordinary bx correctness:

- Transformations need to be synchronizing

- Consistency relations need to fulfill a notion of correctness
- Exkurs:
  - Is compatibility a subclass of correctness? Is every correct set of relations compatible as well?
  - Problematisch: unser Konsistenzbegriff für Relationen (feingranulare Relationen) schließt keine Modelle aus, der Konsistenzbegriff hier aber schon. Wie realisiere ich die feingranularen Relationen, die dafür sorgen, dass nur genau ein Tupel von Modellen konsistent ist?
  - Wir müssen bei der Ableitung unseres Kompatibilitätsbegriffes erklären, dass bei uns der vollständige Ausschluss bestimmter Modelle nicht Teil einer feingranularen Konsistenzrelation sein darf, sondern Teil einer weiteren Spezifikation, die angibt, welche Modelle überhaupt valide sind. Denn so ist es in Transformationssprachen tatsächlich auch.
- Execution function needs to be defined, which potentially induces requirements to the transformations.

Trivialisierung des Problems:

- Ohne weitere Annahmen ist das immer dadurch erreichbar, dass die Transformationen einen beliebigen anderen Zustand der Modelle produzieren. Im einfachsten Fall liefert jede Transformation immer die gleichen konsistenten Modelle zurück, unabhängig von der Änderung. Dann ist der Endzustand der Modelle nach der Ausführung des Netzwerks immer der gleiche.
- Das ist im allgemeinen aber nicht Fall. Letztendlich trifft jede Transformation lokale Entscheidungen. Beispielsweise könnte jede einzelne Transformation gegeben eine beliebige Änderung immer dieselben Modelle (bzw. Änderungen die dazu führen) zurückliefern (im trivialsten Fall leere Modelle). Dann erfüllt jede Transformation ihre Korrektheitseigenschaft bzgl. ihrer Relation, aber das Netzwerk muss nicht korrekt sein, da bspw. T(A,B) und T(B,C) sich immer für verschiedene Instanzen von B entscheiden. Es gäbe somit nie eine

konsistente Lösung für eine beliebige Ausführungsreihenfolge der Transformationen, auch wenn die Relationen das erlauben würden.

- Beispiel mit Namen, wo eine Transformation immer den großen Namen zurückliefert, die andere immer den kleinen. T(A,B) bildet A auf gleiches B ab und beide auf kleine Schreibweise, obwohl beide erlaubt sind. Erzeuge A="a", dadurch B="a". T(B,C) bildet B auf C ab und beide auf große Schreibweise, obwohl beide erlaubt sind. Somit macht sie das zu B="A" und C="A". Nun wird T(A,B) wieder beide klein machen usw. Allerdings wäre eine insgesamt valide Lösung einfach alle groß oder alle klein zu machen, aber die Transformationen finden diesen Zustand nicht.
- Allgemeiner ist zu sagen, dass ein Transformationsnetzwerk eine Turing-Maschine emulieren kann.

  Nachweisen!

  Im allgemeinen terminiert das Netzwerk somit nicht, schlimmer noch, es ist unentscheidbar, ob das Netzwerk hält (siehe Halteproblem).
- Dies zeigt bereits, dass keine Ausführungsfunktion definiert werden kann, die immer ein konsistentes Ergebnis liefert.
- Wir versuchen daher Annahmen an Transformationen zu finden, um diese Fälle auszuschließen bzw. systematisch zu verringern.
- Außerdem möchten wir eine Ausführungsfunktion haben, die ein konsistentes Ergebnis liefert oder einen Fehler, denn es muss nicht immer eine korrekte Lösung geben. Ziel ist es dann die Anzahl der Fälle, in denen sie einen Fehler zurückgibt, zu reduzieren.

Zielsetzung:

- Korrekte Anwendungsfunktion finden (in bestehenden Arbeiten [Ste17]) auch "Resolution" genannt (formal definieren!):
- Welche Anforderungen müssen wir dafür an die Transformationen stellen, damit solch eine Funktion definiert werden kann?
- Wir bezeichnen das Transformationsnetzwerk, in dem eine Transformation eingesetzt wird, als "Kontext"

- Welche dieser Eigenschaften kann die einzelne Transformation (ohne Kenntnis der anderen) erfüllen und für welche muss der Kontext (d.h. die anderen Transformationen) bekannt sein?
- ⇒ Interesse an "kontextfreien" Eigenschaften (lassen sich ohne Kenntnis der anderen Transformationen sicherstellen -> Wiederverwendbarkeit) und "kontextsensitiven" Eigenschaften (Erfüllung der Eigenschaft nur durch Kenntnis über das Transformationsnetzwerk möglich)
- Kontextfreie Eigenschaften involvieren solche, die wir eh schon von Transformationen kennen (Korrektheit einer Transformation, Hippokratie etc.) und solche, die dadurch zustande kommen, dass man weiß, dass diese Transformation in einem Netzwerk eingesetzt werden soll.
- Zielsetzungsoptionen:
  - Wir schränken die Transformationen so ein, dass es immer mindestens eine Ausführungsreihenfolge der Transformationen gibt, sodass für jede beliebige Änderung ein konsistentes Ergebnis durch Anwenden der Transformationen gefunden werden kann
  - Wir akzeptieren, dass es Änderungen gibt, für die das Netzwerk kein konsistentes Ergebnis produzieren kann. Dann muss das Netzwerk (mindestens) in diesen Fällen mit einer Fehlermeldung terminieren.
  - Eine Option ist, dass das Netzwerk dieses Verhalten nur approximiert bzw. approximieren kann, dann muss es sich konservativ verhalten, d.h. im Fall, dass es keine Lösung gibt, auf jeden Fall eine Fehlermeldung geben, und im Fall, in dem es eine Lösung gibt, diese bestenfalls finden oder ausgeben, dass es keine finden kann (d.h. keine False Positives bzw. Nicht-Terminierung). Ziel ist es dann den Grad der Konservativität zu minimieren.
- Lösungsoptionen (Grad der Einschränkung an die Transformationen):

- Hohe Einschränkung: Jede beliebige Reihenfolge von ausgeführten Transformationen führt letztendlich zu einem korrekten Ergebnis (Fixpunktiteration – Allquantifizierung) – Hippokratie-Eigenschaft sorgt dafür, dass keine Transformation wieder etwas ändert, wenn Konsistenz bereits hergestellt ist. Diese Eigenschaft ist in der Praxis möglicherweise zu strikt, da sie sehr starke Anforderungen an die Transformationen stellen müsste. Dafür wäre aber die Anwendungsfunktion trivial.
- Mittlere Einschränkung: Es gibt eine Reihenfolge von ausgeführten Transformationen für jede Änderung die terminiert (Existenzquantifizierung) und die Ausführungsfunktion findet diese Reihenfolge. Utopisch, dass die Anwendungsfunktion aus (potentiell sehr mächtigen) Transformationen die richtige Reihenfolge errechnen kann. Dafür aber (möglicherweise) weniger Anforderungen an die Transformationen (zumindest nicht mehr Anforderungen, denn die Allquantifzierung induziert die Existenzquantifizierung). Eine Funktion könnte dann zumindest nach best-effort versuchen, die richtige Reihenfolge zu finden und konservativ abbrechen, wenn sie diese nicht finden kann (also entweder konsistent terminieren oder terminieren mit der Aussage, dass es entweder keine solche Reihenfolge gibt – bei relaxierten Anforderungen – oder dass es sie nicht finden kann).
- Geringe Einschränkung: Es gibt potentiell keine Reihenfolge der Transformationen, die bei einer Änderung zu einer konsistenten Lösung kommt. Hier müsste die Ausführungsfunktion entsprechend einen Fehler ausgeben.
- Bestehende Arbeiten ([Ste17]) schlagen auch vor eine Baumstruktur zu berechnen (Spannbaum), in dem nur entlang der Baumkanten die Transformationen ausgeführt werden. Dies ist jedoch eine starke Einschränkung daran, was die Transformationen ausdrücken können. Betrachtet man beispielsweise PCM, UML und Java, und hat eine Änderung in PCM. Dann könnte der Spannbaum entweder PCM -> UML ->

Java sein, oder PCM -> UML + PCM -> Java. In ersterem Fall würde Verhaltensbeschreibung, die von PCM nach Java übertragen, aber in UML nicht dargestellt wird, nicht übertragen. Im zweiten Fall würde zusätzliche Information zwischen UML und Java nicht propagiert (Beispiel?) –> Hier sollte auf das Properties-Kapitel verwiesen werden, wo diese "Bottlenecks" erklärt sein sollten, inklusive einem Beispiel, die allgemein Baumstrukturen für Transformationsnetwerke ausschließen.

- Dies setzt voraus, dass die Transformationen und die Anwendungsfunktion mit jeder beliebigen Nutzer-Änderung umgehen kann. Man kann jedoch auch verlangen, dass die Anwendungsfunktion genau dann, wenn es überhaupt eine Ausführungsreihenfolge gibt, diese findet, und sonst einen Fehler ausgibt.
- **Wichtig:** Im Allgemeinen kann eine Ausführungsfunktion keine terminierende Reihenfolge berechnen, da die Transformationen Turing-vollständig sind und deshalb die Frage, welche Reihenfolge zu einer Terminierung führt, unentscheidbar ist (Halteproblem). Daher können wir nur einen konservativen Algorithmus angeben, der ein sinnvolles Abbruchkriterium definiert, mit dem die Ausführung beendet wird, auch wenn potentiell eine Lösung hätte gefunden werden können. Die Fragestellung ist also, wie die Ausführungsfunktion aussehen muss, damit sie in möglichst vielen Fällen, in denen es eine terminierenden Reihenfolge gibt, diese auch findet. Insbesondere lässt sich somit keine geschlossene Form für die Ausführungsfunktion angeben, sondern nur ein Algorithmus, der zur Laufzeit eine Reihenfolge (dynamisch) festlegt.

Problemraum:

- Ziel ist, dass ein Netzwerk von Transformationen nach einer Änderung in einem konsistenten Zustand terminiert. D.h. Korrektheit stellt Anforderungen an *Terminierung*, sowie den *Zustand* bei Terminierung.
- Folgende Abweichungen davon können auftreten:

1. Nicht-Terminierung: Das Netzwerk terminiert nicht. Das bedeutet im Prinzip, dass die Ausführungsfunktion (bzw. der Laufzeit-Algorithmus, der die Funktion dynamisch emuliert) nicht *sound* ist. Soundness der Ausführungsfunktion setzt voraus, dass die berechnet Aufrufsequenz endlich ist. Wenn die Ausführung nicht terminiert, bedeutet das, dass entweder die gleichen Zustände mehrfach durchlaufen werden oder eine Sequenz unendlich vieler Zustände produziert wird. Denn wenn beides nicht der Fall ist, gibt es eine endliche Sequenz unterschiedlicher Zustände, d.h. Terminierung. Das bedeutet, dass es folgende zwei Möglichkeiten gibt:
   - Alternierung: Die gleichen Zustände werden mehrfach durchlaufen.
   - Divergenz: Es werden unendlich viele Zustände produziert.
2. Inkonsistente Terminierung: Die Ausführungsfunktion bzw. der Algorithmus beendet die Ausführung, aber in einem inkonsistenten Zustand. Hier lassen sich ebenfalls wieder zwei Fälle unterscheiden.
   - Unerkannte Inkonsistenz: Der Algorithmus terminiert und denkt, der Zielzustand wäre konsistent. Dies bedeutet aber direkt, dass nicht alle Konsistenzrelationen erfüllt sind, was, zumindest in der Theorie, einfach zu prüfen wäre (entweder durch Prüfung der Relationen oder durch Ausführung der hippokratischen Transformationen, die alle nichts tun dürften)
   - Erkannte Inkonsistenz: Der Algorithmus terminiert, wissend dass die Lösung nicht konsistent ist. Dies kann entweder sein, weil eine Transformation für zwei Modelle in einem inkonsistenten Zustand nicht mehr anwendbar ist, oder weil irgendein anderes Abbruchkriterium erreicht ist.

Annahmen:

- Nutzeränderungen dürfen nicht rückgängig gemacht werden.

- Nutzeränderungen lassen sich so feingranular zerlegen, dass, falls durch die Erzeugung/Änderung eine Konsistenzrelation verletzt wird, es in jeder unabhängigen Teilmenge von Konsistenzrelationen eine verletzte Konsistenzrelation gibt, für die die geänderten Elemente einem Condition Elemente entsprechen, es also insbesondere keine Teilmenge der geänderten Element gibt, die bereits dieses Condition Element sind. Ansonsten ist durch unsere Kompatibilitäts-Definition nicht sichergestellt, dass eine konsistente Modellmenge gefunden werden kann.

Voraussetzungen:

- Relationen müssen korrekt sein, d.h. sie müssen bzgl. einer globalen (meist eher implizit bekannten) n-ären Relation zwischen allen Modellen identisch sein. Eine n-äre Relation lässt sich nicht immer zerlegen (siehe Stevens), aber wir nehmen das an.
- Die einzelne Transformation muss bzgl. ihrer Relation korrekt sein, d.h. sie muss bei Änderungen in beiden Modellen ein zur Relation konsistentes Modell liefern.

Ebenen der Korrektheit:

- Relationen müssen korrekt sein, d.h. gegeben eine Nutzeränderung muss es überhaupt möglich sein eine konsistente Menge an Modellen zu finden. Wenn Transformationen etwas beliebigen tun dürfen geht das immer. Wir nehmen an, dass eine Nutzeränderung nicht rückgängig gemacht werden soll (bzw. wenn sie rückgängig gemacht werden würde eigentlich die Änderung invalide war, d.h. keine Konsistenz im Netzwerk hergestellt werden kann). Daher sind Relationen nur korrekt, wenn für fixierte Elemente, die durch eine Nutzeränderung entstehen können, eine Modellmenge abgeleitet werden kann, die bzgl. der Relationen konsistent ist. D.h. gegeben einige Elemente muss es eine Modellmenge geben, die in allen Relationen liegt und die diese Elemente enthält (-> Kompatibilitätsbegriff). Wir betrachten in Kapitel ?, wie man Kompatibilität präzise definieren und feststellen/garantieren kann. Resultat: Gegeben eine Änderung ist es möglich eine Transformation anzugeben, die aus der Änderung ein konsistentes Modell produziert.

- Einzelne Transformationen müssen korrekt sein: Wir fordern Korrektheit der Transformation sowieso. Allerdings machen in einem Netzwerk verschiedene Transformationen Änderungen an allen Modellen, d.h. wir müssen nicht den "normalen" Transformationsfall unterstützen, dass Deltas in einem Modell ins andere übertragen werden, um Konsistenz herzustellen, sondern die Transformationen müssen *synchronisierend* sein, also Deltas in beiden Modelle annehmen und dann Konsistenz herstellen. Wir definieren diese Synchronisationseigenschaft und betrachten in Kapitel ?, welcher zusätzlichen Anforderungen sich dadurch bzgl. EMOF-Modellen ergeben. Der Input sind Deltas in zwei Modellen, und einzelne Deltas sind potentiell als "authoritative" definiert, was bedeutet, dass die erzeugten/geänderten Elemente nicht noch einmal geändert/gelöscht werden dürfen. Das realisiert die Anforderung, dass Nutzeränderungen nicht rückgängig gemacht werden dürfen.
  Resultat: Gegeben Änderungen in zwei Modellen (mit potentiell authoritativen Änderungen) gibt die Transformation ein konsistentes (bzgl. der Konsistenzrelation) Modellpaar zurück.
- Korrektheit der Anwendungsfunktion: Die Anwendungsfunktion muss die Transformationen in einer

Annahme an Transformationen:

- Muss eine Transformation mit jedem beliebigen Delta umgehen können müssen? Eine Einschränkung auf Monotonie würde dies verhindern. Bzw. wir müssten zeigen, dass es Konsistenzrelationen gibt, die unter der Anforderung an Monotonie nicht wiederhergestellt werden können. Bspw. fügt eine andere Transformation 3 Elemente hinzu, wo zwei mit dem anderen entsprechend der Konsistenzrelationen korrelieren und somit keine Witness-Struktur aufgebaut werden kann, die Konsistenz beweist. Das lässt sich durch Hinzufügen weiterer Elemente potentiell nicht auflösen (siehe Beispiele im SoSym-Paper).

Notwendigkeit Transformationen oder Anwendungsfunktion einzuschränken:

- Zeigen, dass es Beispiele gibt, in denen es keine einzige Ausführungsreihenfolge gibt (All-Quantifizierung), die zu einem konsistenten Ergebnis führt:
- Zeigen, dass es Beispiele gibt, in denen es unabhängig von der Ausführungsreihenfolge immer zu einer Alternierung kommt
- Zeigen, dass es Beispiele gibt, in denen es unabhängig von der Ausführungsreihenfolge immer zu einer Divergenz kommt.
- Die Beispiele sollten zeigen, dass wir keine Einschränkungen an die Transformationen machen können, was das Problem aushebelt. D.h. egal welche Einschränkungen ich an die Transformationen definiere, es lassen sich immer Beispiele konstruieren, in denen es keine Ausführungsreihenfolge gibt, in denen sie terminieren.
- Mathematisch zeigen, dass Alternierung und Divergenz die einzigen Probleme sind. D.h. wenn nicht der gleiche Zustand mehrmals durchlaufen wird (Alternierung) und es nicht unendlich viele Zustände gibt (Divergenz), dann ist die Folge endlich.
- Außerdem mathematisch die Abbildung von Transformationen auf Turing-Maschinen zeigen und damit ableiten, dass allgemeine Netzwerke erstmal nicht terminieren müssen (Abbildung auf Halteproblem)

Zielsetzung die Zweite:

- Wir definieren möglichst minimale Beschränkungen, die dazu führen, dass das Netzwerk terminiert. D.h. es terminiert entweder konsistent oder es terminiert mit einem Fehler, der sagt, dass entweder keine Konsistenz hergestellt werden kann (es gibt keine Ausführungsreihenfolge der Transformationen, die zu Konsistenz führt) oder dass die Anwendungsfunktion nicht in der Lage war eine passende Ausführungsreihenfolge zu finden (Konservativität)
- Zwei Arten von Beschränkungen
  - Beschränkungen an die Transformationen, die dazu führen, dass es in mehr Fällen mindestens eine Ausführungsreihenfolge gibt, in der das Netzwerk konsistent terminiert

  - Beschränkungen an die Ausführungsfunktion, sodass die Ausführung auf jeden Fall terminiert, wenn auch konservativ, d.h. mit Fehler, obwohl es eine korrekte Lösung gegeben hätte.

Exkurs: Menge (konsistenter) Modelle bildet keinen topologischen Raum

- Topologischer Raum besteht aus Grundmenge und Mengensystem von Teilmengen mit den Eigenschaften, dass die Grundmenge offen ist, der Schnitt endlich vieler Mengen offen und die Vereinigung beliebig vieler Mengen offen ist.
- Die Grundmenge wäre die Menge aller Modellelemente
- Diese Menge ist normalweise offen, da z.B. für ein Element mit einem String-Attributwert immer noch das Element mit dem gleichen String-Attributwert plus einem weiteren Symbol in der Menge liegt (und man die Ordnung in der Menge entsprechend definiert). Dass ein Metamodell möglicherweise Einschränkungen definiert und dann im schlimmsten Fall nur ein einziges Modell valide ist, lassen wir hier außen vor.
- Betrachten wir nun eine Topologie auf dieser Menge, also ein Mengensystem aus konsistenten Modellen. Leider ist jedoch der Schnitt zweier konsistenter Modelle nicht zwangsläufig konsistent. Insbesondere sind diese Mengen auch nicht offen, da sie die abgeschlossene Menge darstellen, die genau ein Modell beschreiben.
- Somit lässt sich die Definition von Topologien hier nicht anwenden.

Überlegen, wo hier die Definition von (undirektionalen Relationen) rein muss.

Präzisere Eigenschaften:

- Synchronisationseigenschaft: Eine Transformation kann mit Änderungen an mehreren Modellen umgehen, d.h. gegeben zwei konsistente Modelle + Änderungen an beiden resultiert in zwei Modellen, die konsistent bzgl. der Relation(en) zwischen den Metamodellen sind
-

- Kompatibilität entsprechend Modularisierungsebene
- Synchronisation auf Operationalisierungsebene: Abwägen, dass eine Transformation verschiedene Zustände sehen könnte, auf denen sie ausgeführt wird. Aber letztendlich muss sie damit klarkommen, dass zwei Modelle geändert wurden.

TODO:

- Authoritative Modelle (bzw. eher authoritative Regionen) diskutieren (Verweis Stevens)

6.6. Local Correctness

Simple solution: we define a transformation which normatively implies a relation, thus it is correct by construction. From a theoretical perspective this is easy to reach, from a practical it is not. However, in contrast to our definition of synchronizing transformations, ordinary transformations are only able to process changes in one model and update the other accordingly. Together with the assumption that both models were consistent before does not fit with our scenario, because if one model is modified, the other may be modified as well by another transformation across another path, before a transformation is executed. Thus, both models may have been modified. We consider the following situation: Models A and B were consistent. Model A was changed an we have the changes at hand. Additionally, B was modified because there were other changes propagated through the network. We distinguish all cases of modifications to B that may have violated a consistency relation between A and B (according to our fine-grained consistency notion) and consider what we have to do there (e.g. find-or-create-pattern). Put empirical analysis here.

6.7. Correct APP function

We make the following approach: Always assume there is a solution and start executing the transformation (for now in any order). Finally, the network has to terminate at a fixed point. We investigate, what the reasons may be that it does not try to avoid them.

These reasons can lie in the relations: - relations cannot be completely unfulfillable, as the empty models are always consistent, thus there can always be CPRs that result in a consistent set of models - however, if relations contain pairs that can never be in any consistent model tuple they improve proneness to errors, because a CPR may return that pair, which will never fit to any result of any other transformation. Thus, this should not be allowed -> compatibility

These reasons can also lie in the transformations: - Transformations can make choices and they make choices that are always incompatible to other (refer to example)

Essentially there are two problems: alternation and divergence

6.7.1. Other thought

If each element occurs in each relation only once (so always 1:1 mappings) and if we have compatibility, then any transformation order would return exactly the one model tuple that fits. However: In that case we would have confluence, every information must directly be available in B from A without a transitive propagation over C. This is not what we want. So there must in general be more than one option a transformation is fine with that to reflect the information that another transformation may add or change.

Hippocraticness is not necessary but needs to be discussed

Goal: - Find a solution in as much cases as possible, abort in the others (conservatively) - To do so: reduce cases in which there is no such function - To do so: ensure that relations are defined in a way such that they do not allow a locally correct set of CPRs that has no APP solution. If there is a pair of models (or elements of a fine-grained relation) in a relation, a CPR may return it. But if there is no consistent tuple of models containing these two, it does not make any sense to consider these elements (even worse, if we have monotony, adding these elements makes the network unsolvable). For that reason, we need compatibility. -

6.8. A Formalization of Compatibility

In this section, we precisely define our notion of consistency, and motivate and formally introduce the term *compatibility*. We first discuss properties of transformation networks with an intuitive notion of compatibility, based on considerations in existing work. We then define consistency based on fine-grained consistency relations and, finally, derive a compatibility notion from the consistency formalization and its pursued perception. This serves as our contribution **C1**.

6.8.1. Properties of Transformation Network

Keeping pairs of models consistent by means of incremental, bidirectional transformations has been well researched in recent years [Ste10; Kus+13; Cle+19]. A bidirectional transformation consists of a *relation* that specifies which pairs of models are considered consistent and a pair of directional transformations, denoted as *consistency repair routines*, that take one modified and one originally consistent model and deliver a new model that is consistent to the modified one [Ste10]. Several well-defined properties of such transformations have been identified. The essential *correctness* property states that a consistency repair routine delivers a result such that the models are actually consistent according to the defined relation [Ste10]. Another important property is *hippocraticness*, which states that a consistency repair routine returns the input model if it was already consistent to the modified one [Ste10].

When we combine several transformations to a network to achieve consistency between multiple models, those properties of the single transformations are still relevant, as each transformations on its own has to be at least correct to work properly in a network of transformations. However, correctness of the single transformations does not induce correctness of the transformation network. Taking an arbitrary set of correct transformations and executing them one after another does not necessarily constitute a terminating approach that delivers a result, in which all models are consistent according to the relations of the transformations, because the result of one transformation may violate the relation of another. It is possible that the approach does either not terminate, because there is a divergence or alternation in values changed or elements created, or terminates in a state that is not consistent regarding the relations of all transformations [Kla+19].

Readd

The property of a network to always result in a state in which the models are consistent to all relations of the transformations if they are executed in a specific order, can be seen as a *correctness* property for transformation networks. In this work, we focus on that correctness property and do not discuss further quality properties of transformation networks, such as *modularity*, *evolvability* and *comprehensibility* [Kla18].

A single transformation can only be incorrect in terms of its repair routines, because there are no restrictions regarding its relation that may prevent the repair routine from being able to produce a correct result. In previous work, however, we identified that transformation networks can be incorrect at different levels [Kla+19]. Networks can also be incorrect at the level of relations rather than repair routines, because multiple relations can be contradictory, i.e., they can relate elements in different ways such that the relations cannot be fulfilled at the same time.

Readd

In such a case, the consistency repair routines cannot be result in a state that is consistent according to the relations anymore, thus they may not terminate anymore. We call the relations of such transformations *incompatible*.

In consequence, compatibility of relations is a necessary prerequisite for consistency repair routines to produce correct results in transformation networks. We also found that correctness of the consistency repair routines can already be achieved by construction, whereas compatibility of the relations cannot be achieved by construction but in the best case be checked for a set of relations. In this work, we focus on the possibility to check compatibility of the relations of a set of transformations. In the following, we therefore precisely define the notion of compatibility of relations, which excludes contradictions in relations that can prevent consistency repair routines from fulfilling the relation.

Example!

Finally, the topology of a transformation network directly influences how prone it is to incompatibilities of its relations. Contradictions of consistency relations, as exemplified with the relations R_{PE}, R_{PR}, R'_{ER} in Figure 1.2, can only occur if the same classes are related to each other by different (sequences of) transformations in a different way. For example, in Figure 1.2, each combination of two relations puts the same classes into relation as the third one. This means that a transformation network, in which each pair of classes is only related by one sequence of transformations, cannot have contradictory relations and is thus inherently compatible.

6.8.2. A Fine-grained Notion of Consistency

A common definition of consistency enumerates consistent pairs of models in a relation [Ste10]. However, for our studies on compatibility, we need a more fine-grained notion of consistency. Considering transformations languages, such as QVT-R, first, relations are defined at the level of classes and their properties, i.e. how properties of instances of some classes are related to properties of instances of other classes. Second, they are defined in an *intensional* way, i.e., constraints specify which elements shall be considered consistent, rather than enumerating all consistent instances, known as an *extensional* specification. Both ways have equal expressiveness and especially each intensional specification can be transformed into an extensional one by enumerating all instances that fulfill the constraints. Since mathematical statements are easier to make on extensional specifications, we stick to them. However, we reuse the concept of specifying relations at the level of classes and their properties. This makes it easier to make statements about dependencies between consistency relations. For example, two fine-grained consistency relations considering completely independent sets of classes cannot interfere, and thus especially do not introduce any compatibility problems, which is not easy to express when considering relations at the level of complete models. Finally, from such a fine-grained specification, a holistic relation at level of models can always be derived by enumerating all models that fulfill all the fine-grained specifications, thus it does not restrict expressiveness in any way and can be seen as a *compositional approach* for defining consistency.

In the following, we start with introducing a fine-grained notion of consistency relations. We proceed with considerations on implicit relations, which are induced by a set of consistency relations, such as transitive relations, to finally precisely define a notion of compatibility.

6.8.2.1. Consistency

The first definitions on conditions and consistency relations are based on the work of Kramer [Kra17, sec. 2.3.2, 4.1.1]. The central idea of the consistency notion is to have consistency relations, which contain pairs of objects and, broadly speaking, requires that if the objects in one side of the pair occur in a model, the others have to occur in another model as well.

Definition 16 (Condition). A condition $\mathbb{c}$ for a class tuple $\mathfrak{C}_{\mathbb{c}} = \langle C_{\mathbb{c},1}, \ldots, C_{\mathbb{c},n} \rangle$ is a set of object tuples with:

$$\forall \langle o_1, \ldots, o_n \rangle \in \mathbb{c} : \forall\, i \in \{1, \ldots, n\} : o_i \in I_{C_{\mathbb{c},i}}$$

An element $\mathfrak{c} \in \mathbb{c}$ is called a *condition element*. For a set of models $\mathbb{m} \in I_{\mathbb{M}}$ of a metamodel set $\mathbb{M}$ and a condition element $\mathfrak{c}$, we say that:

$$\mathbb{m} \textit{ contains } \mathfrak{c} :\Leftrightarrow \exists\, m \in \mathbb{m} : \mathfrak{c} \subseteq m$$

Conditions represent object tuples that instantiate the same tuple of classes. They are supposed to occur in models that fulfill a certain condition regarding consistency, i.e., they define the objects that can occur in the previously mentioned pairs of consistency relations, which we specify later. We say that a set of models contains a condition element if any of the models contains all the objects within the condition element. This implies that the metamodel of such a model has to contain all the classes in the class tuple of the condition.

Call class tuple the "signature" of a relation.

Definition 17 (Consistency Relation)**.** Let $\mathfrak{C}_{l,CR}$ and $\mathfrak{C}_{r,CR}$ be two class tuples. A consistency relation CR is a subset of pairs of condition elements in conditions $\mathbb{c}_{l,CR}, \mathbb{c}_{r,CR}$ with $\mathfrak{C}_{l,CR} = \mathfrak{C}_{\mathbb{c}_{l,CR}}$ and $\mathfrak{C}_{r,CR} = \mathfrak{C}_{\mathbb{c}_{r,CR}}$:

$$CR \subseteq \mathbb{c}_{l,CR} \times \mathbb{c}_{r,CR}$$

We call a pair of condition elements $\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \in CR$ a *consistency relation pair*. For a set of models $\mathbb{m}$ and a consistency relation pair $\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle$, we say that:

$$\mathbb{m} \textit{ contains } \langle \mathfrak{c}_l, \mathfrak{c}_r \rangle :\Leftrightarrow \mathbb{m} \textit{ contains } \mathfrak{c}_l \wedge \mathbb{m} \textit{ contains } \mathfrak{c}_r$$

Without loss of generality, we assume that each condition element of both conditions occurs in at least one consistency relation pair, i.e.

$$\begin{aligned}
&\forall\, \mathfrak{c} \in \mathbb{c}_l : \exists \langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \in CR : \mathfrak{c} = \mathfrak{c}_l \\
&\wedge \forall\, \mathfrak{c} \in \mathbb{c}_r : \exists \langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \in CR : \mathfrak{c} = \mathfrak{c}_r
\end{aligned}$$

A consistency relation according to Definition 17 is a set of pairs of object tuples, which are supposed to indicate the tuples of objects that are considered consistent with each other, i.e., if a model contains one of the left object tuples occurs in the relation one of the related right object tuples has to occur in a model as well. It is based on two conditions that define relevant object tuples in each of the two metamodels and defines the ones that are related to each other.

Metamodels in definition may not be different, so consistency relations within a model can be defined. May be a problem if the class tuples are overlapping, so exclude and discuss that?

We can now define a notion of consistency for a set of models based on the definition of consistency relations.

Definition 18 (Consistency)**.** Let *CR* be a consistency relation and let $\mathbb{m} \in I_{\mathbb{M}}$ be a set of models of the metamodels in $\mathbb{M}$. We say that:

$$\begin{aligned}
&\mathbb{m} \textit{ consistent to } CR :\Leftrightarrow \\
&\quad \exists W \subseteq CR : \big(\forall \langle \mathfrak{c}_{l,1}, \mathfrak{c}_{r,1}\rangle, \langle \mathfrak{c}_{l,2}, \mathfrak{c}_{r,2}\rangle \in W : \\
&\qquad\quad \langle \mathfrak{c}_{l,1}, \mathfrak{c}_{r,1}\rangle = \langle \mathfrak{c}_{l,2}, \mathfrak{c}_{r,2}\rangle \vee (\mathfrak{c}_{l,1} \neq \mathfrak{c}_{l,2} \wedge \mathfrak{c}_{r,1} \neq \mathfrak{c}_{l,2})\big) \\
&\qquad \wedge \forall \langle \mathfrak{c}_l, \mathfrak{c}_r\rangle \in W : \mathbb{m} \textit{ contains } \mathfrak{c}_l \wedge \mathbb{m} \textit{ contains } \mathfrak{c}_r \\
&\qquad \wedge \forall \mathfrak{c}'_l \in \mathbb{C}_{l,CR} : \mathbb{m} \textit{ contains } \mathfrak{c}'_l \Rightarrow \mathfrak{c}'_l \in \mathbb{C}_{l,W}
\end{aligned}$$

We call such a W a *witness structure* for consistency of $\mathbb{m}$ to *CR*, and for all elements $\langle \mathfrak{w}_l, \mathfrak{w}_r\rangle \in W$, we call $\mathfrak{w}_l$ and $\mathfrak{w}_r$ *corresponding to* each other.

For a set of consistency relations $\mathbb{CR} = \{CR_1, CR_2, \ldots\}$, we say that:

$$\begin{aligned}
&\mathbb{m} \textit{ consistent to } \mathbb{CR} :\Leftrightarrow \\
&\quad \forall CR \in \mathbb{CR} : \mathbb{m} \textit{ consistent to } CR
\end{aligned}$$

A consistency relation *CR* relates one condition element at the left side to one or more other condition elements at the right side of the relation. The definition of consistency ensures that if one condition element $\mathfrak{c} \in \mathbb{C}_{l,CR}$ in the left side of the relation occurs in a set of models, exactly one of the condition elements related to it by a consistency relation *CR* occurs in another model to consider the set of models consistent.

Give example why this is reasonable, i.e. a counterexample for the more simple notion of Max.

If another element that is related to $\mathfrak{c}$ occurs in the models, this one has to be, in turn, related to another condition element $\mathfrak{c}' \in \mathbb{C}_{l,CR}$ of the left side of condition elements by *CR*, which also occurs in the models. This is a necessary restriction, because usually a single corresponding element is expected, as we will see in examples in the following. To achieve that, the definition uses an auxiliary structure *W*, which serves as a witness structure for those pairs of condition elements that co-occur in the models.

Example 1. The definition of consistency is exemplified in Figure 6.7, which is an alternation of an extract of Figure 1.2 only considering employees and

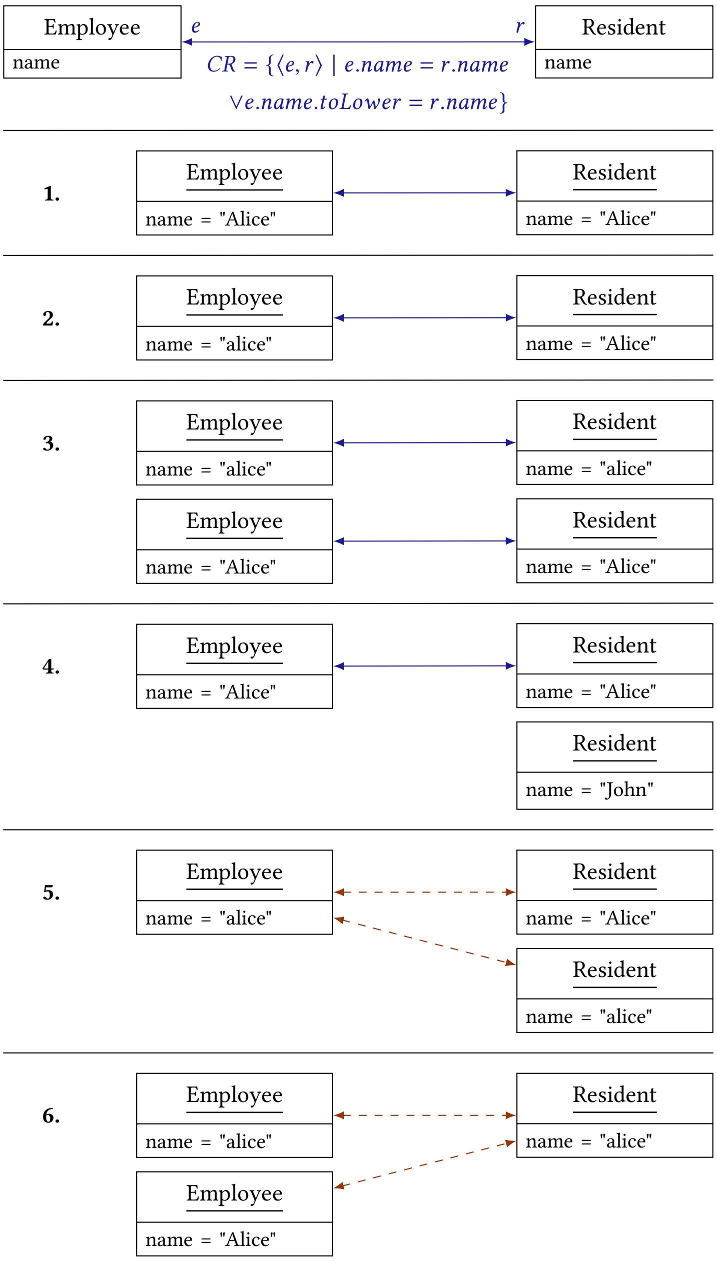


Figure 6.7.: A consistency relation between employee and resident and six example model pairs: model pairs 1–4 being consistent with an appropriate witness structure W shown in blue and model pair 5 and 6 being inconsistent with an inappropriate mapping structure shown in red and dashed.

residents. Models with employees and residents are considered consistent if for each employee exactly one resident with the same name or the same name in lowercase exists. The model pairs 1–3 are obviously consistent according to the definition, because there is always a pair of objects that fulfills the consistency relation. In model pair 4, there is a consistent resident for each employee, but there is no appropriate employee for the resident with $name$ = ”$Alice$”. However, our definition of consistency only requires that for each condition element of the left side of the relation that appears in the models an appropriate right element occurs, but not vice versa. Thus, a relation is interpreted unidirectionally, which we will discuss in more detail in the following. In model pair 5, there are two residents with names in different capitalizations, which would both be considered consistent to the employee according to the consistency relation. Comparably, in model pair 6, there is a resident that fulfills the consistency relations for both employees, each having a different but matching capitalization. However, the consistency definition requires that each element in a model for which consistency is defined by a consistency relation may only have one corresponding element in the model. In this case, there are two residents respectively two employees that could be considered consistent to the employee respectively resident, thus there is no appropriate witness structure with a unique mapping between the elements as required by the consistency definition.

As mentioned before, we define the notion of consistency in a unidirectional way, which means that a consistency relation may define that some elements $\mathfrak{c}_r$ are required to occur in a set of models if some elements $\mathfrak{c}_l$ occur, but not vice versa. Such a unidirectional notion can also be reasonable in our example, as it could make sense to require a resident for each employee, but not every resident might also be an employee. To achieve a bijective consistency definition, for each consistency relation CR its transposed relation $CR^T = \{\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \mid \langle \mathfrak{c}_r, \mathfrak{c}_l \rangle \in CR\}$ can be considered as well. Regarding Figure 6.7, if we consider the relation between employees and residents as well as its transposed, the model pair 4 would also be considered inconsistent, because an appropriate employee for each resident would be required by the transposed relation. We call sets of consistency relations that contain only bijective definitions of consistency *symmetric*.

Definition 19 (Symmetric Consistency Relation Set)**.** Let $\mathbb{CR}$ be a set of consistency relations. We say that:

$$\begin{aligned}&\mathbb{CR} \textit{ is symmetric } :\Leftrightarrow\\ &\quad \forall CR \in \mathbb{CR} : \exists CR' \in \mathbb{CR} : CR' = CR^T\end{aligned}$$

Any description of bijective consistency relations can be achieved by defining a symmetric set of consistency relations. We chose to define consistency in a unidirectional way due to two reasons:

1. Some relevant consistency relations are actually not bijective. Apart from the simple example concerning residents and employees, this situation always occurs when objects at different levels of abstraction are related. Consider a relation between components and classes, requiring for each component an implementation class but not vice versa, or a relation between UML models and object-oriented code, requiring for each UML class an appropriate class in code but not vice versa. These relations could not be expressed if consistency relations were always considered bidirectional for determining consistency.
2. We consider networks of consistency relations, in which, as we will see later, a combination of multiple bijective consistency relations does not necessarily imply a bijective consistency relation again. Thus, we need a unidirectional notion of consistency relations anyway.

6.8.2.2. Implicit Consistency Relations

Each set of consistency relations defines binary consistency relations, each between two sets of classes. However, such consistency relations imply further *transitive* consistency relations. Having one relation between classes A and B and one between B and C implies an additional relation between A and C, for which we define a notion for the concatenation of relations. The goal of this notion is to provide a relation that is induced by the concatenated ones. This means, if a model is consistent to the concatenation, it should also be consistent to the single relations, as otherwise the concatenation would introduce additional consistency constraints. To achieve this, the following definition makes appropriate restrictions to the derived consistency relation pairs.

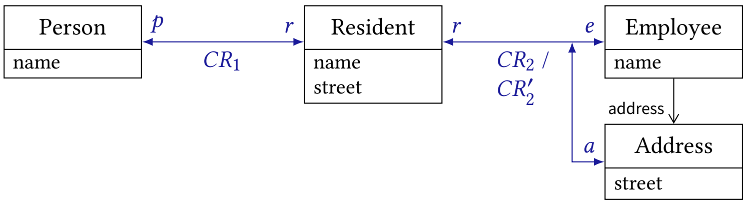
Actually, a concatenation may also consider that two or more relations are concatenated to a single one. I.e. CR1 could map something to A and B, and CR2 could map A to something and CR3 could map B to something. Then there could be a combination of all of them. In fact, each pair of consistency relations between the same metamodels can be combined to "larger" relation that then may be concatenated to other relations. Such a pair could even be a pair of a relation with itself, like if a relation maps on element to two of the same class and another relation then maps one element of the class to another. In summary, our notion of transitivity has to consider that concatenation may not only be sequences, but acyclic graphs.

Definition 20 (Consistency Relations Concatenation)**.** Let CR_1, CR_2 be two consistency relations. The concatenation is defined as follows:

$$\begin{aligned}
CR = CR_1 \otimes CR_2 := \{ & \langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \mid \\
& \exists \langle \mathfrak{c}_l, \mathfrak{c}_{r,1} \rangle \in CR_1 : \exists \langle \mathfrak{c}_{l,2}, \mathfrak{c}_r \rangle \in CR_2 : \mathfrak{c}_{r,1} \subseteq \mathfrak{c}_{l,2} \\
& \wedge \forall \langle \mathfrak{c}_l, \mathfrak{c}'_{r,1} \rangle \in CR_1 : \exists \langle \mathfrak{c}'_{l,2}, \mathfrak{c}'_{r,2} \rangle \in CR_2 : \mathfrak{c}'_{l,2} \subseteq \mathfrak{c}'_{r,1} \}
\end{aligned}$$

with $\mathfrak{C}_{l,CR} = \mathfrak{C}_{l,CR_1}$ and $\mathfrak{C}_{r,CR} = \mathfrak{C}_{r,CR_2}$

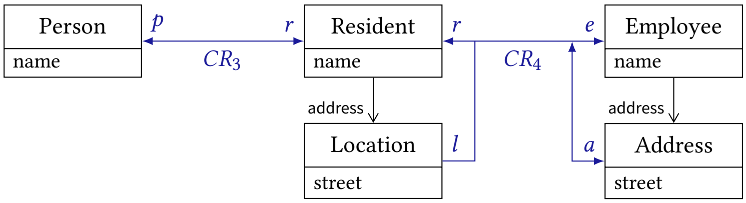
The concatenation of two consistency relations contains all pairs of object tuples that are related across common elements in the right respectively left side of the consistency relation pairs. Such a concatenation may be empty. Two requirements ensure that all models considered consistent to the concatenated relation are also consistent to the single relations: First, it is important that a pair of consistency relations CR_1, CR_2 is only combined if the left condition element of the consistency relation pair from CR_2 is a subset of the right condition element of the consistency relation pair $\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle$ of CR_1. Second, it is necessary that for all elements $\mathfrak{c}_r$ in the right side of CR_1, to which a condition element $\mathfrak{c}_l$ is considered consistent, there must be a matching condition element, i.e. a subset of $\mathfrak{c}_r$, in the left condition of



$$CR_1 = \{\langle p, r\rangle \mid p.name = r.name\}$$

$$CR_2 = \{\langle r, (e, a)\rangle \mid r.name = e.name \wedge r.street = a.street\}$$

$$CR'_2 = \{\langle r, (e, a)\rangle \mid \langle r, (e, a)\rangle \in CR_2 \wedge r.street \neq \text{""}\}$$



$$CR_3 = \{\langle p, r\rangle \mid p.name = r.name\}$$

$$CR_4 = \{\langle (r, l), (e, a)\rangle \mid r.name = e.name \wedge l.street = a.street\}$$

Figure 6.8.: Two scenarios, each with two consistency relations: Consistency relations CR_1 and two options CR_2, CR'_2 with $CR_1 \otimes CR_2 \neq \emptyset$ and $CR_1 \otimes CR'_2 = \emptyset$, and consistency relations CR_3 and CR_4 with $CR_3 \otimes CR_4 = \emptyset$ and $CR_4^T \otimes CR_3^T \neq \emptyset$

CR_2. Otherwise, in both cases the occurrence of $\mathfrak{c}_l$ in a model set would not necessarily impose any consistency requirement by CR_2. In the following, we explain these two requirements at an example.

Example 2. Figure 6.8 extends the initial example (Figure 1.2) with further classes in the consistency relations, such that they do not only relate single classes to each other. It defines an address for employees and in the second example also a location for the address of residents, which are represented in additional classes. Both examples contains a consistency relation CR_1 respectively CR_3 between persons and residents, which define that for each person a resident with the same name has to exist. The examples provide different options for consistency relation between residents (with locations) and employees with addresses (CR_2, CR'_2, CR_4), which exemplify the necessity for the restrictions in Definition 20:

1. $CR_1 \otimes CR_2$: CR_2 requires for each resident an employee with the same name and an address with an arbitrary street name. In consequence, $CR_1 \otimes CR_2$ defines a relation for each person with an employee having the same name and all addresses with possible street names. All models that are consistent to the concatenation are also consistent to the single relations.

2. $CR_1 \otimes CR'_2$: CR'_2 is similar to CR_2 but additionally requires that the street of a resident must not be empty. In consequence, for a resident with an empty address it is not required that an employee exists. This results in $CR_1 \otimes CR'_2 = \emptyset$, because for any person, there must not be an employee, as the person can be consistent to a resident with an empty street name. This shows the necessity of the second restriction in the definition.

3. $CR_3 \otimes CR_4$: The concatenation $CR_3 \otimes CR_4$ is obviously empty, because CR_3 requires a resident for each person, but CR_4 only requires an employee if there is also a location. Such a location does not necessarily exist if a person exists, thus if the models are consistent to CR_3 and CR_4 there must not necessarily be an employee for any contained person. This shows the necessity for the first restriction in Definition 20, which would require a left condition element from CR_4 (resident and location) to be a subset of a right condition element in CR_3 (resident).

4. $CR_4^T \otimes CR_3^T$: The concatenation of the transposed relations $CR_4^T \otimes CR_3^T$ is not empty, but actually contains all combinations of each possible employee with all addresses and relates them to a person with the same name. This is reasonable, because CR_4^T requires for all existing employees and addresses that an appropriate resident with the same name has to exist, which then requires a person with that name to exist due to CR_3^T. The definition does only cover that due to its first restriction, because $\mathfrak{c}_{l,2}$, i.e., the resident related to a person by CR_3^T is a subset of $\mathfrak{c}_{r,1}$, i.e., a tuple of resident and location.

Maybe readd overlapping definition

We can formally show that the defined notion of concatenation does not lead to any restriction of consistency regarding the single relations:

Lemma 2. Let CR_1, CR_2 be two consistency relations and let $CR = CR_1 \otimes CR_2$ be their concatenation. For all model sets $\mathbb{m} \in I_{\mathbb{M}}$ the following statement holds:

$$\mathbb{m} \textit{ consistent to } \{CR_1, CR_2\} \Rightarrow \mathbb{m} \textit{ consistent to } CR$$

Proof. For any set of models $\mathbb{m}$ that is consistent to CR_1 and CR_2, take the witness structure W_1 that witnesses consistency of $\mathbb{m}$ to CR_1 and W_2 that witnesses consistency of $\mathbb{m}$ to CR_2. Now consider the composed witness structure $W = W_1 \otimes W_2$. Let us assume there were $\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle, \langle \mathfrak{c}'_l, \mathfrak{c}'_r \rangle \in W$ with $\mathfrak{c}_l = \mathfrak{c}'_l$ and $\mathfrak{c}_r \neq \mathfrak{c}'_r$. Per definition $\mathfrak{c}_l$ only occurs in one $\langle \mathfrak{c}_l, \mathfrak{c}_{r,1} \rangle \in W_1$. So there must be two $\langle \mathfrak{c}_{l,2}, \mathfrak{c}_r \rangle, \langle \mathfrak{c}'_{l,2}, \mathfrak{c}'_r \rangle \in CR_2$ with $\mathfrak{c}_{l,2} \subseteq \mathfrak{c}_{r,1}$ and $\mathfrak{c}'_{l,2} \subseteq \mathfrak{c}_{r,1}$. However, since $\mathfrak{c}_{l,2}$ and $\mathfrak{c}'_{l,2}$ contain instances of the same classes and are both subsets of the same other object tuples $\mathfrak{c}_{r,1}$, we have $\mathfrak{c}_{l,2} = \mathfrak{c}'_{l,2}$. So we know that:

$$\begin{aligned} &\forall \langle \mathfrak{c}_{l,1}, \mathfrak{c}_{r,1} \rangle, \langle \mathfrak{c}_{l,2}, \mathfrak{c}_{r,2} \rangle \in W : \\ &\quad \langle \mathfrak{c}_{l,1}, \mathfrak{c}_{r,1} \rangle = \langle \mathfrak{c}_{l,2}, \mathfrak{c}_{r,2} \rangle \vee \mathfrak{c}_{l,1} \neq \mathfrak{c}_{l,2} \wedge \mathfrak{c}_{r,1} \neq \mathfrak{c}_{l,2} \end{aligned}$$

Additionally, since W_1 and W_2 are witness structures for consistency of $\mathbb{m}$ to CR_1 and CR_2, the model set contains all condition elements in W_1 and W_2. Consequentially, $\mathbb{m}$ also contains the condition elements in W, as those in W are composed of the ones in W_1 and W_2. This implies that:

$$\forall \langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \in W : \mathbb{m} \textit{ contains } \mathfrak{c}_l \wedge \mathbb{m} \textit{ contains } \mathfrak{c}_r$$

Finally, let us assume that:

$$\exists \, \mathfrak{c}'_l \in \mathbb{c}_{l,CR} : \mathbb{m} \textit{ contains } \mathfrak{c}'_l \wedge \mathfrak{c}'_l \notin \mathbb{c}_{l,W}$$

We know that $\mathbb{c}_{l,CR} \subseteq \mathbb{c}_{l,CR_1}$, because the left condition elements in CR are taken from the left condition elements in CR_1 per definition and thus also contained CR_1. Since $\mathbb{m}$ *contains* $\mathfrak{c}'_l$, there must be a consistency relation pair $\langle \mathfrak{c}'_l, \mathfrak{c}'_{r,1} \rangle \in W_1$, which witnesses consistency of $\mathfrak{c}'_l$ according to CR_1. There must be at least one consistency relation pair $\langle \mathfrak{c}'_{l,2}, \mathfrak{c}'_{r,2} \rangle \in CR_2$ with $\mathfrak{c}'_{l,2} \subseteq \mathfrak{c}'_{r,1}$, because otherwise $\mathfrak{c}'_l$ would per definition not occur in the left condition of CR. For all such tuples $\langle \mathfrak{c}'_{l,2}, \mathfrak{c}'_{r,2} \rangle$, we know that $\mathbb{m}$ *contains* $\mathfrak{c}'_{l,2}$, because $\mathbb{m}$ *contains* $\mathfrak{c}'_{r,1}$ due to its containment in W_1 and due to $\mathfrak{c}'_{l,2} \subseteq \mathfrak{c}'_{r,1}$. In consequence, consistency to CR_2 requires that for one of those $\mathfrak{c}'_{r,2}$ it holds that $\mathbb{m}$ *contains* $\mathfrak{c}'_{r,2}$ and that there is $\langle \mathfrak{c}'_{l,2}, \mathfrak{c}'_{r,2} \rangle \in W_2$ that witnesses this consistency. Summarizing, due to $\langle \mathfrak{c}'_l, \mathfrak{c}'_{r,1} \rangle \in W_1$ and $\langle \mathfrak{c}'_{l,2}, \mathfrak{c}'_{r,2} \rangle \in W_2$ with $\mathfrak{c}'_{l,2} \subseteq \mathfrak{c}'_{r,1}$ and due to the definition of W as the concatenation of W_1 and W_2, we know that $\langle \mathfrak{c}'_l, \mathfrak{c}'_{r,2} \rangle \in W$, which breaks our assumption. So we have shown that:

$$\forall\, \mathfrak{c}'_l \in \mathbb{C}_{l,CR} \mid \mathbb{m} \textit{ contains } \mathfrak{c}'_l : \mathfrak{c}'_l \in \mathbb{C}_{l,W}$$

Summarizing, we have shown that W fulfills all requirements to a witness structure according to Definition 18 for $\mathbb{m}$ being consistent to CR, so we know that $\mathbb{m}$ *consistent to* CR.

□

We can use this notion of concatenation to define a transitive closure for sets of consistency relations, which contains all relations in that set complemented by all possible concatenations of them, i.e., *implicit relations* of that set. Having shown that our definition of consistency relations concatenation is well-defined in the sense that it does not introduce further restrictions for consistency, we are also able to show that the transitive closure does not restrict consistency in comparison to the set of consistency relation itself.

Definition 21 (Transitive Closure of Consistency Relations). Let $\mathbb{CR}$ be a set of consistency relations. We define its transitive closure $\mathbb{CR}^+$ as:

$$\begin{aligned} \mathbb{CR}^+ = \{CR \mid \exists\, CR_1, \ldots, CR_k \in \mathbb{CR} :& \\ CR = CR_1 \otimes \ldots \otimes CR_k\}& \end{aligned}$$

The transitive closure of a set of consistency relations $\mathbb{CR}$ contains all consistency relations of $\mathbb{CR}$ and all concatenations of relations in $\mathbb{CR}$. That means, the transitive closure contains consistency relations that relate all elements that are directly or indirectly related due to $\mathbb{CR}$.

The transitive closure of a consistency relation set does not further restrict consistency in comparison to the original set by construction of concatenation, i.e., if a model set is consistent to a set of consistency relations, it is also consistent to their transitive closure. We show that in the following by first extending the argument of Lemma 2, which shows that concatenation does not further restrict consistency, to the transitive closure, which is only a set of concatenations of consistency relations.

Lemma 3. Let $\mathbb{CR}$ be a set of consistency relations for a set of metamodels $\mathbb{M}$. Then:

$$\begin{aligned}&\forall CR \in \mathbb{CR}^+ \setminus \mathbb{CR} : \exists CR_1, \dots, CR_k \in \mathbb{CR} : \forall \mathbb{m} \in I_{\mathbb{M}} :\\&\quad \mathbb{m} \textit{ consistent to } \{CR_1, \dots CR_k\} \Rightarrow \mathbb{m} \textit{ consistent to } CR\end{aligned}$$

Proof. Per definition, any $CR \in \mathbb{CR}^+$ is a concatenation of consistency relations in $\mathbb{CR}$, i.e.

$$\begin{aligned}&\forall CR \in \mathbb{CR}^+ : \exists CR_1, \dots, CR_k \in \mathbb{CR} :\\&\quad CR = CR_1 \otimes \dots \otimes CR_k\end{aligned}$$

We already know for any two consistency relations CR_1, CR_2 and all model sets $\mathbb{m}$ that if $\mathbb{m}$ *consistent to* $\{CR_1, CR_2\}$, then $\mathbb{m}$ *consistent to* $CR_1 \otimes CR_2$ due to Lemma 2. Inductively applying that argument to $CR_1, \dots, CR_k$ shows that for all models $\mathbb{m}$ with $\mathbb{m}$ *consistent to* $\{CR_1, \dots, CR_k\}$ we know that $\mathbb{m}$ *consistent to* CR. □

As a direct result of the previous lemma, we can now show that the transitive closure of a consistency relation set considers the same sets of models consistent as the consistency relation set itself.

Lemma 4. Let $\mathbb{CR}$ be a set of consistency relations for a set of metamodels $\mathbb{M}$. Then for all sets of models $\mathbb{m} \in I_{\mathbb{M}}$ it is true that:

$$\mathbb{m} \textit{ consistent to } \mathbb{CR} \Leftrightarrow \mathbb{m} \textit{ consistent to } \mathbb{CR}^+$$

Proof. Adding a consistency relation to a set of consistency relations can never lead to a relaxation of consistency, i.e., models becoming consistent that were not considered consistent before. This is a direct consequence of Definition 18 for consistency, which requires models be consistent to all consistency relations in a set to be considered consistent, thus only restricting the set of consistent model sets by adding further consistency relations. In consequence, it holds that:

$$\mathbb{m} \textit{ consistent to } \mathbb{CR}^+ \Rightarrow \mathbb{m} \textit{ consistent to } \mathbb{CR}$$

Due to Lemma 4, we know that a set of models that is consistent to $\mathbb{CR}$ is always consistent to all transitive relations in $\mathbb{CR}^+$ as well. Thus, we know that:

$$\mathbb{m} \textit{ consistent to } \mathbb{CR} \Rightarrow \mathbb{m} \textit{ consistent to } \mathbb{CR}^+$$

In consequence, models are considered consistent equally for $\mathbb{CR}$ and its transitive closure $\mathbb{CR}^+$. □

6.8.3. A Formal Notion of Compatibility

Based on the fine-grained notion of consistency in terms of consistency relations, we can know precisely formulate our initially informal notion of *compatibility* of consistency relations. We stated that we consider consistency relation incompatible if they are somehow contradictory, like the relation between names in our initial example in Figure 1.2. In that example, for residents with non-lowercase names no consistent set of models could be derived. To capture that in a definition, we consider relations compatible if for all condition elements in the consistency relations, i.e., for every tuple of objects for which consistency is somehow constrained by requiring further elements to exist in a set of models to consider it consistent, a consistent model containing those objects can be found. In consequence, a consistency relation is not allowed to prevent objects for which other relations specify consistency from existing in consistent models.

Definition 22 (Compatibility)**.** Let $\mathbb{CR}$ be a set of consistency relations for a set of metamodels $\mathbb{M}$. We say that:

$$\begin{aligned}
&\mathbb{CR} \textit{ compatible } :\Leftrightarrow \\
&\quad \forall CR \in \mathbb{CR} : \forall \mathfrak{c} \in \mathfrak{c}_{l,CR} : \exists \mathfrak{m} \in I_{\mathbb{M}} : \\
&\qquad \mathfrak{m} \textit{ contains } \mathfrak{c} \wedge \mathfrak{m} \textit{ consistent to } \mathbb{CR}
\end{aligned}$$

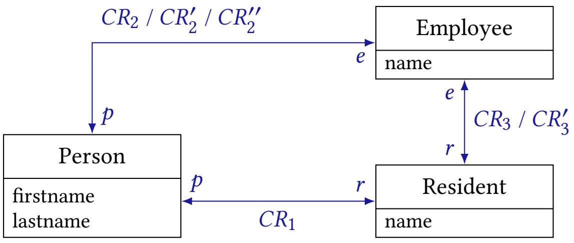
We call a set of consistency relation $\mathbb{CR}$ *incompatible* if it does not fulfill the definition of compatibility.

Definition 22 formalizes the notion of *non-contradictory* relations by requiring that a relation may not restrict that an object tuple, for which consistency is defined in any consistency relation, cannot occur in a model set anymore. We exemplify this notion of compatibility on an extract of the initial example with different consistency relations.

Example 3. Figure 6.9 shows an extract of the three metamodels from Figure 1.2 and several consistency relations, of which different combinations are compatible or incompatible according to the previous definition. We always consider the actual relations together with their transposed ones to have a symmetric set of consistency relations.

$\{CR_1, CR_1^T, CR_2, CR_2^T, CR_3\}$**:** These relations are obviously compatible, because they relate $firstname$ respectively $lastname$ and $name$ in the same way. Thus, for each object with any name, and thus any condition element in all of the consistency relations, a consistent model set can be found by adding instances of the other classes with appropriate names.

$\{CR_1, CR_1^T, CR_2', CR_2'^T, CR_3, CR_3^T\}$**:** These relations are obviously not compatible, because for each person with $firstname = f$ and $lastname = l$, another person with $firstname = f + ","$ and $lastname = l$ has to exist due to CR_2' and the transitive relations requiring the addition of a comma. Thus, each person would require an infinite number of further persons to exist in a consistent set of models. However, models are assumed to be finite, so there is no such model set and the relations are incompatible.



$$CR_1 = \{\langle p, r\rangle \mid r.name = p.firstname + "\textvisiblespace" + p.lastname\}$$

$$CR_2 = \{\langle p, e\rangle \mid e.name = p.firstname + "\textvisiblespace" + p.lastname\}$$
$$CR_2' = \{\langle p, e\rangle \mid e.name = p.firstname + ",\textvisiblespace" + p.lastname\}$$
$$CR_2'' = \{\langle p, e\rangle \mid e.name = p.lastname + "\textvisiblespace" + p.firstname\}$$

$$CR_3 = \{\langle r, e\rangle \mid r.name = e.name\}$$
$$CR_3' = \{\langle r, e\rangle \mid r.name = e.name.toLower\}$$

Figure 6.9.: Three metamodels with different consistency relations. The sets $\{CR_1, CR_1^T, CR_2, CR_2^T, CR_3, CR_3^T\}$ and $\{CR_1, CR_1^T, CR_2'', CR_2''^T, CR_3, CR_3^T\}$ are compatible, whereas the sets $\{CR_1, CR_1^T, CR_2', CR_2'^T, CR_3, CR_3^T\}$ and $\{CR_1, CR_1^T, CR_2, CR_2^T, CR_3', CR_3'^T\}$ are not.

$\{CR_1, CR_1^T, CR_2', CR_2'^T, CR_3, CR_3^T\}$**:** These relations are compatible, although one might not expect that. The relations define that for a resident with $firstname = f$ and $lastname = l$ another resident with $firstname = l$ and $lastname = f$ has to exist, so that the set of models is consistent. Although that behavior may not be intuitive, it does not violate the definition of compatibility, because for any object in the relations, a consistent model can be constructed. In general, such a behavior cannot be forbidden, because comparable behavior might be expected, such as that for a software component an implementation class as well a utility class with different names are created due to different relations, which leads to comparable behavior as in the example. Finally, such a relation would not prevent a consistency repair routine from finding a consistent set of models. So this can be seen as a semantic problem that requires further relation-specific knowledge, as it is necessary to know that a first name should never be mapped to a last name in our example.

$\{CR_1, CR_1^T, CR_2, CR_2^T, CR_3', CR_3'^T\}$: These consistency relations reflect the ones of our motivational example in Figure 1.2. According to the informal notion of incompatibility that we motivated in the introduction with that example, our formal definition of compatibility also considers these relations as incompatible, because it is not possible to create a resident with an uppercase name, such that the containing set of models is consistent. For a resident with $name = "A␣B"$, a person with $firstname = "A"$ and $lastname = "B"$ has to exist, which requires existence of an employee with $name = "A␣B"$. Now CR_3' requires a resident with $name = "a␣b"$ to exist, which in turn requires a resident with $firstname = "a"$ and $lastname = "b"$ and an employee with $name = "a␣b"$ to exist. In consequence, there are two employees, one with the uppercase and one with the lowercase name, for which a resident with the lowercase name has to exist according to the relation CR_3'. So there is no witness structure with a unique mapping between the elements that is required to fulfill Definition 18 for consistency.

To summarize, compatibility is supposed to ensure that consistency relations do not impose restrictions on other relations such that their condition elements, for which consistency is defined, can never occur in consistent models. The goal of ensuring compatibility of consistency relations is especially to prevent consistency repair routines of model transformation from non-termination, as may occur especially in the second scenario, where an infinitely large model would be required to fulfill the consistency relations.

Finally, analogously to the equivalence of a set of consistency relations $\mathbb{CR}$ and its transitive closure $\mathbb{CR}^+$ in regards to consistency of a set of models, we can show that a set of consistency relations and its transitive closure are always equal with regards to compatibility.

Lemma 5. Let $\mathbb{CR}$ be a set of consistency relations for a set of metamodels $\mathbb{M}$. It holds that:

$$\mathbb{CR}\ \mathit{compatible}\ \Leftrightarrow \mathbb{CR}^+\ \mathit{compatible}$$

Proof. The reverse direction of the equivalence is given by definition, since compatibility of a sub of consistency relations implies compatibility of any subset by definition. So we have to show the forward direction by considering the compatibility definition for all $CR \in \mathbb{CR}^+$. We partition $\mathbb{CR}^+$ into $\mathbb{CR}$ and $\mathbb{CR}^+ \setminus \mathbb{CR}$ and consider their consistency relations independently.

First, we consider $CR \in \mathbb{CR}^+ \setminus \mathbb{CR}$. According to Definition 21 for the transitive closure, each $CR \in \mathbb{CR}^+ \setminus \mathbb{CR}$ is a concatenation of consistency relations $CR_1, \ldots, CR_k \in \mathbb{CR}$. In consequence of that definition we know that $\mathbb{c}_{l,CR} \subseteq \mathbb{c}_{l,CR_1}$, so it is given that:

$$\begin{aligned} &\forall\, \mathfrak{c}_l \in \mathbb{c}_{l,CR} : \exists\, \mathfrak{c}'_l \in \mathbb{c}_{l,CR_1} : \forall\, \mathbb{m} \in I_{\mathbb{M}} : \\ &\quad \mathbb{m} \textit{ contains } \mathfrak{c}_l \Rightarrow \mathbb{m} \textit{ contains } \mathfrak{c}'_l \end{aligned} \tag{6.1}$$

Since $\mathbb{CR}$ is compatible, we especially know from Definition 22 for compatibility that:

$$\begin{aligned} &\forall\, \mathfrak{c}'_l \in \mathbb{c}_{l,CR_1} : \exists\, \mathbb{m} \in I_{\mathbb{M}} : \\ &\qquad \mathbb{m} \textit{ contains } \mathfrak{c}'_l \wedge \mathbb{m} \textit{ consistent to } \mathbb{CR} \end{aligned} \tag{6.2}$$

Because of Equation 6.1 and Equation 6.2, we know that:

$$\begin{aligned} &\forall\, \mathfrak{c}_l \in \mathbb{c}_{l,CR} : \exists\, \mathbb{m} \in I_{\mathbb{M}} : \\ &\qquad \mathbb{m} \textit{ contains } \mathfrak{c}_l \wedge \mathbb{m} \textit{ consistent to } \mathbb{CR} \end{aligned} \tag{6.3}$$

Furthermore, Lemma 4 states that for all model sets $\mathbb{m} \in I_{\mathbb{M}}$ it is true that:

$$\mathbb{m} \textit{ consistent to } \mathbb{CR} \Leftrightarrow \mathbb{m} \textit{ consistent to } \mathbb{CR}^+ \tag{6.4}$$

In consequence of equations 6.3 and 6.4, we know that:

$$\begin{aligned} &\forall\, CR \in \mathbb{CR}^+ \setminus \mathbb{CR} : \forall\, \mathfrak{c}' \in \mathbb{c}_{l,CR} : \exists\, \mathbb{m} \in I_{\mathbb{M}} : \\ &\qquad \mathbb{m} \textit{ contains } \mathfrak{c}' \wedge \mathbb{m} \textit{ consistent to } \mathbb{CR}^+ \end{aligned} \tag{6.5}$$

Second, we consider $CR \in \mathbb{CR}$. Due to the definition of compatibility of $\mathbb{CR}$ and Lemma 4 showing equality of consistency of $\mathbb{m}$ regarding $\mathbb{CR}$ and $\mathbb{CR}^+$ it is true that:

$$\begin{aligned}&\forall CR \in \mathbb{CR} : \forall \mathfrak{c}' \in \mathbb{c}_{l,CR} : \exists\, \mathbb{m} \in I_{\mathbb{M}} : \\ &\qquad \mathbb{m} \textit{ contains } \mathfrak{c}' \wedge \mathbb{m} \textit{ consistent to } \mathbb{CR}^+ \end{aligned} \tag{6.6}$$

With Equation 6.5 and Equation 6.6, we have shown compatibility of $\mathbb{CR}^+$ if $\mathbb{CR}$ is compatible. □

6.9. A Formal Approach to Prove Compatibility

In this section, we use the definition of compatibility to derive a formal approach for proving compatibility of consistency relations. The approach bases on two ideas:

1. A set of consistency relations in which each pair of classes is only related across one concatenation of relations is inherently compatible, because there cannot be any contradictory relations. We precisely define this in a specific notion of *consistency relation trees*.
2. A consistency relation that is redundant in a set of relations, i.e., a relation that does not alter the notion of consistency for models regarding the other relations in that set, does not affect compatibility and can thus be removed from that set of relations.

Given a set of consistency relations, compatibility can be proven inductively if a consistency relation tree that is equivalent to the set of relations can be found by only removing redundant relations from that set. Finding such an equivalent consistency relation tree serves as a *witness* for compatibility of a set of relations. In the following, we formalize and prove this inductive approach to check compatibility of a set of consistency relations. This constitutes our contribution **C2**.

The sketched approach for witnessing compatibility is based on a definition of equivalence for sets of consistency relations. We consider two sets of consistency relations equivalent if they consider the same sets of models as consistent:

Definition 23 (Equivalence of Consistency Relations)**.** Let $\mathbb{CR}_1, \mathbb{CR}_2$ be two sets of consistency relations defined for a set of metamodels $\mathbb{M}$. We say that:

$$\begin{aligned}&\mathbb{CR}_1 \textit{ equivalent to } \mathbb{CR}_2 :\Leftrightarrow \forall\, \mathbb{m} \in I_{\mathbb{M}} :\\ &\quad \mathbb{m} \textit{ consistent to } \mathbb{CR}_1 \Leftrightarrow \mathbb{m} \textit{ consistent to } \mathbb{CR}_2\end{aligned}$$

The goal of our approach is to find a set of consistency relations that is compatible and equivalent to a given consistency relation set. We will later use equivalence to introduce a specific notion of redundancy that is compatibility-preserving. In the following, we first consider structures of consistency relation sets that are inherently compatible and afterwards consider redundancy as a means to find an equivalent representation of a relation set that has such a structure.

6.9.1. Compatible Consistency Relation Set Structures

We first consider two essential properties of a consistency relation set that lead to its inherent compatibility:

1. Composability: We show that the union of independent, compatible sets of consistency relations is compatible.
2. Trees: We show that relations fulfilling a special notion of *consistency relation trees* are inherently compatible.

In consequence, we know that a consistency relation set that is composed of independent subsets of consistency relation trees is inherently compatible.

We consider consistency relation sets as independent if there are no transitive consistency relations induced by relations from both sets, i.e., for each object in a model consistency is only restricted by one of those sets.

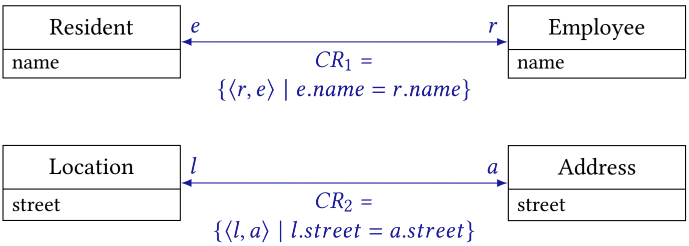

Resident
name
Employee
name
e
r
$CR_1 = \{\langle r, e\rangle \mid e.name = r.name\}$
Location
street
Address
street
l
a
$CR_2 = \{\langle l, a\rangle \mid l.street = a.street\}$

Figure 6.10.: Two independent (sets of) consistency relations

Definition 24 (Independence of Consistency Relation Sets). Let $\mathbb{CR}_1$ and $\mathbb{CR}_2$ be two sets of consistency relations. We say that:

$$\begin{aligned}
&\mathbb{CR}_1 \textit{ and } \mathbb{CR}_2 \textit{ are independent } :\Leftrightarrow\\
&\quad \forall CR \in \mathbb{CR}_1 : \forall CR' \in \mathbb{CR}_2 :\\
&\quad \forall CR_1, \dots, CR_k \in \mathbb{CR}_1 \cup \mathbb{CR}_2 :\\
&\qquad CR \otimes CR_1 \otimes \dots \otimes CR_k \otimes CR' = \emptyset\\
&\qquad \wedge CR' \otimes CR_1 \otimes \dots \otimes CR_k \otimes CR = \emptyset
\end{aligned}$$

We call $\mathbb{CR}$ *connected* if there is no partition of a consistency relation set $\mathbb{CR}$ into two subsets that are independent, i.e.

$$\begin{aligned}
&\forall \mathbb{CR}_1, \mathbb{CR}_2 \subseteq \mathbb{CR} :\\
&\quad \mathbb{CR}_1 \cap \mathbb{CR}_2 = \emptyset \wedge \mathbb{CR}_1 \cup \mathbb{CR}_2 = \mathbb{CR}\\
&\qquad \Rightarrow \neg(\mathbb{CR}_1 \textit{ and } \mathbb{CR}_2 \textit{ are independent}),
\end{aligned}$$

Example 4. Figure 6.10 depicts a simple example with two consistency relations CR_1 and CR_2, each relating instances of two disjoint classes with each other. Since there is no overlap in the objects that are related by the consistency relations, they are considered independent according to Definition 24.

An important property of independent sets of consistency relations is that computing their union is compatibility-preserving, i.e., the union of compatible, independent consistency relation sets is compatible as well:

Theorem 6. Let $\mathbb{CR}_1$ and $\mathbb{CR}_2$ be two compatible sets of consistency relations. Then $\mathbb{CR}_1 \cup \mathbb{CR}_2$ is compatible.

Revise proof with explicit references to independence definition

Proof. Since $\mathbb{CR}_1$ is compatible, per definition there is a model set $\mathfrak{m}$ for each condition element $\mathfrak{c}$ of the left condition of each consistency relation in $\mathbb{CR}_1$ that contains $\mathfrak{c}$ and that is consistent to $\mathbb{CR}_1$. Taking such an $\mathfrak{m}$, we create a new $\mathfrak{m}'$ by removing all elements from $\mathfrak{m}$, which are contained in any condition elements in any consistency relation in $\mathbb{CR}_2$ and thus potentially require other elements to occur to be considered consistent to that consistency relation. In consequence, $\mathfrak{m}'$ does not contain any condition elements from consistency relations in $\mathbb{CR}_2$ and is thus consistent to $\mathbb{CR}_2$ by definition. Additionally, $\mathfrak{m}'$ is still consistent to $\mathbb{CR}_1$, because due to the independence of $\mathbb{CR}_1$ and $\mathbb{CR}_2$, there cannot be any consistency relations in $\mathbb{CR}_1$, which require the existence of the removed elements. In consequence, for each condition element $\mathfrak{c}$ of each consistency relation in $\mathbb{CR}_1$ there is a model set that contains $\mathfrak{c}$ and that is consistent to $\mathbb{CR}_1 \cup \mathbb{CR}_2$. The analogous argumentation applies to the consistency relations in $\mathbb{CR}_2$, which is why the definition of compatibility is fulfilled for all condition elements of all consistency relations in $\mathbb{CR}_1 \cup \mathbb{CR}_2$. □

The constructive proof can also be reflected exemplarily in Figure 6.10: Take any set of models that, for example, contains a resident with an arbitrary name and is consistent to CR_1, i.e., that also contains an employee with the same name. If that set of models contains any addresses or locations, they can be removed without violating consistency to CR_1, because addresses and locations are independently related by CR_2.

Actually, addresses may not be removable because they are referenced by persons. However, if such a reference always needs to be set, this is a restriction that we do not reflect yet in the metamodel formalism. On the other hand, if the reference was relevant for consistency in the first consistency relation, it would have to be considered there as well, thus address would be part of that consistency relation.

Maybe we can remove symmetry and define some restriction for inverse relations. It could be useful to think about implicit relations, which are induced by another one, so that the "signatures" of forward and backward direction match.

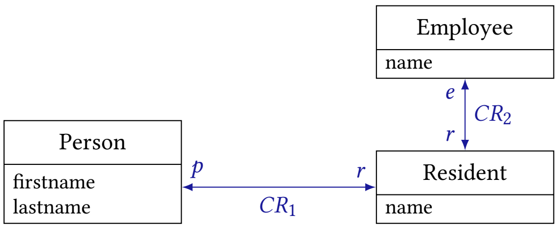
Definition 25 (Consistency Relation Tree)**.** Let $\mathbb{CR}$ be a symmetric, connected set of consistency relations. We say:

$$\begin{aligned}
&\mathbb{CR} \textit{ is a consistency relation tree } :\Leftrightarrow \\
&\quad \forall CR = CR_1 \otimes \ldots \otimes CR_m \in \mathbb{CR}^+ : \\
&\quad \forall CR' = CR'_1 \otimes \ldots \otimes CR'_n \in \mathbb{CR}^+ \setminus CR : \\
&\qquad \forall s, t \mid s \neq t : CR_s \neq CR_t^T \wedge CR'_s \neq CR_t'^T \\
&\qquad \Rightarrow \mathfrak{C}_{l,CR} \cap \mathfrak{C}_{l,CR'} = \emptyset \vee \mathfrak{C}_{r,CR} \cap \mathfrak{C}_{r,CR'} = \emptyset
\end{aligned}$$

We have to assume, that no element is mapped to two elements of the same class, because then it would be possible to have an incompatible network

The definition of a consistency relation tree requires that there are no sequences of consistency relations that put the same classes into relation, i.e. between all pairs of classes there is only one concatenation of consistency relations that puts them into relation. Since we assume a symmetric set of consistency relations, we exclude the symmetric relations from that argument, as otherwise there would always be two such concatenations by adding a consistency relation and its transposed relation to any other concatenation.

Example 5. Figure 6.11 depicts a rather simple consistency relation tree. Persons are related to residents and residents are related to employees, all having the same names respectively a concatenation of $firstname$ and $lastname$, by the relations CR_1, CR_2, as well as their transposed relations CR_1^T, CR_2^T. There are no classes that are put into relation across different paths of consistency relations, thus the definition for a consistency relation tree is fulfilled. If an additional relation between persons and employees was specified, like in Figure 1.2, the tree definition would not be fulfilled.



$$CR_1 = \{\langle p, r\rangle \mid r.name = p.firstname + \text{"␣"} + p.lastname\}$$
$$CR_2 = \{\langle r, e\rangle \mid r.name = e.name\}$$

Figure 6.11.: A consistency relation tree $\{CR_1, CR_1^T, CR_2, CR_2^T\}$

The definition also covers the more complicated case in which multiple classes may be put into relation by consistency relations but there is only a subset of them that is put into relation by different consistency relations.

Subsection with discussion about why hypertrees are not suitable here

We can now prove that a set of consistency relations that is a consistency relation tree is always compatible. We first present a lemma that shows that in a consistency relation tree you can always find an order of the relations such that the classes at the right side of a relation do not overlap with the classes at the left side of a relation that preceded in the order, i.e. there is no cycle in the relations between classes.

Lemma 7. Let $\mathbb{CR} = \{CR_1, CR_1^T, \dots, CR_k, CR_k^T\}$ be a symmetric, connected set of consistency relations. $\mathbb{CR}$ is a consistency relation tree if and only if for each CR there exists a sequence of consistency relations $\langle CR'_1, \dots, CR'_k\rangle$ with $CR'_1 = CR$, containing for each i either CR_i or CR_i^T, i.e.,

$$\begin{aligned}
&\forall\, i \in \{1, \dots, k\} :\\
&\quad \big(CR_i \in \langle CR'_1, \dots, CR'_k\rangle \wedge CR_i^T \notin \langle CR'_1, \dots, CR'_k\rangle\big)\\
&\quad \vee \big(CR_i^T \in \langle CR'_1, \dots, CR'_k\rangle \wedge CR_i \notin \langle CR'_1, \dots, CR'_k\rangle\big)
\end{aligned}$$

such that:

$$\forall s \in \{1, \dots, k-1\} : \forall t \in \{i+1, \dots, k\} :$$
$$\mathfrak{C}_{r,CR'_s} \cap \mathfrak{C}_{r,CR'_t} = \emptyset \wedge \mathfrak{C}_{l,CR'_s} \cap \mathfrak{C}_{r,CR'_t} = \emptyset$$

Proof. We start with the forward direction, i.e., given a consistency relation tree $\mathbb{CR}$ we show that there exists a sequence according to the requirements in Lemma 7 by constructing such a sequence $\langle CR'_1, \dots, CR'_k \rangle$ for any $CR \in \mathbb{CR}$. Start with $CR'_1 = CR$ for any $CR \in \mathbb{CR}$. We now inductively add further relations to that sequence. Take any consistency relation $CR_s = CR_{s,1} \otimes \dots \otimes CR_{s,m} \in \mathbb{CR}^+$ with $\mathfrak{C}_{l,CR_{s,1}} \subseteq \mathfrak{C}_{r,CR}$. Such a sequence must exist because of CR being connected. Now add all $CR_{s,1}, \dots, CR_{s,m}$ to the sequence, which fulfills both requirements to that sequence in Lemma 7 by definition. The following addition of further consistency relations can be inductively applied. Take any other consistency relation $CR_t = CR_{t,1} \otimes \dots \otimes CR_{t,n} \in \mathbb{CR}^+$ such that:

$$\exists CR' \in \{CR, CR_{s,2}, \dots, CR_{s,m}\} : \mathfrak{C}_{l,CR_{t,1}} \subseteq \mathfrak{C}_{r,CR'}$$
$$\wedge\, CR_{t,1}, CR^T_{t,1} \notin \{CR, CR_{s,2}, \dots, CR_{s,m}\}$$

In other words, take any concatenation in the transitive closure of $\mathbb{CR}$ that starts with a relation with a left class tuple that is contained in a right class tuple of a relation already added to the sequence. Again, such a sequence must exist because of $\mathbb{CR}$ being connected and, again, add all $CR_{t,1}, \dots, CR_{t,n}$ to the sequence. Per construction, for each CR' in the sequence, there is a non-empty concatenation of relations within the sequence $CR \otimes \dots \otimes CR'$, because relations were added in a way that such a concatenation always exists. Since all relations in the sequence are contained in $\mathbb{CR}$, such a concatenation was also contained in $\mathbb{CR}^+$. First, we show that the sequence still contains no duplicate elements (1.), i.e., that none of the $CR_{t,i}$ or $CR^T_{t,i}$ is already contained in the sequence $\langle CR, CR_{s,1}, \dots, CR_{s,m} \rangle$. Second, we show that both further conditions for the sequence defined in Lemma 7 are still fulfilled for the sequence $\langle CR, CR_{s,1}, \dots, CR_{s,m}, CR_{t,1}, \dots, CR_{t,n} \rangle$ (2. ,3.).

1. Let us assume that the sequence $\langle CR, CR_{s,1}, \dots, CR_{s,m} \rangle$ already contained one of the $CR_{t,i}$ or $CR^T_{t,i}$. If $CR_{t,i}$ is contained in the sequence, there is a concatenation $CR \otimes \dots \otimes CR_{t,i}$ with relations in $\langle CR, CR_{s,1}, \dots, CR_{s,m} \rangle$, as well as a concatenation

$CR \otimes \ldots \otimes CR_{t,1} \otimes \ldots \otimes CR_{t,i}$. Since $CR_{t,1} \notin \{CR, CR_{s,2}, \ldots, CR_{s,m}\}$ by construction, these two concatenations relate the same class tuples, i.e., they contradict the definition of a consistency relation tree. If $CR^T_{t,i}$ was contained in the sequence $\langle CR, CR_{s,2} \otimes \ldots \otimes CR_{s,m} \rangle$, there is a concatenation $CR \otimes \ldots \otimes CR_w \otimes CR^T_{t,i}$ with relations in $\langle CR, CR_{s,1}, \ldots, CR_{s,m} \rangle$ and, like before, the concatenation $CR \otimes \ldots \otimes CR_{t,1}, \ldots, CR_{t,i}$. Due to $\mathfrak{C}_{r,CR_w} \cap \mathfrak{C}_{l,CR_{t,i}} \neq \emptyset$ and $CR^T_{t,1} \notin \{CR, CR_{s,2}, \ldots, CR_{s,m}\}$ by construction, the two concatenations $CR \otimes \ldots \otimes CR_w$ and $CR \otimes \ldots \otimes CR_{t,1} \otimes \ldots \otimes CR_{t,i}$ have an overlap in both their left and right class tuples, i.e., they contradict the definition of a consistency relation tree. In consequence, the sequence $\langle CR, CR_{s,1}, \ldots, CR_{s,m} \rangle$ cannot have contained any $CR_{t,i}$ or $CR^T_{t,i}$ before.

2. Let us assume there were any CR'_u and CR'_v in the sequence $\langle CR, CR_{s,1}, \ldots, CR_{s,m}, CR_{t,1}, \ldots, CR_{t,n} \rangle$ such that $\mathfrak{C}_{r,CR'_u} \cap \mathfrak{C}_{r,CR'_v} \neq \emptyset$. As discussed before, for each of these relations exists a concatenation of relations in the sequence $CR \otimes \ldots \otimes CR'_u$ and $CR \otimes \ldots \otimes CR'_v$, which is contained in $\mathbb{CR}^+$. This contradicts the definition of a consistency relation tree, so there cannot be two such relations with overlapping classes in the right class tuple.
3. Let us assume there were any CR'_u and CR'_v $(u < v)$ in the sequence $\langle CR, CR_{s,1}, \ldots, CR_{s,m}, CR_{t,1}, \ldots, CR_{t,n} \rangle$ such that $\mathfrak{C}_{l,CR'_u} \cap \mathfrak{C}_{r,CR'_v} \neq \emptyset$. Again per construction, there must be a non-empty concatenation $CR \otimes \ldots \otimes CR'_w \otimes CR'_u$ with $w < u$. Since $\mathfrak{C}_{l,CR'_u} \subseteq \mathfrak{C}_{r,CR'_w}$ per definition, it holds that $\mathfrak{C}_{r,CR'_w} \cap \mathfrak{C}_{r,CR'_v} \neq \emptyset$. In other words, the relation CR'_v introduces a cycle in the relations. We have already shown in (2.) that this contradicts the definition of a consistency relation tree.

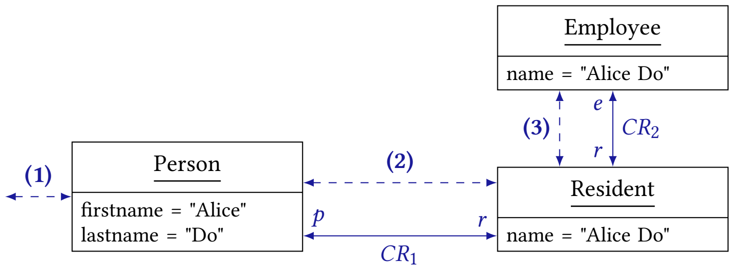
The previous strategy for adding relations to the sequence can be continued inductively by adding relations of the transitive closure of $\mathbb{CR}$ if their relations were not already added to the sequence. This process can be continued until finally all relations in $\mathbb{CR}$ are added to the sequence. Inductively applying the same arguments as before, the final sequence still fulfills all requirements for the sequence in Lemma 7.

We proceed with the reverse direction, i.e., given that a sequence according to the requirements in Lemma 7 exists for all $CR \in \mathbb{CR}$, we show that the set of consistency relations fulfills the definition of a consistency relation tree. Let us assume that the tree definition was not fulfilled, i.e., that there were two consistency relations $CR_s = CR_{s,1} \otimes \ldots \otimes CR_{s,m} \in \mathbb{CR}^+$ and $CR_t = CR_{t,1} \otimes \ldots \otimes CR_{t,n} \in \mathbb{CR}^+$ such that $\mathfrak{C}_{l,CR_s} \cap \mathfrak{C}_{l,CR_t} \neq \emptyset$ and $\mathfrak{C}_{r,CR_s} \cap \mathfrak{C}_{r,CR_t} \neq \emptyset$. Without loss of generality, we assume that $CR_{s,m} \neq CR_{t,n}$, because otherwise we could instead consider the sequence without those last relations and still fulfill the defined requirements. Any sequence according to Lemma 7 containing both $CR_{s,m}$ and $CR_{t,n}$ would contradict the assumption, because $\mathfrak{C}_{r,CR_{s,m}} \cap \mathfrak{C}_{r,CR_{t,n}} \neq \emptyset$ in contradiction to the assumptions regarding the sequence. Thus, the sequence has to contain either $CR^T_{s,m}$ or $CR^T_{t,n}$. Let us assume that the sequence contains $CR^T_{s,m}$. Then the sequence cannot contain $CR_{s,m-1}$, because $\mathfrak{C}_{r,CR^T_{s,m}} \cap \mathfrak{C}_{r,CR_{s,m-1}} \neq \emptyset$, which, again, would contradict the assumptions regarding the sequence. This argument can be inductively applied to all $CR_{s,i}$, such that the sequence has to contain all $CR^T_{s,i}$. Since the sequence contains $CR^T_{s,1}$, it must contain $CR_{t,1}$, because $\mathfrak{C}_{r,CR^T_{s,1}} \cap \mathfrak{C}_{r,CR^T_{t,1}} \neq \emptyset$. In consequence of $CR_{t,1}$ being contained in the sequence, all $CR_{t,i}$ have to be contained as well, due to the same reasons as before. So we have these conditions, which introduce a cycle in the overlaps of the class tuples of the relations within the sequence:

$$\begin{aligned}
&\mathfrak{C}_{l,CR^T_{s,i-1}} \cap \mathfrak{C}_{r,CR^T_{s,i}} \neq \emptyset \wedge \mathfrak{C}_{l,CR_{t,1}} \cap \mathfrak{C}_{r,CR^T_{s,1}} \neq \emptyset \\
&\wedge \mathfrak{C}_{l,CR_{t,i}} \cap \mathfrak{C}_{r,CR_{t,i-1}} \neq \emptyset \wedge \mathfrak{C}_{l,CR^T_{s,m}} \cap \mathfrak{C}_{r,CR_{t,n}} \neq \emptyset
\end{aligned}$$

Because of that cycle in the overlap of class tuples, there is no order of these relations $CR''_1, \ldots, CR''_{m+n}$ such that for all of them it holds that $\mathfrak{C}_{l,CR''_u} \cap \mathfrak{C}_{r,CR''_v} \neq \emptyset$ $(u < v)$, which contradicts the assumptions regarding the sequence in Lemma 7. The analog argument holds when we assume that the sequence contains $CR^T_{t,n}$ instead of $CR^T_{s,m}$. In consequence, there cannot be two such concatenations CR_s and CR_t without breaking the assumptions for the sequence in Lemma 7. □

The previous lemma shows that the definition of consistency relation trees based on unique concatenations of the same class tuples is equivalent the possibility to find sequences of the relations that do not contain cycles in the related class tuples. The definition is supposed to be easier to check in practice. However, we can now show that a consistency relation tree is



$$CR_1 = \{\langle p, r\rangle \mid r.name = p.firstname + \text{"␣"} + p.lastname\}$$

$$CR_2 = \{\langle r, e\rangle \mid r.name = e.name\}$$

Figure 6.12.: An example for constructing a model with the condition element of CR_1 containing the person named "Alice Do" for a consistency relation tree according to the consistency relations in Figure 6.11.

always compatible with a constructive proof that requires the equivalent definition from Lemma 7.

Theorem 8. Let $\mathbb{CR}$ be a consistency relation tree, then $\mathbb{CR}$ is compatible.

Proof. We prove the statement by constructing a set of models for each condition element in the left condition of each consistency relation that contains the condition element and is consistent, i.e., that fulfills the compatibility definition. The basic idea is that because $\mathbb{CR}$ is a consistency relation tree, we can simply add necessary elements to get a model set that is consistent to all consistency relations, by following an order of relations according to Lemma 7. Thus, we explain an induction for constructing such a model set, which is also exemplified for a simple scenario in Figure 6.12, based on the relations in the consistency relation tree in Figure 6.11.

Base case: Take any $CR \in \mathbb{CR}$ and any of its left side condition elements $\mathfrak{c}_l = \langle o_{l,1}, \dots, o_{l,m}\rangle \in \mathbb{c}_{l,CR}$. Select any $\mathfrak{c}_r = \langle o_{r,1}, \dots, o_{r,n}\rangle \in \mathbb{c}_{r,CR}$, such that $\mathfrak{c}_l$ and $\mathfrak{c}_r$ constitute a consistency relation pair $\langle \mathfrak{c}_l, \mathfrak{c}_{r,}\rangle \in CR$. Now construct the model set $\mathbb{m}$ that contains only $o_{l,1}, \dots, o_{l,m}$ and $o_{r,1}, \dots, o_{r,n}$. In consequence, we have a minimal model set $\mathbb{m}$, such that $\mathbb{m}$ *contains* $\mathfrak{c}_l$ and $\mathbb{m}$ *consistent to* CR. Additionally, $\mathbb{m}$ is consistent to CR^T due to symmetry of CR and CR^T: It is $\mathfrak{c}_r \in \mathbb{c}_{l,CR^T}$ and $\langle \mathfrak{c}_r, \mathfrak{c}_l \rangle \in CR^T$ and no other condition element of $\mathbb{c}_{l,CR^T}$ is contained in $\mathbb{m}$ by construction, thus $\mathbb{m}$ is consistent to CR^T. In consequence, we know that for all $CR \in \mathbb{CR}$, $\{CR, CR^T\}$ is compatible. Considering the example in *Figure* 6.12, for the selection of any person as a condition element in $\mathbb{c}_{l,CR_1}$ (1), we select a resident in $\mathbb{c}_{r,CR_1}$ with the same name (2), such that the elements are consistent to CR_1.

Induction assumption: According to Lemma 7, there is a sequence $\langle CR_1, \ldots, CR_k \rangle$ of the relations in $\mathbb{CR}$ with $CR_1 = CR$, such that:

$$\forall s \in \{1, \ldots, k-1\} : \forall t \in \{i+1, \ldots, k\} :$$
$$\mathfrak{C}_{r,CR'_s} \cap \mathfrak{C}_{r,CR'_t} = \emptyset \land \mathfrak{C}_{l,CR'_s} \cap \mathfrak{C}_{r,CR'_t} = \emptyset$$

Considering the example in Figure 6.12, such a sequence would be $\langle CR_1, CR_2 \rangle$, because the elements in the right condition of CR_2 are not represented in the left condition of CR_1. If, in general, we know that $\{CR_1, CR_1^T \ldots, CR_i, CR_i^T\}$ $(i < k)$ is compatible, for every $\mathfrak{c}_l \in \mathbb{C}_{l,CR}$, we can find a model set $\mathbb{m}$ that contains $\mathfrak{c}_l$ and is consistent to $\{CR_1, CR_1^T, \ldots, CR_i, CR_i^T\}$ by definition. We can especially create a minimal model according to our construction for the base case and the following inductive completion.

Induction step: Consider CR_{i+1}. There is at most one condition element $\mathfrak{c}_l \in \mathbb{c}_{l,CR_{i+1}}$ with $\mathbb{m}$ *contains* $\mathfrak{c}_l$. If there were at least two condition elements $\mathfrak{c}_l, \mathfrak{c}'_l \in \mathbb{c}_{l,CR_{i+1}}$, both contained in $\mathbb{m}$, then by construction there is a consistency relation CR_s $(s < i+1)$ with $\mathfrak{c}_l, \mathfrak{c}'_l \in \mathbb{c}_{r,CR_j}$. Let us assume there were two consistency relations CR_s, CR_t, each containing one of the condition elements in the right condition, then there would be non-empty concatenations $CR \otimes \ldots \otimes CR_s$ and $CR' \otimes \ldots \otimes CR_t$ with $\mathfrak{C}_{l,CR} \cap \mathfrak{C}_{l,CR'} \neq \emptyset$, because we started the construction with elements from the left condition of CR, so every element is contained because of a relation to those elements, and with $\mathfrak{C}_{r,CR_s} \cap \mathfrak{C}_{r,CR_t} \neq \emptyset$, because both condition elements $\mathfrak{c}_l$ and $\mathfrak{c}'_l$ instantiate the same classes, as they are both contained in $\mathbb{c}_{l,CR_{i+1}}$. This would violate Definition 25 for a consistency relation tree, thus there is only one such consistency relation CR_s. Consequently, there must be two condition elements $\mathfrak{c}_{ll}, \mathfrak{c}'_{ll} \in \mathbb{C}_{l,CR_s}$ with $\langle \mathfrak{c}_{ll}, \mathfrak{c}_l \rangle, \langle \mathfrak{c}'_{ll}, \mathfrak{c}'_l \rangle \in CR_s$, because per construction $\mathbb{m}$ was consistent to CR_s and so there must be a witness structure with a unique mapping between condition elements contained in $\mathbb{m}$. The above argument can be applied inductively until we finally find that there must be two condition elements $\mathfrak{c}_{lll}, \mathfrak{c}'_{lll} \in \mathbb{C}_{l,CR}$, which are contained in $\mathbb{m}$. This is not true by construction, as we started with only one element from $\mathbb{C}_{l,CR}$, so there is only one such condition element $\mathfrak{c}_l \in \mathbb{C}_{l,CR_{i+1}}$ with $\mathbb{m}$ *contains* $\mathfrak{c}_l$.

For this condition element $\mathfrak{c}_l \in \mathbb{C}_{l,CR_{i+1}}$, select an arbitrary $\mathfrak{c}_r = \langle o_1, \ldots, o_s \rangle \in \mathbb{C}_{r,CR_{i+1}}$, such that $\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \in CR_{i+1}$. Now create a model set $\mathbb{m}'$ by adding the objects $o_1, \ldots, o_s$ to $\mathbb{m}$. Since $\mathfrak{c}_l$ is the only of the left condition elements of CR_{i+1} that $\mathbb{m}$ contains, model set $\mathbb{m}'$ is consistent to CR_{i+1} per construction. $\mathbb{m}'$ is also consistent to CR^T_{i+1}, because due to the symmetry of CR_{i+1} and CR^T_{i+1}, it is $\mathfrak{c}_r \in \mathbb{C}_{l,CR^T_{i+1}}$ and due to $\langle \mathfrak{c}_r, \mathfrak{c}_l \rangle \in CR^T_{i+1}$, a consistent corresponding element exists in $\mathbb{m}'$. Furthermore, there cannot be any other $\mathfrak{c}' \in \mathbb{C}_{l,CR^T_{i+1}}$ with $\mathbb{m}'$ *contains* $\mathfrak{c}'$, because otherwise there would have been another consistency relation CR' that required the creation of $\mathfrak{c}'$, which means that there are two concatenations of consistency relations $CR \otimes \ldots \otimes CR'$ and $CR \otimes \ldots \otimes CR_{i+1}$ that both relate instances of the same classes, which contradicts Definition 25 for a consistency relation tree.

Additionally, due to Lemma 7, for all CR_s $(s < i+1)$, we know that $\mathfrak{C}_{l,CR_s} \cap \mathfrak{C}_{r,CR_{i+1}} = \emptyset$. Since the newly added elements $\mathfrak{c}_r$ are part of $\mathbb{C}_{r,CR_{i+1}}$, these elements cannot match the left conditions of any of the consistency relations CR_s $(s < i+1)$. So $\mathbb{m}'$ is still consistent to all CR_s $(s < i+1)$. Finally, due to Lemma 7, for all CR_s $(s < i+1)$, we know that $\mathfrak{C}_{r,CR_s} \cap \mathfrak{C}_{r,CR_{i+1}} = \emptyset$. Again, since the newly added elements $\mathfrak{c}_r$ are part of $\mathbb{C}_{r,CR_{i+1}}$, these elements cannot match the left conditions of any of the consistency relations CR^T_s $(s < i+1)$. So $\mathbb{m}'$ is still consistent to all CR^T_s $(s < i+1)$. In consequence, we know that $\mathbb{m}'$ *consistent to* $\{CR_1, CR^T_1 \ldots, CR_{i+1}, CR^T_{i+1}\}$.

Considering the example in Figure 6.12, we would select CR_2 and add for the resident, which is in the left condition elements of CR_2, an appropriate employee to make the model set consistent to CR_2 (3).

Conclusion Taking the base case for CR and the induction step for CR_{i+1}, we have inductively shown that

$$\mathbb{m}' \textit{ consistent to } \{CR_1, CR_1^T \ldots, CR_k, CR_k^T\} = \mathbb{CR}$$

Since the construction is valid for each condition element in every consistency relation in $\mathbb{CR}$, we know that a consistency relation tree $\mathbb{CR}$ is compatible.

□

Summarizing, Theorem 6 and Theorem 8 have shown that consistency relation sets fulfilling a special notion of trees are compatible and that combining compatible independent sets of relations is compatibility-preserving. In consequence, having a consistency relation set that consists of independent subsets that are consistency relation trees, this set of relations is inherently compatible. An approach that evaluates whether a given set of consistency relations fulfills Definition 24 and Definition 25 for independence and trees can be used to prove compatibility of those relations.

However, consistency relations fulfill such a structure only in specific cases. In general, like in our motivational example in Figure 1.2, there may be different consistency relations putting the same elements into relation, such that the definition for consistency relation trees is not fulfilled. In the following, we discuss how to find a consistency relation tree that is equivalent to a given set of consistency relations, such that this equivalence witnesses compatibility.

6.9.2. Redundancy as Witness for Compatibility

Add a definition for a *compatibility-preserving consistency relation* that states that is preserved compatibility and use that for clarifying the following lemmas and theorems.

We have introduced specific structures of consistency relations that are inherently compatible. If a given set of consistency relations does not represent one of those structures, especially because there are multiple consistency relations putting the same classes into relation, it is unclear whether such a set is compatible.

In the following, we present an approach to reduce a set of consistency relations to a structure of independent consistency relation trees. The essential idea is to find relations within the set, which do not change compatibility of the consistency relation set whether or not they are contained in it. An approach that finds such relations and—virtually—removes them from the set until the remaining relations form a set of independent consistency relation trees, proves compatibility of the given set of relations. We first define the term of a *compatibility-preserving* relation.

Definition 26. Let $\mathbb{CR}$ be a compatible set of consistency relations and let CR be a consistency relation. We say that:

$$\begin{aligned}&CR \textit{ compatibility-preserving to } \mathbb{CR} :\Leftrightarrow\\&\quad \mathbb{CR} \cup \{CR\} \textit{ compatible}\end{aligned}$$

To be able to find such a compatibility-preserving relation, we introduce the notion of *redundant* relations and prove the property of being compatibility preserving. Informally speaking, a relation is redundant if it is expressed transitively across others, i.e., if it does not restrict or relax consistency compared to a combination of other relations. We precisely specify a notion of redundancy in the following.

Definition 27 (Redundant Consistency Relation)**.** Let $\mathbb{CR}$ be a set of consistency relations for a set of metamodels $\mathbb{M}$. For a consistency relation $CR \in \mathbb{CR}$, we say that:

$$\begin{aligned}&CR \textit{ redundant in } \mathbb{CR} :\Leftrightarrow\\&\quad \exists\, CR' \in (\mathbb{CR} \setminus \{CR\})^+ : \forall\, \mathfrak{m} \in I_{\mathbb{M}} :\\&\qquad \mathfrak{m} \textit{ consistent to } CR' \Rightarrow \mathfrak{m} \textit{ consistent to } CR\end{aligned}$$

Add examples for redundancy! How do the elements of the redundant relation have to be related to the ones in CR'?

Can we define an even more general notion of redundancy, not stating about the relation to a single consistency relation but the set of consistency relation, abstracting the implication to consistency to the whole set of relations?

The definition of redundancy of a consistency relation CR ensures that there is another consistency relation, possibly transitively expressed across others, such that if a model is consistent to that other relation, it is also consistent to CR. This means that there are no model sets that are considered inconsistent to CR, but not to another relation, thus CR does not restrict consistency. Actually, the definition of redundancy implies that the set of consistency relations with and without the redundant one are equivalent according to Definition 23, thus both consider the same model sets as consistent.

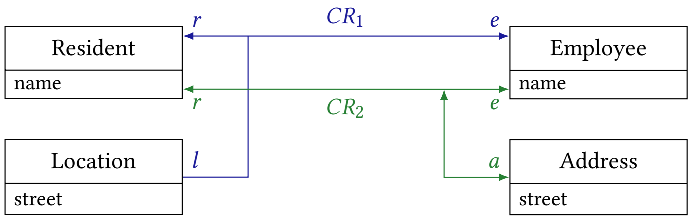
Explain that we do not require equality of elements in CR and CR' because, e.g., CR might only related names, whereas CR' related names and addresses, thus we only require that there are elements that are co-indicating consistency.

Lemma 9. Let $CR \in \mathbb{CR}$ be a redundant consistency relation in a relation set $\mathbb{CR}$. Then $\mathbb{CR}$ is equivalent to $\mathbb{CR} \setminus \{CR\}$.

Proof. Like discussed in Lemma 4, adding a consistency relation to a set of consistency relations can never lead to a relaxation of consistency, i.e., models becoming consistent that were not considered consistent before. This is a direct consequence of Definition 18 for consistency, which requires models be consistent to all consistency relations in a set to be considered consistent, thus restricting the set of consistent model sets by adding further consistency relations. In consequence, it holds that:

$$\mathbb{m} \textit{ consistent to } \mathbb{CR} \Rightarrow \mathbb{m} \textit{ consistent to } \mathbb{CR} \setminus \{CR\}$$

Additionally, a direct consequence of Definition 27 for redundancy is that a redundant consistency relation does not restrict consistency, as it considers all models to be consistent that are also considered consistent to another consistency relation in the transitive closure of the consistency relation set.



$$CR_1 = \{\langle (r, l), e\rangle \mid r.name \neq \text{""} \wedge (r.name = e.name \vee r.name = e.name.toLower)\}$$

$$CR_2 = \{\langle r, (e, a)\rangle \mid r.name = e.name \wedge a.street \neq \text{""}\}$$

Figure 6.13.: Redundant consistency relation CR_1 in $\{CR_1, CR_2\}$

Thus, all models that are considered consistent to the transitive closure of $\mathbb{CR} \setminus \{CR\}$ are also consistent to CR and thus to all relations in $\mathbb{CR}$:

$$\mathbb{m} \textit{ consistent to } (\mathbb{CR} \setminus \{CR\})^+ \Rightarrow \mathbb{m} \textit{ consistent to } \mathbb{CR}$$

According to Lemma 4, each set of models that is consistent to a consistency relation set is also consistent to its transitive closure an vice versa. In consequence, the previous implication is also true for $\mathbb{CR} \setminus \{CR\}$ rather than $(\mathbb{CR} \setminus \{CR\})^+$. Summarizing, $\mathbb{CR}$ and $\mathbb{CR} \setminus \{CR\}$ are equivalent. □

Possibly add that lemma

In general, to consider a consistency relation redundant in $\mathbb{CR}$, it has to define equal or weaker requirements for consistency than one of the other relations in $\mathbb{CR}$. Informally speaking, such weaker requirements mean that the redundant relation must have weaker conditions, i.e., it must require consistency for less objects and consider the same or more objects consistent to each of the left condition elements.

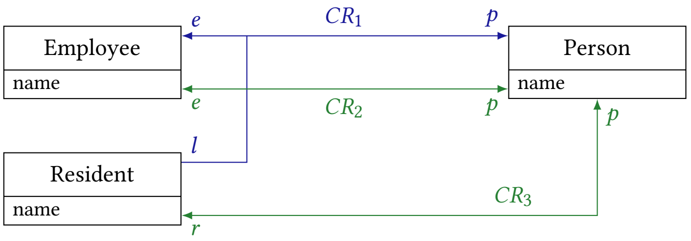
Example 6. Such weaker consistency requirements are exemplified in Figure 6.13, which shows a consistency relation CR_1 that is redundant in $\{CR_1, CR_2\}$.

A redundant consistency relation, such as CR_1, must have weaker requirements in the left condition, such that it requires consistent elements to exist in less cases. This means that it may have a larger set of classes that are matched and that there may be less condition elements for which consistency is required. In case of CR_1, the left condition contains both a resident and a location, whereas the left condition of CR_2 only contains residents. Thus CR_1 requires consistent elements, i.e., employees, only if a resident and a location exists, whereas CR_2 requires that already for an existing resident. Furthermore, the residents for which CR_1 defines any consistency requirements are a subset of those for which CR_2 defines consistency requirements, as CR_1 does not make any statements about residents having an empty name. Thus, the left condition elements of CR_1 are a subset of those of CR_2. In consequence, if CR_1 requires consistency for a resident and a location, CR_2 requires it anyway, because it already defines consistency for the contained resident.

Additionally, a redundant consistency relation, such as CR_1, must have weaker requirements for the elements at the right side, such that one of the consistent right condition elements is contained anyway because another relation already required them. This means that the relation may have a smaller set of classes, of whom instances are required to consider the models consistent, and there may be more condition elements of the right side that are considered consistent with condition elements of the left side to not restrict the elements considered consistent. In case of CR_1, it only requires an emploee to exist for a resident compared to CR_2, which also requires a non-empty address to exist. Additionally, CR_1 does not restrict the employees that are considered consistent to employees compared to CR_2, as it also considers employees with the same name as consistent, but additionally those having the name of the resident in lowercase.

Add proposition about redundancy properties

Our goal is to have a compatibility-preserving notion of redundancy, i.e., adding a redundant relation to a compatible relation set should preserve compatibility. Unfortunately, our intuitive redundancy definition is not compatibility-preserving.



$$CR_1 = \{\langle (e, r), p\rangle \mid e.name = r.name.toUpper \land e.name = p.name\}$$

$$CR_2 = \{\langle e, p\rangle \mid e.name = p.name\}$$

$$CR_3 = \{\langle p, r\rangle \mid r.name = p.name.toLower\}$$

Figure 6.14.: A consistency relation CR_1 being redundant in $\{CR_1, CR_2, CR_3\}$, with $\{CR_2, CR_3\}$ being compatible and $\{CR_1, CR_2, CR_3\}$ being incompatible.

Proposition 10. Let $\mathbb{CR}$ be a compatible set of consistency relations and let CR be a consistency relation that is redundant in $\mathbb{CR} \cup \{CR\}$. Then CR is not necessarily compatibility-preserving, i.e., $\mathbb{CR} \cup \{CR\}$ is not necessarily compatible.

Proof. We prove the proposition by providing a counterexample. Consider the example in Figure 6.14. CR_2 relates each employee to a person with the same name and CR_3 relates each person to a resident with the same name in lowercase. The consistency relation set $\{CR_2, CR_3\}$ is obviously compatible, because for each employee and each person, which constitute the left condition elements of the consistency relations, a consistent model set containing the person respectively employee can be created by adding the appropriate person or employee with the same name and a resident with the name in lowercase. Furthermore, CR_1 is redundant in $\{CR_1, CR_2, CR_3\}$ according to Definition 27, because if a model is consistent to CR_2 it is also consistent to CR_1, since CR_1 also requires persons with the same name as an employee to be contained in a model set but in less cases, precisely only those in which the model also contains a resident such that the employee name is the one of the resident in uppercase.

However, $\{CR_1, CR_2, CR_3\}$ is not compatible. Intuitively, this is due to the fact that CR_1 and CR_3 define an incompatible mapping between the names of residents and persons. This is also reflected by Definition 22 for compatibility. Take a model with an employee and a resident named A. This is a condition element in $\mathfrak{c}_{l,CR_1}$. Consequentially, CR_1 requires a person A to exist. Furthermore CR_3 requires a resident with name a to exist. In consequence, there are two tuples of employees and residents, both with employee A and one with resident A respectively resident a each, for which a consistent person with name A is required by CR_1. However, CR_1 actually forbids to have two residents, one having the lowercase name of the other, because both are condition elements in CR_1 requiring an appropriate person to occur in a consistent model, but there is only one person that to which both can be mapped, namely the one with the uppercase name, so there is no witness structure with a unique mapping as required by Definition 18 for consistency. This example shows that adding a redundant consistency relation to a compatible set of consistency relations does not lead to a compatible consistency relation set. □

In consequence of Proposition 10, we need a stronger definition of redundancy that is compatibility-preserving. In the example in Figure 6.14 showing Proposition 10, we have seen that it is problematic if a redundant consistency relation considers more classes in its left condition than the relation it is redundant to. Therefore, we restrict the left class tuple.

Definition 28 (Left-equal Redundant Consistency Relation)**.** Let $\mathbb{CR}$ be a set of consistency relations for a metamodel set $\mathbb{M}$. For a consistency relation $CR \in \mathbb{CR}$, we say:

$$\begin{aligned}
&CR \textit{ left-equal redundant in } \mathbb{CR} :\Leftrightarrow \\
&\quad \exists CR' \in (\mathbb{CR} \setminus \{CR\})^+ : \forall \mathfrak{m} \in I_{\mathbb{M}} : \\
&\quad\quad \mathfrak{m} \textit{ consistent to } CR' \Rightarrow \mathfrak{m} \textit{ consistent to } CR \\
&\quad\quad \wedge \mathfrak{C}_{l,CR} = \mathfrak{C}_{l,CR'}
\end{aligned}$$

The definition of left-equal redundancy is similar to the redundancy definition but restricts the notion of redundancy to cases in which the left condition of the redundant consistency relation CR considers the same classes than the other relation in the set of consistency relations that induces consistency of a model set to *CR*. As discussed before, redundancy in general allows that the left condition of a redundant consistency relation can consider a superset of those classes.

Lemma 11. Let *CR* be a consistency relation that is left-equal redundant in a set of consistency relations $\mathbb{CR}$. Then *CR* is redundant in $\mathbb{CR}$.

Proof. Since the definition of left-equal redundancy is equal to the one for redundancy, apart from the additional restriction for the class tuples, redundancy of a left-equal redundant relation is a direct implication of the definition. □

Before showing that left-equal redundancy is compatibility-preserving, we introduce an auxiliary lemma that shows that if a model set contains any left condition element of a left-equal redundant relation, i.e., if that redundant relation requires the model set to contain corresponding elements for that object tuple to be consistent, there is also another relation that requires corresponding elements for that object tuple.

Lemma 12. Let *CR* be a consistency relation that is left-equal redundant in a set of consistency relations $\mathbb{CR}$ for a set of metamodels $\mathbb{M}$. Then it holds that:

$$\begin{aligned}&\exists\, CR' \in (\mathbb{CR} \setminus \{CR\})^+ : \forall\, \mathfrak{c}_l \in \mathbb{c}_{l,CR} : \exists\, \mathfrak{c}'_l \in \mathbb{c}_{l,CR'} :\\ &\quad \forall\, \mathbb{m} \in I_{\mathbb{M}} : \mathbb{m} \textit{ contains } \mathfrak{c}'_l \Rightarrow \mathbb{m} \textit{ contains } \mathfrak{c}_l\end{aligned}$$

Proof. Due to left-equal redundancy of *CR* in $\mathbb{CR}$, we know per definition that:

$$\begin{aligned}&\exists\, CR' \in (\mathbb{CR} \setminus \{CR\})^+ : \forall\, \mathbb{m} \in I_{\mathbb{M}} :\\ &\quad \mathbb{m} \textit{ consistent to } CR' \Rightarrow \mathbb{m} \textit{ consistent to } CR\\ &\quad \wedge \mathfrak{C}_{l,CR} = \mathfrak{C}_{l,CR'}\end{aligned}$$

This implies that:

$$\exists\, CR' \in (\mathbb{CR} \setminus \{CR\})^+ : \forall\, \mathfrak{c}_l \in \mathbb{c}_{l,CR} : \mathfrak{c}_l \in \mathbb{c}_{l,CR'}$$

Because if there was a $\mathfrak{c}_l \in \mathbb{c}_{l,CR}$ so that $\mathfrak{c}_l \notin \mathbb{c}_{l,CR'}$, then the model set $\mathbb{m}$ only consisting of $\mathfrak{c}_l$ would be consistent to CR', because it does not require any other elements to exist for considering the model set consistent, whereas there is at least one $\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \in CR$, so that $\mathbb{m}$ needs to contain $\mathfrak{c}_r$ for considering $\mathbb{m}$ consistent to CR, which is not given by construction. This shows that $\mathbb{c}_{l,CR'}$ contains all elements in $\mathbb{c}_{l,CR}$, so there is always at least one element from $\mathbb{c}_{l,CR'}$ that a model set $\mathbb{m}$ contains if it contains an element from $\mathbb{c}_{l,CR}$, which proves the statement in the lemma. □

The following lemma derived the property of left-equal redundancy from redundancy, which was not correct. Maybe we can find a more general notion of redundancy from which we can derive the contains implication, reviving this lemma gain.

Theorem 13. Let $\mathbb{CR}$ be a compatible set of consistency relations for a set of metamodels $\mathbb{M}$ and let CR be a consistency relation that is left-equal redundant in $\mathbb{CR} \cup \{CR\}$. Then $\mathbb{CR} \cup \{CR\}$ is compatible.

Proof. Due to left-equal redundancy of CR in $\mathbb{CR} \cup \{CR\}$, which also implies general redundancy according to Definition 27, $\mathbb{CR}$ and $\mathbb{CR} \cup \{CR\}$ are equivalent, according to Lemma 9. Due to that equivalence, we know that for any model set $\mathbb{m} \in I_{\mathbb{M}}$:

$$\mathbb{m} \textit{ consistent to } \mathbb{CR} \Leftrightarrow \mathbb{m} \textit{ consistent to } \mathbb{CR} \cup \{CR\} \tag{6.1}$$

It follows from Definition 22 for compatibility and Equation 6.1:

$$\forall CR' \in \mathbb{CR} : \forall \mathfrak{c}_l \in \mathbb{c}_{l,CR'} : \exists \mathbb{m} \in I_{\mathbb{M}} : \\ \mathbb{m} \textit{ contains } \mathfrak{c}_l \wedge \mathbb{m} \textit{ contains } \mathbb{CR} \cup \{CR\} \tag{6.2}$$

This already shows that for $\mathbb{CR}$ the compatibility definition is fulfilled, so we need to prove that the compatibility definition is fulfilled for CR as well. Due to compatibility of $\mathbb{CR}$ and Lemma 5 showing equality of compatibility for a consistency relation set and its transitive closure, we know that:

$$\forall CR' \in \mathbb{CR}^+ : \forall \mathfrak{c}_l \in \mathbb{c}_{l,CR'} : \exists \mathbb{m} \in I_{\mathbb{M}} : \\ \mathbb{m} \textit{ contains } \mathfrak{c}_l \wedge \mathbb{m} \textit{ consistent to } \mathbb{CR}^+ \tag{6.3}$$

Due to left-equal redundancy of CR in $\mathbb{CR} \cup \{CR\}$, we have shown in Lemma 12 that the following is true:

$$\begin{aligned}&\exists\, CR' \in \mathbb{CR}^+ : \forall\, \mathfrak{c}_l \in \mathbb{c}_{l,CR} : \exists\, \mathfrak{c}'_l \in \mathbb{c}_{l,CR'} : \forall\, \mathbb{m} \in I_{\mathbb{M}} : \\ &\quad \mathbb{m} \textit{ contains } \mathfrak{c}'_l \Rightarrow \mathbb{m} \textit{ contains } \mathfrak{c}_l\end{aligned} \tag{6.4}$$

The combination of Equation 6.3 and Equation 6.4 gives:

$$\begin{aligned}&\exists\, CR' \in \mathbb{CR}^+ : \forall\, \mathfrak{c}_l \in \mathbb{c}_{l,CR} : \exists\, \mathfrak{c}'_l \in \mathbb{c}_{l,CR'} : \\ &\quad (\forall\, \mathbb{m} \in I_{\mathbb{M}} : \mathbb{m} \textit{ contains } \mathfrak{c}'_l \Rightarrow \mathbb{m} \textit{ contains } \mathfrak{c}_l) \\ &\quad \wedge (\exists\, \mathbb{m} \in I_{\mathbb{M}} : \mathbb{m} \textit{ contains } \mathfrak{c}'_l \wedge \mathbb{m} \textit{ consistent to } \mathbb{CR}^+)\end{aligned}$$

A simplification by combining the two last lines of that statement leads to:

$$\begin{aligned}&\forall\, \mathfrak{c}_l \in \mathbb{c}_{l,CR} : \exists\, \mathbb{m} \in I_{\mathbb{M}} : \\ &\quad \mathbb{m} \textit{ contains } \mathfrak{c}_l \wedge \mathbb{m} \textit{ consistent to } \mathbb{CR}^+\end{aligned}$$

Due to Equation 6.1 and Lemma 4, which shows equality of consistency for a consistency relation set and its transitive closure, this is equivalent to:

$$\begin{aligned}&\forall\, \mathfrak{c}_l \in \mathbb{c}_{l,CR} : \exists\, \mathbb{m} \in I_{\mathbb{M}} : \\ &\quad \mathbb{m} \textit{ contains } \mathfrak{c}_l \wedge \mathbb{m} \textit{ consistent to } \mathbb{CR} \cup \{CR\}\end{aligned} \tag{6.5}$$

The combination of Equation 6.2 and Equation 6.5 shows that $\mathbb{CR} \cup \{CR\}$ fulfills Definition 22 for compatibility. □

Corollary 14. Let $\mathbb{CR}$ be a compatible set of consistency relations and let $CR_1, \ldots, CR_k$ be consistency relations with:

$$\begin{aligned}&\forall\, i \in \{1, \ldots, k\} : \\ &\quad CR_i \textit{ left-equal redundant in } \mathbb{CR} \cup \{CR_1, \ldots, CR_i\}\end{aligned}$$

Then $\mathbb{CR} \cup \{CR_1, \ldots, CR_k\}$ is compatible.

Proof. This is an inductive implication of Theorem 13, because $\mathbb{CR}$ is compatible and sequentially adding CR_i to $\mathbb{CR} \cup \{CR_1, \ldots, CR_{i-1}\}$ ensures that $\mathbb{CR} \cup \{CR_1, \ldots, CR_i\}$ is compatible, because $\mathbb{CR} \cup \{CR_1, \ldots, CR_{i-1}\}$ was compatible as well. □

With Corollary 14, we have shown that if we have a set of consistency relations $\mathbb{CR}$ and are able to find a sequence of redundant consistency relations $CR_1, \ldots CR_k$ according to Corollary 14 such that we know that $\mathbb{CR} \setminus \{CR_1, \ldots CR_k\}$ is compatible, then it is proven that $\mathbb{CR}$ is compatible.

6.9.3. Summary

In the previous sections, we have proven the following three central insights:

1. Compatibility is composable: If independent sets of consistency relations are compatible, then their union is compatible as well (Theorem 6).
2. Consistency relation trees are compatible: If there are no two concatenations of consistency relations in a consistency relation set that relate the same classes, then that set is compatible (Theorem 8).
3. Left-equal redundancy is compatibility-preserving: Adding a left-equal redundant consistency relation to a compatible set of consistency relations, that set unified with the redundant relation is still compatible (Corollary 14).

These insights enable us to define a formal approach for proving compatibility of a set of consistency relations. Given a set of relations for which compatibility shall be proven, we search for consistency relations in that set that are left-equal redundant to it. If iteratively removing such redundant relations—virtually—from the set leads to a set of independent consistency relation trees, it is proven that the initial set of consistency relations is compatible.

Such an approach to prove compatibility of consistency relations is *conservative*. If the approach finds redundant relations, such that a consistency relation set can be reduced to a set of independent consistency relations trees, the set is proven compatible, as we have shown by proof. If the approach is not able to find such relations, the set may still be compatible, but the approach is not able to prove that. Conceptually, this can be due to the fact that there may be compatibility-preserving relations that do not fulfill the definition of left-equal redundancy. Furthermore, an actual technique to identify left-equal redundant relations may not be able to find all of them automatically, as we will see later.

Formalize conservativeness

Prove that if the dual of the meta graph is a tree, it is a consistency relation tree

In the following, we discuss how such an approach can be operationalized. First, we discuss how actual transformations, at the example of QVT-R, can be represented in a graph-based structure, such that it conforms to our formal notion and allows to check whether the structure is an independent set of consistency relation trees. Second, we present an approach for finding consistency relations that are left-equal redundant, by the means of an SMT solver applied to the constraints defined in QVT-R relations.

Construction of valid models. Valid models may restrict the usable instances of a metamodel. Discuss impact on definitions and theorems and especially the constructive discussions within the proofs. Especially consider the meaning of references in models.

7. Proving Compatibility of Relations [30 p.]

8. Constructing Correct Transformations [20 p.]

9. Orchestrating Transformation Networks [20 p.]

10. Evaluation and Discussion [20 p.]

Part IV.

Improving Quality of Transformation Networks [50 p.]

11. Specifying Consistency with Commonalities [20 p.]

Modern software-intensive systems are usually described by different *models*, also considered as *views*, such as code, architecture, deployment specifications and other role- or concern-specific models. Since all these models describe the same system, but focus on specific properties or use different levels of abstraction, they typically share information that is represented in the models redundantly, or at least induces dependencies between them. Such redundancies have to be kept consistent to achieve a contradiction-free specification of the system.

In practice, redundancies are often kept consistent manually [Sax+17; Gui+18]. A common means to automate consistency preservation are *incremental model transformations*. Such transformations define how an instance of one metamodel has to be updated to restore consistency whenever an instance of another was modified. A transformation can be declared in different ways: it may either specify what has to be changed to restore consistency (*imperative*), or only the consistency constraints that have to hold, from which the rules to restore consistency are derived automatically (*declarative*). But no matter how a transformation is defined, it specifies a *relation* between two (or more) metamodels. However, redundant elements are representations of a *common concept* rather than independent elements that have to be directly related. In consequence, we think that it is natural to make the common concept, which the redundant elements are supposed to describe, explicit. We can then specify how this concept manifests itself in the different metamodels, instead of defining directly how the redundant elements are related. For example, it appears to be more natural to say that classes in UML and Java are different manifestations of the concept of a class in object-oriented design, rather than saying that a UML class should be related to a corresponding Java class. Such a common concept is what we call a *Commonality*.

In this paper, we present the *Commonalities approach*. It defines Commonalities between metamodels explicitly and thereby allows to state clearly which common concepts they share. From such a specification, transformations are derived that keep instances of those metamodels consistent. We discuss options how to derive such transformations, strategies to hierarchically compose Commonalities, as well as benefits and limitations of the approach. Additionally, we discuss design options for a language that supports the specification of Commonalities and present the *Commonalities language*, which we have developed as a proof-of-concept. It is based on the Bachelor's thesis of Gleitze [Gle17].

Our main contributions in this paper are:

Commonalities Approach (C1): We propose an approach for making common concepts of different metamodels explicit rather than encoding them implicitly in constraints of a transformation.

Commonalities Language (C2): We discuss design options for a language to define Commonalities and outline one language to specify them.

Proof-of-Concept (C3): We give an indicator for the applicability of the approach by providing a proof-of-concept implementation and applying it to a scenario with four simple metamodels sharing common concepts.

We expect several benefits from our approach, i.e. specifying Commonalities, in comparison to direct transformation specifications between metamodels. First, we claim to achieve *better understandability* of relations between metamodels, because common concepts are made explicit. Second, the approach *reduces errors* when more than two metamodels are to be kept consistent. Transformations usually relate two metamodels, especially because multidirectional relations are hard to express [Ste17], and therefore have to be combined to a network of transformations to keep instances of more than two metamodels consistent. Such a network can be regarded as a graph, formed by metamodels as its nodes and transformations as its edges. However, such a network can easily raise compatibility problems if there exists more than one path of transformations between two metamodels. A hierarchy of Commonality specifications is, by design, not prone to such problems. Finally, we *improve reusability* in comparison to a network of transformations, because an arbitrary subset of metamodels, between which Commonalities are defined, can be selected to keep their instances consistent. In contrast, removing metamodels from a transformation network can easily lead to missing transformation paths between two metamodels.

11.1. Running Example

We use a running example throughout the paper to explain the Commonalities idea. It relies on three metamodels: UML class models, Java code, whose grammar definition can be treated as a metamodel [Hei+10], and the PCM [Reu+16], a component-based architecture description language. The consistency relations between Java and UML class models are mostly one-to-one mappings, as they provide the same concepts of object-oriented design. Consistency relations between PCM and Java were proposed by Langhammer et al. [LK15]. For this paper, only the one-to-one mapping between components in PCM and classes in Java—or generally all object-oriented languages—are relevant, whereby each component is mapped to a class but not vice versa.

For the examples in this paper, a minimalist subset of those metamodels, depicted in Figure 11.1, is sufficient. It only comprises classes in UML and Java and components in PCM, which all have a name. The name shall be equal between corresponding classes in UML and Java, whereas the classes realizing a component shall have the same name as the component, complemented by an "Impl" suffix (cf. [LK15]). With state-of-the-art techniques, those constraints could be implemented as relations in declarative or as enforcing routines in imperative transformation languages.

In this paper, we consider metamodels that conform to the EMOF standard [Obj16b]. Such metamodels consist of classes, which we denote as *metaclasses* to avoid confusion with classes in exemplary metamodels such as Java and UML. Metaclasses can in turn contain attributes and associations to other classes, which may be containments.

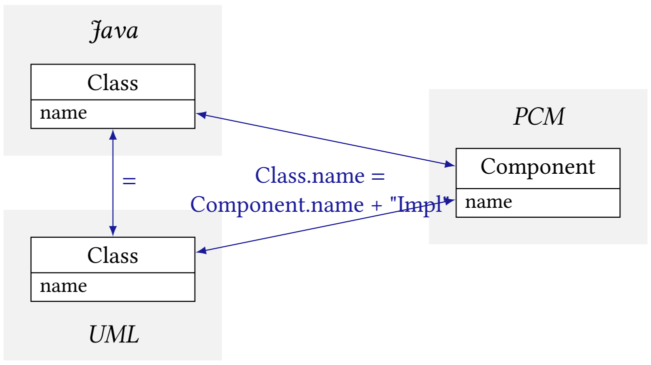


Figure 11.1.: Metamodel extracts for Java, UML and PCM and consistency relations ($\longleftrightarrow$) between them

11.2. Making Common Concepts Explicit

Based on the decomposition of consistency relations, we propose another approach to achieve a tree structure of transformations, which inherently optimizes *modularity*. The idea bases on the insight that we can distinguish two kinds of consistency relations:

Descriptive consistency relations are “naturally” given when two metamodels represent common *concepts* redundantly or at least with dependent properties. This is, for example, the case for UML class models and Java realizing Object Orientation (OO).

Normative consistency relations prescribe consistency and do not exist “naturally”. This is especially the case if metamodels represent different abstractions or domains of a system, which have no implicit relation, such as an ADL and Java.

While descriptive consistency relations between two metamodels are usually definite, such as those between OO design in UML and Java, normative consistency relations may vary depending on the project context. For example, several possible relations can be defined between an ADL and OO design, for example the realization of each component as a class or as a complete project [Lan17].

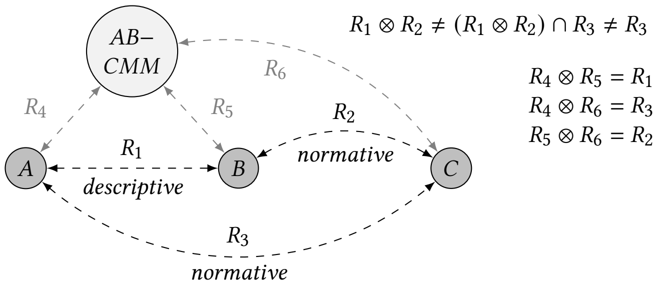


Figure 11.2.: Definition of a concept metamodel

We can use this for showing how to integrate commonalities with ordinary direct relations, maybe there should be a section about that.

Transformations for descriptive consistency relations implicitly encode the common concepts. Instead, we propose to make these common concepts explicit in so-called *Concept Metamodels (CMMs)* and define relations between them and the concrete metamodels. We illustrate this in Figure 11.2. The descriptive consistency relation R_1 is converted into a CMM for the metamodels A and B with new relations R_4 and R_5 between the concrete metamodels and the CMM. The existing normative consistency relations 11.2 and R_3 to metamodel C are replaced by a new relation R_6 to the CMM. The CMM and its consistency relations have to be appropriately defined to replace the original ones, as depicted in Figure 11.2. It will be part of our research to figure out how to define such a CMM, so that it can also be combined with other metamodels. While it basically has to contain the common concepts of the metamodels sharing a descriptive consistency relations, it may also need to contain additional information depending on consistency relations to other metamodels, which are not known a-priori.

This approach addresses all identified challenges and solves them under the assumption that all consistency relations are described using CMMs. We achieve inherent transformation compatibility by avoiding more than one transformation path between two metamodels by design. Since metamodels are only coupled across CMMs and thus represent leaves of the transformation tree, any subset of them can be selected, maximizing modularity.

Finally, we assume that making common concepts explicit improves comprehensibility.

The most crucial part of this approach is the necessity to build a tree of CMMs. This will not be possible if always considering whole metamodels and their relations, but can be possible if independent concepts are extracted to be treated individually. For that, we will apply our findings on consistency relation decomposition (see section 11.3). Additionally, the approach specifically aims to improve the specification of transformations for descriptive relations. In consequence, it must be combined with transformations expressing the normative relations between CMMs and other metamodels.

11.3. The Commonalities Approach

The state-of-the-art approach to keep models consistent automatically is the application of transformation languages. If instances of multiple (i.e., more than two) metamodels are to be kept consistent, one can either use multidirectional transformation approaches, or compose bidirectional transformations to a network of transformations [Cle+19]. When an instance of one metamodel is changed in such a network, the transformations are executed successively to propagate the change transitively across all models. There are strategies to find one ordering of transformations to apply [Ste17] and strategies to perform a fixpoint iteration until no further changes are conducted [Kla+19].

In this section, we propose a different approach for keeping two or more models consistent by specifying their common concepts rather than their direct consistency relations. This forms our contribution **C1**.

11.3.1. Making Common Concepts Explicit

The redundancies between different metamodels are an expression of common concepts that are represented redundantly. We already gave the example of a class in UML and Java, which are different representations of

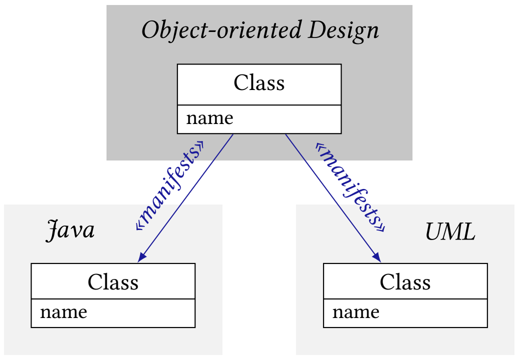


Figure 11.3.: Concept metamodel for object-oriented design with a `Class` Commonality and its relations to UML and Java

the common concept of a class in general object-oriented design. We propose to make common concepts explicit rather than encoding them into the rules of a transformation. This can be achieved by creating a *concept metamodel*, which defines those common concepts, and specifying the relations between the concept metamodel and the existing metamodels. We refer to the existing metamodels as *concrete metamodels*. The relation specifications can be used to derive transformations between the concrete metamodels and the concept metamodel.

Figure 11.3 shows the metaclasses for a `Class` as extracts of the concrete metamodels for UML and Java and the metaclass for the common concept of a `Class` in the concept metamodel for object-oriented design. We denote a single common concept as a *Commonality*. Further Commonalities in object-oriented design could, for example, be interfaces or methods. The relation between the `Class` Commonality and its realizations in the concrete metamodels are shown by a *«manifests»* relation. In our simplified example, the relation would especially define that the names of the classes have to be equal.

When another concrete metamodel that represents the same concepts shall be added, it is only necessary to define its relation to the concept metamodel. For example, adding C++ as another metamodel representing object-oriented design would require the definition of the relation between the

`Class` Commonality in object-oriented design and its representation in C++. Adding an additional metamodel may require the concept metamodel to be extended by Commonalities that were not relevant for the already considered metamodels. In general, a concept metamodel has to contain Commonalities for redundancies in all concrete metamodels, which—mathematically speaking—can be expressed as the union of all pairwise intersections of the concrete metamodels.

11.3.2. Composing Commonalities

We have explained how multiple metamodels can be kept consistent using one concept metamodel. This allows, theoretically, the definition of one large concept metamodel that contains all Commonalities for all concrete metamodels. It would at first sight be similar to a Single Underlying Metamodel (SUMM), as introduced by Atkinson et al. [ASB10]. However, it would be less complex than a SUMM, which is able to express all information about the software system and thus contains the union of all concrete metamodels. Nevertheless, one large concept metamodel would still become unmanageably large due to the fact that it had to contain the union of all pairwise intersections of the concrete metamodels, as mentioned before.

To avoid the specification of such a monolithic concept metamodel, we propose to compose Commonalities from different concept metamodels. Instead of having only Commonalities that relate to metaclasses in concrete metamodels, Commonalities may also have relations to other Commonalities. Consider the concept metamodel for component-based design in Figure 11.4. It contains the Commonality `Component`, which is represented by an equally named metaclass in PCM, as well as in the Commonality `Class` in the concept metamodel for object-oriented design, conforming to the relations proposed by Langhammer et al. [LK15]. This induces a tree structure with Commonalities as inner nodes and metaclasses of concrete metamodels as leaves. With such a composition structure, a *«manifests»* relation may not only exist between a Commonality of a concept metamodel and a metaclass in a concrete metamodel but also between two Commonalities. However, a concrete or concept metamodel that is lower in the hierarchy is supposed to represent how a metaclass or Commonality in the higher one manifests,

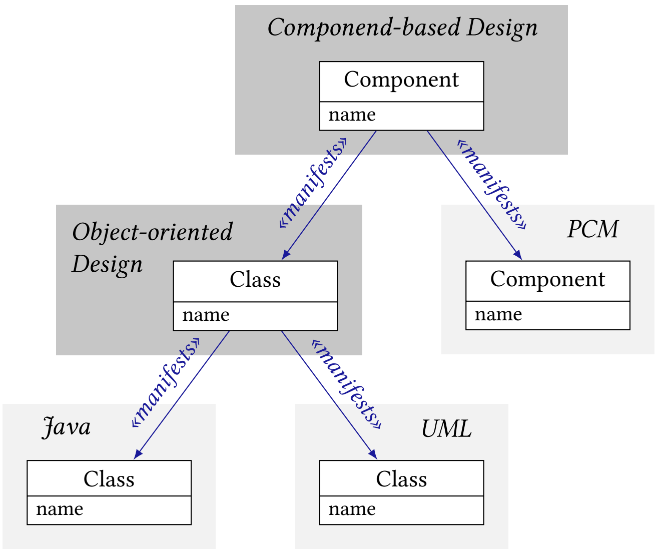


Figure 11.4.: Concept metamodels (dark) and their relations to concrete metamodels (light) for the running example

which is why we call it a *manifestation*. For example, the object-oriented design concept metamodel is a manifestation of the component-based design concept metamodel.

Our goal is to achieve a tree structure of commonalities. In the extended example in Figure 11.5, a `Component` in the concept metamodel for component-based design does not only manifest in a PCM `Component` as well as a `Class` in object-oriented design, but also in UML. Since a `Class` in object-oriented design manifests both in Java and UML, we do not have a tree structure of the induced relations between the metamodels anymore, due to `Class` and `Component` both being represented in UML. However, This still induces a tree structure between metaclasses and Commonalities, with the Commonalities being inner nodes and metaclasses of concrete metamodels being leaves.

A metamodel may have several Commonalities in different concept metamodels with different other metamodels. For example, in Figure 11.4, the

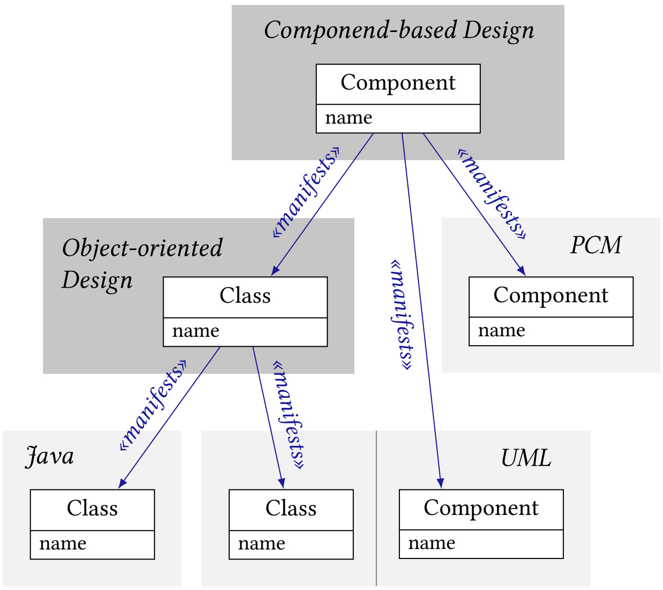


Figure 11.5.: Concept metamodels (dark) and concrete metamodels (light) of the running example, extended by UML components, with their relations

UML metamodel contains a `Class` and a `Component` metaclass, which have two different Commonalities in two different concept metamodels.

11.3.3. Transformation Operationalization

To actually keep models consistent, the specification of a hierarchy of concept metamodels has to be operationalized. Two options for operationalization can be distinguished:

Concept metamodels as additional metamodels: The specified concept metamodels are actually instantiated and the transformations are executed as they are defined between the concept metamodels and their manifestations. In consequence, instances of the concept metamodels have to be maintained.

Transformations between concrete metamodels: The concept metamodels and the relations between them and their manifestations are used to derive bidirectional transformations between the concrete metamodels. For example, from the concept metamodel for object-oriented design in Figure 11.3, a bidirectional transformation between Java and UML is derived.

A drawback of the first option is that additional models have to be managed and persisted. In consequence, the user has to version these models although they should be transparent to him or her, as long as no appropriate framework abstracts from such tasks. A drawback of the second option is that the types of supported relations that can be described in the transformations are limited. First, only relations may be defined that can be composed with any other relation, such that a direct transformation between two metamodels can be derived. Second, it is possible to define n-ary relations between more than two metamodels that cannot be decomposed into binary relations between them, but only into n binary relations between those metamodels and an additional one [Ste17]. In consequence, the first option provides higher expressiveness.

While the first option can be realized without an additional language by just defining the concept metamodels and the transformations with existing languages, the second option requires a mechanism that generates the transformations between the concrete metamodels from those between the concept metamodels and their manifestations.

11.3.4. Benefits of Commonalities

We suppose the Commonalities approach to provide two kinds of benefits: First, we expect that it improves understandability of relations between metamodels, because common concepts are not encoded in transformations implicitly but modelled explicitly. This is even a benefit if instances of only two metamodels shall be kept consistent. Second, it reduces problems that can occur if several bidirectional transformations are combined into a network of transformations to keep multiple models consistent.

Networks of transformations can have two extremes of topologies, as depicted in Figure 11.6. If transformations between all metamodels are defined, the network forms a dense graph (see Figure 11.6a). In contrast, if

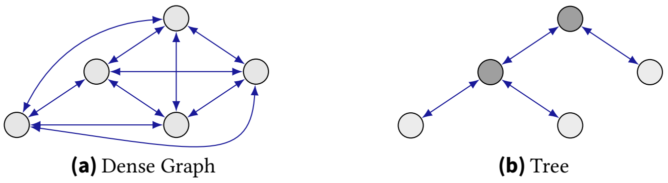


Figure 11.6.: Extremes of transformation network topologies: nodes represent metamodels, edges represent transformations (concept metamodels in a tree of Commonalities in dark gray), adapted from [Kla18]

there exists exactly one path of transformations between each pair of metamodels, the network forms a tree (see Figure 11.6b). Several properties for such networks have been identified by Gleitze [Gle17] and Klare [Kla18]. Two essential properties are *compatibility* and *modularity* [Kla18], which, unfortunately, contradict each other. The Commonalities approach, however, improves both of them. *Compatibility* means that transformations do not define contradictory constraints. Consider the relations introduced for the running example in Figure 11.1. The names of the same class in Java and UML are defined to be equal. If a class in Java and UML realizes a PCM component, it shall have the same name appended with an "Impl" suffix. If transformations realize the three relations between PCM, UML and Java, and the one between PCM and Java adds that suffix whereas the one between PCM and UML omits it, the constraints can never be fulfilled. In that case, the transformations are considered incompatible. Incompatibility may arise whenever more than one transformation path between two metamodels exists. In consequence, compatibility cannot be guaranteed in dense network, whereas it is inherently high if the network forms a tree. *Modularity* means that any subset of the metamodels can be used without loosing consistency because of missing transformations. Modularity is high if any metamodel can be removed from the network and the remaining transformations still define consistency between all remaining metamodels. In consequence, modularity is high in a dense network, because all metamodels are directly related, while it is low if the network is a tree, because inner nodes cannot be removed without their children not being related by a transformation anymore. Since redundant paths between metamodels improve modularity but reduce compatibility, these properties are inherently contradicting.

The Commonalities approach improves both these properties due to the fact that additional metamodels are introduced in the specification. The transformations between metaclasses in concrete metamodels and Commonalities in concept metamodels induce a tree, thus compatibility is high. Additionally, only the leaves of the tree are concrete metamodels, which are actually used to describe a system and whose instances are modified, whereas the inner nodes only represent auxiliary metamodels, exemplarily marked in Figure 11.6b. In consequence, taking an arbitrary subset of concrete metamodels removes only leaves and can thus be done without removing any transformations that are necessary to keep instances of the remaining metamodels consistent. This constitutes a major benefit of the Commonalities approach as compared to ordinary networks of transformations.

11.4. Composition of Commonalities with Views

Discuss here, that a commonalities structure can be encapsulted in views (ref to Vitruv), which are then used to combine with other such structures in an ordinary network of BX. E.g. let there be an OO commonality for Java and UML and one CBS commonality for UML and PCM. Both are encapsulated in a projective view-based approach, which, e.g., exposes the concept metamodel. These views can than be combined by ordinary bx. This allows to build concept metamodels for subsets of the problem (subsets of the metamodels), especially for scenarios in which descriptive relations exist, which are then combined by ordinary networks. This gives the benefits of commonalities, such as extendability, modularity (which are preserved even if the concept is combined with others by bx), but also provides the flexibility of bx networks, but reduced the proneness to errors in the networks as parts are handled by inherently compatible commonalities.

See for example Figure 11.2 for a scenario combining normative and descriptive relations. We could compare a scenario where Java, UML class, UML comp and PCM are connected in a network, connected in an overall commonality and with two commonalities combined in a network.

Drawback of this approach is that the views exposed the structures have to provide all required information to be kept consistent with other structures. For example, the CBS commonality only contains the information shared between UML and PCM, thus if there is information in PCM to be shared with Java, but not with UML component, the concept metamodel does not contain that, but has to be exposed to be kept consistent with the OO concept. Thus, there may be more extensive views than only exposing the commonality. In fact, the structure would need to be a SUM, for which any information can be extracted. However, it is an open issue how consistency is preserved if information is derived to different views which are all modified, or if a heterogeneous view is created (cf. ModelJoin). Imagine the consistency preservation derives the commonalities view for components to modify the information shared between UML and PCM and uses the PCM view to change information only present in PCM (e.g. functionality). If a change in Java requires modifications in both views, these changes both have to be propated to the underlying models. If there are conflicts, they have to be resolved like in a synchronization scenario (several user modifiy views concurrently). This problem is yet unsolved.

11.5. Processes for Defining Commonalities

Discuss how commonalities can be defined. Which roles are involved, especially how different domain experts can communicate. E.g. bottom-up approach: Take the most related metamodels and define their commonalities. Then define higher-level commonalities relating these concept metamodels or even the concrete metamodels. The problem is that combining two concept metamodels requires them to contain all necessary information, thus a concept metamodel design is not only driven by the metamodels to keep directly consistent, but also by the information that is needed to preserve consistency to other commonalities. This refers to the same scenario as if multiple commonality structures are encapsulated into projective view environments which are combined by BX. They require the exposed views to provide all information necessary to preserve consistency.

12. Designing a Language for Expressing Commonalities [15 p.]

Having introduced the concepts of the Commonalities approach, we now present the Commonalities language, as proposed by Gleitze [Gle17]. The language is a prototypical realization of the approach. We discuss design options for such a language and outline its syntax and its usage. This forms our contribution **C2**. For a detailed discussion of the language capabilities, we refer to [Gle17]. In this paper, we focus on the presentation of fundamental concepts and therefore only outline the language to give an impression of what our proof-of-concept evaluation is based on.

12.1. Design Options

The development of a language for realizing the Commonalities approach provides at least two areas of design options. First, there are different possibilities to operationalize the approach, which covers decisions that are not visible to the user of the language. Second, different options regarding how to specify concept metamodels and their relations to their manifestations exist, which are decisions that are visible to the user of the language.

We already discussed in subsection 11.3.3 that there are two different options for operationalizing a specification of Commonalities: one that uses instances of the concept metamodels and defined transformations at runtime, and one that generates direct transformations between the concrete metamodels from the specification. Since the way a specification is operationalized does not affect the user of the approach, it is up to the language and its designer to choose one of the options. We chose to build a language following the former approach, because it does not limit the possible relations that can be expressed.

Regarding the specification of concept metamodels and the transformations, there are two options:

External concept definition: Concept metamodels are defined as ordinary metamodels and the relations to their manifestation are defined in an individual transformation specification for each manifestation.

Internal concept definition: A specialized language allows to define the concept metamodel and the Commonalities it consists of together with relations of all Commonalities to their manifestations.

A benefit of the first option is that the relation to each manifestation can be specified independently, which reduces dependencies between the different manifestations of one concept metamodel. Additionally, it could be realized without developing a new language. The concept metamodels can be described just like any other ordinary metamodel and one can use any existing transformation language, such as QVT, for the transformation definitions. The second option, in contrast, requires a dedicated language that enables an integrated specification of the concept metamodel and its relations to manifestations. We chose to build a language following the second option because we expect its benefits to outweigh the mentioned disadvantages. First, it improves locality: all information about one Commonality is represented in one place. Because of that, we expect that it is easier for developers to understand the combined transformation logic concerning one Commonality. Additionally, the concept metamodel can be easily extended with this solution whenever adding a manifestation requires defining a new Commonality. Finally, we expect that, compared to other options, grouping transformations by their Commonality reduces the possibility for developers to forget defining one or more relevant manifestations for a Commonality.

Listing 12.1: An exemplary specification of the `Component` Commonality between PCM, UML and the object-oriented design concept in the Commonalities language

12.2. Language Description

As introduced before, our realization of the Commonalities language provides an internal concept definition and uses the concept metamodels as additional metamodels in the operationalization. An example for the syntax of the Commonalities language is depicted in Listing 12.1.

The language allows to define concept metamodels by declaring Commonalities, each representing one commonality between different manifestations, such as the `Component` Commonality in our example. Relations between the concept metamodels and their manifestations are supposed to be specified *declaratively*. For every Commonality, the metaclasses in the manifestations that realize them are specified. In the example, the `Component` in PCM and the `Class` in the object-oriented design concept metamodel are related to the `Component` Commonality. In our language, a Commonality is realized by a metaclass in the metamodel that is generated for a concept, so the `Component` Commonality is realized by a `Component` metaclass.

Within a Commonality, attributes and references can be defined, similar to an ordinary metaclass. The relations of an attribute to the manifestation are declared directly at the attribute. In the example, a `name` attribute is specified, which maps to the name of the component in PCM and the name appended with an “Impl” suffix in Java. The language provides several operators for attribute relations, apart from equality relations. The example depicts a prefix operator that allows to compose a String attribute. Such operators can be defined independently and added to the language dynamically. References can be defined comparably to attributes but can be enriched with a definition of containment relations.

The actually conceptualized and implemented language by Gleitze [Gle17] is far more sophisticated than the simple overview we provide here. It supports different kinds of bidirectional operators for attribute mappings, containment specifications (so-called *participations*), attribute checks as preconditions for Commonality instantiation, and more.

13. Evaluation and Discussion [15 p.]

13.1. Proof-of-Concept

We have proposed the Commonalities approach and a realizing language. We have explained that we expect them to improve understandability of transformations and to reduce problems of transformation networks, such as compatibility and modularity. Although we gave arguments that justify this expectation, it has to be evaluated empirically to increase evidence. However, before evaluating the benefits of our approach, we first have to investigate its feasibility. For that reason, we built an initial prototype of the language and applied it to a simple evaluation case as a proof-of-concept. This forms our contribution **C3**.

13.1.1. Case Study

We have implemented a prototype of the Commonalities language, which allows to define Commonalities with simple attribute and reference mappings and to compose Commonalities. The syntax is an extension of the example shown in Listing 12.1. The language comprises a compiler that derives a concept metamodel, as well as a set of transformations from a specification in the language. The generated transformations are in turn defined in the Reactions language [Kla16], which is a delta-based transformation language that is, just like the Commonalities language itself, part of the Vitruvius approach [KBL13]. Vitruvius is a view-based development approach that uses transformations to keep models consistent. The implementation of the Commonalities language can be found in the GitHub repository of the Vitruvius project [Vit].

We have applied the implementation to a simple case study that consists of four metamodels, each containing one metaclass that represents a root element and one that represents a contained element. Both elements have an identifier and a name in all metamodels, and an additional single-valued and multi-valued feature of integers in two of the metamodels. The root metaclass additionally has a containment reference to the contained metaclass. We have defined two Commonalities, one for the root element and one for the contained element, which redundantly represent the same concepts in all the metamodels. The root Commonality references the contained Commonality. This results in one concept metamodel with four manifestations.

To validate that the specifications in the Commonalities language are correctly defined and operationalized, we have defined test cases that perform 21 different model modifications, which create and delete all possible types of elements and modify all attribute and reference values in instances of every metamodel. They cover the set of all possible modifications that can be performed on instances of those metamodels. This also includes change propagation across composed Commonalities. The tests successfully validate that the modifications are correctly propagated to all other models in all cases. The test cases and the used example metamodels are also available in the GitHub repository of the Vitruvius project [Vit].

13.1.2. Discussion

Our proof-of-concept validates the feasibility of the proposed Commonalities approach: It demonstrates that it is possible to apply the concept of defining consistency relations between multiple metamodels through a central metamodel in a simple scenario. It also shows that an operationalization can be derived that preserves consistency of all instances of such metamodels. The results only give an indicator that the Commonalities concept can be applied and that a language with an internal concept definition can be designed. To further evaluate the capabilities of such an approach, the language would have to be extended to be able to define more complex relationships. Additionally, the approach has to be applied to larger parts of more complex metamodels and metamodels for different contexts to improve external validity of the results. This could also reveal whether the assumption of having a tree of Commonalities is practical in realistic scenarios.

Since evaluating functional capabilities of the approach is only an—essential—first step, the evaluation of further properties such as applicability, appropriateness, effectiveness and scalability are part of ongoing work with further case studies. As a central benefit of our approach, we claim to improve understandability of relations between metamodels, but can only give arguments for that by now. An evaluation of that claim would require a user experiment that compares our approach to specifications of direct transformations between multiple metamodels.

Finally, one might argue that defining concept metamodels leads to additional effort, as for two metamodels it is necessary to define one additional metamodel and two transformations rather than only a single transformation. First, this is only true as long as only two metamodels are related by one concept metamodel. If three metamodels shall be related, there would be a network of three transformations, which are not necessarily compatible without using the Commonalities approach, and one metamodel with three transformations using the Commonalities approach. When the Commonalities approach is applied, the number of necessary transformations increases linearly with the number of metamodels that are related, whereas it increases quadratic without them. Second, the effort for defining transformations can be reduced by using an appropriate language to define concept metamodels and transformations, as we have proposed in chapter 12. Our language only requires developers to write one specification that contains both the concept metamodel and all transformations to its manifestations.

Part V.

Epilogue [25 p.]

14. Related Work [15 p.]

14.1. From DocSym Overview

Model consistency preservation, also referred to as *model repair*, is an active field of current research. Nevertheless, most approaches are restricted to consistency between pairs of models [Ste17]. The kinds of dependencies between multiple models and types of inconsistencies were discussed by Kolovos et al. [KPP08]. A summary and classification of model consistency approaches, also regarding their multi-model support, was presented by Macedo et al. [MJC17b].

As stated by Stevens [Ste17], it is reasonable to target multi-model consistency by combining binary transformations. Especially incremental model transformation languages can be applied by executing the transformations transitively. They were surveyed by Kusel et al. [Kus+13], including the Atlas Transformation Language (ATL) [Jou+06; Xio+07], VIATRA [Ber+15] and Triple Graph Grammars (TGGs) [Anj+14; Anj14]. For TGGs, initial concepts for supporting multiple models were proposed [TA15; TA16]. Another transformation-based tool is *Echo* [MGC13]. It is based on *Alloy* [MC13], which uses QVT-R [Obj16a] and model finding to repair inconsistencies. For QVT-R, challenges and steps towards supporting transformations between multiple model were discussed [MCP14]. Kramer proposed an approach combining a language for declarative mappings between metamodels with a fallback language for imperative consistency repair [Kra17; Kla16], developed in the context of the Vitruvius framework [Kra+15]. Except for the explicitly mentioned works, those approaches do not explicitly deal with challenges introduced by combining binary transformations. Another topic regarding transformations is *uncertainty*. Not all decision can be made automatically, e.g., whether a created Java class shall represent an ADL component or not. To handle this, different solutions can be calculated [EPR15] or the developer can be asked for his intent [LK14].

Approaches regarding the combination of binary transformations are rather focused on composing transformations between the same two metamodels [Wag08; WVD10; Wag+11]. Additionally, those approaches do usually not consider transformations as black boxes, but provide intrusive composition operators that require adaptions of the composed transformations [SG08]. Some approaches also deal with processes for composition and simply assume interoperability [Old05].

An approach specific for consistency between different ADLs is DUALLy [Mal+10; Era+12]. It requires the specification of relations between concrete ADLs to a central, predefined ADL metamodel. DUALLy is based on a generic model consistency approach, which uses Answer Set Programming (ASP) [CDE06; Era+08] based on logical programming techniques. Such an approach of adding additional metamodels to represent consistency relations is also shortly discussed in [Ste17].

There has also been some research on design patterns for transformations [ISH08; Lan+14]. They have been surveyed by Lano et al. [Lan+18], but mainly unify how specific kinds of consistency relations can be expressed in transformation languages and not on achieving non-intrusive transformation interoperability.

14.2. From Interoperability Case Study

Macedo et al. [MJC17a] provide a classification of consistency preservation approaches also considering support for multi-model scenarios. In the following, we compare our work to research areas related to preserving consistency between multiple model types.

Networks of Bidirectional Transformations

Networks of BX are the focus of our research. Stevens [Ste17] investigates the ability to split global into binary constraints. She gives arguments to stick to networks of BX rather than using multidirectional transformations.

Important for such networks is the transformation execution order. While we aim to allow arbitrary execution orders, other approaches focus on finding or defining appropriate orders [Ste18].

Multidirectional Transformations

Multidirectional transformations are an alternative to networks of BX. Although they benefit from being less prone to interoperability issues, they do not allow for modular definitions of consistency specifications. The QVT-R standard [Obj16a] considers multidirectional transformations, but Macedo et al. [MCP14] reveal several limitations of its applicability. An extension of TGGs to multiple models [TA15; TA16] focuses on the specification of multidirectional rules but not on potential conceptual and operational issues that we investigated. Commonalities metamodels offer a different approach to reduce the number of transformations and potential issues. Gleitze [Gle17] proposes a generic idea for them, whereas DUALLy [Mal+10; Era+12] uses a domain-specific commonalities metamodel for architecture description languages. Stünkel et al. [Stü+18] and Diskin et al. [DKL18] discuss such commonalities metamodels from a theoretical viewpoint. Several topics of multidirectional transformations, especially the usage of networks of bidirectional transformations and the interaction of several bidirectional transformations, were discussed in a Dagstuhl seminar [Cle+19]. The focus in related working groups was the investigation of scenarios, in which networks of bidirectional transformations do not suffice and thus checked our assumption in section 3.1.

Transformation Chains

Transformation chains are sets of transformations executed one after another to transform one (high-level) model into one (low-level) model across one or more others. It is a special case of networks of BXs, in which chains between all pairs of metamodels are realized. Specification languages for transformation chains, such as FTG+PM [Lúc+13], allow to combine transformations to chains. Another approach is UniTI [Van+06; Van+07; Pil+08], which treats and combines transformations as black-boxes like we do. However, it derives compatibility from external specifications rather than achieving compatibility by construction. To improve maintainability, approaches for separating transformation chains into smaller concern-specific ones [Yie+12] and to support evolution [Yie+09] have been developed.

Transformation Composition

Transformation composition techniques are a means to build networks of BX. They can be separated into internal techniques, which are white-box approaches integrated into the language [Wag08; WVD10; Wag+11], e.g. inheritance or superimposition techniques, and external techniques. External approaches consider the transformations as black-boxes, which makes them related to our work. Most approaches especially focus on factorization and re-composition as a refactoring technique for transformations [SG08] and consider syntactic compatibility on the level of external specifications and matching metamodels rather than investigating techniques to achieve interoperability by construction. Lano et al. [Lan+14] present a catalog of patterns that foster correct composition of transformations. This also includes patterns for unique instantiation like we proposed in **??**. In contrast, our contribution primarily comprises a categorization of mistakes and only uses one specific pattern that is appropriate to avoid mistakes of a certain category. TODO: TraCo composition system [HKA10] for composing transformations, specification of transformation components and properties that are anaylzed: analytical approach (good for modularization level), not by design

Model Merging and Constraint Solving

Model merging and constraint solving are further approaches to achieve consistency preservation between multiple models. For example, Eramo et al. [Era+08] consider the usage of ASP for preserving model consistency. We, however, focus on transformation-based techniques and issues related to that, which is why we do not discuss that research area in more detail.

14.3. From Commonalities

The Commonalities approach is related to the highly researched field of model consistency and especially of model transformations. In the following, we compare our approach to others that rely on commonality specifications, to both multidirectional transformations and transformation networks that also allow consistency preservation between multiple models and finally to constraint solving, a different paradigm for preserving model consistency.

Commonality Approaches

The idea of defining commonalities to express consistency of multiple models was especially researched from a theoretical viewpoint. That research is based on the idea of using an additional $n + 1$-th metamodel to decompose the n-ary consistency relation between n metamodels into n binary relations [Stü+18; DKL18].

Existing approaches to practically use commonalities for keeping multiple models consistent are domain-specific. The DUALLy approach [Mal+10; Era+12] uses a domain-specific concept metamodel for architecture description languages, which is a fixed metamodel to which relations of arbitrary architecture description languages can be defined.

Multidirectional Transformations

Without defining additional metamodels, multidirectional transformations are an approach to directly define the relations between multiple metamodels. The QVT-R standard [Obj16a] considers multidirectional transformations, but Macedo et al. [MCP14] reveal several limitations of its applicability and propose strategies to circumvent them. TGGs are a graph-based approach to define transformations, which has been extended to enable the specification of multidirectional rules [TA15; TA16]. In contrast to our work, these approaches support the specification on n-ary relations between n metamodels, but do not provide means to improve their understandability as we expect the definition of Commonalities to do.

Networks of Bidirectional Transformations

We introduced networks of bidirectional transformations as the state-of-the-art for specifying consistency relations between multiple metamodels. Stevens [Ste17] investigates the ability to decompose n-ary relations into binary ones and also discusses confluence issues, which arise from incompatibilites of transformations, as discussed in subsection 11.3.4. Such a decomposition of relations is not always possible, thus such approaches are restricted to cases where all n-ary relations can be decomposed into binary ones. Additionally, such networks are prone to compatibility errors or reduced modularity, as discussed in subsection 11.3.4.

Transformation composition and transformation chains deal with specific problems of transformation networks. Composition techniques deal with internal composition of transformations [Wag08], which are techniques that are integrated into a language, and external composition of transformations, which work independently from the language. Those approaches especially comprise factorization and re-composition of transformations [SG08] and investigations of compatibility of transformations for different versions of the same metamodels. Transformation chains deal with specific networks that occur when transformations from metamodels with a high level of abstraction to those with a low level of abstraction are defined. Specification languages for transformation chains allow to combine transformations to chains [Lúc+13] and to treat them as black-boxes [Van+06; Van+07].

Constraint Solving

Consistency relations between multiple metamodels can also be expressed as logical constraints. Restoring consistency for a set of models can be achieved by constraint solving. Eramo et al. [Era+08] consider the usage of ASP for that. The approach derives a set of candidates that fulfill the constraints after a model is modified. However, that research focuses on solving constraints rather than designing an appropriate way how to define them, in contrast to our Commonalities approach.

14.4. From SoSym MPM4CPS Paper

In this article, we have presented an approach for proving compatibility of transformation network. Thus, our work contributes to the goal of achieving consistency respectively consistency preservation between multiple models and is related to other approaches with that goal. It is highly related to the area of transformation networks and multi-directional transformations, especially to validation techniques for them. Combining transformations to a network is also related to transformation composition and transformation chain construction, as it is a more general case of these specific problems. Finally, we used formal techniques including a theorem prover to make statements about OCL expressions in QVT-R relations, which is why other comparable formal techniques are related to our work. We discuss the relation of our work to work in these areas in the following.

14.4.1. Consistency Preservation of Multiple Models

Preserving consistency of software artifacts (i.e., models) has been long researched. Starting with approaches for specific modeling languages, such as the UML [DMW05], the relevance of model-driven engineering, accompanied by OMG's Model Driven Architecture [Obj14a] process specification, rose. Several approaches provide domain-specific solutions for consistency problems, such as for consistency between SysML [Obj19] and AUTOSAR [Sch15] in the automotive domain [GHN10]. Modeling frameworks, such as Eclipse Modeling Framework (EMF) [Ste+09], enabled the definition of tools that are independent from concrete models, such as transformation languages, model merging tools and so on.

Based on such modeling framework, different approaches considering model consistency have been developed. They can be distinguished into approaches that are only able to check consistency of models [RE12; KD17] and those that are able to also enforce consistency. Consistency-enforcing approaches are sometimes also referred to as *model repair* approaches, which were surveyed by Macedo et al. [MJC17a]. They also considered whether the approaches are able to handle multiple models or only pairs, but found that only one of the considered approaches handles that case by considering the pairwise relations between models. Consistency preservation approaches are based on heterogeneous ideas, ranging from model merging [Man+15; RC13], macro- and megamodeling [SME08; Sal+15], model finding and constraint solving [MC13; MGC13] and model transformations [CH06; Kus+13; SZK16; MJC17a]. Most of these approaches, if supporting the case of multiple models at all, assume that there is a common knowledge about how all involved models shall be related. With modular knowledge, like assumed when creating transformation networks, incompatibilities in the way consistency is considered always lead to problems, regardless of the approach chosen, so the finding of our work is relevant for all these approaches.

14.4.2. Multi-directional Transformations

Of the previously presented approaches for consistency preservation, model transformations is the approach that provides the highest degree of freedom to influence the way in which consistency is restored. The area of incremental, bidirectional transformations is most relevant for consistency preservation purposes. The concept of bidirectional transformation can be generalized to multi-directional transformations [Ste17; Cle+19], i.e., specification with relations as well as consistency repair routines between multiple models. So consistency preservation between multiple models can be achieved with two transformation approaches, first with multi-directional transformations, and second by combining bidirectional transformations to networks. However, those topics have only been considered in research since recently [Cle+19].

Only few approaches presented in the recent years explicitly consider the case where multiple models shall be kept consistent. Several transformation languages have been proposed in the recent years, surveyed by Kusel et al. [Kus+13]. Among popular languages such as QVT [Obj16a], ATL [Jou+06; Xio+07], VIATRA [Ber+15] and TGGs [Anj+14; Anj14], originally developed by Schürr [Sch95], only the QVT-R standard explicitly considers the case in which more than two models shall be transformed into each other by allowing the definition of multi-directional transformations. However, Macedo et al. [MCP14] revealed several limitations of its applicability. Extensions of TGGs to multiple models called Multi Graph Grammars (MGGs) [KS06] and Graph Diagram Grammars [TA15; TA16] consider the specification of multi-directional rules, but focus on the specification concept and do not yet consider what happens if several such rules are conflicting. Although multi-directional transformation approaches are inherently less prone to compatibility problems, we already discussed drawbacks of the necessity to have no modular specification of consistency.

The case that transformations are combined to networks is not considered by any existing transformation language. Most of the existing considerations for such networks are rather theoretical. For a single bidirectional transformation, several relevant properties, such as correctness, hippocraticness or undoability have been found and researched [Ste10]. In our work, in contrast, we are interested in further properties that are relevant when combining transformations to networks. Stevens [Ste17] started to discuss problems that arise from the combination of several transformations, such as potential non-termination or the problem of not finding a consistent solution. She defined in which situations it is not possible to express a multiary relation by means of binary relations at all. She also discussed orchestration problems for the execution order of transformations [Ste18]. However, compatibility of relations have not been considered yet.

An approach to emulate multi-directional transformations in terms of bidirectional transformation networks are commonalities models. They introduce further models that contain the information that is shared between models and thus has to be kept consistent. They serve as a hub with bidirectional transformations to the actual models, acting like a multi-directional transformations. This concept has been considered on a rather theoretical basis [Stü+18; DKL18], discussing which kinds of relations can be expressed with such an approach, and from an engineering perspective [KG19], discussing the modular specification and composition of commonalities. However, all these approaches do not allow a combination of independently developed consistency specifications for subsets of the models, which is the goal of our work.

14.4.3. Transformation (De-)Composition

Our approach can be seen as a technique to decompose transformations into sets of transformations that are either essential or redundant. Transformation composition has especially been researched in terms of creating chains of transformations, composing larger transformations from smaller ones and finding and extracting common parts in different transformations, known as *factorization*.

A transformation chain defines a sequence of transformations, which transforms one abstract, high-level model into one low-level model across one or more others of different abstraction levels. Languages like FTG+PM [Lúc+13] and UniTI [Van+07] allow to specify the combination of transformations to chains. However, tools like UniTI derive compatibility from additional, external specifications of the transformations, for which conformance to the actual transformation is not guaranteed. Additionally, transformation chains are only a special case of transformation networks, as each transformation network is also aware of the individual transformation chains between all pairs of models. They are, by construction, not that prone to compatibility problems, because there cannot be any cycles in the transformations.

Transformation composition techniques can be seen as a means to build transformation networks. Internal composition techniques can be separated into white-box approaches, which are integrated into languages [Wag08; WVD10; Wag+11], e.g., inheritance or superimposition techniques, and external techniques, which consider the transformations as black boxes. For such transformation compositions, Lano et al. [Lan+14] present a catalog of patterns that foster correct composition. Our approach considers the transformations as white boxes, or at least requires knowledge about the defined consistency relations, but is, in contrast to existing work, not integrated into a transformation language. Additionally, existing approaches have the goal of enhancing composition of transformations between the same metamodels, thus providing benefits like improved reusability, whereas we combine transformations between different metamodels. However, our findings on compatibility can also be applied to composition of transformations between the same metamodels. Finally, factorization approaches identify common parts of transformations and extract them into a base transformation from which the individual parts are extended [SG08]. Such approaches use intrusive operators that adapt the transformations for composition, whereas we only non-intrusively analyze the transformations.

14.4.4. Formal Methods in Consistency Preservation

Some approaches consider consistency preservation as a constraint solving problem rather than a transformation problem. They use constraints to represent consistency relations, like we do for the relations of transformations, and then try to find valid solutions after an inconsistency-introducing modification was made by model finding. For example, some approaches use ASP to preserve consistency of models [CDE06; Era+08]. For QVT-R and Echo, implementations with Alloy were proposed to resolve inconsistencies [MC13; MC16], which were also implemented in the transformation tool Echo. However, these approaches find consistent models based on the defined constraints rather than checking whether those constraints can be fulfilled under specific conditions, like our definition of compatibility specifies and our presented approach is able to prove.

Finally, there are several approaches for the validation of OCL constraints used to define conditions on valid models or to define model transformations. To validate the existence of models that fulfill certain OCL constraints, Kuhlmann et al. [KHG11] and González et al. [Gon+12] propose an approach using SAT solvers. For the validation of model transformations, different approaches have been proposed. Cabot et al. [Cab+10] derive invariants from transformations, which they use for verification purposes, such as to find whether a model exists that can fulfill a transformation rule. Comparably, Cuadrado et al. [CGL17] analyze ATL transformations to find errors in transformations and to find out whether a source model exists that may trigger a transformation. Rather than using constraint logic for verifying a transformation, Azizi et al. [AZK17] verify correctness of an ETL transformation using the symbolic execution of the transformation. Instead of checking a transformation on its own, Vallecillo et al. [Val+12] propose to define a formal specification of transformations, against which they can be validated. Finally, Büttner et al. [BEC12] propose an approach for proving correctness of ATL transformations against pre- and postconditions using SMT solvers. Most approaches use some kind of constraint logic or theorem proving for validating correctness of transformations, which is comparable to our approach. Our defined notion of compatibility is comparable to correctness notions in the approaches of Cuadrado et al. [CGL17] and Cabot et al. [Cab+10], as they try to figure out if a rule can be triggered by any model.

However, all these approaches consider correctness of a single transformation. In contrast, we consider correctness of a transformation network.

15. Future Work [5 p.]

Based on the presented work, there are plenty of possibilities and necessities for follow-up research. In the following, we present an overview of the topics that are most relevant to be considered next from our point of view. We first discuss conceptual extensions by considering weaker redundancy notions and processes to use the approach in, as well as its impact on other correctness requirements in terms of consistency repair. Afterwards, we shortly discuss the possibility to evaluate alternatives for the realization of our formal approach, as well as ideas for completing our realization.

15.0.1. Relaxation of Redundancy Notion

In subsection 6.9.2, we have introduced the notion of *left-equal redundancy*, since an intuitive notion of redundancy is not strong enough to be compatibility-preserving. We based the decision for that stronger definition on insights from a counterexample for redundancy being compatibility-preserving. However, there might be a weaker notion than left-equal redundancy that is still strong enough to be compatibility-preserving. In that case, it would be interesting to investigate application scenarios in which compatibility-preserving relations that are not left-equal redundant occur and how the definition can be appropriately relaxed, such that our approach supports that notion as well.

15.0.2. Interactive Process

Our approach enables a user to check a network of consistency relations regarding compatibility. If the approach identifies a given network as compatible, it is actually compatible as the algorithm operates conservatively. However, the approach is not able to prove incompatibility. If the approach does not identify a network as compatible, it may be incompatible or not. For that reason, we should define a holistic process for the usage of the approach, which integrates further information given by the user into the process of proving compatibility. If the algorithm is not able to prove compatibility, it can present the network, in which it removed some redundant relations, to the user. The user could then be asked to declare a cycle of relations as compatible, for which the algorithm is not able to prove it, or which are actually not compatible, but whose restriction of relations regarding consistency according to other relations is intended. Afterwards, the algorithm could proceed with finding further redundant relations to prove compatibility, based on the decision of the user. As a result, the approach would be applicable to more cases in which compatibility is intentionally not given or in which the algorithm on its own is not able to prove it.

15.0.3. Impact on Consistency Repair

In subsection 6.8.1, we introduced different levels at which a transformation network can be incorrect [Kla+19]. While the approach in this paper is concerned with the correctness of consistency relations, correctness can also be considered at the level of consistency repair routines that ensure consistency according to the relations. Correctness of consistency relations is a necessary requirement for consistency repair routines to work properly, as otherwise, for example, non-terminating loops that try to fulfill consistency relations that can never be fulfilled may occur. Based on the formal notion of compatibility presented in this article, we will evaluate in future work how compatibility affects correctness of consistency repair routines, and whether all remaining correctness issues can be avoided by construction, as proposed in [Kla+19]. This would result in a holistic approach based on construction guidelines and our technique to prove compatibility that enables developers to build correct transformation networks.

15.0.4. Validation of Operationalization Alternatives

We have chosen to translate OCL expressions in QVT-R relations into first-order logic formulae and to use an SMT solver, namely Z3, to evaluate Horn clauses of those expressions that represent the potential redundancy in cycles of consistency relations. However, such a theorem prover is not able to evaluate satisfiability of clauses in all cases, as we already discussed in ??. It is possible to implement the approach in ?? by means of other formal methods. For example, interactive theorem provers may be able to prove redundancy of consistency relations in more cases. This hypothesis can be evaluated in future work. Another possibility is the use of multiple formal methods in the decomposition procedure. Although this requires translating OCL expressions into multiple languages, some formal methods can sometimes provide proofs where others cannot. Thus, the simultaneous use of different symbolic computation tools can increase the chances of finding redundancy proofs.

15.0.5. Completion of Operationalized Approach

The operationalization of our approach presented in ?? and ?? is currently limited to the parts of QVT-R relations and OCL operations that we presented in those sections. In future work, we will extend the set of supported OCL operations, which the approach is able to translate into first-order logic formulae. This will allow us to apply the approach to more sophisticated case studies and provide further evaluation to indicate general applicability of the approach. Moreover, SMT solvers come with heuristics (sometimes called *strategies*) to fine-tune their performances. Strategies should be chosen according to the nature of tested SMT instances, i.e.. consistency specifications. Thus, a better integration of the SMT solver can improve the realization of the current approach for proving compatibility in transformation networks.

16. Conclusions [5 p.]

16.0.1. Compatibility

In this article, we presented an approach to prove compatibility of consistency relations in transformation network. We introduced a formal notion of compatibility, describing when consistency relations are considered contradictory, and we proved correctness of a formal approach that checks whether a transformation network is compatible. We defined an operationalization of that approach for QVT-R and OCL, which uses the translation of OCL to first-order logic formulae and an SMT solver to prove compatibility.

Applying the approach to different scenarios in an evaluation, we found that the approach operates correctly in the sense that it produces *conservative* results. Further, we found that conservativeness is rather low, i.e., only few actually compatible transformation networks were not identified as such. More precisely, only 20% of the compatible transformations were not identified as such, which indicates the practical applicability of the approach. The current limitations of the approach and the degree of conservativeness especially arise from limitations due to undecidability of OCL and missing translations of OCL constructs to first-oder logic formulae. We did not identify conceptual issues that limit the expressiveness or applicability of our approach.

The presented approach enables developers of transformations to independently define their transformations and combine them afterwards, without the necessity to align the underlying consistency relations a priori or to check their compatibility manually when combining them. This is an important contribution to the overall goal of being able to build properly working networks of independently developed transformations to foster the development of large software and cyber-physical systems that involve several models and views to describe that system under construction.

16.0.2. Interoperability Issues

Issues that can arise from the combination of independently developed BX to networks have not been systematically investigated yet. In this paper, we therefore categorized failures that can occur when executing faulty networks of BX. Additionally, we structured the process of specifying consistency into three levels: the global level, the modularization level and the operationalization level. These levels carry the danger for different kinds of mistakes, which we categorized and related to potential failures they can result in. We found that each of the levels is prone to different types of mistakes, and that each type of failure is specific for one category of mistake. This enables developers to easily identify the kind of mistake they made when recognizing a failure. Additionally, the systematic knowledge about potential mistakes, failures, and their relations makes it possible to further develop techniques to avoid them. We have discussed two general avoidance strategies at the modularization and operationalization level in this paper. In future work, we will especially investigate how far and under which assumptions BX can be analyzed regarding contradictions at the modularization level when they are combined.

16.0.3. Quality Properties by Commonalities

In this paper, we proposed the Commonalities approach, which allows to make common concepts of different metamodels explicit. The central idea of the presented approach is to define *concept metamodels* that represent the common concepts, i.e., the Commonalities of two or more existing metamodels, and to define the relation of those metamodels to the concept metamodels. Concept metamodels can be hierarchically composed to enable the separate definition and combination of independent concepts. We discussed different options for designing a language that supports the specification of such Commonalities and their relations, as well as for operationalizing such a specification to executable transformations. We outlined a language for the Commonalities approach and explained which of the aforementioned options we chose, and why. Finally, we have applied an implementation of that language to simple scenarios as a proof-of-concept. The results indicate the feasibility of applying the Commonalities approach and implementing an appropriate language.

The expected benefit of our approach is a better understandability of relations between metamodels compared to their implicit encoding in transformations. Additionally, we argued why our approach improves the essential properties *compatibility* and *modularity*, which usually contradict each other in other approaches to keep multiple models consistent, like networks of transformations that define the direct relations between metamodels. In ongoing work, we extend the capabilities of the language to perform a comprehensive evaluation of the functionality of the approach as well as its applicability to more sophisticated case studies. We will validate our claim of improved understandability in a controlled experiment. Nevertheless, the initial results of our proof-of-concept are a promising indicator for the applicability of the Commonalities approach.

Bibliography

[Anj14] A. Anjorin. “Synchronization of Models on Different Abstraction Levels using Triple Graph Grammars”. PhD thesis. Technische Universität Darmstadt, 2014.

[Anj+14] A. Anjorin et al. “Efficient Model Synchronization with View Triple Graph Grammars”. In: *Modelling Foundations and Applications.* Vol. 8569. LNCS. Springer International Publishing, 2014, pp. 1–17.

[ASB10] C. Atkinson, D. Stoll, and P. Bostan. “Orthographic Software Modeling: A Practical Approach to View-Based Development”. In: *Evaluation of Novel Approaches to Software Engineering.* Vol. 69. Communications in Computer and Information Science. Springer, 2010, pp. 206–219.

[AZK17] B. Azizi, B. Zamani, and S. Kolahdouz-Rahimi. “Contract verification of ETL transformations”. In: *2017 7th International Conference on Computer and Knowledge Engineering (ICCKE).* 2017, pp. 154–160.

[Ber+15] G. Bergmann et al. “Viatra 3: A Reactive Model Transformation Platform”. In: *Theory and Practice of Model Transformations.* Springer, 2015, pp. 101–110.

[BEC12] F. Büttner, M. Egea, and J. Cabot. “On Verifying ATL Transformations Using ‘off-the-shelf’ SMT Solvers”. In: *Model Driven Engineering Languages and Systems.* Springer Berlin Heidelberg, 2012, pp. 432–448.

[Cab+10] J. Cabot et al. “Verification and Validation of Declarative Model-to-Model Transformations through Invariants”. In: *Journal of Systems and Software* 83.2 (2010), pp. 283–302.

[CDE06] A. Cicchetti, D. Di Ruscio, and R. Eramo. “Towards Propagation of Changes by Model Approximations”. In: *2006 10th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW’06).* IEEE Computer Society, 2006, p. 24.

[Cle+19] A. Cleve et al. “Multidirectional Transformations and Synchronisations (Dagstuhl Seminar 18491)”. In: *Dagstuhl Reports* 8.12 (2019), pp. 1–48.

[CGL17] J. S. Cuadrado, S. Guerra, and J. de Lara. “Static Analysis of Model Transformations”. In: *IEEE Transactions on Software Engineering* 43.9 (2017), pp. 868–897.

[CH06] K. Czarnecki and S. Helsen. “Feature-based Survey of Model Transformation Approaches”. In: *IBM Systems Journal* 45.3 (2006), pp. 621–645.

[DMW05] C. R. Dantas, L. G. P. Murta, and C. M. L. Werner. “Consistent evolution of UML models by automatic detection of change traces”. In: *Eighth International Workshop on Principles of Software Evolution (IWPSE’05).* 2005, pp. 144–147.

[DKL18] Z. Diskin, H. König, and M. Lawford. “Multiple Model Synchronization with Multiary Delta Lenses”. In: *Fundamental Approaches to Software Engineering.* Springer International Publishing, 2018, pp. 21–37.

[EPR15] R. Eramo, A. Pierantonio, and G. Rosa. “Managing Uncertainty in Bidirectional Model Transformations”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering.* SLE 2015. ACM, 2015, pp. 49–58.

[Era+12] R. Eramo et al. “A model-driven approach to automate the propagation of changes among Architecture Description Languages”. In: *Software and Systems Modeling* 11 (1 2012), pp. 29–53.

[Era+08] R. Eramo et al. “Change Management in Multi-Viewpoint System Using ASP”. In: *Enterprise Distributed Object Computing Conference Workshops.* 2008, pp. 433–440.

[ETA] ETAS Group. *ASCET-DEVELOPER.* URL: `https://www.etas.com/ascet` (visited on 02/12/2020).

[GHN10] H. Giese, S. Hildebrandt, and S. Neumann. “Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent”. In: *Graph Transformations and Model-Driven Engineering*. Vol. 5765. LNCS. Springer Berlin / Heidelberg, 2010, pp. 555–579.

[Gle17] J. Gleitze. “A Declarative Language for Preserving Consistency of Multiple Models”. Bachelor’s Thesis. Karlsruhe Institute of Technology (KIT), 2017.

[Gon+12] C. A. González et al. “EMFtoCSP: A tool for the lightweight verification of EMF models”. In: *2012 First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)*. 2012, pp. 44–50.

[Gui+18] H. Guissouma et al. “An Empirical Study on the Current and Future Challenges of Automotive Software Release and Configuration Management”. In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2018, pp. 298–305.

[HKA10] F. Heidenreich, J. Kopcsek, and U. Aßmann. “Safe Composition of Transformations”. In: *Theory and Practice of Model Transformations*. Springer Berlin Heidelberg, 2010, pp. 108–122.

[Hei+10] F. Heidenreich et al. “Closing the Gap between Modelling and Java”. In: *Software Language Engineering*. Vol. 5969. LNCS. Springer Berlin Heidelberg, 2010, pp. 374–383.

[ISH08] M.-E. Iacob, M. W. A. Steen, and L. Heerink. “Reusable Model Transformation Patterns”. In: *2008 12th Enterprise Distributed Object Computing Conference Workshops*. 2008, pp. 1–10.

[ITE] ITEA. *AMALTHEA4public – An Open Platform Project for Embedded Multicore Systems*. URL: `http://www.amalthea-project.org/` (visited on 02/12/2020).

[Jou+06] F. Jouault et al. “ATL: A QVT-like Transformation Language”. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA ’06. ACM, 2006, pp. 719–720.

[Kla16] H. Klare. "Designing a Change-Driven Language for Model Consistency Repair Routines". Master's Thesis. Karlsruhe Institute of Technology (KIT), 2016.

[Kla18] H. Klare. "Multi-model Consistency Preservation". In: *21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS): Companion Proceedings*. 2018, pp. 156–161.

[KG19] H. Klare and J. Gleitze. "Commonalities for Preserving Consistency of Multiple Models". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2019, pp. 371–378.

[Kla+19] H. Klare et al. "A Categorization of Interoperability Issues in Networks of Transformations". In: *Journal of Object Technology* 18.3 (2019). The 12th International Conference on Model Transformations, 4:1–20.

[KPP08] D. Kolovos, R. Paige, and F. Polack. "Detecting and Repairing Inconsistencies across Heterogeneous Models". In: *2008 1st International Conference on Software Testing, Verification, and Validation*. 2008, pp. 356–364.

[KD17] H. König and Z. Diskin. "Efficient Consistency Checking of Interrelated Models". In: *Modelling Foundations and Applications*. Springer International Publishing, 2017, pp. 161–178.

[KS06] A. Königs and A. Schürr. "MDI: A Rule-based Multi-document and Tool Integration Approach". In: *Software and Systems Modeling (SoSyM)* 5.4 (2006), pp. 349–368.

[KBL13] M. E. Kramer, E. Burger, and M. 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. ACM, 2013, 5:1–5:6.

[Kra+15] M. E. Kramer et al. "Change-Driven Consistency for Component Code, Architectural Models, and Contracts". In: *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*. CBSE '15. ACM, 2015, pp. 21–26.

[Kra17] M. E. Kramer. “Specification Languages for Preserving Consistency between Models of Different Languages”. PhD thesis. Karlsruhe Institute of Technology (KIT), 2017. 278 pp.

[KHG11] M. Kuhlmann, L. Hamann, and M. Gogolla. “Extensive Validation of OCL Models by Integrating SAT Solving into USE”. In: *Objects, Models, Components, Patterns*. Springer Berlin Heidelberg, 2011, pp. 290–306.

[Kus+13] A. Kusel et al. “A Survey on Incremental Model Transformation Approaches”. In: *ME 2013 – Models and Evolution Workshop Proceedings*. 2013, pp. 4–13.

[Lan17] M. Langhammer. “Automated Coevolution of Source Code and Software Architecture Models”. PhD thesis. Karlsruhe Institute of Technology (KIT), 2017. 259 pp.

[LK14] M. Langhammer and M. E. Kramer. “Determining the Intent of Code Changes to Sustain Attached Model Information During Code Evolution”. In: *Fachgruppenbericht des 2. Workshops “Modellbasierte und Modellgetriebene Softwaremodernisierung”*. Vol. 34 (2). Softwaretechnik-Trends. GI e.V., 2014.

[LK15] M. Langhammer and K. Krogmann. “A Co-evolution Approach for Source Code and Component-based Architecture Models”. In: *17. Workshop Software-Reengineering und-Evolution*. Vol. 4. 2015.

[Lan+14] K. Lano et al. “Correct-by-construction synthesis of model transformations using transformation patterns”. In: *Software & Systems Modeling* 13.2 (2014), pp. 873–907.

[Lan+18] K. Lano et al. “A Survey of Model Transformation Design Pattern Usage”. In: *Journal of Systems and Software* 140 (2018), pp. 48–73.

[Lúc+13] L. Lúcio et al. “FTG+PM: An Integrated Framework for Investigating Model Transformation Chains”. In: *SDL 2013: Model-Driven Dependability Engineering*. Springer Berlin Heidelberg, 2013, pp. 182–202.

[MC13] N. Macedo and A. Cunha. “Implementing QVT-R Bidirectional Model Transformations Using Alloy”. In: *Fundamental Approaches to Software Engineering*. Vol. 7793. LNCS. Springer Berlin Heidelberg, 2013, pp. 297–311.

[MC16] N. Macedo and A. Cunha. “Least-change bidirectional model transformation with QVT-R and ATL”. In: *Software & Systems Modeling* 15.3 (2016), pp. 783–810.

[MCP14] N. Macedo, A. Cunha, and H. Pacheco. “Towards a framework for multi-directional model transformations”. In: *3rd International Workshop on Bidirectional Transformations - BX*. Vol. 1133. CEUR-WS.org, 2014.

[MGC13] N. Macedo, T. Guimaraes, and A. Cunha. “Model Repair and Transformation with Echo”. In: *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. 2013, pp. 694–697.

[MJC17a] N. Macedo, T. Jorge, and A. Cunha. “A Feature-based Classification of Model Repair Approaches”. In: *IEEE Transactions on Software Engineering* 43.7 (2017), pp. 615–640.

[MJC17b] N. Macedo, T. Jorge, and A. Cunha. “A Feature-based Classification of Model Repair Approaches”. In: *IEEE Transactions on Software Engineering* PP.99 (2017), p. 1.

[Mal+10] I. Malavolta et al. “Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies”. In: *IEEE Transactions of Software Engineering* 36.1 (2010), pp. 119–140.

[Man+15] U. Mansoor et al. “MOMM: Multi-objective model merging”. In: *Journal of Systems and Software*. Vol. 103. 2015.

[Mat] MathWorks. *Simulink – Simulation and Model-Based Design – MATLAB & Simulink*. URL: `https://www.mathworks.com/products/simulink.html` (visited on 02/12/2020).

[Maz16] M. Mazkatli. “Consistency Preservation in the Development Process of Automotive Software”. MA thesis. Karlsruhe Institute of Technology (KIT), 2016.

[Maz+17] M. Mazkatli et al. “Automotive Systems Modelling with Vitruvius”. In: *15. Workshop Automotive Software Engineering*. Vol. P-275. Lecture Notes in Informatics (LNI). GI, Bonn, 2017, pp. 1487–1498.

[Obj16a] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification.* Version 1.3. 2016.

[Obj14a] Object Management Group (OMG). *Model Driven Architecture (MDA) – MDA Guide rev. 2.0.* 2014.

[Obj14b] Object Management Group (OMG). *Object Constraint Language.* Version 2.4. 2014.

[Obj16b] Object Management Group (OMG). *OMG Meta Object Facility (MOF) Core Specification.* Version 2.5.1. 2016.

[Obj19] Object Management Group (OMG). *OMG System Modeling Language (OMG SysML).* Version 1.6. 2019.

[Obj17] Object Management Group (OMG). *OMG Unified Modeling Language (OMG UML).* Version 2.5.1. 2017.

[Old05] J. Oldevik. “Transformation Composition Modelling Framework”. In: *Distributed Applications and Interoperable Systems.* Springer Berlin Heidelberg, 2005, pp. 108–114.

[PRV08] M. Petrenko, V. Rajlich, and R. Vanciu. “Partial Domain Comprehension in Software Evolution and Maintenance”. In: *2008 16th IEEE International Conference on Program Comprehension.* 2008, pp. 13–22.

[Pil+08] J. von Pilgrim et al. “Constructing and Visualizing Transformation Chains”. In: *Model Driven Architecture – Foundations and Applications.* Springer Berlin Heidelberg, 2008, pp. 17–32.

[RE12] A. Reder and A. Egyed. “Incremental Consistency Checking for Complex Design Rules and Larger Model Changes”. In: *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS 2012).* Vol. 7590. LNCS. Springer-Verlag, 2012, pp. 202–218.

[Reu+16] R. H. Reussner et al. *Modeling and Simulating Software Architectures – The Palladio Approach.* MIT Press, 2016. 408 pp.

[RC13] J. Rubin and M. Chechik. “N-way model merging”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. 2013.

[SME08] R. Salay, J. Mylopoulos, and S. Easterbrook. “Managing Models through Macromodeling”. In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 2008, pp. 447–450.

[Sal+15] R. Salay et al. “Enriching megamodel management with collection-based operators”. In: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2015, pp. 236–245.

[SZK16] L. Samimi-Dehkordi, B. Zamani, and S. Kolahdouz-Rahimi. “Bidirectional Model Transformation Approaches – A Comparative Study”. In: *2016 6th International Conference on Computer and Knowledge Engineering (ICCKE)*. 2016, pp. 314–320.

[SG08] J. Sánchez Cuadrado and J. García Molina. “Approaches for Model Transformation Reuse: Factorization and Composition”. In: *Theory and Practice of Model Transformations*. Springer Berlin Heidelberg, 2008, pp. 168–182.

[Sax+17] E. Sax et al. *A Survey on the State and Future of Automotive Software Release and Configuration Management*. Tech. rep. Karlsruhe Institute of Technology, Department of Informatics, 2017.

[Sch15] O. Scheid. *AUTOSAR Compendium - Part 1: Application and RTE*. AUTOSAR - Compendium Series. CreateSpace Independent Publishing Platform, 2015.

[Sch95] A. Schürr. “Specification of graph translators with triple graph grammars”. In: *Graph-Theoretic Concepts in Computer Science*. Vol. 903. LNCS. Springer Berlin Heidelberg, 1995, pp. 151–163.

[SV06] T. Stahl and M. Völter. *Model-Driven Software Development*. John Wiley & Sons, 2006.

[Ste+09] D. Steinberg et al. *EMF - Eclipse Modeling Framework*. 2nd ed. Addison-Wesley Professional, 2009.

[Ste10] P. Stevens. “Bidirectional model transformations in QVT: semantic issues and open questions”. In: *Software & Systems Modeling* 9.1 (2010), pp. 7–20.

[Ste17] P. Stevens. “Bidirectional Transformations in the Large”. In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS).* 2017, pp. 1–11.

[Ste18] P. Stevens. “Towards sound, optimal, and flexible building from megamodels”. In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems.* ACM, 2018, pp. 301–311.

[Stü+18] P. Stünkel et al. “Multimodel Correspondence Through Intermodel Constraints”. In: *Conference Companion of the 2Nd International Conference on Art, Science, and Engineering of Programming.* Programming’18 Companion. ACM, 2018, pp. 9–17.

[TA16] F. Trollmann and S. Albayrak. “Extending Model Synchronization Results from Triple Graph Grammars to Multiple Models”. In: *Theory and Practice of Model Transformations.* Springer International Publishing, 2016, pp. 91–106.

[TA15] F. Trollmann and S. Albayrak. “Extending Model to Model Transformation Results from Triple Graph Grammars to Multiple Models”. In: *Theory and Practice of Model Transformations.* Springer International Publishing, 2015, pp. 214–229.

[Val+12] A. Vallecillo et al. “Formal Specification and Testing of Model Transformations”. In: *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures.* Springer Berlin Heidelberg, 2012, pp. 399–437.

[Van+06] B. Vanhooff et al. “Towards a Transformation Chain Modeling Language”. In: *Embedded Computer Systems: Architectures, Modeling, and Simulation.* Springer Berlin Heidelberg, 2006, pp. 39–48.

[Van+07] B. Vanhooff et al. “UniTI: A Unified Transformation Infrastructure”. In: *Model Driven Engineering Languages and Systems.* Springer Berlin Heidelberg, 2007, pp. 31–45.

[Vit] Vitruv Tools. *Vitruvius Framework (GitHub).* URL: `https://github.com/vitruv-tools/Vitruv` (visited on 03/23/2020).

[Wag08] D. Wagelaar. “Composition Techniques for Rule-Based Model Transformation Languages”. In: *Theory and Practice of Model Transformations.* Springer Berlin Heidelberg, 2008, pp. 152–167.

[WVD10] D. Wagelaar, R. Van Der Straeten, and D. Deridder. “Module superimposition: a composition technique for rule-based model transformation languages”. In: *Software & Systems Modeling* 9.3 (2010), pp. 285–309.

[Wag+11] D. Wagelaar et al. “Towards a General Composition Semantics for Rule-Based Model Transformation”. In: *Model Driven Engineering Languages and Systems.* Springer Berlin Heidelberg, 2011, pp. 623–637.

[Wol+15] C. Wolff et al. “AMALTHEA – Tailoring tools to projects in automotive software development”. In: *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS).* Vol. 2. 2015, pp. 515–520.

[Xio+07] Y. Xiong et al. “Towards Automatic Model Synchronization from Model Transformations”. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering.* ASE ’07. ACM, 2007, pp. 164–173.

[Yie+09] A. Yie et al. “An Approach for Evolving Transformation Chains”. In: *Model Driven Engineering Languages and Systems.* Springer Berlin Heidelberg, 2009, pp. 551–555.

[Yie+12] A. Yie et al. “Realizing Model Transformation Chain interoperability”. In: *Software & Systems Modeling* 11.1 (2012), pp. 55–75.