Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

A Categorization of Interoperability Issues in Networks of Transformations

Heiko Klare[a] Torsten Syma[a] Erik Burger[a] Ralf Reussner[a]

a. Chair for Software Design and Chair (SDQ),
Institute for Program Structures and Data Organization (IPD),
Karlsruhe Institute of Technology (KIT), Germany

Abstract Bidirectional transformations (BX) are a common approach for keeping two types of models consistent, but consistency preservation between more than two types of models is not researched well. One solution is the composition of BX to networks of transformations. Nevertheless, such networks are prone to failures due to interoperability issues between the individual BX, which are independently developed by various experts. We therefore systematically identify and categorize such issues. First, we structure the process of consistency specification into different conceptual levels. Then we develop a catalog of potential mistakes, which we derive from those levels, and consequential failure types. Finally, we discuss strategies to avoid mistakes at the different levels. This catalog is beneficial for transformation developers and transformation language developers. It improves awareness in developers of potential mistakes and consequential failures, enables the development of techniques to avoid specific mistakes by construction, and eases the identification of reasons for failures.

Keywords Model Transformation; Multidirectional Transformation; Transformation Composition; Transformation Interoperability

1 Introduction

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 Bidirectional Transformations (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 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

Heiko Klare, Torsten Syma, Erik Burger, Ralf Reussner. *A Categorization of Interoperability Issues in Networks of Transformations*. In Journal of Object Technology, vol. 18, no. 3, 2019, pages 4:1–20.
doi:10.5381/jot.2019.18.3.a4

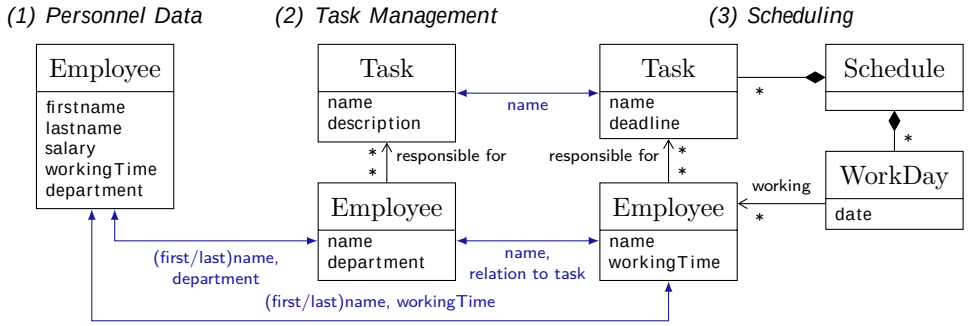


Figure 1 – Exemplary Consistency Relations (↔) between Three Simple Metamodels

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 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.

2 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 EMOF [Obj16b].

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

In the following, let a model M_i be always an instance of model type $\mathbb{M}_i$. 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 [Obj14, 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 *$\mathbb{CS}$ for model types $\mathbb{M}_1, \ldots, \mathbb{M}_n$ is a relation $\mathbb{CS} \subseteq \mathbb{M}_1 \times \ldots \times \mathbb{M}_n$ between models that are consistent. We denote a binary consistency specification for model types $\mathbb{M}_i$ and $\mathbb{M}_j$ as $\mathbb{CS}_{i,j}$.*

Enumerating consistent instances to define consistency is comparable to [Ste17]. If there are no restrictions on when models are consistent, $\mathbb{CS}$ contains all tuples of models. We denote restrictions for models to be in $\mathbb{CS}$ as *consistency constraints*. It would, in theory, also be possible to define $\mathbb{CS}$ 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) $CS_{i,j}$ that maps a tuple of model pairs, each containing an original model M_k M_k and a modified model M_k M_k, to a new tuple of model pairs:*

$$CPS_{CS_{i,j}} : \big((M_1, M_1), \ldots, (M_n, M_n)\big) \to \big((M_1, M_1), \ldots, (M_n, M_n)\big),$$

$$\big((M_1, M_1), \ldots, (M_i, M_i), \ldots, (M_j, M_j), \ldots, (M_n, M_n)\big)$$

$$\to \begin{cases} \big((M_1, M_1), \ldots, (M_i, M_i), \ldots, (M_j, M_j), \ldots, (M_n, M_n)\big) & (M_i, M_j) \quad CS_{i,j} \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

so that

$$(M_i, M_j) \quad CS_{i,j}$$

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 [Ste07], so that no changes are performed when models are already consistent. Let CPS be a set of preservation specifications for consistency specifications CS. We denote the set of consistent model tuples regarding CS as $\mathfrak{M}_{CS} = \{(M_1, \ldots, M_n) \mid CS_{i,j} \quad CS : (M_i, M_j) \quad CS_{i,j}\}$. We want to achieve that:

$$(M_1, \ldots, M_n) \quad \mathfrak{M}_{CS} : \quad M_1 \quad M_1, \ldots, M_n \quad M_n : \quad CPS_1, \ldots, CPS_k \quad CPS :$$
$$CPS_1 \circ \cdots \circ CPS_k\big((M_1, M_1), \ldots, (M_n, M_n)\big) = \big((M_1, M_1), \ldots, (M_n, M_n)\big)$$
$$CS_{i,j} \quad CS : (M_i, M_j) \quad CS_{i,j}$$

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.

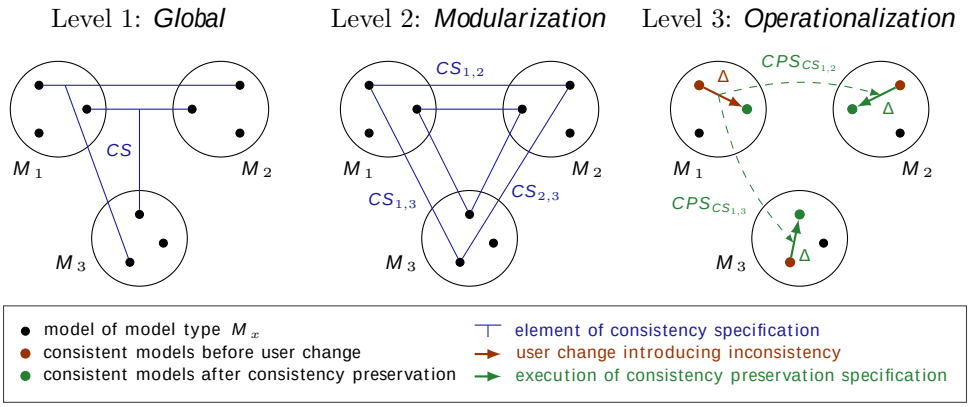


Figure 2 – Examples for Abstraction Levels in the Consistency Specification Process

3 The Consistency Specification Process

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 2 and explain them in more detail in the following.

3.1 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 2.

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 2, 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 1.

3.2 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 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 [Ste07], but is used for the extension to networks of BX here.

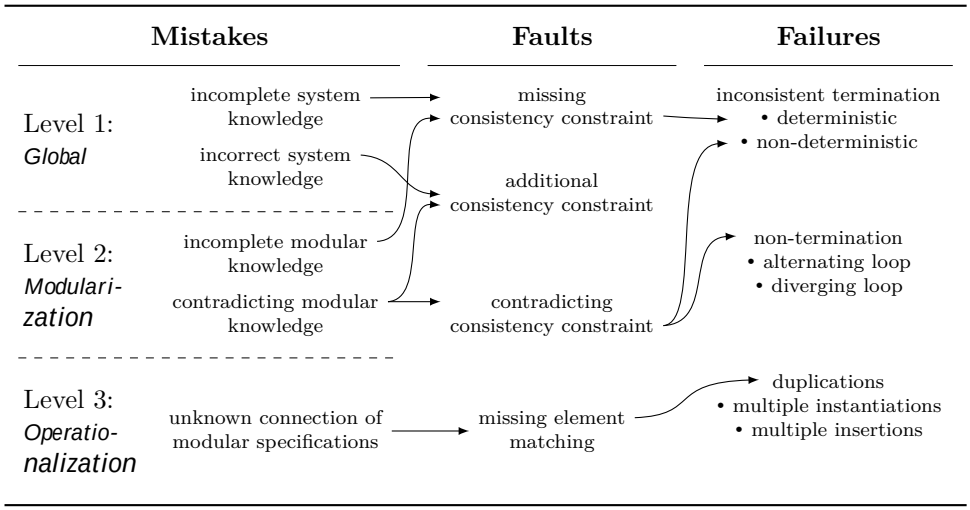


Figure 3 – Categorization and Dependencies of Mistakes, Faults and Failures

4 Issues in Networks of Bidirectional Transformations

In this section, we categorize potential *failures* that can occur when executing BX in a network to preserve consistency. We then consider *mistakes* that a developer can make and that lead to *faults* in the specifications of consistency and its preservation. We derive them from the specification levels introduced in Subsection 3.1, as each kind of mistake is specific for one of those levels. We finally relate the mistakes to the failures that can occur while executing the operationalization of a faulty consistency specification. That categorization forms our contribution **C2**. In the following, we only discuss failures and their causing mistakes, but no strategies to solve or avoid them. Such strategies are discussed in Section 5.

4.1 Potential Failures

Mistakes in the specification of consistency, no matter on which of the specification levels, can lead to failures when executing the preservation of consistency according to that specification. Before identifying the causal mistakes, we first categorize the types of potential failures into three categories. We depict them in Figure 3.

First, consistency preservation can fail by **resulting in an inconsistent state**. This can either occur *deterministically* or *non-deterministically*, if the result depends on the execution order of the consistency preservation specifications.

Second, consistency preservation can fail by **not terminating**. This can either manifest in an *alternating loop*, when a feature, e.g., an attribute, alternates between two or more values, or in a *diverging loop*, when at least one feature value diverges, e.g., a number counting up or a string being repeatedly appended.

Third, consistency preservation can result in **duplications**. *Multiple instantiation* can occur because different consistency preservation specifications instantiate an element multiple times, although all of them represent the same element. For example, an element is created by transformations $M_1 \rightarrow M_2 \rightarrow M_3$ and another is created by transformation $M_1 \rightarrow M_3$, although there should be only one element. *Multiple*

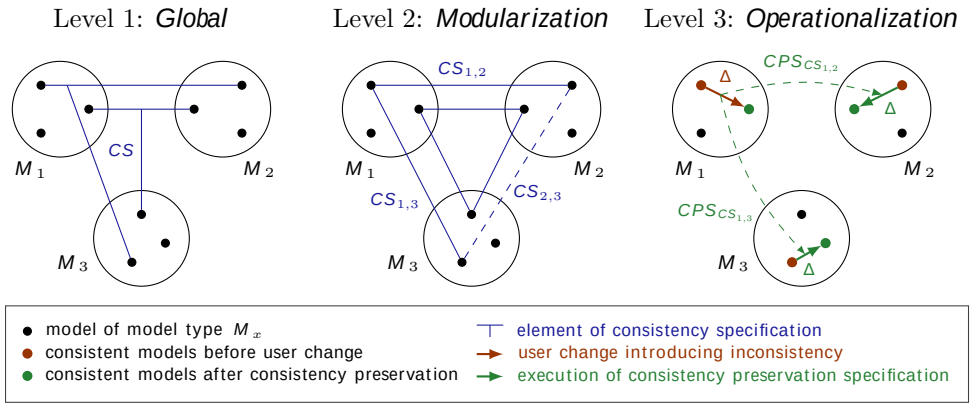


Figure 4 – Examples for Mistakes on Different Specification Levels

referencing can occur due to the same reason because an element is inserted into a reference or attribute list several times, although it should be inserted only once.

4.2 Mistakes and Faults

Developers or the transformation engine can make different kinds of mistakes on each of the specification levels, which lead to faults in the specification and finally to different kinds of failures during consistency preservation. In the following, we derive mistakes and faults from the specification levels, depicted in Figure 3.

Global Level

Regarding global consistency specifications for a set of model types, two basic mistakes can be made. These mistakes concern compliance of the defined consistency specification with the actual notion of consistency between the involved model types. First, a specification can be incomplete (*underspecified*), which means that some consistency constraints are missed. As a result, the consistency specification according to Definition 3 would contain more tuples of models than are actually consistent to each other. Another potential mistake are too restricted (*overspecified*) consistency specifications, which means that additional, faulty consistency constraints are considered. As a result, actually consistent tuples of models would be missing in the consistency specification according to Definition 3.

Modularization Level

When developers modularize the global consistency specification by defining binary consistency specifications, these modular specifications can be non-compliant with the global one. Two kinds of mistakes, similar to those at the global level, can be distinguished, regarding compliance of modular and global specifications. First, modular consistency specifications can be incomplete (*underspecified*), so that there are global constraints which are not covered by them. The modular consistency specifications $CS_{1,2}$, $CS_{2,3}$ and $CS_{1,3}$ in Figure 4 are incomplete iff

M_1, M_2, M_3 :

(M_1, M_2) $CS_{1,2}$ (M_2, M_3) $CS_{2,3}$ (M_1, M_3) $CS_{1,3}$ (M_1, M_2, M_3) CS

This finally leads to *false positives* when investigating whether a given tuple of models is consistent regarding the global specification. Modular consistency specifications cannot only be incomplete because of an actual specification mistake, but also because of n-ary relations on the global level that cannot be expressed by a set of binary relations. We excluded that case by our assumption made in Subsection 3.1, as otherwise a modularization into binary relations would not be possible at all. If such cases have to be supported, the modularization would have to be extended to also consider n-ary relations.

Second, a modular specification can be too restricted (*overspecified*) regarding the global consistency specification if additional constraints are added. The modular consistency specifications in Figure 4 are overspecified iff

$$M_1, M_2, M_3 :$$
$$(M_1, M_2, M_3) \quad \mathit{CS} \quad \big[(M_1, M_2) \quad \mathit{CS}_{1,2} \quad (M_2, M_3) \quad \mathit{CS}_{2,3} \quad (M_1, M_3) \quad \mathit{CS}_{1,3}\big]$$

In Figure 4, omitting the dashed relation in $\mathit{CS}_{2,3}$ would lead to such an overspecifiation. Overspecifications lead to additional constraints regarding the global specification, but also, and more severe, to contradicting constraints regarding other modular specifications. In case of contradictions, the modular consistency specifications cannot be fulfilled at the same time. In such a case, the graph of consistency relations would contain no cylces, i.e. sets of models that are consistent to each other. We have discussed an example for such contradicting specifications in Section 1, where constraints for transferring an employee name contradicted. Such mistakes lead to *false negatives* as actually consistent models (regarding the global specification) are identified as inconsistent.

Operationalization Level

The types of mistakes that can be made at the operationalization level are different from those at the other levels, because this level does not concern the definition of consistency specifications (Definition 3), but of consistency *preservation* specifications (Definition 4). Such specifications are faulty if no composition of them exists that returns a consistent tuple of models for each possible change. In Figure 4, an exemplary application of a single consistency preservation specification is depicted that leads to models that are not consistent according to the (global and modular) consistency specifications. Let CPS be a set of consistency preservation specifications for the binary consistency specifications CS and let $\mathfrak{M}_{\mathit{CS}}$ be the set of model tuples that are consistent regarding CS (cf. Section 2). The consistency preservation specifications are faulty iff

$$(M_1, \ldots, M_n) \quad \mathfrak{M}_{\mathit{CS}}, (M_1, \ldots, M_n) \quad \mathit{M}_1 \times \cdots \times \mathit{M}_n : \quad \mathit{CPS}_1, \ldots, \mathit{CPS}_k \quad \mathit{CPS} :$$
$$\mathit{CPS}_1 \circ \cdots \circ \mathit{CPS}_k\big((M_1, M_1), \ldots, (M_n, M_n)\big) = \big((M_1, M_1), \ldots, (M_n, M_n)\big)$$
$$\mathit{CS}_{i,j} \quad \mathit{CS} : (M_i, M_j) \;/\; \mathit{CS}_{i,j}$$

In practice, mistakes at the operationalization level occur due to missing identification of equal elements in different consistency preservation specifications. In our motivational example (Figure 1), consider that an employee is created in the personnel management system, transformed to the task management system and from that to the scheduling system. The additional direct specification between personnel management and scheduling system has to consider the already created employee rather than instantiating a new one.

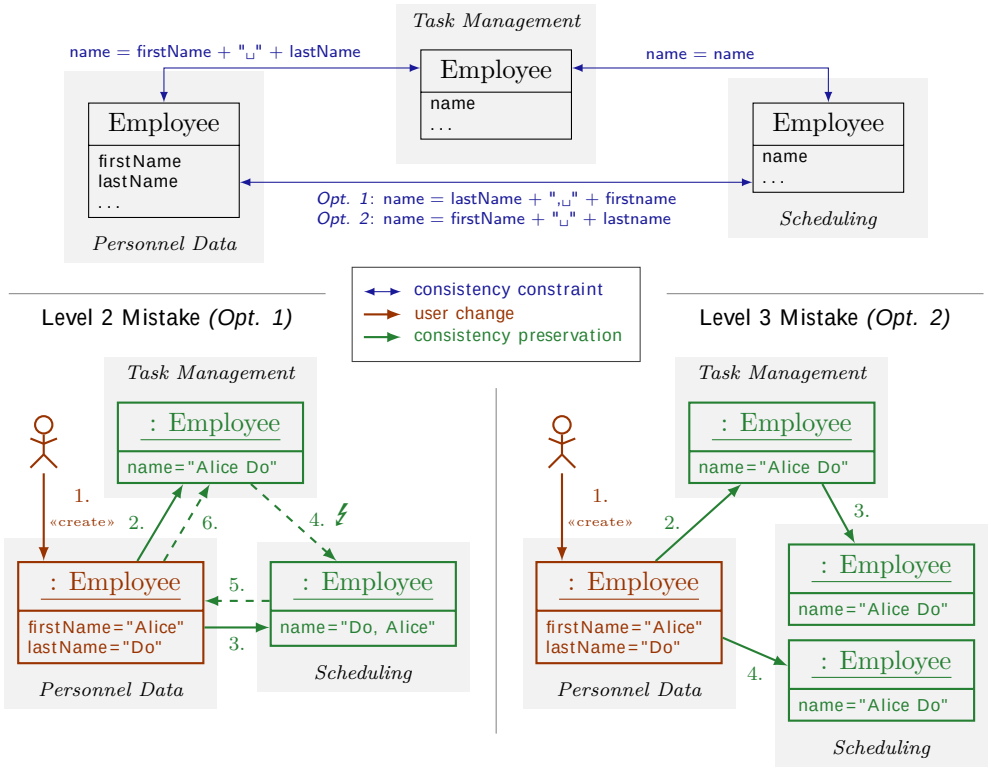


Figure 5 – Consistency Constraints on Metamodel Extract (top), Failure due to Mistake on Modularization Level (left), Failure due to Mistake on Operationalization Level (right)

4.3 Categorization and Discussion

Although all failures occur during operationalization, the mistakes that lead to them can also be made at a higher specification level, such as the modularization or global level. More importantly, each type of failure can be traced back to specific types of mistakes, or, vice versa, specific mistakes lead to specific kinds of failures. Figure 5 shows extracts of the three metamodels from our motivation, as well as consistency constraints between them. There are two options for a constraint between personnel data and scheduling system. The first option is contradictory to the one defined between personnel data and task management system, as already discussed in Section 1. This demonstrates that contradictory constraints are a typical fault that can result from contradicting modular knowledge, when different persons define such constraints independently. If, nevertheless, such a contradictory consistency specification is operationalized to a consistency preservation specification, the propagation of changes may never terminate. This is shown in the left scenario in Figure 5, where the name is replaced repeatedly in an *alternating loop* as indicated by the dashed arrows.

If no mistakes are made on the modularization level, so that no contradictions exist, missing matching of equal elements in the consistency preservation specifications can still lead to duplicate element instantiations. With the second option for the constraint in Figure 5, no mistakes on modularization level exist. However, a missing matching of elements can lead to the situation shown in the right scenario of Figure 5, in which two employees are instantiated across different transformation paths.

These were two of several causal chains for mistakes and faults to resulting failures. We give a full overview of those dependencies in Figure 3. Missing constraints lead to deterministic inconsistencies, because such inconsistencies are not modelled and thus resolved. Additional consistency constraints do not lead to any actual failures, but reduce the set of consistent models. The only consequence is that consistency preservation does not consider models that would actually be consistent. Contradicting constraints, which can arise from a faulty modularization, are more severe, as we have seen in the example: They can either lead to non-deterministic inconsistencies, e.g., depending on the execution order of consistency preservation specifications, or to loops that alternate or diverge values. Finally, the missing element matching at the operationalization level can lead to multiple instantiations, as we have seen in the example, or multiple insertions.

5 Avoiding Interoperability Issues

To ensure that a network of BX operates properly, potential mistakes must be avoided. We evaluate our categorization regarding correctness and completeness in a case study that combines independently developed transformations. In that case study, we classify occurring failures with our categorization and trace them back to a causal mistake. To identify whether such a classification is correct, we need to be able to fix the mistake and validate that the failure disappears. Therefore, we discuss general strategies to avoid mistakes at the different levels as our contribution **C3** and apply them in the evaluation.

At the global level, mistakes occur due to non-conformance with an informal notion of consistency and can only be avoided by careful requirements elicitation. We therefore have to assume that global level mistakes are reliably avoided by the developers. Analytic approaches [Kla18] can ensure that specifications at the modularization and operationalization level are free of faults. Nevertheless, the drawback of such an approach is that it works a-posteriori, when transformations are combined to a network. We, in contrast, want to achieve avoidance of interoperability issues a-priori, so that transformations can be developed independently and combined afterwards. It is easy to see that mistakes at the modularization level cannot be avoided a-priori. Ensuring that transformations are non-contradictory would require developers to have knowledge about the other transformations, which breaks the assumption of independent development. Finally, mistakes regarding element matching at the operationalization level are domain-independent. This enables the development of generic mechanisms to ensure interoperability at the operationalization level by construction, without knowing about other transformations.

In the following, we discuss one strategy to avoid mistakes at the modularization and one to avoid those at the operationalization level. Developers can use these strategies to build networks that are free of faults, or can use them to fix mistakes if failures occur.

5.1 Contradiction-free Modularizations

Contradictions in binary consistency specifications cannot be avoided by design. However, the structure of the network of specifications influences how prone to mistakes it is [Kla18]. Two extremes of networks are depicted in Figure 6: One is to have a specification for each pair of model types, inducing a dense graph. This

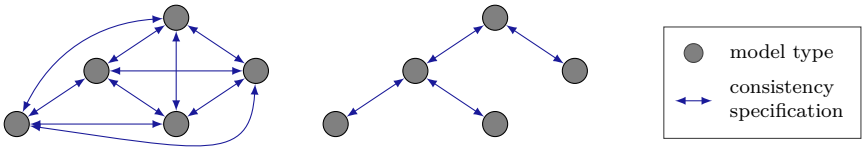


Figure 6 – Extremes of Strategies for Modularizing Consistency Specifications

extreme is prone to contradictions, because all relations are redundantly specified across several paths. Another extreme is to define each relation, potentially indirectly, only once, so that only one path of consistency specifications exists between each pair of model types. This leads to a tree of specifications, which is inherently free of contradictions and avoids modularization mistakes by design. However, it requires that such a network structure exists at all, because between three model types there must always be one relation that can be expressed transitively across the other two (cf. [Kla18]). For example, if $CS_{1,3}$ in Figure 4 shall be omitted and transitively expressed across $CS_{1,2}$ and $CS_{2,3}$, it must hold that:

$$M_1, M_2, M_3 : (M_1, M_2, M_3) \quad CS \Leftrightarrow (M_1, M_2) \quad CS_{1,2} \quad (M_2, M_3) \quad CS_{2,3}$$

In consequence, if a network of transformation can be built that is a tree, mistakes at the modularization level are avoided by design. If a tree cannot be achieved, it is necessary to find and fix mistakes when transformations are combined to a network. In this case, the consistency specifications must be revised whenever non-termination or non-deterministic termination of consistency preservation is observed (see Figure 3).

5.2 Matching Elements in Operationalizations

To avoid failures due to mistakes at the operationalization level, transformations must respect that other transformations may have already created elements. In the binary case, this is unnecessary. A single incremental BX can assume that elements are either created by the user, or were created by the transformation itself. To identify corresponding elements, transformation languages usually use trace models, which are created by the transformations. When BX are combined to networks, direct trace links may be missing because a sequence of other transformations created the elements and trace links only indirectly across elements in other models. In this scenario, corresponding elements can be matched by information at three levels:

1. *Explicit unique*: The information that elements correspond is unique and represented explicitly, e.g., within a trace model.
2. *Implicit unique*: The information that elements correspond is unique, but represented implicitly, e.g., in terms of key information within the models such as element names.
3. *Non-unique*: If no unique information exists, heuristics must be used, e.g. based on ambiguous information or transitive resolution of indirect trace links.

Indirect trace links, which link elements transitively across other models, usually exist for elements that correspond, because other transformations have already created them. Nevertheless, indirect trace links cannot be used to unambiguously identify such elements. An element can correspond to multiple elements in another model,

which is why most transformation languages offer tagging of trace links with additional information to identify the correct element. For example, a language may tag trace links with the transformation rule they were instantiated in. This is helpful in the bidirectional case, but when links are resolved transitively, these tags have been created by other, independently developed transformations, and are thus unknown. Therefore, resolving indirect trace links is only a heuristic, but does not unambiguously retrieve corresponding elements.

Finally, it is up to the transformation engine or the transformation developer to ensure that elements are correctly matched. In contrast to the bidirectional case, direct trace links cannot be assumed in case of networks of BX. Therefore, key information within the models must always be considered to identify matching elements. Whenever direct trace links or unique key information exists, relevant elements can be unambiguously matched. In all other cases, heuristics must be used, which potentially leads to failures.

6 Evaluation

We have systematically constructed the categorization in Section 4 from the potentials for mistakes that are induced by the different specification levels. To further improve evidence regarding completeness and correctness of our categorization, we validate it in a case study as our contribution **C4**. The goal is to show completeness of the identified mistakes and failures, and to investigate correctness of the dependencies between them.

6.1 Case Study

The evaluation is based on a case study developed for the Ecore-based Vitruvius framework[1] [KBL13] for consistent system development. Vitruvius uses incremental, delta-based consistency preservation. It records atomic changes in models and executes consistency preservation specifications, according to Definition 4, to inductively preserve consistency. Those specifications are written in the Reactions language [Kla16], which is a language for unidirectional transformations at the operationalization level.

The case study is based on consistency between UML class models, instances of the Palladio Component Model (PCM), which is an architecture description language for performance prediction [RHK16], and Java code. For these metamodels, different persons have independently developed transformations [Kra17], especially without knowing about the other transformations with which they shall be combined. This made the specifications prone to mistakes at the modularization and operationalization level. The specifications are available on GitHub[2]. For the evaluation, we employ the pairs of unidirectional specifications between PCM and UML, as well as between UML and Java. Although this induces only two bidirectional specifications, we have four transformations since both directions of the transformations have been specified independently. They have to interoperate correctly, may also contradict, and need to perform element matching. Thus, our scenario is prone to the same mistakes as a scenario with three or more BX.

The transformations realize rather trivial constraints between UML and Java. Most elements are mapped one-to-one, whereas multi-valued parameters and associations

[1] `http://vitruv.tools`
[2] `https://github.com/vitruv-tools/Vitruv-Applications-ComponentBasedSystems`

are mapped to collection types in Java. The relations between PCM and UML were proposed by Langhammer et al. [LK15]. Interfaces are equally represented, PCM components and data types are mapped to classes in UML. PCM components contain Service Effect Specifications (SEFFs), which are an abstraction of their behavior specification used for performance prediction. Those SEFFs are mapped to methods in UML and Java. In total, the transformations between PCM and UML react to 57 change types in PCM and 65 change types in UML, and the transformations between UML and Java react to 66 change types in UML and to 48 change types in Java to restore consistency in the other model.

In total, we have used 187 test cases that perform different kinds of relevant fine-grained changes in instances of all metamodels, such as insertions, modifications and deletions of all types of elements that have to be kept consistent. Additionally, we have simulated the construction of the Media Store system [SK16], which is a sophisticated case study system for the PCM. This system is available as a PCM model as well as Java code.

6.2 Methodology

Process

We executed the test cases on a transformation network, which we created as a combination of the existing transformations. They were executed until no further changes occurred. We then classified the occurring failures according to Subsection 4.1. Based on our categorization in Subsection 4.3, we traced back the failures to mistakes and fixed them according to the strategies discussed in Section 5. Failures can be hidden by others: For example, an incompatible constraint may produce no failure because the scenarios fail earlier due to missing element matching or vice versa. For this reason, we re-executed the process until no further failures occurred. Finally, we applied the transformations to the more complex Media Store construction case to validate that all mistakes were fixed.

Measurements

We measured the number of failures in each of the iterations. We relate the number of failures that we were able to categorize to the total number of recognized failures ($\mathit{identifiedFailureRatio} = \frac{\#\ \mathit{of\ categorized\ failures}}{\#\ \mathit{of\ total\ failures}}$) to show completeness of the identified failure types. This metric is rather weak, because it does not identify whether a failure is categorized correctly. We therefore relate the total number of resolved failures, which are those that do not occur in the subsequent iteration anymore, to the number of detected failures ($\mathit{resolvedFailureRatio} = \frac{\#\ \mathit{of\ resolved\ failures}}{\#\ \mathit{of\ total\ failures}}$). If a failure disappears after fixing the causing mistakes, the classification of the failure and also the relation to the causing mistake was correct. Therefore, this metric gives an indicator for both completeness of the identified failure types and the relation of mistakes to failures.

6.3 Results

We had to perform two iterations of the previously described process. In the first iteration, we faced failures due to mistakes at the operationalization level, whereas in the second iteration only failures due to remaining mistakes at the modularization level occurred. We have tagged the states before and after the evaluation process in the GitHub repository[2].

In the first iteration, all 187 tests failed. The reason was that all transformations assumed that new elements are only created by the user or the transformation itself. In consequence, we observed multiple instantiations and insertions in 187 cases, which we could trace back to 35 missing matchings of elements in the transformations. After adding appropriate matchings, all these failures disappeared in a second iteration, so for the first iteration $\mathit{identifiedFailureRatio} = \mathit{resolvedFailureRatio} = 1$, since all detected failures were identified and resolved.

In the second iteration, 5 new failures occurred. Three of them were diverging loops, which were caused by a namespace repeatedly prefixed to the name of classes, interfaces and enumerations in Java. The causing mistakes were incompatible constraints: The Java model contains the fully qualified name of a class, whereas the UML model only contains the simple name, which was correctly propagated from UML to Java, but the namespace prefix was not removed in the opposite direction. The two other failures were alternating loops, which were caused by alternations of element visibilities. For methods and constructors, the visibilities were repeatedly changed due to an inconsistent mapping of visibilities from UML to Java and vice versa. After fixing those mistakes, no failures remained. So we again have $\mathit{identifiedFailureRatio} = \mathit{resolvedFailureRatio} = 1$, since all detected failures were identified and resolved.

Summarizing, we were able to classify and resolve all failures in the case study and trace them back to mistakes with our classification in Section 4. This demonstrates the applicability of our categorization and is an indicator for the completeness and correctness of our catalog. Most important, we did not find any failures that were caused by mistakes at a different specification level than we expected. To further validate the catalog, we should apply it to further case studies. It is however hard to find existing, independently developed transformations between at least three metamodels. They would have to be developed in a schema similar to the one proposed by Kramer et al. [Kra+16].

7 Related Work

Macedo et al. [MJC17] 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 Triple Graph Grammars (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 2.

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+07], 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 Subsection 5.2. 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.

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 Answer Set Programming (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.

8 Conclusion

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.

References

[Cle+19] A. Cleve, E. Kindler, P. Stevens, and V. Zaytsev. "Multidirectional Transformations and Synchronisations (Dagstuhl Seminar 18491)". In: *Dagstuhl Reports* 8.12 (2019), pp. 1–48. DOI: 10.4230/DagRep.8.12.1.

[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. DOI: 10.1007/978-3-319-89363-1_2.

[Era+08] R. Eramo, A. Pierantonio, J. R. Romero, and A. Vallecillo. "Change Management in Multi-Viewpoint System Using ASP". In: *Enterprise Distributed Object Computing Conference Workshops, 2008 12th*. 2008, pp. 433–440. DOI: 10.1109/EDOCW.2008.22.

[Era+12] R. Eramo, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio. "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. DOI: 10.1007/s10270-010-0170-z.

[Gle17] J. Gleitze. "A Declarative Language for Preserving Consistency of Multiple Models". Bachelor's Thesis. Karlsruhe Institute of Technology (KIT), 2017. DOI: 10.5445/IR/1000076905.

[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. DOI: 10.1145/2489861.2489864.

[Kla16] H. Klare. "Designing a Change-Driven Language for Model Consistency Repair Routines". Master's Thesis. Karlsruhe Institute of Technology (KIT), 2016. DOI: 10.5445/IR/1000080138.

[Kla18] H. Klare. "Multi-model Consistency Preservation". In: *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS 2018*. 2018, pp. 156–161. DOI: 10.1145/3270112.3275335.

[Kra+16] M. E. Kramer, G. Hinkel, H. Klare, M. Langhammer, and E. Burger. "A Controlled Experiment Template for Evaluating the Understandability of Model Transformation Languages". In: *Proceedings of the Second International Workshop on Human Factors in Modeling*. (Saint Malo, France). Vol. 1805. CEUR-WS.org, 2016, pp. 11–18.

[Kra17] M. E. Kramer. "Specification Languages for Preserving Consistency between Models of Different Languages". PhD thesis. Karlsruhe Institute of Technology (KIT), 2017. 278 pp. DOI: 10.5445/IR/1000069284.

[Lan+14] K. Lano, S. Kolahdouz-Rahimi, I. Poernomo, J. Terrell, and S. Zschaler. "Correct-by-construction synthesis of model transformations using transformation patterns". In: *Software & Systems Modeling* 13.2 (2014), pp. 873–907. DOI: 10.1007/s10270-012-0291-7.

[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.

[Lúc+13] L. Lúcio, S. Mustafiz, J. Denil, H. Vangheluwe, and M. Jukss. "FTG+PM: An Integrated Framework for Investigating Model Transformation Chains". In: *SDL 2013: Model-Driven Dependability Engineering*. Springer Berlin Heidelberg, 2013, pp. 182–202. DOI: 10.1007/978-3-642-38911-5_11.

[Mal+10] I. Malavolta, H. Muccini, P. Pelliccione, and D. A. Tamburri. "Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies". In: *IEEE Transactions of Software Engineering* 36.1 (2010), pp. 119–140. DOI: 10.1109/TSE.2009.51.

[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.

[MJC17] 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. DOI: 10.1109/TSE.2016.2620145.

[Obj14] Object Management Group (OMG). *OMG Object Constraint Language (OCL)*. Version 2.4. 2014.

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

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

[RHK16] R. H. Reussner, J. Henss, and M. Kramer. "Introduction". In: *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016. Chap. 1, pp. 3–15.

[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. DOI: 10.1007/978-3-540-69927-9_12.

[SK16] M. Strittmatter and A. Kechaou. *The Media Store 3 Case Study System*. Tech. rep. 2016,1. Faculty of Informatics, Karlsruhe Institute of Technology, 2016. DOI: 10.5445/IR/1000052197.

[Ste07] P. Stevens. "Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions". In: *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2007, pp. 1–15. DOI: 10.1007/978-3-540-75209-7_1.

[Ste17] P. Stevens. "Bidirectional Transformations in the Large". In: *ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2017, pp. 1–11. DOI: 10.1109/MODELS.2017.8.

[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. DOI: 10.1145/3239372.3239378.

[Stü+18] P. Stünkel, H. König, Y. Lamo, and A. Rutle. "Multimodel Correspondence Through Inter-model Constraints". In: *Conference Companion of the 2Nd International Conference on Art, Science, and Engineering of Programming*. Programming'18 Companion. ACM, 2018, pp. 9–17. DOI: 10.1145/3191697.3191715.

[TA15] F. Trollmann and S. Albayrak. "Extending Model to Model Transformation Results from Triple Graph Grammars to Multiple Models". In: *Proceedings of the 8th International Conference on Theory and Practice of Model Transformations*. Springer International Publishing, 2015, pp. 214–229. DOI: 10.1007/978-3-319-21155-8_16.

[TA16] F. Trollmann and S. Albayrak. "Extending Model Synchronization Results from Triple Graph Grammars to Multiple Models". In: *Proceedings of the 9th International Conference on Theory and Practice of Model Transformations*. Springer International Publishing, 2016, pp. 91–106. DOI: 10.1007/978-3-319-42064-6_7.

[Van+07] B. Vanhooff, D. Ayed, S. Van Baelen, W. Joosen, and Y. Berbers. "UniTI: A Unified Transformation Infrastructure". In: *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2007, pp. 31–45. DOI: 10.1007/978-3-540-75209-7_3.

[Wag+11] D. Wagelaar, M. Tisi, J. Cabot, and F. Jouault. "Towards a General Composition Semantics for Rule-Based Model Transformation". In: *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2011, pp. 623–637. DOI: 10.1007/978-3-642-24485-8_46.

[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. DOI: 10.1007/978-3-540-69927-9_11.

[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. DOI: 10.1007/s10270-009-0134-3.

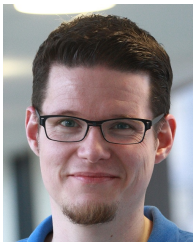
[Yie+09] A. Yie, R. Casallas, D. Wagelaar, and D. Deridder. "An Approach for Evolving Transformation Chains". In: *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2009, pp. 551–555. DOI: 10.1007/978-3-642-04425-0_42.

[Yie+12] A. Yie, R. Casallas, D. Deridder, and D. Wagelaar. "Realizing Model Transformation Chain interoperability". In: *Software & Systems Modeling* 11.1 (2012), pp. 55–75. DOI: 10.1007/s10270-010-0179-3.

About the authors

Heiko Klare is a doctoral researcher at the chair for Software Design and Quality (SDQ) at the Karlsruhe Institute of Technology (KIT) since 2016. His research interests involve model consistency preservation, consistency-aware collaborative modelling, variants and versions management, as well as view-based modelling and development processes. Contact him at klare@kit.edu, or visit https://sdq.ipd.kit.edu/people/heiko-klare/.

Torsten Syma wrote his Master's thesis at the chair for Software Design and Quality (SDQ) about model transformation composition and interoperability issues in transformation networks. He received his M.Sc. from the Karlsruhe Institute of Technology (KIT) in 2018. Contact him at torsten.syma@student.kit.edu.



Erik Burger is a postdoctoral researcher and head of the model-driven development group at the chair for Software Design and Quality (SDQ) at the Karlsruhe Intitute of Technology (KIT). He received his PhD in 2014. His research interests are view-based development, metamodel evolution and model co-evolution, as well as distributed development. Contact him at burger@kit.edu, or visit https://sdq.ipd.kit.edu/people/erik-burger/.

Ralf Reussner is a computer science professor at the Karlsruhe Institute of Technology (KIT). He holds the chair for Software Design and Quality (SDQ) since 2006 and heads the Institute for Program Structures and Data Organization. His research group works in the interplay of software architecture and predictable software quality as well as on view-based design methods for software-intensive technical systems. Contact him at reussner@kit.edu, or visit https://sdq.ipd.kit.edu/people/ralf-reussner/.