Classifying and Avoiding Compatibility Issues in Networks of Bidirectional Transformations

Timur Sağlam[1], Heiko Klare[1]

[1]*Karlsruhe Institute of Technology, Am Fasanengarten 5, Karlsruhe, 76131, Germany*

Abstract

Consistency preservation allows keeping interdependent models consistent with each other. A common approach for preserving consistency between two models is using a bidirectional transformation (BX). To keep multiple models consistent, BXs can be combined into a network of transformations. Since developing each BX requires individual domain knowledge, they could be independently developed. Moreover, a BX might be reused in different contexts and therefore not be specifically designed for the context at hand. Consequentially, networks of independently developed BXs are prone to compatibility issues. However, BXs need to be compatible by design to facilitate their reuse. Thus, we investigate such compatibility issues in networks of BXs. For that, we conduct a case study, in which we gradually build a circular network with three metamodels and three independently developed BXs. We introduce a classification for such issues and propose avoidance patterns to prevent compatibility issues in networks of BXs by design. This work helps transformation developers, as it improves awareness for possible issues in networks of independently developed BXs and enables them to prevent specific issues by design.

Keywords

Model Transformation, Transformation Composition, Transformation Network, Model Consistency

1. Introduction

Consistency preservation deals with the problem of keeping interdependent models consistent with each other when one is modified. Especially in agile development, models may often change, and interdependent models must be updated accordingly. For two models, consistency preservation mechanisms are often based on an incremental bidirectional transformation (BX) [1]. One approach for keeping multiple models consistent is combining multiple BXs to networks of transformations, where each BX is concerned with keeping two models consistent [2]. Models can also be transitively kept consistent. When one model is changed, the BXs are executed iteratively, until no BX execution will lead to further changes. Changes, therefore, propagate through the network from model to model. Since the development of BXs requires individual domain knowledge, they can be independently developed by different experts [3]. Additionally, they are often reused in different contexts and, therefore, not specifically designed for the context at hand. Thus, networks of independently developed BXs are prone to *compatibility issues* [4]. However, to facilitate the reuse of BXs, they need to be *compatible* by design. Multiple BXs are compatible if there is at least one order of execution that delivers the expected result under the assumption that each BX on its own produces the expected result.

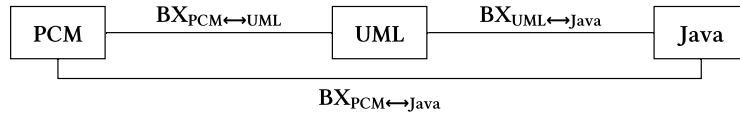CEUR Workshop Proceedings (CEUR-WS.org)

Figure 1: A network of independently developed BXs based on three pre-existing metamodels.

For example, there may be two chains of BXs that relate the same metamodels across different other metamodels. Yet, they may relate the elements in different ways. We refer to these chains of BXs in a network as *paths* in the network. Especially when multiple paths in the network allow propagating the same changes between two models, failures due to incompatibilities can be caused by redundant change propagation. For example, the duplicate creation of two semantically identical model elements: Two BXs are not designed with the expectation in mind that other BXs modify the target model. When a model element is manually created in the source model, changes are propagated along both paths. Since both BXs cannot find a corresponding target element, they each create that element, thus leading to two duplicate elements. These failures appear during the use of the network when the BXs are executed. They lead to inconsistencies between models, and can, in some cases, lead to non-terminating change propagation. Failures during productive use will not only render the consistency preservation unusable but might also affect the systems that depend on the models. It is not feasible to remove paths from the network to avoid issues, as usually one path only partly subsumes another.

Previous work [4] did only consider linear networks without multiple paths between models. However, issues as the previously mentioned duplicate element creation, by definition, only arise in more complex networks with multiple paths that allow redundant change propagation. To the best of the authors' knowledge, there is no work on the systematic investigation of issues that arise in networks of independently developed BXs due to their incompatibility. In this paper, we introduce a classification for such issues, which concerns the knowledge that is required to avoid them. The classification is based on the issues we observed in a case study on networks of BXs. In this case study, we gradually built up the network illustrated in Figure 1. We use three metamodels: The UML [5] class metamodel, a metamodel for Java source code, and the Palladio component model (PCM) [6, 7] for performance analysis of software architectures. We use three independently developed BXs, which are systematically tested for single use, however not for network use. We employ the VITRUVIUS framework [8] as the underlying mechanism for consistency preservation. In this paper, we make the three following contributions:

Classification of Compatibility Issues (C1): We classify issues in networks of independently developed BXs with respect to the knowledge required to avoid them during the BX design.

Issue Prevention Patterns (C2): We identify patterns to systematically prevent individual compatibility issue classes during the design of a BX, one of which we present in detail.

Unavoidable Issues (C3): We discuss issues that cannot be systematically prevented by design and how to spot them while assembling a network of independently developed BXs.

2. Assumptions and Terminology

In the following, we discuss the assumptions this paper is based on and the terminology we use.
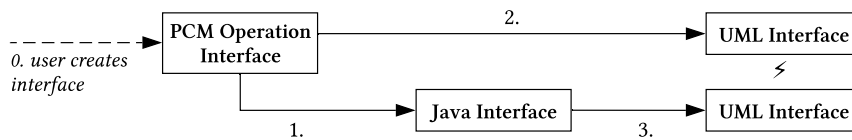
Figure 2: Duplicate creation of two identical UML interfaces, as changes are propagated from the PCM model along two paths: A direct one and a transitive one via the Java model.

2.1. Assumptions

We examine issues in networks of independently developed BXs, which are unlikely to arise at the same scale when a single expert designs multiple BXs specifically to keep a set of models consistent. However, in a large-scale system, a single expert cannot design all BXs. The more models need to be consistent, and the more heterogeneous the domains of these models are, the more domain knowledge is required to design the BXs. Thus, it is reasonable to expect the BXs to be developed independently by different experts [3]. Additionally, two further assumptions affect the compatibility of BXs in a network. First, networks of BXs encourage the use of existing BXs to assemble a network instead of creating them from scratch. Thus, BXs in a network might not be designed for being chained with all other transformations. Specifically, a BX might not be designed for scenarios where another BX changes the same target model. Second, even if the BX was designed for network use, it may be re-used in another network with different BXs, increasing the possibility for incompatible BXs. Finally, we assume the existence of a trace model, as used in QVT [9], which allows using trace links to track corresponding model elements across model instances.

2.2. Terminology

We discuss terminology regarding the differences of mistakes, faults, and failures, as well as what we understand as compatibility issues. We reuse terminology [4] for classifying issues in networks of BXs to differentiate issues according to their cause, manifestation, and impact:

1. ***Mistakes*** are a wrong judgment made by a person, for example, a developer. Mistakes might occur during the design or implementation of software or BXs.
2. ***Faults*** are the manifestation of the mistakes in an artifact, for example, the code or a BX.
3. ***Failures*** are how faults show themselves, for example, during code execution or change propagation in a network of BXs.

Mistakes can potentially manifest themselves as faults. A fault can potentially show itself through the occurrence of failures. However, not every mistake leads to a fault, and not every fault causes a failure. When repairing a fault to prevent the corresponding failures, the corresponding mistake usually needs to be understood. **Compatibility Issues** are issues based on the interaction of BXs that are not designed with each other in mind. Thus, the BXs are incompatible. For example, two or more paths in the network may relate the same metamodels. Yet, they may relate the elements in different ways, leading to issues upon transformation execution. They especially arise when different paths in the network allow propagating the same changes between two models. For example, in our case study, we frequently encountered a compatibility issue that leads to the creation of duplicated, but semantically identical model elements. We observed this when a user manually creates an interface in the PCM component model, as depicted in Figure 2. The BXs define that for each PCM interface, a corresponding

```
1    routine createUmlInterfaceIfRequired(java::Interface jInterface) {
2      match {
3        val uModel = retrieve uml::Model
4        require absence of uml::Interface corresponding to jInterface
5      }
6      action {
7  -      call createUmlInterface(jInterface)
8  +      call {
9  +        val uInterface = uModel.findUmlType(jInterface.name, namespace, Interface)
10 +        if(uInterface === null) {
11 +          createUmlInterface(jInterface)
12 +        } else {
13 +          addTypeCorrespondence(jInterface, uInterface) // add missing trace link
14 +        }
15 +      }
16      }
17    }
```

Listing 1: Routine responsible for the creation of a UML interface for a corresponding Java interface. The developer should have written the plus-prefixed lines instead of the minus-prefixed lines.

UML interface should be created, as well as a corresponding Java interface. However, after the consistency preservation terminates, the UML class model contains two identical interfaces. Both BXs do not expect another BX to modify *their* target model. Thus, if no UML interfaces can be located through trace links, non-existence is assumed. As the first BX to be executed creates such a UML interface, this assumption is incorrect for the second BX. The BX developers made the mistake of not considering the interaction between BXs. Thus, the BXs are not designed to check on external creation whenever the trace links are missing. Listing 1 illustrates this mistake. It shows a *routine* written in the Reactions language [10] which creates a UML interface whenever a Java interface is created (see Figure 2). The minus-prefixed lines show what the developer wrote, while the plus-prefixed lines show what a developer should have written to avoid any duplication failures. The trace link check in Line 4 is sufficient to check for existence in the single use of the BX. However, to check for external creation, Line 9 is required, as the absence of a trace link does not guarantee the absence of the corresponding UML interface.

3. Methodology

We conduct a case study on compatibility issues in networks of independently developed BXs. In this case study, we incrementally assemble the network illustrated in Figure 1 based on three pre-existing metamodels and three pre-existing, independently developed BXs. During each increment, we execute fine-grained scenarios to observe the consistency restoration. Some approaches try to find an appropriate execution order for networks of BXs [11]. However, as finding an appropriate strategy for the execution order proves challenging [12], we aim to facilitate the compatibility of BXs for any arbitrary execution order. Thus, we execute the scenarios multiple times for different execution orders. Our scenarios do not involve concurrent modification of models [13], as even the modification of a single model can lead to failures in networks of independently developed BXs. The catalog of mistakes, faults, and failures is published separately in [14]. From this catalog, we derive the classification of compatibility issues (C1) and then develop avoidance patterns (C2), which can be applied pro-actively during the BX design to systematically prevent compatibility issues.

3.1. Network of Transformations

The case study is developed for the Ecore-based VITRUVIUS framework [8]. This framework preserves consistency between models in a delta-based way, which means fine-grained sequences of atomic changes to the source model are transformed by the framework with the correlating BXs to the resulting change sequence for the target model. In the VITRUVIUS framework, BXs are written in the Reactions language [10], an imperative, domain-specific language for defining unidirectional consistency preservation. This means that a single BX is designed as two separate unidirectional incremental transformations. The developer needs to ensure that a BX is internally consistent. As illustrated in Figure 1, we base the case study network on three metamodels. First, the Palladio Component Model (PCM) [6, 7] is a model for performance prediction of software based on their component-based architecture description. PCM models offer a structural view on a software system at the component level. It allows, among other aspects, modeling components, their interfaces, and the assembly of a system. Second, the UML [5] metamodel with UML class models offering a structural view on the object-oriented architecture of software systems. They contain information about classes, their properties, and their relations. Third, the Java Model Printer and Parser (JaMoPP) [15] is a model representation of the Java programming language. JaMoPP allows bridging between Java models and arbitrary Java code. The network thus enables the consistency preservation between Java code, UML class models, and component-based architecture models.

Between each pair of metamodels, we employ an incremental BX. These independently developed BXs are systematically tested for their intended single use, however not for use in a network. The BXs sometimes require user interactions when consistency relations are ambiguous. Thus, these BXs are not necessarily fully automated. For the consistency preservation between the UML and Java metamodels, a BX keeps the structural information of the Java code consistent with the correlating information of UML class models [8]. Behavioral information such as the semantics of method bodies is not transformed. For the PCM and UML metamodel, a BX designed by Syma [16] keeps PCM repository models consistent with UML class models. For example, it keeps the required and provided interfaces of the components consistent with the correlating UML interfaces. For PCM and Java, a BX designed by Langhammer [17] keeps PCM repository models consistent with Java code. For example, components are kept consistent with the Java classes that implement them. Provided interfaces are kept consistent with Java interfaces and the correlating realization relation; required interfaces are kept consistent with fields in the implementation classes correlating to the component. Some concepts are only shared between two of the metamodels at a time, while others are shared among all three. As an example, Figure 3 shows how the concept of an assembly context between two components is represented in instances of all three metamodels. Note that each BX consists of some consistency relations that are not covered by the other BXs, meaning that it is not possible to remove a BX to transform the circular network into a linear one without losing some consistency constraints.

3.2. Scenarios

We used a set of 39 fine-grained scenarios to observe the consistency restoration in the network. These scenarios were initially designed as test cases to ensure the functionality of the BX between PCM and UML models by Syma [16]. We extended these test cases to include the Java
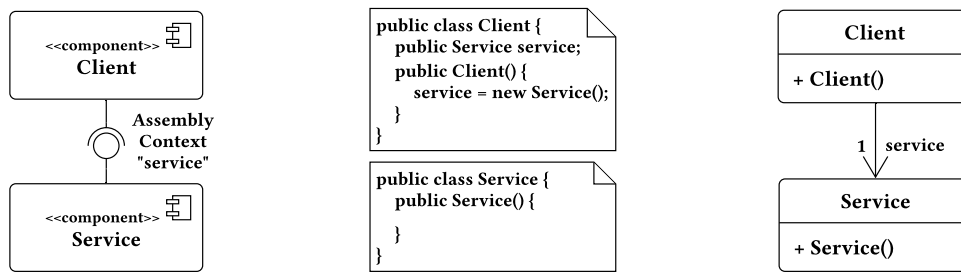
Figure 3: Different representations of the assembly of two components on the instance level in PCM models (left), Java code (middle), UML class models (right).

metamodel. Each scenario concerns the consistency preservation of a few model elements. The scenarios systematically cover create operations, update operations, and delete operations for individual consistency constraints. These operations affect both model elements themselves as well as their properties. While a few update operations are not represented by the scenarios, however, the create operations cover all relevant information that can also be changed by an update operation. In these scenarios, a single model is altered, resulting in an inconsistent state. Then, the consistency preservation mechanism executes the BXs one by one in an arbitrary order to restore consistency. The execution stops when no BX execution leads to further changes. Afterward, the testing framework automatically checks whether the model instances match the expectations specified by the scenario. Failures are either inconsistent model states, crashes during the consistency preservation, or non-terminating BX execution.

3.3. From Failures to Mistakes

We analyze the encountered failures one by one and derive the underlying faults that caused them. For example, when a duplicate element is created, we backtrack the propagated change sequences from the failure to pinpoint the fault. The fault is not necessarily located in the BX that was executed when the failure occurred. In the next step, we infer what mistake led to that fault. For example, a duplicate creation due to a missing existence check may have been the consequence of the BX developer not considering the use of the BX in a network. Reasoning retroactively on what mistakes lead to a fault in the BXs can only achieve a limited certainty since this is based on assumptions. However, we are interested in what knowledge is required to avoid a class of mistakes in the worst case. Even when a complex fault might have been caused by a mere technical mistake, we want to know what the minimal knowledge is to systematically avoid this mistake every time. Thus, it is acceptable to reason with some uncertainty.

4. Classification of Compatibility Issues

In this section, we present our classification of issues in networks of BXs (C1). In general, we want to distinguish two types of mistakes: Mistakes that can be prevented during the design of the BXs and mistakes that cannot be prevented at design time and, therefore, need to be dealt with when the BXs are assembled into a network. To systematically avoid mistakes during the design of the BXs, it is also essential to know which knowledge is required to do so. We present different categories regarding the knowledge required for avoidance in Table 1. Mistakes that cannot be systematically avoided during the BX design generally require in-depth

Table 1
Systematic avoidability of mistakes during the BX design and the knowledge to avoid the mistakes.

Avoidability during BX Design	Required Knowledge
Systematically Avoidable	Technical Knowledge
	Transformation Knowledge
	Possible Use of the BX in a Network
Not Systematically Avoidable	Specific BX Interaction in a Network

knowledge about the specific network and how BXs interact in this network. This knowledge is not available at design time for a network in which a BX is reused. We distinguish mistakes that can be avoided during the BX design into three categories according to the knowledge required to do so. First, *technical knowledge* such as knowledge concerning the transformation language. Second, *transformation knowledge*, which includes all domain knowledge required to design a specific BX. Third, *network knowledge*, meaning that the BX might be used in a network and that it interacts with other BXs. To understand which mistakes correlate to the occurring failures, we need to trace the failures to the mistakes via the corresponding faults in the BXs. Thus, we propose three separate classifications for mistakes, faults, and failures.

4.1. Failures

A failure will show itself through inconsistent model states after the consistency restoration terminates or in some cases through non-terminating consistency restoration. The latter causes the models to continuously loop through several inconsistent states. We classify the failures based on the states of the models in the network of BXs. We base this classification on the comparison of the expected model states with the actual model states. Thus, we derive three failure categories, which are *excess elements*, *missing elements*, or *incorrect elements*.

Excess Elements contains all failures where the actual state of a model contains more elements than the expected state. We further distinguish three failure classes of excess elements: *Duplicate element creation* describes a failure where two semantically identical elements are created in the same model (as illustrated in Figure 2). This can result in the co-existence of duplicates with one element overwriting the other. *Accidental creation* describes all failures where an excess element was created, but not as a duplicate. *Missing deletion* describes a failure where an element should have been deleted to achieve a consistent state of all the models in the network.

Missing Elements contains all failures with fewer elements in the actual state of a model than correctly expected. We identify two failure classes with missing elements: *Unwanted element deletion* describes all failures where a model element was erroneously deleted. This means a BX deleted a model element that is required for a consistent state between the models. *Missing element creation* describes all failures where a model element was not created, although it should have been created to achieve a consistent state.

Incorrect Elements contains all failures where the elements themselves are not matching the expectations. We differentiate three failure classes with incorrect elements: *Incorrect property value* includes all failures where any property value of a model element differs from the expected value. *Misplaced element* contains all failures where either a model element was misplaced, or

Table 2
Classification of the failures according to the states of the models in the network.

Model State	Failures	Failure Class	Failures
Excess Elements	84	Duplicate Creation	82
		Missing Deletion	2
		Accidental Creation	0
Incorrect Elements	25	Incorrect Property Value	20
		Misplaced Elements	3
		Wrong Element Type	2
Missing Elements	10	Unwanted Deletion	6
		Missing Creation	4

a root element was persisted in the wrong location. *Wrong element type* includes all failures where an element created by a BX was not an instance of the expected metaclass.

Table 2 shows the distribution of encountered failures. The most frequent failures, accounting for 68.9% of all failures, are duplicate element creations. The second most frequent are incorrect property values with 16.8%, which mostly affected the names or namespaces of model elements. This can be explained by the importance of names in the models of the case study: As not all models use unique identifiers, elements are often identified by their names.

4.2. Faults

Each failure is caused by a fault, which is located in one or more BXs. Multiple failures can be caused by a single fault. Some faults are located in multiple BXs at once. They are based on how BXs interact which each other in a network. Thus, we derive the fault classification based on the scope of the fault that needs to be considered when repairing that fault. We derive three fault categories: A fault is either a *technical fault*, a *BX-internal fault*, or a *BX interaction fault*.

Technical Faults are introduced during the BX implementation, not during the BX design. They stem from the incorrect usage of the transformation language or incautious implementation. While these faults are less interesting regarding issues in networks of BXs, they show the importance of a cautious and methodical implementation.

BX-Internal Faults are faults where the scope does not exceed a single BX. Thus, the fault can be independently repaired without considering other BXs. This requires, in contrast to the technical faults, some domain knowledge about the consistency relations between the two affected metamodels. We identify three fault classes within the category BX-internal faults: *Missing change propagation* describes faults where a modification of the source model does not lead to a required modification of the target model. *Unwanted change propagation* includes all faults where a modification of the source model mistakenly propagates to a modification of the target model. *Incorrect change propagation* are faults where a modification of the source model propagates to an incorrect modification of the target model.

BX Interaction Faults describe faults that can only be repaired at the network level because multiple BXs conflict with each other due to incompatibilities. We identify three fault classes

Table 3
Classification of the faults according to their scope.

Fault Scope	Faults	Fault Class	Faults
Technical	5	Technical Fault	5
BX-Internal	6	Missing Change Propagation	4
		Unwanted Change Propagation	2
		Incorrect Change Propagation	0
BX Interaction	18	Element Creation Conflicts	12
		Property Convention Conflict	4
		Deviating Root Management	2

within the category of BX interaction faults: *Element creation conflict* describes faults where multiple BXs creating elements of the same type in the same model leads to conflicts. This includes faults regarding missing checks for external element creation. *Property convention conflict* contains all faults where two or more BXs deviate in how they apply conventions regarding property values to semantically identical elements. For example, two BXs naming identical elements in the same model differently. *Deviating root management* describes faults where different BXs deviate in how they manage the root element of the same model.

Table 3 shows the distribution of the encountered faults. The majority of faults regard transformation interaction. Preemptively avoiding these faults requires the BX developer to reason in advance on possible BX interaction. As this might not be on a developer's mind when designing a BX, it explains why these faults are so common. Element creation conflicts between transformations, which frequently lead to failures based on duplicate element creation, are the most common faults and account for 41,3% of all observed faults.

4.3. Mistakes

As established, it is preferable to avoid mistakes as early as during the design of the BX. This depends on the knowledge that is required to avoid mistakes. Thus, we categorize the mistakes accordingly and derive three mistake categories: A mistake is based on either a lack of *Technical knowledge*, a lack of *Transformation Knowledge*, or a lack of *Network Knowledge*.

Lack of Technical Knowledge includes mistakes that occur due to missing technical knowledge and due to the incautious design of a BX. Preventing these technical mistakes requires no BX-specific domain knowledge.

Lack of BX-specific Knowledge describes all mistakes where the knowledge on a single BX is required to avoid the mistake. This includes understanding the domain of the transformed models and how to transform between them. We identify three mistake classes regarding a lack of BX-specific knowledge: *Disregarded source model modification* describes all mistakes where a modification to the source model was not considered as a relevant trigger for the BX to modify the target model. *Disregarded property convention* describes mistakes where conventions on the property values of model elements are not considered. For example, the naming schemes of the source and target elements are not considered when transforming names

Table 4

Classification of the mistakes according to the knowledge to avoid them.

Knowledge	Mistakes	Mistake Class	Mistakes
Technical	5	Incautious BX Design	5
BX-specific	8	Disregarded Source Model Change	3
		Disregarded Property Convention	3
		Overlooked Intra-Model Dependency	2
Network	16	Overlooked BX Interaction	13
		Unpredictable BX Interaction	3

between two correlating elements. *Overlooked intra-model dependencies* describes mistakes where a modification of one element should be accompanied by a modification of another element in the same model, as the second element depends in some way on the first. Ideally, the model itself should be responsible for intra-model consistency. However, when using pre-existing metamodels, the BXs are nonetheless required to deal with such idiosyncrasies.

Lack of Network Knowledge includes all mistakes that require network-specific knowledge to be avoided. At a minimum, this includes considering the potential interaction of BXs in a network. However, it also includes details on the structure of the network and how changes are propagated in the network. Thus, these mistakes are based on the interaction of multiple BXs. We identify two mistake classes regarding lacking network knowledge: *Overlooked BX interaction* describes mistakes where the interaction of a BX with other BXs was either not considered at all or partly misunderstood. *Unpredictable BX interaction* describes mistakes where the interaction of a BX with other BXs could not be predicted during the BX design.

Table 4 shows the distribution of the encountered mistakes. Most mistakes are based on network knowledge. In total, 44,8% of all mistakes are based on overlooked BX interaction. This suggests that BX developers do not consider that other BXs might concurrently modify the same models. Thus, many failures could be avoided if the BX developers are aware of the possible BX interaction. Mistakes based on unpredictable BX interaction require detailed network knowledge to be avoided and thus cannot be systematically prevented during the BX design, which matches the distinction made in Table 1.

5. Preventing Mistakes by BX Design

Avoiding failures by systematically preventing the correlating mistakes during the BX design allows reliably reusing the BX in an arbitrary network. Vice versa, mistakes that cannot be prevented at design time can cause failures during the BX execution if they are not detected when assembling the network. While every fault can eventually be fixed, not every correlating mistake can be systematically avoided during the design of the BX. All technical mistakes can be, by definition, systematically prevented during the design of the BXs. Moreover, all encountered mistakes regarding BX-specific knowledge can also be systematically prevented during the BX design. This requires the domain knowledge to design the BXs firsthand. For the mistakes that require network knowledge, however, we need to make a distinction: Mistakes

based on unconsidered BX interaction can be prevented during the BX design. Others cannot be systematically prevented during the BX design, as detailed knowledge of the network is required at design time to avoid them. In our case study, 13 of the 16 mistakes based on network knowledge can be systematically prevented during the BX design. Since these 16 mistakes are responsible for 67.2% of the 119 observed failures, it is crucial to systematically avoid these mistakes. We identified three patterns to systematically avoid mistakes during the design of the BXs. We discuss one of them in detail, while the others are published separately [14]. As for C3, we discuss the mistakes that cannot be systematically prevented during the design of a BX.

5.1. Find-or-Create Pattern

When a network allows for the creation of semantically identical model elements via different paths, the BXs need to ensure that there is no duplicate element creation or overwriting of elements (see Figure 2). To achieve that, the BXs need to identify existing elements. A strategy for matching model elements at three levels is proposed by Klare et al. [4]: First, through *explicit unique* information like trace links. Second, through *implicit unique* information such as element names. Third, through *non-unique* information heuristics, for example, based on ambiguous information. When designing a BX for single use, model elements are either created by the user or the BX itself. To track corresponding model elements, a BX usually creates trace links. The absence of a trace link means there is no corresponding element, so the BX needs to create it. In a network of BXs, a trace link might be absent as another BX might have created the correlating element. Matching model elements is always required when creating model elements. Therefore, we can define a pattern for this matching, which then needs to be implemented in the BXs wherever elements are created. We thus define the *find-or-create pattern*, which is using implicit unique information to identify corresponding elements whenever the explicit unique information is missing. It may use any model information, such as element names or the containment structure of the models. The find-or-create pattern works as follows: First, if there is an explicitly corresponding element, do nothing. In that case, the element and the correspondence information exist. Second, if there is a target element without the explicit correspondence information, retrieve it through containment structure and unique identifying information. Then, add the missing correspondence information. Third, if no such target element exists, create the element and the correspondence information. This concludes the find-or-create pattern. Listing 1 illustrates an implementation of the pattern in the Reactions language [10]. In practice, several typical scenarios arise of how to retrieve a target element without explicit correspondence information. As other BXs create candidate elements, not all elements might be explicitly known. Additionally, the target element needs to be identified among all the candidates. We present three containment-based identification approaches:

Identification via Target Containment When the parent element of the target element is known to the BX, the containment relation between them can be leveraged to identify the corresponding element. If the containment relation is a single element containment, the corresponding element can be directly retrieved. If it is a multiary containment, additional identifying information, such as a name, can be used to identify the correct element. As an example, when locating the corresponding UML classifier of a PCM component, the UML package that contains the classifier is known, as it is required to insert the classifier in case no corresponding element exists. To identify the classifier amongst all classifiers contained by the

UML package, the name of the PCM component can be used.

Identification via Parent Correspondence If the parent element is not directly known, it is possible to utilize the parent element of the source element and retrieve its correlating counterpart in the target model. This target parent can then be used to identify the corresponding target element. Again, through the containment relation directly for a single containment, or with additional information for a multiary containment. As an example, when locating the corresponding Java interface for a UML interface, the Java package that corresponds to the UML package containing the UML interface can be retrieved with trace links or other explicit information. Then, the corresponding Java interface can be identified amongst all Java interfaces contained in the Java package using the UML interface name.

Identification via Model Traversal If locating the parent of the target element is not feasible, a corresponding element can be identified through additional identifying information by traversing the target model. As an example, the used Java metamodel represents each package as an individual model and provides no containment structure between them. Thus, the two previous approaches are not feasible. However, all elements of the UML model are contained under a single root. Consequentially, the corresponding UML package can be identified by traversing the UML model and comparing the names of the packages with the Java package namespace, which contains the names of all parents of the package.

5.2. Unpreventable Mistakes
We found three mistakes during the case study that cannot be systematically prevented during the BX design. Preventing them would require knowledge on the interaction of the BXs in the network, which is not always available during the design of a BX. The developers are forced to restrict the behavior of the BX during its design. They are forced to decide among several options, without an inherently correct choice. Different BXs in a network may implement different incompatible options, leading to failures when used in the network. However, these mistakes might not lead to failures when these incompatible decisions are recognized when assembling the network. Then, the faults can be resolved by implementing uniform decision-making across all BXs. Ideally, transformation developers should avoid making these choices or similar constraints whenever possible. If forced to make a choice, it is essential to consider how other BXs might choose. Moreover, they need to be aware of these faults to recognize and resolve them when assembling a network of BXs. The three unpreventable mistakes observed in the case study belong to two mistake types. Note that there may be more such types, which are yet to be observed. In the following we discuss both mistake types:

Mismatching Root Element Management The UML metamodel utilizes a root element to contain all model elements. Each BX that transforms to UML models needs to check if a root element exists and create it if it does not. For that, the BXs used in the case study make assumptions on the name and location of the root element. In the case study, one BX requires the user to specify the name and location of the UML root element, while another BX keeps track of root elements through trace links. These deviating ways of managing UML root elements based on different assumptions lead to failures during the execution of the BXs. The first BX to be executed creates a root element based on the information specified by the user. When the other is executed, trace links do not exist. Thus, a second root element is erroneously created.
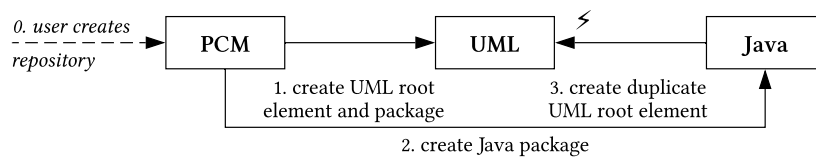
Figure 4: Mismatching root element management that results in a duplication failure.
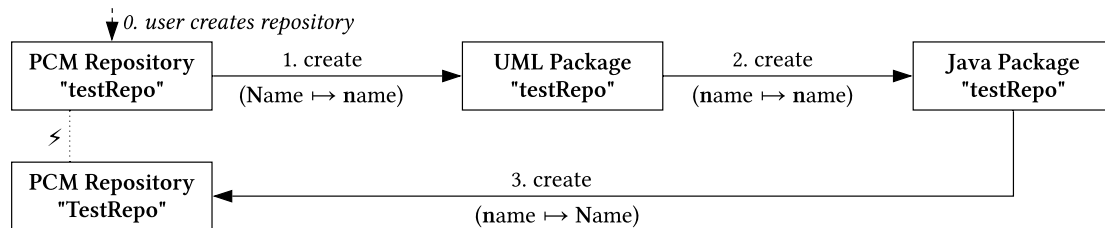

Figure 5: Mismatching element names cause the creation of two repositories, as the BX between PCM and Java enforces capitalized repository names.

This failure is illustrated in Figure 4. While each BX is functioning correctly in single use, their interaction in a network leads to failures. This failure cannot be systematically prevented during the design of a single BX, as it might not be clear if and how other BXs manage root elements. More specifically, this mistake cannot even be prevented with the find-or-create pattern, as a corresponding root cannot be reliably identified due to the potentially infinite number of possibilities for its location and name.

Mismatching Element Naming In the case study, the BXs ensure the consistency of UML and Java packages, as well as PCM Repositories. However, different naming conventions apply to these three element types. Both Java and UML packages should have lowercase names, which is enforced by the BXs. The PCM metamodel does not enforce a naming convention for repositories. Figure 5 shows how this leads to a failure. Initially, a PCM repository with a lowercase name is created manually. This change leads to the automatic creation of a UML package with the same name, as the name already is in lowercase. Consequentially, this new change then leads to the creation of a Java package with an identical name. Third, this change propagates back to the PCM model. As no trace links exist yet, the BX checks if a corresponding PCM repository exists with a matching name. However, because it expects an uppercase name, no corresponding repository is located and a duplicate repository with a capitalized name is erroneously created. While BX developers have a uniform understanding of the naming convention, *how* they enforce it may vary. This can only be systematically avoided with cooperation among the developers. However, this is not possible when a BX is reused in a different network.

6. Comparison with Previous Work

Our work extends the case study from Klare et al. [4] and analyzes issues in a more complex network. Klare et al. utilize the same case study setup, but they assemble a network with linear topology. As depicted in Figure 1, we use an additional BX to assemble a circular network that is fully connected. The network of Klare et al. contains no redundant paths between models.

However, we observed failures, such as the duplication issue depicted in Figure 2, that can only appear in a network with redundant paths. The classification by Klare et al. explicitly excludes technical issues, as they require each BX to be implemented correctly. Klare et al. classify failures based on how the consistency preservation terminates, while we classify them based on the model states after the termination. Moreover, they classify faults according to the state of the consistency preservation specifications, while we classify them based on the scope to be considered for their repair. They classify mistakes based on three specification levels at which a consistency preservation mechanism can be conceptually defined: *Global* meaning multiary relations, *Modularization* meaning binary relations for model pairs, and *Operationalization* meaning the transitive flow of information in a network [4]. We classify mistakes based on the minimal knowledge required to systematically avoid them. While the global level explicitly captures n-ary consistency relations, we solely consider binary relations.

7. Threats to Validity

We discuss threats to the validity according to Runeson and Höst [18]. We consider construct validity as given, as the operational measures studied are the issues in networks themselves.

Internal Validity We conducted the case study carefully and systematically to guarantee its internal validity. We used pre-existing, independently developed BXs, which are systematically tested for single use. They are based on pre-existing metamodels from established frameworks and tools, each from a different context. We used pre-existing test cases as change scenarios to systematically cover the consistency constraints preserved by the BXs. We were able to classify all encountered mistakes, faults, and failures. We were also able to resolve all failures by repairing the underlying faults. This shows that our classification is, in itself, consistent. While we analyzed 119 failures, the correlating number of faults and mistakes is comparatively low. Thus, their relative frequency could vary depending on the case study. However, the frequency is less important, since the primary concern is *how many* of the mistakes can be avoided. Our results show that many mistakes can be avoided during the BX design.

External Validity Overfitting a classification makes it less applicable for other contexts. To counteract this, we design this classification as abstractly as possible. We also compare our classification to the one by Klare et al. [4]. While this confirms that we can classify the problems of both works interchangeably, Klare et al. used the same metamodels and a subset of our BXs, which could affect which issues occur. However, the classification is also applicable for other metamodels and BXs, as the categories were derived from domain-independent properties. The failure categories are based on the model states, which is thus applicable to any EMOF-based [19] metamodel. The fault categories are based on the scope of the fault of the BX level, which is independent of the case study and is thus applicable to any network of BXs. Analogously, the mistake categories are based on the knowledge required to avoid mistakes (technical knowledge, BX-specific knowledge, network knowledge) and thus also applicable to mistakes for any network of independently developed BXs. We argue that the high-level categories of the classification are complete, as they were derived as partitions of their respective supersets. For example, the mistake categories (technical knowledge, BX-specific knowledge, network knowledge) are partitions of the possible knowledge required to avoid a mistake. When categorizing a mistake according to our classification, it can, by definition, only be

in one of these categories, as there is no other knowledge required to design a network of BXs. Moreover, we classified all failures and faults from Klare et al. [4] with our classification, showing its completeness. Note that this does not necessarily mean that the fine-grained classes are complete. Especially for the faults, there might arise different faults in the category of BX interaction faults when looking at different networks of BXs. The avoidance patterns (C2) are designed abstractly, meaning they are independent of a specific transformation language and framework. They can thus be applied in different networks of BX.

8. Related Work

To the best of our knowledge, there is no other work that investigates compatibility issues in networks of independently developed BXs to contribute towards compatibility by design.

Consistency Preservation We investigate issues in networks of BXs for model consistency. Thus, our work relates to approaches for the consistency preservation of multiple models. Meier et al. [20] compare such approaches, one of them being VITRUVIUS [8], which we use in our case study. Macedo et al. [21] present a feature-based classification of model repair approaches concerning inter-model consistency. Pepin [22] and Klare et al. [23] discuss the decomposition of consistency relations to detect redundant information. This assists in finding incompatibilities in consistency specifications. This work assists in preventing such issues.

Bidirectional Transformations BXs are a well-researched subject [24, 25, 26, 27]. There are several approaches, languages, and tools to specify BXs. Triple Graph Grammars (TGGs) are a common technique for defining BXs [28, 29, 30], while delta-lenses [31] are a technique for delta-based BXs. Stevens [32] introduces an algebraic framework for BX, which focuses mainly on lenses. We use incremental BXs written in the Reactions language [10]. We also relate to model synchronization and concurrent editing, as in these cases duplicate creation of elements must also be avoided [13]. However, in our scenarios changes may not be conflicting, which usually requires merging techniques [33], thus making the problem significantly easier to solve.

Networks of BXs and Transformation Composition Networks of BXs are the focus of our research. Stevens [34] investigates how to split multiary consistency relations into multiple binary consistency relations. Some approaches try to find appropriate orders for the execution of BXs in networks [11]. Since finding an appropriate strategy for that proves challenging [12], we aim to facilitate the compatibility of BXs for any arbitrary execution order. Transformation composition allows building networks of BXs. Lano et al. [35] propose composition patterns for BXs, while we propose patterns to prevent mistakes in BX networks. There are composition approaches that integrate into transformation languages [36, 37, 38], while others treat BXs as black boxes. For transformation chaining, there are language-based approaches [39], and approaches that consider BXs as black-boxes, such as UniTI [40, 41]. In our work, the VITRUVIUS framework treats BXs as black boxes. However, while the black box approaches for BX composition and chaining modify the BXs, we aim for compatibility by BX design.

Multiary Transformations Multiary transformations are an alternative to networks of BXs [42]. Macedo et al. [43] propose an extension for QVT-R (OMG 2016) to specify multiary transformations, while Trollmann and Albayrak [44, 45] extend TGGs to support multiary transformations. Stevens [34] details how multiary consistency relations can be expressed

through binary consistency relations. Stevens also discusses *non-interference*, which defines that BXs modify a model without interfering with other BXs. However, for pre-existing BXs, we cannot rely on non-interference to avoid issues in networks of BXs. Commonalities metamodels allow reducing the number of BXs and potential issues. Gleitze [46] introduces a generic idea for commonalities metamodels, whereas DUALLy [47, 48] uses a domain-specific approach. Stünkel et al. [49] and Diskin et al. [50] discuss them from a theoretical viewpoint. We analyze networks of binary BXs, as defining multiary transformations for an increasing number of models becomes increasingly difficult. It is arguably easier to design multiple binary BXs.

9. Conclusion

There is little to no systematic knowledge on preventing issues in networks of independently developed BXs. Ideally, these issues should be prevented during the design of the BXs to facilitate reusing BXs in any network. If they cannot be avoided at design time, this can lead to failures during consistency preservation if their corresponding faults are not detected when assembling the network. To address this issue, we introduced a classification of compatibility issues in networks of independently developed BXs. Previous work uses a linear network without any redundant paths. However, some issues, such as duplications due to external creation, can only arise in networks with redundant paths. We conducted a case study, in which we gradually built a circular network with three metamodels and three pre-existing BXs based on the Vitruvius framework. We identified 29 mistakes made by BX developers, which manifest in 29 faults, which, in turn, cause 119 failures. Six of these mistakes are based on a lack of technical knowledge, eight were based on a lack of BX-specific knowledge, and 16 were based on a lack of network knowledge. We identified three mistakes that cannot be systematically prevented during the design of the BX, as the developers are forced to decide among several options without an inherently correct choice. Different BXs in a network may implement different incompatible options, leading to failures when used in the network. As it may not yet be clear how other BX developers choose, these mistakes cannot be systematically prevented during BX design. They need to be detected and addressed when assembling a network. We proposed patterns to systematically prevent failures in networks of BXs that can be applied during BX design. The proposed patterns alone prevent 19 of the 29 faults, especially avoiding all preventable mistakes based on network knowledge. As only three mistakes cannot be systematically avoided at all, 96 of the 119 failures in our case study could have been prevented. Hence, this work helps transformation developers, as it improves awareness for possible issues in networks of BXs and enables them to prevent specific classes of issues by design. This work also serves as a foundation to understand issues in networks of independently developed BXs.

Verifiability

The implementation of Vitruvius [8] is available at GitHub [51]. We provide all artifacts of our case study in a dedicated reproduction package [52]. This includes a history of the case study, as well as the developed transformation network with the test scenarios of the case study.

Acknowledgments

References

[1] N. Kahani, M. Bagherzadeh, J. R. Cordy, J. Dingel, D. Varró, Survey and classification of model transformation tools, Software & Systems Modeling 18 (2018) 2361–2397. doi:`10.1007/s10270-018-0665-6`.

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

[3] H. Klare, Multi-model Consistency Preservation, in: 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`.

[4] H. Klare, T. Syma, E. Burger, R. Reussner, A categorization of interoperability issues in networks of transformations, Journal of Object Technology 18 (2019) 4:1–20. doi:`10.5381/jot.2019.18.3.a4`, the 12th International Conference on Model Transformations.

[5] S. Cook, C. Bock, P. Rivett, T. Rutt, E. Seidewitz, B. Selic, D. Tolbert, Unified Modeling Language (UML) Version 2.5.1, Standard, Object Management Group (OMG), Consortium, 2017.

[6] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, K. Krogmann, Modeling and Simulating Software Architectures – The Palladio Approach, MIT Press, Cambridge, MA, 2016.

[7] S. Becker, H. Koziolek, R. Reussner, The Palladio component model for model-driven performance prediction, Journal of Systems and Software 82 (2009) 3–22. doi:`10.1016/j.jss.2008.03.066`.

[8] H. Klare, M. E. Kramer, M. Langhammer, D. Werle, E. Burger, R. Reussner, Enabling consistency in view-based system development — the vitruvius approach, Journal of Systems and Software 171 (2021) 110815. doi:`10.1016/j.jss.2020.110815`.

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

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

[11] 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`.

[12] J. Gleitze, H. Klare, E. Burger, Finding a universal execution strategy for model transformation networks, in: Fundamental Approaches to Software Engineering, Springer International Publishing, Cham, 2021, pp. 87–107. doi:`10.1007/978-3-030-71500-7_5`.

[13] F. Orejas, E. Pino, M. Navarro, Incremental concurrent model synchronization using triple graph grammars, in: Fundamental Approaches to Software Engineering, Springer

International Publishing, Cham, 2020, pp. 273–293. doi:`10.1007/978-3-030-45234-6_14`.

[14] T. Sağlam, A Case Study for Networks of Bidirectional Transformations, Master's thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2020. doi:`10.5445/IR/1000120806`.

[15] F. Heidenreich, J. Johannes, M. Seifert, C. Wende, Closing the gap between modelling and java, in: Software Language Engineering, Springer Berlin Heidelberg, 2010, pp. 374–383. doi:`10.1007/978-3-642-12107-4_25`.

[16] T. Syma, Multi-model Consistency through Transitive Combination of Binary Transformations, Master's thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, 2018. doi:`10.5445/IR/1000104128`.

[17] M. Langhammer, Automated Coevolution of Source Code and Software Architecture Models, Ph.D. thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2017. doi:`10.5445/IR/1000069366`.

[18] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Engineering 14 (2008) 131–164. doi:`10.1007/s10664-008-9102-8`.

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

[20] J. Meier, C. Werner, H. Klare, C. Tunjic, U. Aßmann, C. Atkinson, E. Burger, R. Reussner, A. Winter, Classifying approaches for constructing single underlying models, in: Model-Driven Engineering and Software Development, Springer International Publishing, Cham, 2020, pp. 350–375. doi:`10.1007/978-3-030-37873-8_15`.

[21] N. Macedo, T. Jorge, A. Cunha, A Feature-based Classification of Model Repair Approaches, IEEE Transactions on Software Engineering 43 (2017) 615–640. doi:`10.1109/TSE.2016.2620145`.

[22] A. Pepin, Decomposition of Relations for Multi-model Consistency Preservation, Master's thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, 2019. doi:`10.5445/IR/1000100374`.

[23] H. Klare, A. Pepin, E. Burger, R. Reussner, A Formal Approach to Prove Compatibility in Transformation Networks, Technical Report 3, Karlsruhe Institute of Technology (KIT), Karlsruhe, 2020. doi:`10.5445/IR/1000121444`.

[24] P. Stevens, A Landscape of Bidirectional Model Transformations, Springer Berlin Heidelberg, 2008, pp. 408–424. doi:`10.1007/978-3-540-88643-3_10`.

[25] A. Kusel, J. Etzlstorfer, E. Kapsammer, P. Langer, W. Retschitzegger, J. Schoenboeck, W. Schwinger, M. Wimmer, A survey on incremental model transformation approaches, in: ME 2013 – Models and Evolution Workshop Proceedings, 2013, pp. 4–13.

[26] L. Samimi-Dehkordi, B. Zamani, S. Kolahdouz-Rahimi, Bidirectional model transformation approaches a comparative study, in: 2016 6th International Conference on Computer and Knowledge Engineering (ICCKE), IEEE, 2016, pp. 314–320. doi:`10.1109/ICCKE.2016.7802159`.

[27] S. Hidaka, M. Tisi, J. Cabot, Z. Hu, Feature-based classification of bidirectional transformation approaches, Software & Systems Modeling 15 (2016) 907–928. doi:`10.1007/s10270-014-0450-0`.

[28] A. Schürr, Specification of graph translators with triple graph grammars, in: Graph-Theoretic Concepts in Computer Science, Springer Berlin Heidelberg, 1995, pp. 151–163.

[29] A. Schürr, F. Klar, 15 years of triple graph grammars, in: Graph Transformations, Springer Berlin Heidelberg, 2008, pp. 411–425.

[30] H. Giese, R. Wagner, Incremental model synchronization with triple graph grammars, in: Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg, 2006, pp. 543–557.

[31] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, F. Orejas, From state- to delta-based bidirectional model transformations: The symmetric case, in: Model Driven Engineering Languages and Systems, volume 6981, Springer Berlin Heidelberg, 2011, pp. 304–318. doi:`10.1007/978-3-642-24485-8_22`.

[32] P. Stevens, Towards an algebraic theory of bidirectional transformations, in: Graph Transformations, Springer Berlin Heidelberg, 2008, pp. 1–17.

[33] Y. Xiong, H. Song, Z. Hu, M. Takeichi, Synchronizing concurrent model updates based on bidirectional transformation, Software and Systems Modeling 12 (2013) 89–104. doi:`10.1007/s10270-010-0187-3`.

[34] P. Stevens, Maintaining consistency in networks of models: bidirectional transformations in the large, Software and Systems Modeling 19 (2020) 39–65. doi:`10.1007/s10270-019-00736-x`.

[35] K. Lano, S. Kolahdouz-Rahimi, I. Poernomo, J. Terrell, S. Zschaler, Correct-by-construction synthesis of model transformations using transformation patterns, Software & Systems Modeling 13 (2014) 873–907. doi:`10.1007/s10270-012-0291-7`.

[36] 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`.

[37] D. Wagelaar, R. Van Der Straeten, D. Deridder, Module superimposition: a composition technique for rule-based model transformation languages, Software & Systems Modeling 9 (2010) 285–309. doi:`10.1007/s10270-009-0134-3`.

[38] D. Wagelaar, M. Tisi, J. Cabot, 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`.

[39] L. Lúcio, S. Mustafiz, J. Denil, H. Vangheluwe, 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`.

[40] B. Vanhooff, S. Van Baelen, A. Hovsepyan, W. Joosen, Y. Berbers, Towards a Transformation Chain Modeling Language, in: Embedded Computer Systems: Architectures, Modeling, and Simulation, Springer Berlin Heidelberg, 2006, pp. 39–48. doi:`10.1007/11796435_6`.

[41] B. Vanhooff, D. Ayed, S. Van Baelen, W. Joosen, 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`.

[42] A. Cleve, E. Kindler, P. Stevens, V. Zaytsev, Multidirectional Transformations and Synchronisations (Dagstuhl Seminar 18491), Dagstuhl Reports 8 (2019) 1–48. doi:`10.4230/DagRep.8.12.1`.

[43] N. Macedo, A. Cunha, H. Pacheco, Towards a framework for multi-directional model transformations, in: 3rd International Workshop on Bidirectional Transformations - BX, volume 1133, CEUR-WS.org, Athens, Greece, 2014, pp. 71–74.

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

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

[46] J. Gleitze, A Declarative Language for Preserving Consistency of Multiple Models, Bachelor's thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2017. doi:10.5445/IR/1000076905.

[47] I. Malavolta, H. Muccini, P. Pelliccione, D. A. Tamburri, Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies, IEEE Transactions of Software Engineering 36 (2010) 119–140. doi:10.1109/TSE.2009.51.

[48] R. Eramo, I. Malavolta, H. Muccini, P. Pelliccione, A. Pierantonio, A model-driven approach to automate the propagation of changes among Architecture Description Languages, Software and Systems Modeling 11 (2012) 29–53. doi:10.1007/s10270-010-0170-z.

[49] P. Stünkel, H. König, Y. Lamo, 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, New York, USA, 2018, pp. 9–17. doi:10.1145/3191697.3191715.

[50] Z. Diskin, H. König, M. Lawford, Multiple Model Synchronization with Multiary Delta Lenses, in: Fundamental Approaches to Software Engineering, Springer International Publishing, Cham, 2018, pp. 21–37. doi:10.1007/978-3-319-89363-1_2.

[51] Vitruv Tools, VITRUVIUS GitHub Organization, https://github.com/vitruv-tools, 2021. Accessed: 2019-04-24.

[52] T. Sağlam, H. Klare, Reproduction package for the paper on classifying and avoiding compatibility issues in networks of bidirectional transformations, 2021. doi:10.5445/IR/1000133796.