Derivation of Change Sequences from State-Based File Differences for Delta-Based Model Consistency

Master's Thesis of

Jan Willem Wittler

at the Department of Informatics
KASTEL – Institute of Information Security and Dependability

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

15th December 2020 – 14th June 2021

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

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

Karlsruhe, 14th June 2021

. . . . . .

(Jan Willem Wittler)

Abstract

In view-based software development, views may share concepts and thus contain redundant or dependent information. Keeping the individual views synchronized is a crucial property to avoid inconsistencies in the system. In approaches based on a *Single Underlying Model* (SUM), inconsistencies are avoided by establishing the SUM as a single source of truth from which views are projected. To synchronize updates from views to the SUM, *delta-based consistency preservation* is commonly applied. This requires the views to provide fine-grained change sequences which are used to incrementally update the SUM. However, the functionality of providing these change sequences is rarely found in real world applications. Instead, only state-based differences are persisted. Therefore, it is desirable to also support views which provide state-based differences in delta-based consistency preservation. This can be achieved by estimating the fine-grained change sequences from the state-based differences.

This thesis evaluates the quality of estimated change sequences in the context of model consistency preservation. To derive such sequences, matching elements across the compared models need to be identified and their differences need to be computed. We evaluate a sequence derivation strategy that matches elements based on their unique identifier and one that establishes a similarity metric between elements based on the elements' features. As an evaluation baseline, different test suites are created. Each test consists of an initial and changed version of both a UML class diagram and consistent Java source code. Using the different strategies, we derive and propagate change sequences based on the state-based difference of the UML view and evaluate the outcome in both domains.

The results show that the identity-based matching strategy is able to derive the correct change sequence in almost all (97 %) of the considered cases. For the similarity-based matching strategy we identify two reoccurring error patterns across different test suites. To address these patterns, we provide an extended similarity-based matching strategy that is able to reduce the occurrence frequency of the error patterns while introducing almost no performance overhead.

Zusammenfassung

In der sichtenbasierten Software-Entwicklung ist es möglich, dass mehrere Sichten das gleiche Konzept abbilden, wodurch Sichten redundante oder abhängige Informationen darstellen können. Es ist essenziell, diese individuellen Sichten synchron zu halten, um Inkonsistenzen im System zu vermeiden. In Ansätzen mit einem *Single Underlying Model* (SUM) werden Inkonsistenzen vermieden, indem das SUM als zentrale und einzige Informationsquelle genutzt wird, von welcher Sichten projiziert werden. Um Sichten mit dem SUM zu synchronisieren, wird in den meisten Fällen eine *deltabasierte Konsistenzhaltung* verwendet. Diese nutzt feingranulare Änderungssequenzen, welche von den einzelnen Sichten bereitgestellt werden müssen, um das SUM inkrementell zu aktualisieren. In realen Anwendungsfällen ist die Funktionalität zur Bereitstellung dieser Änderungssequenzen jedoch selten verfügbar. Stattdessen werden nur zustandsbasierte Änderungen persistiert. Es ist insofern wünschenswert Sichten, welche nur zustandsbasierte Änderungen bereitstellen, in deltabasierter Konsistenzhaltung zu unterstützen. Dies kann erreicht werden, indem die feingranularen Änderungssequenzen von den zustandsbasierten Änderungen abgeleitet werden.

In dieser Arbeit wird die Qualität von abgeleiteten Änderungssequenzen im Kontext von Modellkonsistenzhaltung evaluiert. Um eine solche Sequenz abzuleiten, müssen übereinstimmende Elemente aus den verglichenen Modellen identifiziert und deren Unterschiede bestimmt werden. Um übereinstimmenden Elemente zu identifizieren, nutzen wir zwei Strategien. Bei der einen Strategie werden übereinstimmende Elemente anhand ihres eindeutigen Bezeichners erkannt. Bei der anderen Strategie wird eine Ähnlichkeitsmetrik basierend auf den Eigenschaften der Elemente genutzt. Als Evaluationsgrundlage werden verschiedene Testszenarien erstellt. Für jeden Test wird eine initiale und eine geänderte Version von sowohl einem UML-Klassendiagramm als auch Java-Code bereitgestellt. Wir nutzen die verschiedenen Strategien, um Änderungssequenzen basierend auf den zustandsbasierten Änderungen der UML-Sicht abzuleiten, geben diese an das SUM weiter und untersuchen die Ergebnisse in beiden Domänen.

Die Ergebnisse zeigen, dass die Strategie, welche eindeutige Bezeichner nutzt, in fast allen betrachteten Fällen (97 %) die korrekte Änderungssequenz liefert. Bei der Nutzung der ähnlichkeitsbasierten Strategie können wir zwei wiederkehrende Fehlermuster identifizieren. Bezüglich dieser Probleme stellen wir eine erweiterte ähnlichkeitsbasierte Strategie vor, welche in der Lage ist, die Auftrittshäufigkeit der Fehlermuster zu reduzieren ohne die Ausführungsgeschwindigkeit signifikant zu beeinflussen.

Contents

List of Figures

List of Tables

1. Introduction

Due to the rising size of current software systems, it is hardly possible for any engineer to fully understand the developed system in its entirety. One approach to deal with this complexity is *View-Based Software Development*. Here, the system description is split up into different *Views* each describing a part of the system under development. By splitting up the system into smaller parts, only domain experts for the individual views are required instead of for the entire system. A single view, or model, can contain arbitrary information, like the documentation of requirements, a software architecture diagram, or executable source code. One problem introduced by the fragmentation of the system description are redundant or dependent information across views, as views can share overlapping information. If these redundancies or dependencies are not correctly managed, *inconsistencies* can occur. However, manually identifying and correctly managing redundancies to avoid inconsistencies is a challenging and time-consuming task. Therefore, various approaches for automated model consistency preservation exist.

One approach for automated model consistency preservation is the usage of a *Single Underlying Model* (SUM) [3]. A SUM contains the entire system description and serves as the single source of truth. It cannot be directly accessed but rather only indirectly from views. In the SUM-based context, a view is a partial projection of the SUM. Since the SUM is transparent for the engineer, the benefits of view-based software development still apply for this approach. However, since internally there is no information fragmentation anymore, no redundancies and thus no inconsistencies can occur.

A challenge with the SUM approach is propagating the changes of a view to the SUM as view changes introduce temporary inconsistencies. Changes to a view can either be expressed *state-based* or *delta-based*. In the state-based representation, the initial and changed view state are provided. In the delta-based representation, the initial view state and a sequence of fine-grained changes that need to be applied to the view to obtain its changed state are provided. A common approach for propagating view changes is *delta-based consistency preservation* [16] which propagates the delta-based changes of a view to the SUM. Using delta-based instead of state-based changes introduces the benefit that updates to the SUM are only performed incrementally instead of regenerating entire parts which might lose information. To obtain the fine-grained change sequences, they need to be recorded by the editor which is used to modify the view. However, this limits the real world applicability drastically as most industry tools do not offer this functionality and will not be extended to support it (due to closed source, legacy software, or simply budget limits). Therefore, it is highly desirable to also support delta-based consistency in cases where only state-based information is available.

To convert state-based changes to delta-based ones, the fine-grained change sequence needs to be estimated from the state-based information. This task is a complex and ambiguous task with an infinite search space as multiple change sequences can lead to the

same final state. Therefore, although broad tooling support for this task is available, none of them can guarantee to derive a correct change sequence in every case. Even though an incorrect change sequence may lead to the correct final state of the considered view, the propagation of these incorrect changes may lead to incorrect updates in the SUM. To illustrate this, renaming an element may look identical to deleting the old and inserting a new element with the modified name in one view but may trigger distinct operations in the SUM.

For the derivation of change sequences several strategies exist. A common concept is to first identify matching elements across the compared models and then derive the differences from those matches. A match should combine those elements of different models which are considered to represent the same (but possibly modified) information. In identity-based matching, these matches are constructed based on the unique static identifier of each element. However, since identifiers are not always available or not consistent across the compared models, another approach is to use similarity-based matching. Here, the features of elements are compared on their similarity to obtain a distance metric.

1.1. Goals of the Thesis

In this thesis we evaluate two existing strategies for change sequence derivation on their correctness in the context of model consistency preservation. Therefore, we construct test suites that are considered to pose challenges to the strategies and identify problems occurring when running the strategies against them. The evaluated strategies apply identity- respectively similarity-based matching. We show that for the identity-based matching strategy, almost always a correct change sequence is derived. For the similarity-based matching strategy, two reoccurring error patterns are identified. To address these, we provide a customized similarity-based matching strategy that is able to reduce the occurrence probability of the identified errors.

To derive a change sequence, the model comparison framework *EMF Compare* [9] is employed. EMF Compare is embedded into the *Eclipse Modeling Framework* (EMF) [52], supports binary and ternary model comparisons of arbitrary domains, and is highly customizable and extensible. Both evaluated matching strategies are provided by EMF Compare and are evaluated unmodified.

For model consistency preservation the Vitruvius framework [34] is used. Vitruvius realizes the SUM concept using a *Virtual Single Underlying Model* (V-SUM). A V-SUM leverages the SUM requirements to allow individual coupled models instead of a monolithic one. As this again introduces redundancies, the individual models are kept consistent by applying *Model-to-Model* (M2M) transformations whenever one of the models changes. The definition of these transformations is expressed as *Consistency Specifications* between the models.

Although our test suites aim to generate generic conclusions, providing tests for every domain is not feasible. Therefore, we limit our test suites to contain only views of the *Unified Modeling Language* (UML) [26] and the Java [44] domain. More precisely, in every test changes to a UML class diagram are performed and its influence on Java source code is observed. These domains form good candidates due to their large semantic overlap

and their broad usage [17, 5]. Furthermore, they are already supported by the chosen consistency preservation framework.

In this thesis we present an early approach towards the integration of real world applications into automated model consistency preservation. While a correct derivation of change sequences is an important aspect, other problems also need to be solved to fully support unobserved editors. Defining triggers to start the consistency preservation, dealing with data not covered by the SUM like layout information, or supporting outdated view states are just a few of them.

1.2. Structure of the Thesis

This thesis is divided into an introduction to the topic in chapters 1 to 3, a presentation of our approach in chapters 4 to 6, its evaluation and obtained findings in chapters 7 and 8, and finally an overview of future work and a conclusion in chapters 9 and 10. In more detail, the individual chapters contain the following:

Chapter 2 introduces the foundations for our work. It details model comparison and model consistency preservation and presents the used frameworks for these tasks. In *chapter 3*, related work is presented. Alternative approaches to automated model consistency preservation and different model comparison strategies are discussed. Furthermore, identified common model editing operations are introduced which are reused in the created test suites.

In *chapter 4*, the general concept of this thesis is explained. We highlight how the evaluated test suites are embedded into the view-based approach, present the chosen metrics for the evaluation, and reason why the considered tests are generalizable. *Chapter 5* gives insights on the implementation details, adjustments required to the applied consistency specifications, and limitations of the chosen model domains. In *chapter 6* the designed test suites are explained. We discuss the selected test cases, why they pose challenges to change derivation, and how they can be transformed into abstract concepts.

Chapter 7 presents the findings from evaluating the strategies on the designed test suites. We identify generic problems that may occur and present a strategy optimized for model consistency preservation. In *chapter 8* the results of our evaluation are presented. We evaluate our optimized and the existing strategies based on their correctness of the generated models and their performance. Additionally, we discuss threats to the validity of our approach. Finally, *chapter 9* suggests topics for future work, and *chapter 10* concludes the thesis and summarizes the obtained results.

2. Foundations

In this chapter, the foundations on which this thesis is based are explained. We discuss approaches towards automated model consistency preservation using a *Single Underlying Model* and *Delta-Based Consistency Preservation*. Then, we present a specific realization of this concept which is the Vitruvius framework. Furthermore, we show how model differences are derived and its practical realization in the *EMF Compare* framework [9].

2.1. Automated Model Consistency Preservation

With the increasing complexity of modern software systems, different approaches are developed to abstract away from this complexity. In *View-Based Software Development*, this complexity is reduced by splitting the system into role-specific *Views*, each presenting only a part of the system. A view can contain arbitrary information about the system, like requirements descriptions, modeling artifacts, or software parts. Since view semantics can overlap, the same information may appear in different views. This information redundancy is a problem as inconsistent states can get introduced if only some of the redundant information is updated [18]. However, manually preserving the consistency across views is hardly feasible, especially with a growing number of views and their sizes. Therefore, various approaches for automated model consistency preservation exist. One of these is using a *Single Underlying Model* from which views are projected, and *Delta-Based Consistency Preservation* [16] to propagate changes from the views to the SUM.

2.1.1. Single Underlying Model

To solve the problem of possible inconsistencies across views, in the *Orthographic Software Modeling* (OSM) paradigm the concept of a *Single Underlying Model* (SUM) is established [3]. "A SUM is a complete definition of a system and contains all known information about it. It contains no redundant or implicitly dependent information and is thus always free of contradictions, i.e., inconsistencies" [34]. Access to the SUM is not provided directly, but through editable views which provide a partial projection of the underlying model. By having the SUM as the transparent single source of truth while still providing individual views on the systems, the benefits of view-based software development are preserved although internally a monolithic system description is used.

Since views are projections of the SUM, they need to be generated dynamically. For this purpose, different model transformations are proposed [4, 10, 11, 56]. Using the transformations, a view can be generated from the SUM and changes performed on the view can be propagated back to the SUM. Although the SUM allows no inconsistencies, a

view is allowed to be temporarily inconsistent while editing. However, eventually changes need to be synchronized with the SUM to restore consistency.

2.1.2. Delta-Based Consistency Preservation

Changes to a view can either be represented *state-based* or *delta-based*. The latter one represents changes explicitly by providing the fine-grained sequence of change operations performed to obtain the modified state. In the former one, changes are implicitly represented by providing the initial and changed state of the view.

In the current SUM-based model consistency approaches, most frameworks use delta-based change representations [34, 40, 56, 57] to apply *delta-based consistency preservation* [16]. In delta-based consistency preservation, the fine-grained changes are incrementally applied to the SUM to perform small updates. An alternative approach was to use coarse-grained changes which would require to regenerate parts of the model on every change. However, in the context of model consistency the regeneration of model parts might lose those information which are projected away when generating the view as they are not included in the changed view state.

2.2. The Automated Model Consistency Approach Vitruvius

One practical realization of the SUM-based model consistency approach is the Vitruvius framework [34]. Since the definition of a metamodel for a SUM, the *Single Underlying Metamodel* (SUMM), is a challenging task as the engineer needs to be an expert for all involved domains, the constraints on the SUM are leveraged. Instead of a SUM, a *Virtual SUM* (V-SUM) is employed which consists of individual, but coupled models together with *Consistency Specifications* between those models.

2.2.1. Virtual Single Underlying Model

For the application of a SUM, there are several challenges in practice. First, the definition of a SUMM is a highly complex task. Not only does the information of every domain needs to be know, but also the dependencies and shared concepts across domains. With increasing system sizes, finding a domain expert for all considered domains is difficult. Second, the extension, evolution, or maintenance of a SUMM may easily introduce unintended side effects as every change may impact various domains. Therefore, in the Vitruvius approach the monolithic SUM concept is replaced by a modular *Virtual SUM* (V-SUM) concept [34].

A V-SUM provides the same guarantees as a SUM, namely being a complete and consistent system description. However, it realizes this by integrating individual, but coupled models instead of using one single large model. The coupling between the models is expressed in *Consistency Specifications*. Therefore, the metamodel of a V-SUM, the *Virtual SUMM* (V-SUMM) consists not only of the metamodels of the individual models but also contains the consistency specifications between them. A benefit of the modular design is that existing metamodels are compatible to the V-SUMM, only requiring the addition

of consistency specifications. Furthermore, the mentioned problems of SUMM definition, extension, evolution, and maintenance are reduced as multi-domain knowledge is only required for creating the consistency specifications. Since multiple specifications between different subsets of domains can be formulated, there is no need for a single expert for all domains.

2.2.2. Consistency Preservation

While the usage of a V-SUM simplifies the application for the engineer and the domain expert, it introduces new consistency constraints between the internal models. Since the coupled models may contain redundant information, it is crucial that this shared information is synchronized. In the Vitruvius approach, consistency is guaranteed inductively on changes to the model. This requires the initial state of the V-SUM to be consistent and transitions between states to preserve consistency. Since empty models are implicitly consistent, a consistent initial state is trivial to obtain. An approach to integrate existing models while still preserving a consistent state is presented by Leonhardt et al. [38]. To preserve consistency when transitioning states, delta-based consistency is used.

Redundant or dependent information between models is expressed using consistency specifications. Whenever one of the internal models changes, the consistency specifications define which *Model-to-Model* (M2M) transformations need to be executed to keep the coupled models consistent. Since these transformations apply changes to another model and thus may trigger other consistency transformations, transformations may be applied transitively or even form cycles. To preserve the trace of the consistency transformations, corresponding elements can be made explicit using *Correspondences*. These link two elements of different domains together such that for future transformations the corresponding element can be retrieved by traversing the correspondence.

To realize the consistency specification, Vitruvius uses *Consistency Preservation Rules* (CPRs) between metamodel pairs [36]. Each rule defines conditions for a pair of metaclasses that need to be met by their instances to be consistent. To express these, three consistency languages are provided [33, 36]. The *Reactions* language enables imperative, unidirectional specifications from a source to a target metamodel. The *Mappings* language provides support for declarative, bidirectional specifications between a pair of metamodels. In the *Commonalities* language, rules are expressed by defining shared concepts of different metamodels.

2.3. Model Comparison

One requirement of delta-based consistency preservation is that fine-grained changes are provided. To obtain them when only state-based differences are available, they need to be estimated. This can be accomplished using model comparison approaches. The aim of model comparison is to detect and represent differences between models. Common applications are the detection of version differences between models or of similarities for distinct models. While theoretically an arbitrary number of models can be compared, in practice binary and ternary comparisons are dominant. The process of model comparison

can be split into the two phases *Model Matching* and *Model Differencing* [8]. In the matching phase, similar elements from the compared models are matched together. In the differencing phase, differences based on the computed matches are generated. These phases are also established in the model comparison framework *EMF Compare* [9]. EMF Compare is used in this thesis as the basis for the generation of model differences.

2.3.1. Model Matching

In the model matching phase, corresponding elements between the considered models are matched together. Elements correspond to one another if the model matching strategy identifies them to represent the same (but possibly modified) information. While commonly matches contain one element of each model, matches with multiple elements of the same model or unmatched elements are also possible. The computed matches are used in the following phase to compute differences in the local scope of elements rather than in the global scope of the entire model. Since the problem of model matching can be reduced to the graph isomorphism problem, which is known to be NP-hard [47], different approaches have been proposed to deal with this complexity. These approaches can be grouped into four different categories.

Static Identity-Based Matching For the static identity-based matching approach every model element is required to have a persisted, non-volatile unique identifier. Two elements are matched if their identifiers are identical. Since only identifiers are considered, this approach promises fast performance and low susceptibility to errors, and is therefore the preferred approach whenever unique identifiers can be guaranteed. However, in common real-world applications such identifiers cannot be guaranteed. Models may get constructed independently of another (thus having completely different identifiers) or simply do not support identifiers at all.

Signature-Based Matching To overcome the requirement for static unique identifiers, the signature-based matching approach computes identifiers dynamically. The different features of an element are combined using a user-defined function to compute a unique identifier. Like in identity-based matching, elements are matched if their identifiers are identical. While this approach allows any models to be compared, it is highly dependent on the function used to calculate the identifiers.

Similarity-Based Matching Another approach which does not rely on static identifiers is similarity-based matching. In contrast to the previous two approaches, a match is not based on strict equality of some identifier but rather uses a distance metric to compute how similar two elements are. A side effect of this non-binary matching is that many-to-many matches are possible. To determine the distance between two elements, the aggregated similarity of the elements' features is computed. Since features tend to have different importance on the similarity of elements — e.g. an element's name is often a good similarity indicator while the exact numerical type of some attribute does rarely influence the semantics of the element — weight parameters for the features need to be

specified. Even though this weighting function still strongly influences the result, the approach enables less strict matching compared to signature-based matching. A drawback of similarity-based matching is its performance. Since distances are computed on a pair of elements instead of just once per element, the search space increases.

Custom Language-Specific Matching Algorithms While the three previous approaches are meant to be used independently of the modeling domain, this approach tailors a matching strategy for one specific modeling language. Commonly, it is based on similarity-based matching but provides a weighting function specific for the considered domain. Using this approach, results with high accuracy and great performance can be achieved. However, the algorithm needs to be adjusted for every modeling language individually, thus lacking generalizability.

2.3.2. Model Differencing

In the model differencing phase, the computed matches are translated into differences. A local differencing on the matches instead of a global one on the entire model offers the benefits of reduced complexity, as only elements need to be compared, and increased performance, as all matches can be processed in linear time. The drawback of this local comparison is that errors done in the matching phase cannot be detected or undone and will result in incorrect differences. Therefore, it is of crucial importance to receive optimal matches.

To generate the differences, for every match its affected elements are compared and differences of each feature are extracted. Furthermore, creation or deletion differences are created for every unmatched element, depending on whether the element is unmatched in the old (creation) or the new (deletion) model. For a created element, additional differences are generated to update its features to the presented state.

Differences can be represented in different ways which should abstract away from the used matching technology and the underlying metamodels. In the scope of this thesis, we use the *diff Metamodel* [20] to describe differences. Here, differences are represented as a sequence of atomic change operations. These operations are the creation, deletion, or movement of an element, or the modification of some element feature. However, other representations, like *Edit scripts* [30], exist.

2.3.3. The Model Comparison Framework EMF Compare

EMF Compare [9] is a model comparison framework integrated into the *Eclipse Modeling Framework* (EMF) [19]. It provides generic support for the binary or ternary comparison of any EMOF-compatible models. EMOF is the meta-metamodel for any model in the EMF environment and is compliant with the MOF standard [25]. Internally, the comparison algorithm follows the introduced two phase approach. To allow a high reusability and customization, the two phases are completely separated such that the logic for one phase can be exchanged while still being compatible.

The matches and differences are represented as models conforming to the *match* and *diff* Ecore metamodels respectively [20]. Their representation is optimized to be model-based,

compact, self-contained, transformative, compositional, and metamodel independent. This enables a high abstraction from the underlying technology and enables further processing such as conflict detection or model transformations [9].

By default, EMF Compare uses a combination of an identifier- and similarity-based matching approach. If the compared elements all provide an identifier, they are matched on their identifier equality, otherwise the similarity metric is used. The similarity is computed by analyzing an element's name, its content, its metaclass, and its relations. Several heuristics, like limiting the search space to only elements in a certain neighborhood, are used to filter out noise and reduce the execution time. Modifying the framework to only use identifier- *or* similarity-based matching or even integrating a custom matching strategy is also possible.

In addition to model matching and differencing, EMF Compare adds three additional phases which are executed after the differences are computed. First, in the *Equivalences* phase, changes representing the same operation are identified. This is mainly required as in EMOF-based models a change to a reference may automatically adjust the opposite of the reference, thus the two changes need to be executed together. Second, the *Requirements* phase identifies dependencies between changes. Here changes that form preconditions for other changes are identified, such that they can be applied in the correct order. Last, for ternary comparisons *Conflicts* are detected.

3. Related Work

Since this thesis is an early approach on evaluating change sequence derivation strategies for delta-based model consistency, there is, to the best of the author's knowledge, very few literature for this specific topic. Therefore, in this chapter we rather give an overview of work related to the individual parts of the chosen approach. These are the underlying model consistency framework, the chosen model comparison tooling and the considered strategies, and the editing operations considered for evaluation.

3.1. Automated Model Consistency Preservation

Besides of the already introduced Vitruvius approach, there exist several other solutions for automated model consistency preservation. These can be grouped into approaches which are also based on the Single Underlying Model (SUM) approach and those which follow different strategies.

3.1.1. Single-Underlying-Model-Based Approaches

Based on their original idea for Orthographic Software Modeling (OSM) [3], Tunjic and Atkinson use asymmetric, bidirectional model transformations to synchronize views and the SUM. Using the transformations, deltas of a view can be transformed to deltas for the SUM or vice versa such that all artifacts can be kept consistent. Another approach based on OSM is presented by Cicchetti, Ciccozzi, and Leveque [13]. In contrast to the previous work, they do not use deltas but generate difference metamodels [14] for the underlying model and each view. When a change occurs, it is represented in the associated difference metamodel which can be transformed into the other difference metamodels using higher-order transformations, and then applied to the corresponding views or the underlying model.

An approach similar to the Vitruvius approach is the *Role-Based Single Underlying Model* (RSUM) approach [57]. It also uses consistency preservation rules to keep individual models consistent. However, it additionally allows to combine and reconnect existing metamodels and uses role-based programming [37] rather than object-oriented programming. *MOdel CONSistency Ensured by Metamodel Integration* (MoConseMI) is another approach that combines the usage of an essential SUM with the support for integrating existing (meta)models [40]. Existing models are integrated into one SUM by applying a chain of fine-grained transformations (*Operators*). Due to the reversibility of the operators, the initial models are preserved as views which are updated using the chain of operators.

An overview of the described approaches, including the Vitruvius approach, and a comparison between them can be found in [41]. One similarity of them is their usage of

delta-based consistency. The focus of the existing literature is rather on the modeling and preservation of consistency *after* the fine-grained change sequences have been obtained. In contrast, this work uses the existing consistency preservation mechanism provided by the Vitruvius framework as a foundation to evaluate the quality of change sequence derivation strategies.

3.1.2. Further Approaches

If different models are kept consistent without a SUM as the ground truth, the individual views combined form the entire system description. Thus, changes in one view need to get directly translated to other views. One approach for this is to use coarse-grained changes and coupled transformations to synchronize view points [58]. In this approach, fine-grained changes are derived using identity-based matching and transformed into coarse-grained changes using detection rules written in *Maude* [15]. These changes are then propagated to other view points using explicit correspondence links. The authors motivate the usage of coarse-grained changes as the system modeler rather thinks in these dimensions than in low-level ones and thus a more natural environment is provided to the engineer.

Another approach that has direct synchronization between views is the *CoWolf* framework [24]. It aims at delivering user friendly utilities for model driven development by providing textual and graphical editors for different model types. Models are kept consistent in a delta-based approach using *Henshin* rules [2] which can either be created manually or using the *SiLift* environment [29].

3.2. Model Comparison Strategies

Model comparison has a wide scope of applications. Besides its usage in model consistency preservation, one of the most important applications is model versioning. As the text-based comparison algorithms commonly used in version control systems like SVN or GIT had been shown to be insufficient for model versioning [42], numerous model comparison approaches have been proposed. Since the vast number of approaches is out of scope for this work, the interested reader can find a broad overview in [31, 53]. One frequently cited early approach is *UMLDiff* [60] which provides model comparison based on custom, for UML models optimized, similarity metrics. Since incorporating the specific UML semantics into the metric limits its application to the UML domain, Lin et al. proposed a similar approach but rather for arbitrary metamodels (*DSMDiff*) [39].

Another metamodel-independent approach is *EMF Compare* [9]. It uses similarity metrics for model matching but provides a high degree of customizability to adapt for custom strategies. It is model-based, i.e. its output is again represented by a model to enable further model-to-model or model-to-text transformations.

Since model comparison is an inherently complex task, taking the benefit of being metamodel-agnostic introduces the drawback of reduced accuracy which has been shown for EMF Compare and DSMDiff by Kolovos et al. [35]. Furthermore, Pietsch et al. presented

a basic benchmark suite for model change scenarios which can pose problems for model comparison techniques [46].

As an approach to overcome these limitations, Addazi et al. extend the EMF Compare matching algorithm with semantic matching [1]. They compute the semantic distance of objects using a lexical database to identify semantic similar words and extend the matching policy by this semantic distance measure. While this approach outperforms the default EMF Compare algorithm in particular benchmarks, it suffers from a low precision value due to overly many matches and is influenced heavily by the chosen ontological specification.

Müller and Rumpe extend the EMF Compare framework with the possibility for the developer to provide presettings which specify how certain elements have changed [43]. While this can resolve any incorrect matches and enables the storage of the presettings together with the changed model such that matching is performed correctly not only on the local machine, the user has to *know* which matches need to be corrected manually and thus additional effort is required.

3.3. Common Model Editing Operations

Even though models are used in various use cases, there are certain editing operations that occur more frequently. Parul and Sidhu propose five abstract refactoring operation categories for UML class diagrams into which more specific operations can be grouped [45]. These are adding or removing features, moving some method, and generalizing or specializing an element. More specific refactoring operations for UML class diagrams are presented in [51]. Here, seventeen model smells are identified by grouping results from previous literature. For each model smell, a sequence of refactoring operations to correct it is listed. For this purpose, 34 refactoring operations are provided. These scale from single operations like adding a parameter to complex changes like extracting common features to a superclass. Another catalog for model smells and refactoring operations is collected by Fowler [22]. In contrast to the previous work, this catalog is not domain specific and listed refactorings are based on the author's experience and preferences.

To emphasize that refactoring operations are not limited on intra-model editing, Wimmer et al. present a catalog of model-to-model refactoring operations [59]. The catalog includes 24 operations for transformation rules and is derived by analyzing existing transformation examples in the Atlas Transformation Language (ATL) [28]. Like in the UML class diagram catalog, the presented operations scale from atomic operations to complex changes. Interestingly, even though the three presented catalogs are collected for different domains, they share similar operations. As an example, collapsing or expanding hierarchies are expressed by the refactoring operations *Collapse Hierarchy* and *Extract Superclass* in [22, 51] and by *Eliminate Superrule* and *Extract Superrule* in [59].

To examine the application of refactoring operations in real-world applications, Tsantalis et al. perform an empirical study on three Java projects with a history of six, seven, and twelve years. To automatically identify refactoring operations, detection rules originally introduced by Biegel et al. [6] are adopted and applied to model differences generated using DSMDiff [39]. In their study, they identify movement of classes and methods and

the renaming of classes as the dominant refactoring operations, followed by adjustments to the hierarchy like pulling a method up or extracting a superclass. Basic operations like the addition or removal of some element are not considered a refactoring operation in this study and thus do not appear in the results.

4. Concept

In this chapter we present our concept for embedding and evaluating views with state-based differences in delta-based consistency preservation. First, we discuss how state-based differences can be supported using model comparison and which existing comparison strategy approaches we consider in this thesis. Second, we motivate how requirements to results of the model comparison strategies can be relaxed in the context of consistency preservation. Third, application scenarios which pose non-trivial challenges to delta-based consistency preservation are identified. Finally, we present our evaluation baseline which uses the discussed findings to allow generalizable conclusions.

4.1. State-Based Differences in Delta-Based Consistency Preservation

In the theoretical concept for SUM-based approaches, there are no constraints on how views are synchronized with the SUM. However, almost all current approaches use delta-based consistency preservation [34, 40, 56, 57]. As discussed in subsection 2.1.2, delta-based approaches allow to perform only small changes to models instead of regenerating them. A requirement for its application is the availability of fine-grained change sequences. We call views that provide such sequences *delta-providing views* and views which only provide state-based differences *state-providing views*. Intuitively, delta-providing views are supported by delta-based consistency preservation. In contrast, a state-providing view needs to be extended with the functionality to derive a fine-grained change sequence from its state-based changes to be supported by delta-based consistency preservation.

To extend a state-providing view with this derivation functionality, we use model comparison. There are four different categories of model matching approaches in model comparison: identity-based, signature-based, similarity-based, and custom language-specific algorithms. For the scope of this thesis we compare an identity-based matching and a similarity-based matching strategy. Although identity-based matching approaches provide the best matching accuracy and performance, their application scenarios are limited. Identifiers may not be available for some or all elements of the model, or they may not be consistent across the compared models. The latter may especially occur when multiple people work simultaneously on a project, maybe even using different model editors. For these cases we consider similarity-based approaches to be the best choice. Similarity-based approaches provide similar functionality as signature-based ones but are more flexible by not requiring an exact match of the computed signature but use a distance metric. Language-specific algorithms are not considered as we want to obtain results generalizable to arbitrary domains.

Although bridging from state-based differences to delta-based ones is a crucial functionality for integration real world applications, there are further challenges out of scope for this thesis. One problem we assume to pose major challenges to model consistency are simultaneous changes to the SUM and a view. If a SUM is modified, views become outdated and need to be regenerated. However, if the view is in a dirty state, a regeneration would lose the changed information and thus the update is postponed. If an outdated view provides delta-based changes — independent of whether they are observed or estimated — they are based on an outdated SUM as its baseline but need to be resolved against the updated SUM. This can introduce problems, as referenced elements got modified or deleted. While this is not a problem specific to state-providing views but may occur for any view, we consider it to occur more frequently for state-providing views as changes are less often synchronized. To avoid this problem, in this thesis we only consider scenarios in which the changed view is based on the current SUM state.

Besides of the simultaneous change problem, other challenges are to define triggers to start consistency preservation and to deal with information of the application which is not representable in the SUM. To limit the scope of this thesis, we only consider the trivial cases of these challenges. Although additional data in the form of layout information is produced during the creation of our models, we ignore it. Excluding layout information from the SUM is reasonable, as it has no semantic value but only helps the user to understand the model easier. In future work, one could evaluate how this *decorative* information can be recovered by the SUM. Regarding triggers, only manual ones are used. This means that the consistency preservation is initiated by manually providing the initial and changed view states. In future steps, this should be extended to be executed automatically at meaningful occasions. We imagine committing the model to version management or saving it to disk to be good candidates for automatic triggers.

4.2. Relaxation of Change Sequence Correctness

For the derivation of change sequences, model comparison is used. Commonly, its main goal is to derive the *correct* change sequence, i.e. the change sequence that was originally used to create the changed model. However, for model consistency preservation this can be relaxed. In particular, the derived change sequence is only an intermediate artifact which is discarded after consistency is restored. Therefore, though the main target of this work is to optimize the derived change sequence, its validation does only rely on the correctness of the SUM. To emphasize this shift of focus, we define two new properties of change sequences.

Definition 1 (Conservative Change Sequence)**.** Let V be a view projected from the single underlying model SUM. Let V' be its changed version. Let $S\Delta_{V,V'}$ be the change sequence derived from V and V'. Let SUM' be the updated underlying model obtained by applying $S\Delta_{V,V'}$ on SUM using the consistency preservation mechanism.

$S\Delta_{V,V'}$ is conservative if V' is projected from SUM'.

Definition 2 (Admissible Change Sequence)**.** Let V be a view projected from the single underlying model SUM. Let V' be its changed version obtained by applying the change

sequence $S^*\Delta_{V,V'}$. Let $S\Delta_{V,V'}$ be a change sequence derived from V and V'. Let SUM' be the updated underlying model obtained by applying $S^*\Delta_{V,V'}$ on SUM using the consistency preservation mechanism.

$S\Delta_{V,V'}$ is admissible if applying $S\Delta_{V,V'}$ on SUM using the consistency preservation mechanism produces SUM'.

A conservative change sequence preserves the changed state of the considered view after change propagation while giving no warranties for other information of the SUM. An admissible change sequence guarantees the correctness of the SUM and thus implicitly also of the changed view. As a consequence, an admissible change sequence is always conservative. Following the definitions, the goal of change sequence derivation for model consistency can be reformulated to deriving an admissible change sequence, as this is exactly the case when the SUM is correct. We require every change sequence to be at least conservative. Generating an invalid changed view should never be an accepted outcome. To distinguish that for the scope of model consistency preservation the admissibility of a change sequence is sufficient to produce correct results, we avoid the term *correct* change sequence in the latter of this work. Rather, we refer to the change sequence that was used to produce the changed state as the *actual* change sequence. Since the actual change sequence produces the correctly updated SUM, it is by definition admissible and thus also conservative.

4.3. Modeling Domain Considerations

Since we require any derived change sequence to be at least conservative, we want to consider only scenarios where a conservative change sequence is not automatically admissible. A conservative change sequence can always be obtained by deleting all elements of the original view and creating all elements of its changed version. Thus, if a conservative change sequence was always admissible, the change sequence derivation would be trivial. We call scenarios in which a change sequence might be conservative but not admissible *critical cases*. As a SUMM is constructed by combining the information of different domains, we call domains that form a critical case *critical domains*. Furthermore, as we use a V-SUMM for consistency preservation, the used consistency specifications can also influence which cases are critical. To make our results independent of the used consistency specifications, we require certain properties from the specifications.

4.3.1. Critical Domains for Consistency Preservation

Whether a SUMM constructed by combining certain domains forms a critical case or not is dependent on the view type from which changes are provided. The same SUMM can be a critical case for some view types and not critical for others. To illustrate this, propagating changes from a view which represents the entire SUM is trivially not critical. However, a view which only contains a fraction of the information for some element can be a critical case. Here, renaming an element preserves the information of the element not in the view, while a deletion and re-insertion with the new name loses it. However, in the view

both changes look identically. Both described views can be projected from the same SUM, resulting in one critical and one not critical case.

The described examples already highlight one aspect to identify a critical case: the portion to which elements of the SUM are represented in the view. In the first described example, every element is completely covered by the view, thus forming no critical case. In the second example, an element is only partially covered, therefore being a critical case. The third case are views which only cover elements completely but do not cover every element. In this case, the classification is depending on the relation between the covered and stripped elements. If there is a relation of any kind (for example association, inheritance, or containment) between the covered and stripped elements, the scenario is critical. Since the relation to the stripped element can be seen as a property of the covered element, this property is not covered by the view and thus the element can be considered to be not completely covered. If all covered elements have no relation to stripped elements, it is not a critical case as any changes to the covered elements cannot influence the stripped elements.

We can apply this classification also to the V-SUM approach. Since individual coupled models are used instead of one monolithic model, the covered portion of an element transforms to whether there is a correspondence of one element in another model. Therefore, to identify critical cases, we need to assess how much the domains overlap semantically. For two domains, there can either be none, a partial, or a complete semantic overlap. Since for the scope of this thesis we only consider a view which is projected from a single internal model, we can simplify this further to the semantic overlap of the affected model to the other models. We say the other domains are *distinct* from the changed model, have a *partial overlap*, or are *contained*. Distinct and contained domains form the extreme cases where no or a complete semantic overlap is present. In all other cases, the domains are partially overlapping. An additional property that may appear in a V-SUM are ambiguous representations between models. One semantic concept of one domain may allow multiple different representations in another domain. A concept is only considered as ambiguous representable if it is represented in both domains to some extent. Otherwise, for partially overlapping domains every exclusive information would be considered to be an ambiguous representation as it can be arbitrarily changed while preserving consistence.

Since the amount of semantic overlap and the possibility of ambiguity are distinct properties, there is a total of six combinations of which five are possible; distinct domains cannot be ambiguous as there are no shared semantic concepts. The combinations are shown in Table 4.1. Any possible ambiguous case (partial overlap or contained) and every case with partially overlapping domains forms a critical case. For ambiguous representations, different change sequences may lead to the construction of a different of the multiple possible representations of an ambiguous concept. For partially overlapping domains, exclusive information of some domain may get lost or incorrectly preserved depending on the change sequence. Distinct domains are always consistent with each other as they share no semantic overlap. Therefore, they can never form a critical case. Finally, contained domains with non-ambiguous representations are also not critical. Since we require every change sequence to be conservative, the changed domain is updated to its correct new state. As all information of the contained domains is also available in

	Distinct	Partial Overlap	Contained
Non-Ambiguous	−	+	−
Ambiguous	(−)	+	+
	− non-critical	+ critical	

Table 4.1.: Critical Cases for Domain Relationships in a V-SUM

the changed domain, it can be regenerated from the changed domain independent of the applied change sequence.

4.3.2. Consistency Specification Requirements

For our classification of critical domains, certain properties of the consistency specification are implied. If the consistency specification is wrongly configured, conservative change sequences may lead to incorrect results even in non-critical cases. To avoid these, a consistency specification needs to be *complete* and *correct*. A consistency specification is *complete* if every possible operation is supported. A consistency specification is *correct* if every occurring semantic concept is correctly configured. The latter property includes that constructed correspondences should not dependent on the order in which a model is constructed. Both requirements can be weakened to apply only for the occurring scenarios and their affected elements if the scope of occurring scenarios is known for some application. This simplification is necessary for real-world applications, as the domains in these contexts are often too complex for taking every single element and every possible change sequence into account. We think every consistency specification in consistency preservation applications should fulfill these completeness and correctness requirements. Otherwise, incorrect models may occur even for delta-providing views as certain operations are not supported or the order of distinct operations produces different results.

4.4. Evaluation Baseline Construction

In order to evaluate how state-providing views perform in delta-based consistency preservation, an evaluation baseline is necessary. However, providing test scenarios for all domains and consistency specifications is not feasible. Therefore, we chose a critical domains pair on which our evaluation is based. To reduce the bias introduced by the choice of the domains, we avoid exploiting certain particularities of the domains but rather apply general concepts that can be found across domains. Nevertheless, it is not claimed that there are no cases which are not considered by this thesis.

4.4.1. Test Case Setup

The chosen case study includes the UML [26] and Java [44] domains. In particular, the modified state-providing view is a UML class diagram and the coupled model consists of Java source code. Since in the current Vitruvius implementation views can only implicitly

defined by representing one internal model, the state-providing view and the internal UML model are identical. As changes are propagated from UML to Java, we call UML the *source domain* and Java the *target domain*. These domains were chosen for several reasons. UML class diagrams are widely used and can be serialized using an international standardized *eXtensible Markup Language* (XML) representation, thus even if the generalization for some insight is not possible, it is still valid for the UML domain which makes it applicable to a wide range of scenarios. Furthermore, its metamodel is MOF-compatible, which makes it very representative for other metamodels. Java is one of the most used programming languages in academia and industry [5]. In addition, for most UML elements there is an unambiguous 1:1 mapping to some Java element which makes the consistency specification easy to understand. Finally, Java contains additional information not present in the UML domain (e.g. method bodies) and some ambiguous consistency mappings (e.g. accessor methods) which makes the case study a critical case. Besides of the scientific reasons, the case study was also chosen due to some practical motivation. First, both domains are already supported by the Vitruvius framework such that there is a serialization and deserialization functionality to transform from file representation to in-memory model representation (or vice versa respectively). Second, consistency specifications from UML to Java and from Java to UML already exist (though incomplete, see section 5.3).

For every test case, we construct the V-SUM with initial UML and Java models which are consistent to each other. Since we want to identify cases when a conservative change sequence is not admissible, the SUM needs to be extended with exclusive or ambiguous information not present in the changed view. In our case, the Java domain needs to be extended. Although critical domains are chosen, the properties that make them critical, namely partial overlap and ambiguous, also need to be exploited. If no such information was present in the SUM, it could be stripped from the SUM thus making the domains non-critical. Following on the V-SUM setup, we provide a changed state of our state-providing view and observe the results in the V-SUM. More precisely, a modified UML class diagram is provided to derive a change sequence from it. This change sequence can then be used to update the coupled models using the consistency preservation.

4.4.2. Model Correctness Metric

To automatically evaluate the resulting V-SUM of our test cases, we need to introduce some metric. The target of a test case is to produce an admissible change sequence. Also we want to verify that our precondition of a conservative change sequence is hold. Although the conservative and admissible properties are defined in the scope of view and SUM, due to our test setup we are able to simplify them. Since the changed view exactly matches the internal UML model, we can assess whether a change sequence is conservative by comparing the internal UML model to the expected one. As the V-SUM only consists of the UML and the Java model, this leaves only a validation of the Java model when checking whether a change sequence is admissible.

Since a conservative change sequence is a requirement, any deviation from the expected UML model in the internal one should cause the test to fail. Therefore, a sufficient metric for the source model is its equality to the expected source model. Any deviating source

model is caused by a non-conservative change sequence which breaks our assumptions on the derivation strategies.

For the target model we know due to the correct and complete consistency specification and the conservative change sequence that every target domain element that is covered by the consistency specification will be correct independent of whether the derived change sequence is admissible. Therefore, in the case of an incorrect change sequence, only a small portion of the target model will be incorrect. Consequentially, common metrics like *precision*, *recall*, or *F1-score* would produce misleading high accuracy scores even though the change sequence could be far from the actual change sequence. To shift the evaluation focus to the faults in the model, another metric would be to count the number of incorrect model elements. However, this requires to match the elements of the two models to identify the incorrect ones. As model matching is already performed during change sequence derivation, the metric would be dependent on a feature of the evaluated strategy which reduces its meaningfulness. Therefore, the target model metric is simplified to also evaluate the equality of the generated target model compared to the expected target model. Since the target model is split over multiple files, the equality can be evaluated for each file separately. For incorrect models, the incorrect elements can be annotated manually.

4.4.3. XML as a Prototype for Semi-Structured Files

As the considered domains are selected to serve as a generalizable case study, we need to assess the influence of the underlying file serialization. In both the chosen consistency preservation framework and the model comparison framework, models are represented using an in-memory representation. Although this requires some deserialization logic to convert from the file to the in-memory representation, it abstracts away from underlying file semantics. This already weakens the influence of the chosen file serialization. Furthermore, the file serialization of our considered UML view, the *eXtensible Markup Language* (XML), can be considered as prototypical for a wide range of file structures.

The structure of a file can be categorized into *structured*, *semi-structured*, and *unstructured* data [23]. Structured data requires a predefined data model and is represented in a tabular format. One of the most common examples for this are SQL databases. Semi-structured data uses tags or markers to self-describe the structure of its file in a predefined matter. Famous representatives of semi-structured data are XML or JSON files, or No-SQL databases. Unstructured data does not adhere to some data model or some predefined structure. Audio or video files belong to this category but also text-heavy data like messages or documents; a common keyword associated with unstructured files is *Big Data*.

We can consider XML as a prototype for both structured and semi-structured data as any file of these structure categories can be converted to XML. Structured data can be represented in XML by converting each table row to an XML element with children representing the different table cells. An example of modeling relational databases in XML is described in [48]. For semi-structured data, they all share the concept of a hierarchical self-describing structure such that this structure can be simulated in XML. Note that even though the file itself can be represented in XML, specific traits of the file (like fast queries and indexes in relational databases) are not necessarily preserved. Nevertheless, as we only consider the model comparison properties of the files, this can be neglected.

Although unstructured data can be embedded into XML, it cannot be converted to a self-structured format with a semantic concept as there is no predefined structure or a describing metamodel. While this prevents to make any conclusions from XML to unstructured data, unstructured data cannot be described by an in-memory model neither — due to the absence of a metamodel — and is therefore not applicable to the considered model comparison technology.

5. Implementation

Before being able to evaluate different change sequence derivation strategies, a test environment needs to be built that enables change propagation from state-providing views and validation of the achieved results. Furthermore, a correct and complete consistency specification is needed to be assured that test scenarios are failing due to incorrectly derived change sequences and not due to an incomplete or inconsistent consistency specification.

In this section, the implementation steps for building the test environment are discussed. The pipeline used for automatic model loading and change propagation is shown in section 5.1 and following, its output validation is described in section 5.2. In section 5.3, implemented solutions for inconsistencies and missing features of the consistency specification are highlighted. Lastly, particular traits of the chosen domains and required adjustments for these are presented in section 5.4. The test environment is built as a test application of the Vitruvius framework and is available on GitHub[1]. In order to build the system, adjusted versions of the Vitruvius framework[2] and its domains[3] are needed.

5.1. Pipeline Setup

In the current implementation of the Vitruvius approach, there is no explicit concept of views. Rather, a view is defined implicitly as a projection of one of the coupled models contained in the V-SUM and assumed to provide fine-grained change sequences. While it is desirable to eventually support defining and customizing view types, it is beyond the scope of this thesis. Therefore, instead of providing a full implementation to support different types of views, a lightweight adapter to integrate state-providing views in the model consistency preservation is implemented.

This adapter is integrated in a pipeline which allows to automatically setup the V-SUM, propagate changes, and validate the results. In order to setup the V-SUM, the pipeline loads the baseline state of the state-providing view from file into the V-SUM which triggers internal transformations. More precisely, the triggered transformations generate a coupled consistent model of the target domain. As a second step, the changed state of the state-providing view is used to derive a change sequence which is then applied to the V-SUM. This triggers again transformations which update the considered view and modify the coupled model of the target domain. To distinguish between conservative and admissible change sequences, the generated target domain model can be modified — with restricted operations to preserve consistency — before change propagation. An overview of the pipeline is shown in Figure 5.1.

[1] `https://github.com/JanWittler/Vitruv-Applications-ComponentBasedSystems`
[2] `https://github.com/JanWittler/Vitruv`
[3] `https://github.com/JanWittler/Vitruv-Domains-ComponentBasedSystems`

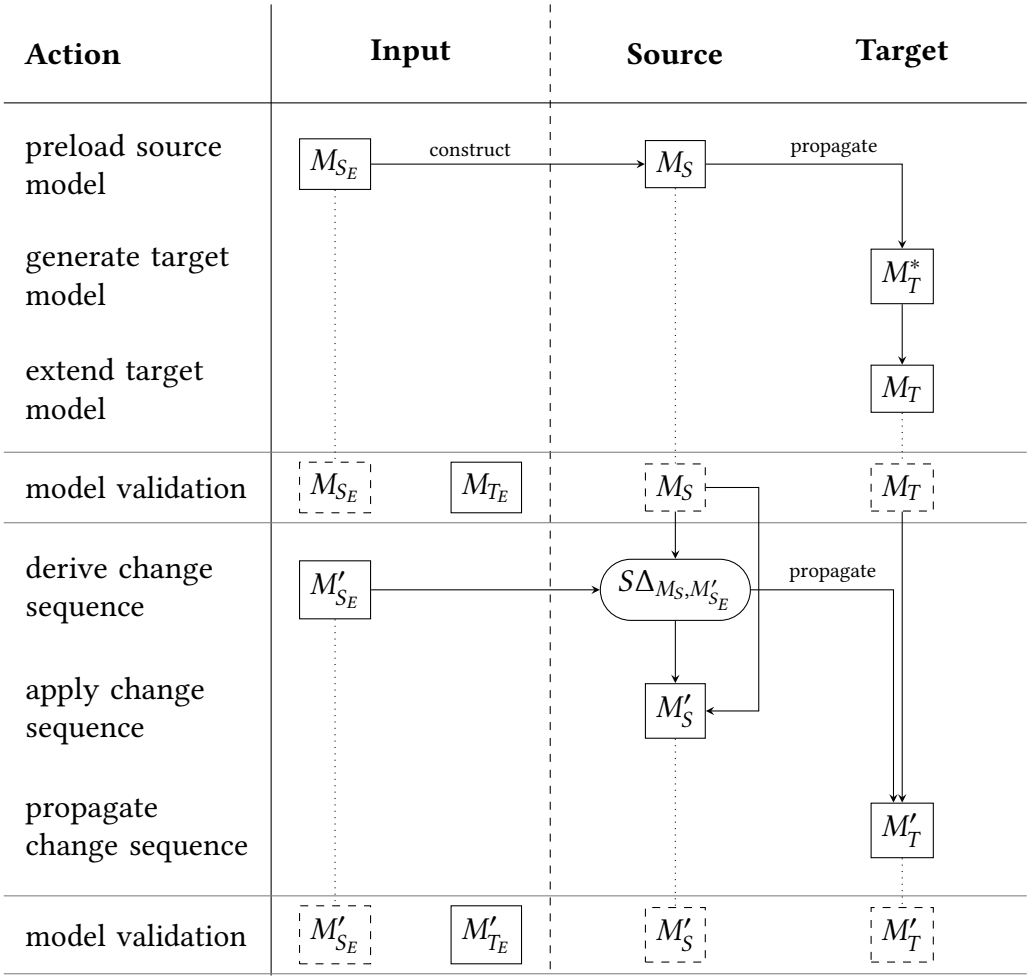


Figure 5.1.: Test Case Pipeline

To allow a more precise description of the pipeline steps, in this section we use the terms *source model* and *target model* instead of referring to views and the SUM. As in our test cases the SUM only consists of two models, we can directly reference them. The source model is the state-providing view that is modified, and the target model is the coupled model which is kept consistent.

5.1.1. Virtual Single Underlying Model Initialization

Since as of implementing the pipeline, the Vitruvius framework did not support to store and reload its V-SUM in a version control compatible way — as it relied on absolute paths of the local machine — each test case reconstructs the V-SUM from an initial empty state. Although this introduces a performance overhead, it makes the pipeline not susceptible to changes of how the Vitruvius framework stores its internal state.

The V-SUM is based on the initial source model state M_{S_E} which is provided as an external file. To setup the V-SUM, the external source model is copied into the V-SUM. However, copying the file to the appropriate location is not sufficient, as the Vitruvius instance needs to be aware of the newly added view to adapt its internal state. Instead, an empty internal source model M_S is created, the external source model is loaded into memory, and every element of the external model is copied to the internal model. Using this approach, Vitruvius is able to register the creation and modification of every model element as if the model was created in an observed editor. Thus, delta-based changes are made available to the consistency preservation.

Since delta-based changes are available, the delta-based consistency preservation can be applied to generate a coupled consistent target model M_T^*. Although the delta-based changes are not necessarily matching the actual change sequence, we can assume them to be admissible. This can be done as the target model is empty before propagating changes and thus does not contain domain-inherent information, and the consistency specification is required to be correct. Generating an admissible change sequence is crucial as otherwise the test evaluation is based on a wrong initial state. However, to completely assure the model correctness, it is additionally validated after the V-SUM setup.

Although after change propagation already coupled source and target models are generated, the V-SUM setup requires the additional step of extending the target model (obtaining M_T). This is necessary such that not every conservative change sequence is also admissible (see subsection 4.4.1). However, the target model cannot be arbitrarily modified as changes may trigger some consistency preservation modifications in the source domain. Performing a change that triggered modifications in the source model would break the assumption taken for this thesis that the baseline state of the state-providing view is identical to the regenerated view from the SUM.

To not perform changes that affect the source model, the presented test cases modify the target model only in two different ways. Either additional information that cannot be represented in the source domain is added, or information where the consistency specification allows multiple representations is changed to an alternative representation. In our test cases, we allow two specific modifications. First, a method body may be added to some Java method or constructor. Since method bodies do not have a representation in UML, no consistency transformations are triggered. Second, Java property accessory may

be deleted. Property accessors are automatically generated on UML field creation but have no explicit representation in UML. Thus they form an ambiguous case, their existence or absence results in the same consistent UML model. Since once more simply copying the changed Java file to the appropriate file system location is not sufficient as the Vitruvius instance needs to be aware of the changes, modifications are specified programmatically using the *JaMoPP*-based model representation of Java files [27].

5.1.2. State-Based Change Application and Propagation

With the initial V-SUM state being setup, the state-based changes can be applied. Therefore, a second external model of the source domain M'_{S_E} is provided. This model represents the changed state of the evaluated state-providing view. The Vitruvius framework already provides the function `propagateChangedState(Resource newState, URI oldLocation)` to propagate state-based changes. This function derives the fine-grained change sequence, and applies them to the source and target models.

To derive the change sequence, the derivation strategy of the changed model is used. While in the existing Vitruvius implementation each modeling domain has a fixed strategy associated with it, we extend the implementation to allow fast switching between our evaluated strategies. The strategies are provided with the base state M_S and the modified state M'_{S_E} of the source model and return a list of differences in the EMF Compare difference model format. The conversion from these differences to the Vitruvius change model is already supported by the framework. Therefore, the differences are replayed on a copy of the base state of the source model and an attached observer records these to produce Vitruvius changes.

For the implementation of our evaluated identity- and similarity-based strategies, we use the existing model comparison logic of EMF Compare. By default, EMF Compare uses identity-based matching if identifiers are available, and similarity-based matching if not. As we know that all our evaluated elements have an identifier, we use the default logic without modifications as the identity-based strategy. For the similarity-based strategy, we also use the default logic but disable identity-based matching, such that always the similarity metric is used.

Although fine-grained change sequences are derived for the changed source model state, this changed state is not yet reflected in the V-SUM. Therefore, the internal source model is updated to M'_S by applying the derived change sequence. As the update is performed with estimated delta-based changes, the delta-based consistency preservation is triggered. Thus, the target model is updated to M'_T by propagating the applied changes.

By using an existing function of the Vitruvius framework to propagate state-based changes, design decisions of the pipeline influence the consistency preservation only marginally. We only modify the existing workflow to allow the insertion of our own derivation strategies which implement the same protocol as the existing derivation strategy. A challenge of this approach is to obtain the actual derived change sequences which is valuable for reasoning about test results. Since the sequence is only an intermediate artifact to restore consistency, it is by default discarded as soon as the change application and propagation is completed. Furthermore, we do not want to overload our own strategies with this logging functionality. To solve this, we insert an additional object in the strategy

call chain between the Vitruvius instance and the actual strategy. This object forwards any change derivation requests to the actual strategy but stores the derived sequence before returning it to the Vitruvius instance. With this approach, the logging of the derived change sequences is transparent to both the strategy and the Vitruvius instance.

5.2. Output Validation

Each test generates source and target model files, the derived change sequence, and the internal Vitruvius virtual model state as its output. Although the derived change sequence and the Vitruvius state are important artifacts to debug potential failing test cases, they do not require any validation. Validating the internal Vitruvius state would give no benefit as it is tool-dependent and might change in future versions of the framework. The derived change sequence is only an intermediate representation which is discarded after the change is applied. Additionally, by definitions 1 and 2, it is implicitly conservative whenever the changed view, i.e. the source model, is correct, and admissible whenever the SUM, i.e. both the source and target model, is correct. Since admissibility is the sufficient requirement for each change sequence, the correctness of each test is determined only by the correctness of the output model files. To prevent errors in the change sequence derivation process by incorrect inputs, in addition to validating the models after each test case, they are also validated after the V-SUM preloading is completed (see Figure 5.1 model validation). This ensures that the system is in the expected state and there are no undetected errors in the setup logic.

Although the Vitruvius framework already offers functionality to compare two model files, the model validation relies on custom model comparison logic. This is primarily motivated by the fact that the model preloading and change sequence derivation, application, and propagation rely on the same model representations that could be used to compare models. Thus, any issues introduced by the model representations would remain undetected by the model validation. Furthermore, this makes the model validation independent of the used model consistency framework. However, simply bit-wise comparing the files is not sufficient (as e.g. for XML the order of attributes can change), therefore each domain uses a custom file comparison to check for semantic, instead of syntactic, equality.

5.2.1. UML Model Validation

To validate the UML model, the model file managed by the Vitruvius instance is compared to the already provided external model file. Since UML models use an XML file representation, the model validation is performed using XMLUnit [7]. This tool parses XML files into a tree-like structure which allows to ignore whitespaces and the order of XML attributes. XMLUnit already offers a default comparison mechanism which performs an element-wise comparison of the XML files by iterating through their hierarchies. Two files are considered as equal if both files share the same hierarchy and all element comparisons are succeeding. For an element comparison to succeed, the compared elements need to match their node name, all their attributes, and all their descendants.

Node condition			Missing attribute	Default value
tag name	=	uml:Model	xmlns:ecore	*any*
tag name	=	packageImport	xmi:type	uml:PackageImport
tag name	=	ownedAttribute	xmi:type	uml:Property
tag name	=	ownedOperation	xmi:type	uml:Operation
tag name	=	ownedParameter	xmi:type	uml:Parameter
tag name	=	eAnnotations	xmi:type	ecore:EAnnotation
tag name	=	details	xmi:type	ecore:EStringToStringMapEntry
tag name	=	ownedEnd	xmi:type	uml:Property
tag name	=	generalization	xmi:type	uml:Generalization
xmi:type	=	uml:Class	visibility	public

Table 5.1.: UML File Comparison Default Values

Even though the default comparison is already quite elaborate, some modifications are required for the UML file comparison. These adjustments are necessary as the serialization used by the external UML model tool and the serialization used by the Vitruvius framework are different. In particular, the external serialization stores default attributes, whereas the Vitruvius serialization does not. Therefore, the absence of default attributes is ignored. Table 5.1 shows the list of default values for some attribute of some node. Furthermore, association members are stored in different orders. As the order is arbitrary, it is ignored.

5.2.2. Java Model Validation

For the validation of the Java model, a folder containing the expected Java files in their expected file hierarchy is required. This folder is provided as additional input to the tests for each model validation phase (M_{T_E} respectively M'_{T_E} in Figure 5.1). Since there are two model validation phases, two expected Java file hierarchies need to be provided. However, since most test cases within one suite share their base model (see chapter 6), the Java file hierarchy for the setup validation can be shared. The Java model is considered to be correct if the file hierarchy managed by the V-SUM matches the provided one and each internal Java file has semantic equality to its expected counterpart.

The Java file comparison works line-based. Starting with the first line in both file, the current line of each file is tested for equality with the current line of the expected file. To adjust for particularities of the JaMoPP serialization, empty lines are skipped and whitespaces at the start or end of a row and before semicolons are ignored. Furthermore, the generated source file contains fully-qualified type references instead of only the classifier name together with an according `import` statement. However, as the expected Java files should serve as a gold standard, they apply the preferred combination of an `import` statement and the classifier name only. To still being able to correctly compare these two different representations, lines with `import` statements are also skipped for

comparison and fully-qualified type references are simplified to their classifier name if the referenced type got already imported in any of the two compared files.

It is worth noting that the proposed Java file comparison will incorrectly mark files as not matching when the order of methods and attributes is changed, even though the order is arbitrary in Java. While this could be addressed with additional implementation effort, such cases appeared only in two out of 144 tests and were corrected manually with little effort.

5.3. Reactions Refinements

As motivated in subsection 4.3.2, an incorrect or incomplete consistency specification may distort the evaluation. Consequentially, a correct and complete consistency specification is required. However, providing a complete consistency specification is a challenging task due to complexity of the chosen domains and their large semantic overlap. Therefore, the properties of correctness and completeness are weakened to only apply for those elements and scenarios that are considered. More specifically, the consistency specification is only required to be complete for those elements which are covered by the test suites and to be correct within the scope of tested scenarios. Since the test space is constructed and thus known to be completely covered by the consistency specification, it is valid to assume that test failures are only caused by incorrectly derived change sequences.

In the Vitruvius framework, the consistency specification between UML and Java is implemented in the Reactions language [36] and was initially provided by Chen [12]. Though the majority of cases and elements considered in this thesis are already covered by the consistency specification, some limitations have to be overcome to consider it as correct and complete.

5.3.1. *Move*-Operations

When deriving change sequences, one of the most common operations is the movement of objects to another container. Though they also might occur in the actual change sequence, especially for similarity-based strategies they are often a side-effect of a conservative matching for the parent container. As an example, renaming a container may result in a deletion and re-insertion of the renamed container and as a consequence a move of all the container's children. This observation is discussed more detailed in subsection 7.2.1. In the Vitruvius difference model, a movement is represented by a removal from and a subsequent insertion into a container. In the existing consistency specification, any reaction for an insertion or removal is bound to trigger only when a prior creation respectively a subsequent deletion of the object occurred (*compound trigger*). Since an object's lifetime extends before and beyond a movement, this does not hold true for this kind of operation which results in no reaction trigger for a movement operation causing failures in a majority of test cases.

Since the Reactions language limits the possible combination of atomic triggers in compound triggers to $\{creation, insertion\}$ and $\{removal, deletion\}$, it is not possible to distinguish between, for example, a creation and an insertion or an insertion without cre-

ation of an object. In order to support insertion without creation changes while preserving reactions to creation and insertion, one either could add an additional reaction to react to the atomic trigger of an insertion or avoid compound triggers altogether. With the latter approach, insertions with or without a previous creation are unified to the same scenario. While the prior option may seem like a valid alternative, it imposes additional constraints on both reactions (the newly added and the existing one with a compound trigger) as a creation and insertion pattern would trigger both of them. As the execution order of reactions is by design non-deterministic [32], the result needs to be identical independent of the execution order of the reactions. Since this might be difficult to achieve and, more importantly, introduces new requirements to testing — as a test should cover all possible reaction execution orders — it is favorable to avoid compound reaction triggers and rather use atomic triggers only.

To extend the consistency specification with the support for movement operations by splitting the compound triggers into atomic ones, three design alternatives are considered. Besides of providing the missing support for movement operations, we aim at preserving as much of the existing consistency specification logic as possible. For the refactoring, especially the reactions triggered by creation and subsequent insertion require detailed attention as the correspondences are setup in these routines. If the setup of the correspondences fails, any subsequently triggered reaction will not execute as it requires a correctly setup correspondence.

Correspondence at creation The first approach is the most intuitive approach. It splits any reaction with compound triggers into reactions for each atomic trigger and adjusts the model in those reactions accordingly. For example, a trigger for creation and insertion is split into a reaction that reacts to a creation and creates the object and its correspondence; and another reaction that reacts to an insertion and inserts the corresponding object. This implies that correspondences are created at the creation of the object and need no special considerations afterwards, thus removing the special treatment in insertion reactions.

Though the approach appears elegant, it does not fulfill all requirements. An instance of the same metamodel element may get inserted in different containers resulting in different representations in the target model. For example, a UML property may get inserted in a UML class or a UML association which results in different Java representations. However, since the correspondence has to be setup during creation, the target metamodel element type needs to be known before insertion which is not possible in this case. Another problem is introduced by transitive cases. In these scenarios, an object may already be created by some other consistency specification from other domains and only the correspondence to the current object is missing. Therefore, before creating any object it is searched for a possibly matching object which correspondence is just missing (pattern *find-or-create* [49]). This pattern does fail when applied directly after creation, as the created object has no identifying attributes set yet. As a consequence, a new object would be created each time, leading to duplication of objects in transitive scenarios.

Correspondence at insertion The second approach aims at reducing the compound triggers to their latest executed atomic trigger. Namely, a trigger for creation and insertion

is reduced to a trigger only for insertion; a trigger for removal and deletion is reduced to a trigger only for deletion. Consequentially, the correspondences are setup in the insertion-triggered reaction. This implies that for every insertion the find-or-create pattern is executed. However, the introduced execution overhead is only marginal as the pattern terminates early when a correspondence is already setup, which holds true in the majority of cases. Another consequence of the refactoring would be that there was no reaction that reacts explicitly to a removal of an object from its container. While this may seem like a gap in the consistency specification, it can be safely assumed that each removal is either followed by the object's deletion or an insertion to some other container. This is mandatory as objects without a container are not allowed by the Eclipse Modeling Framework. Therefore, removals are implicitly captured by the other reactions.

Correspondence at renaming The third approach is similar to the previous approach as that compound triggers are reduced to their latest executed atomic trigger. The difference with this approach is that correspondences are created when the object is initially named. In contrast to creating the correspondences on insertion, the execution of the find-or-create pattern can be reduced to once per object — on initial naming — instead of on every insertion.

Approach selection After evaluating the different approaches, the second approach, *correspondence at insertion*, was implemented. While creating the correspondences at creation seems appealing, the problems introduced with it cannot be resolved within the current language implementation thus making it no alternative. As the other two approaches are very similar, the second approach is favored as it is closer to the previous consistency specification implementation, thus allowing to reuse more existing code.

Nevertheless, the chosen implementation alternative as well as the option with creating correspondences on renaming impose some implicit constraints on the order of the changes. In most cases, the find-or-create pattern requires both the object's name and the object's container to be set to identify a matching object, thus for the chosen alternative the name must be set before the insertion, and for the renaming alternative an element must be inserted before named. While this order assumption holds for all test cases in the Vitruvius framework and all tests evaluated in this thesis, it cannot be assumed. If the order does not match the assumptions, the find-or-create pattern fails and thus will create a new object even if potentially the object is already existing. Furthermore, changes to other attributes of an element need to be performed only after the correspondence is setup, as otherwise the executed reaction does not have a corresponding object to modify yet and no changes are propagated. However, these problems were also present in the existing consistency specification.

5.3.2. Java Accessor Tagged Correspondences

Though most commonly one element of the source domain is represented by at most one element of the target domain, there also exist one-to-many relationships. One of these cases are UML attributes and Java fields and accessors. One UML attribute always

corresponds to exactly one Java field but may additionally have correspondences with a *getter* or *setter* Java method. Since accessors are optional in Java, the representation of the UML attribute is ambiguous in Java. As pointed out in subsection 5.1.1, this ambiguity of the Java accessors representation is exploited in the test scenarios to create a target model that contains information not present in the source model. Therefore, it is of central importance that the consistency specification correctly deals with accessors and especially does not accidentally recreate or delete them when the UML attribute changes.

In our application, the consistency specification automatically creates both accessor methods alongside the Java field when a UML field is created. It is worth noting that this is an explicit design decision. Other options — like creating no accessors at all automatically — would also result in correct consistency specifications. However, the existing consistency specification does not model an explicit correspondence between the UML field and the accessors. Rather, the correspondence is implicitly resolved by matching the methods by their name. This introduces issues, as renamed accessors cannot be matched anymore or methods matching the accessor naming pattern are incorrectly recognized as an accessor method. Furthermore, if transitive transformations are enabled, i.e. a triggered change to the Java domain may trigger changes back in the UML domain, the creation of the accessor methods would trigger the creation of a corresponding operation in the UML domain. Therefore, accessor correspondences were made explicit using tagged correspondences. A tagged correspondence is a correspondence that uses a textual tag to distinguish it from regular correspondences. By using a unique tag for each of the *getter* and the *setter* methods, each accessor method can be unambiguously identified. This allows to correctly find the corresponding elements even if the accessor got renamed, prevents incorrect correspondence matching for similarly named methods, and the creation of UML operations for Java accessor methods.

5.4. UML Domain Particularities

While this thesis aims at utilizing the UML to Java case study only as a foundational example which provides generalizable conclusions, it is unavoidable that certain particularities of the chosen domains will uncover during the research. For the UML domain we identify two problems. First, XML identifiers are incorrectly managed when having multiple resources for the same XML document in memory. Second, the usage of UML associations may lead to incorrect matching behavior.

5.4.1. XML Identifiers

Since UML is serialized as an XML document, it uses XML identifiers (tag attribute `xmi:id`). As these are specific to XML, they are not part of the Eclipse UML metamodel but are rather stored in a mapping owned by the `XMLResource`. An `XMLResource` is the in-memory representation of a UML XML file.

In the Vitruvius framework, it may happen that there are multiple objects based on the same file simultaneously in memory. Commonly, the virtual model hosts one object per resource, and the test view hosts another object for each resource that it accesses. While

```
1 <packagedElement xmi:type="uml:Association" xmi:id="1"
    memberEnd="2 3">
2   <eAnnotations xmi:type="ecore:EAnnotation" xmi:id="4"
      source="org.eclipse.papyrus">
3     <details xmi:type="ecore:EStringToStringMapEntry"
        xmi:id="5" key="nature" value="UML_Nature"/>
4   </eAnnotations>
5   <ownedEnd xmi:type="uml:Property" xmi:id="3" name="
      example" type="6" association="1"/>
6 </packagedElement>
```

Figure 5.2.: UML Association XML Example

in most execution scenarios this does not pose any harm as changes in a test view are observed and thus the other resource representations can be updated accordingly, changes to the identifier mapping of the `XMLResource` are not observed. This leads to inconsistent XML identifiers across the Vitruvius instance. More precisely, the test view resource instance does contain the correct identifiers — those of the external model loaded from file — but the models hosted by the virtual model contain newly generated identifiers.

To overcome this, the XML identifiers are copied manually to the resources hosted by the virtual model after the initial source model loading and after resolving the changed model. If this workaround was left out, the identifiers of the internal model would not match the ones of the preloaded model and thus also not match those of the changed model. This would lead to incorrect change sequences for any strategy that relies on identifiers.

5.4.2. Associations

Associations are a common element in UML diagrams as they are used to describe the relationship between different classes, data types, and interfaces. Unfortunately, they are also a common error source for an incorrectly derived change sequence as the default EMF Compare implementation has difficulties to correctly match associations and their descendants. Since associations also have identifiers, this is less critical for identity-based matching strategies but even for these strategies errors may appear which can lead to non-conservative change sequences. We discuss the occurrence of these cases in detail in chapter 8.

To understand why associations are a problematic type of element, Figure 5.2 provides an extract of a UML model that shows an association. An association has two endpoints which are defined by listing their identifiers in the `memberEnd` attribute separated by spaces. Note that the endpoints do not need to be contained in the association. In the provided example, one endpoint is contained in the association (`xmi:id` 3) as an `ownedEnd`, the other one is not (and therefore not included in the shown XML). Endpoints not contained in their association are commonly defined as a property contained in a classifier with an `association` attribute pointing to the association. Furthermore, each association contains

the shown `eAnnotations` which are used for internal handling of the UML resource and do not provide additional semantic information.

When it comes to matching associations with EMF Compare, there are several properties that do not work well with the similarity-based matching strategy. First, besides of their identifiers, annotations and their children are identical for all associations. Second, associations are hard to match as they do not provide a name attribute. Third, there may be multiple endpoints across different associations with the same name. Fourth, associations are stored directly in the root element of a UML model. The combination of these four properties makes it hard for the EMF Compare match engine to identify matching associations as neither the association container, its name attribute, nor children attributes can be used to identify similarities. The only uniquely identifying attribute is the `memberEnd` attribute. However, as the default match engine is optimized to support matching of arbitrary elements, it is not optimized for this specific case and therefore does not weigh the `memberEnd` attribute as strong as an identifier.

While EMF Compare offers lightweight extension points to optimize the matching for this specific case, this thesis does not provide some optimized matching logic for the UML domain. This was left out as the purpose of this work is to obtain generalized findings for any domain and not to generate a logic optimized for the UML domain; an overview of optimized UML comparison approaches is presented by [50]. However, we still examine the impact of incorrect association matching on the overall error for incorrect change sequences in chapter 8.

6. Test Suites

In order to assess different change sequence derivation strategies, reference models and change sequences are needed to have an evaluation baseline on which the derived change sequences can be compared. However, the number of possible systems and change sequences is unlimited and thus cannot all be covered. Therefore, models and test cases need to be selected in such a way that they are not specific to their underlying reference system and provide some representative characteristics for a larger class of change sequences.

For the scope of this thesis, three different approaches are used to choose representative systems and test scenarios. First, systems are artificially constructed to highlight basic changes and recurring change patterns (section 6.1). Second, systems already described in previous work that are stated to pose problems to change sequence derivation are reused (section 6.2). Third, large-scale systems are generated to evaluate the performance of the different strategies under high load (section 6.3). For all systems, the considered elements and attributes are constrained to a small subset of the entire UML domain. By limiting the considered elements, there are less domain-specific cases to consider which could break the assumption of a correct and complete consistency specification. Furthermore, since only a few concepts of the chosen domain are considered, elements can be better embedded into an abstract environment. Thus, occurring problems can be better generalized to arbitrary domains. The test systems are constructed using the external editor *Papyrus* [21]. Its features and the tool selection process are described in section 6.4.

6.1. Artificially Constructed Systems

For the creation of meaningful artificial systems, common change sequences need to be identified. This section presents test suites for two classes of common change sequences. First, a system to test atomic changes is presented. Evaluating the strategies against atomic changes is a crucial part of the evaluation as any further test suites rely on the correct behavior for atomic changes. Second, a test suite for common refactoring operations is presented. Besides of their common appearance, testing refactoring operations provides the benefit that their change sequences mostly consist of a larger number of atomic changes and therefore are assumed to pose a bigger challenge to the evaluated strategies.

6.1.1. Atomic Change Operations

Any change sequence, regardless of the model's size or the number of changes, can be considered as a composition of atomic change operations. To assure that each of these atomic change operations is supported by the consistency preservation tool and covered

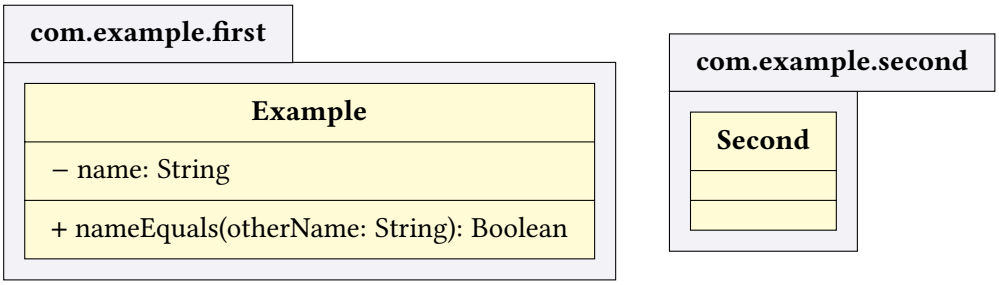


Figure 6.1.: UML Model for Atomic Change Operations

by the consistency specification, the test suite for atomic change operations contains tests for all atomic change kinds for various model element types.

By testing all kinds of atomic changes, it can be assured that the consistency specification covers all relevant cases and that the consistency preservation framework and the used pipeline work correctly. This will also help to evaluate larger change sequences as for strategies passing the test suite for atomic changes it can be assumed that occurring errors are related to the derived change sequence and not to some internal limitations or failures.

Since the change sequences are derived using EMF Compare, the kinds of atomic changes are also defined by EMF Compare. In particular, the difference model used by EMF Compare defines the four atomic change kinds *add*, *change*, *delete*, and *move*. Note that, since Vitruvius uses a different metamodel for change sequence representation, an atomic change of EMF Compare is not necessarily an atomic change in Vitruvius. More specifically, an *add* operation is translated into a creation and insertion-into-container sequence, a *move* operation is translated into a removal-from-container and insertion-into-container sequence, and a *delete* operation is translated into a removal-from-container and deletion sequence. Only a *change* operation is translated to an atomic change of an attribute operation. However, other combinations of changes in the Vitruvius change sequence representation cannot occur since all change sequences originate from the difference metamodel of EMF Compare and are therefore limited to its kinds.

The UML model which is used for all test cases of the *Atomic Change Operations* (ACO) test suite as the initial model is shown in Figure 6.1. Since the purpose of this test suite is to identify fundamental issues of the environment setup rather than evaluating the change derivation strategies, the model is kept minimal on purpose to reduce the probability of deriving an incorrect change sequence. While the support for atomic changes could have also been tested by directly providing the specific atomic change to the consistency preservation engine, incorporating the state-based change derivation was chosen to examine the test pipeline as a whole. For the target model extension (see section 5.1), the setter method for the `name` attribute of the class `Example` is deleted, and the method `nameEquals` is extended with the method body `return this.name == otherName;`.

The different test cases are listed in Table 6.1. To test a *change* operation, an element's name attribute is changed (*rename*); the other change kinds map to their equally named tests. For each change kind, the change is performed for an attribute of a class, a method of a class, and a class each. These three elements are chosen to represent different concepts

Name	Changes
Add Attribute	Add attribute `counter` of type `Integer` to class `Example`
Add Class	Add empty class `New` to package `com.example.first`
Add Method	Add method `+doSomething(i: Integer)` to class `Example`
Move Attribute	Move attribute `name` from class `Example` to class `Second`
Move Class 1	Move class `Second` to package `com.example.first`
Move Class 2	Move class `Example` to package `com.example.second`
Move Method	Move method `nameEquals` from class `Example` to class `Second`
Remove Attribute	Remove attribute `name` from class `Example`
Remove Class	Remove class `Example`
Remove Method	Remove method `nameEquals` from class `Example`
Rename Attribute	Rename attribute `name` of class `Example` to `newName`
Rename Class	Rename class `Example` to `Renamed`
Rename Method	Rename method `nameEquals` of class `Example` to `nameNotEquals`

Table 6.1.: Atomic Change Operations Test Description

in terms of generalizability. A class represents a container without siblings, a method a container with siblings, and an attribute an element without children.

6.1.2. Common Refactoring Operations

When it comes to developing tests for a larger number of changes, the number of potential change sequences is unlimited. To select a reasonable subset of these, already identified refactoring operations were used. While this certainly does not cover all scenarios that may be encountered in real applications, it can be assumed that the identified patterns will be common change sequences and that the sizes of their change sequences lie within the average expected range. Furthermore, since EMF Compare exploits the locality principle and therefore only matches within a certain neighborhood [9], any composition of refactoring operations in different parts of one model can be considered to result in similar results as deriving the change sequences successively for the different refactoring operations.

The initial UML model for all test cases of the *Common Refactoring Operations* (CRO) test suite is shown in Figure 6.2. It is based on the `mediastore.basic` package of the Media Store case study [54] which is implemented in Java, and adapted to enable certain refactoring operations and to adjust for limitations of the used UML model editor.

To extend the Java model, the setter methods for the attributes `providedMethods` of class `ProvidedInterface`, `requiredMethods` of class `RequiredInterface`, `requiredInterface` of class `LegacyData`, `encoding` of class `Metadata`, and all setter methods contained in classes `EJB`, `Config`, and `CurrentUser` are deleted. Additionally, all getter methods contained in the class `Config` are also deleted. Furthermore, the methods `toString` of class `Method`, `printRequiredInterfaces` and `printProvidedInterfaces` of class `EJB`, `isReconfigurable` and `getEJBs` of class `Config`, and the constructor of class `CurrentUser` are extended with appropriate method bodies.

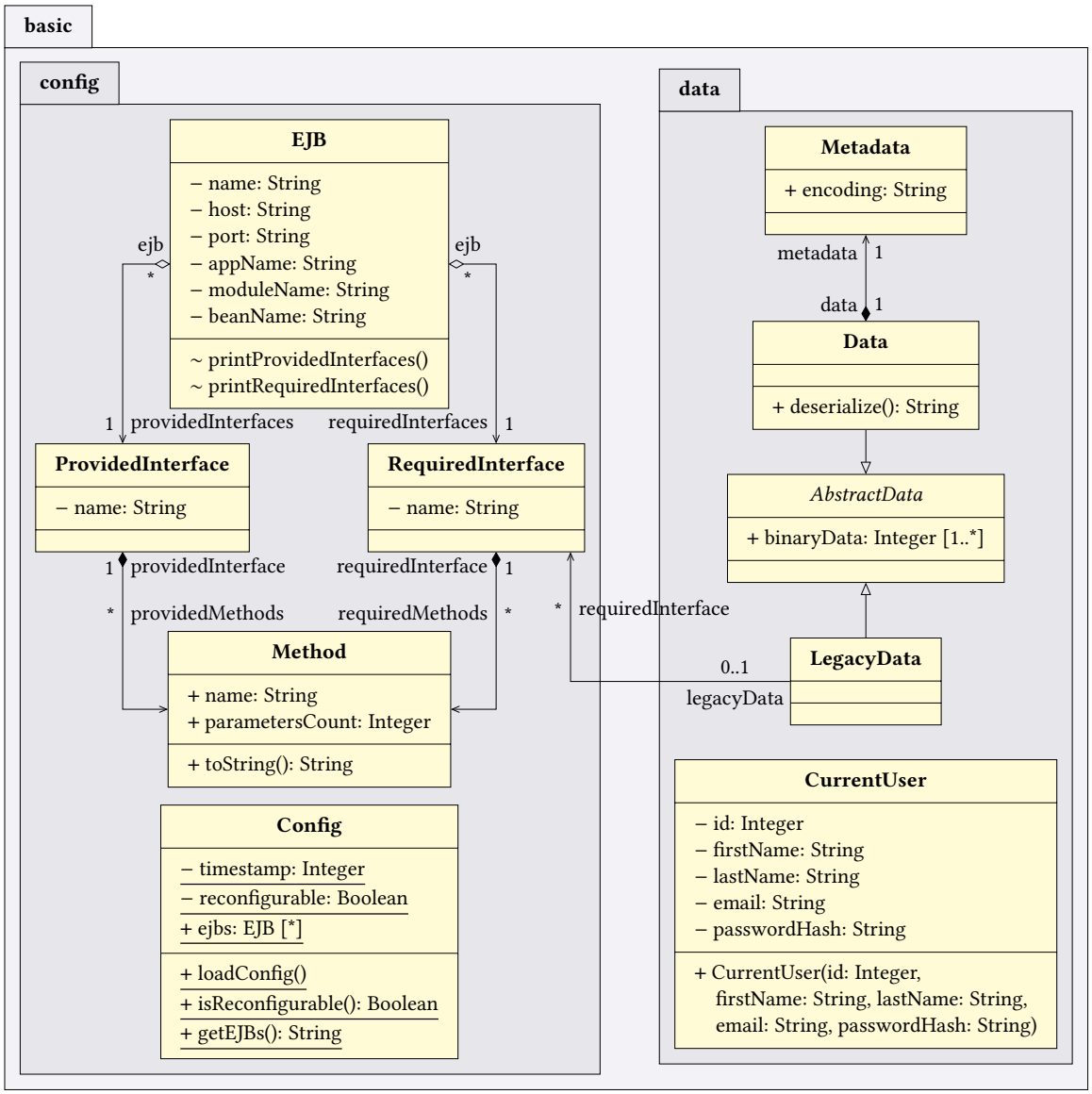


Figure 6.2.: UML Model for Common Refactoring Operations

Name	Changes
Change Method Signature	Make method `deserialize` of class `Data` static
	Change return type of `deserialize` to `Boolean`
	Add parameter `data` of type `Data` to method `deserialize`
	Rename method `deserialize` to `deserializeData`
Collapse Hierarchy	Move attribute `binaryData` of class `AbstractData` to class `Data`
	Delete class `AbstractData`
Extract Associated Class	Create class `Printer`
	Move method `printProvidedInterfaces` of class `EJB` to class `Printer`
	Add parameter `ejb: EJB` to `printProvidedInterfaces`
	Move method `printRequiredInterfaces` of class `EJB` to class `Printer`
	Add parameter `ejb: EJB` to `printRequiredInterfaces`
	Add 1:1 composition `printer` from `EJB` to `Printer`
Extract Superclass	Create class `Interface`
	Make class `ProvidedInterface` a subclass of `Interface`
	Move attribute `name` of class `ProvidedInterface` to class `Interface`
	Move attribute `providedMethods` of class `ProvidedInterface` to class `Interface`
	Rename attribute `providedMethods` to `methods`
	Make class `RequiredInterface` a subclass of `Interface`
	Delete attribute `name` of class `RequiredInterface`
	Delete attribute `requiredMethods` of class `RequiredInterface`
Inline Class	Move attribute `encoding` of class `Metadata` to class `Data`
	Delete class `Metadata`
Remove Associated Class	Delete class `LegacyData`

Table 6.2.: Common Refactoring Operations Test Description

The modeled test cases are based on the common model editing operations presented in section 3.3 and follow the refactoring operations catalog presented by Sidhu, Singh, and Sharma [51]. Since various of their identified refactorings are too simple to reveal any issues with the change sequence derivation (like *hide attribute* or *move class* are just atomic changes), only those operations that require a larger number of change operations are implemented. To pertain a broad coverage of scenarios, the test cases are selected to cover all categories of refactoring operations identified by Parul and Sidhu [45] (except for method movement which is already covered by atomic operations, see Table 6.1). The implemented test cases are listed in Table 6.2. Even though the implemented refactorings are defined for UML class diagrams, similar operations occur also for other domains as discussed in section 3.3. Therefore, we consider the chosen test cases to be representative even beyond the UML domain.

6.2. Exemplary Medium-Scale Systems

Due to the large amount of research already performed on change sequence derivation and model matching, there already exist different systems that are shown to pose a challenge to model matching algorithms. Therefore, two of these systems are reused as test suites for this work. The selected test systems are the *Model Match Challenge* system by Pietsch, Müller, and Rumpe (subsection 6.2.1) which was designed as a general benchmark for model matching strategies, and the *Thesis System* example by Addazi et al. (subsection 6.2.2) which was specifically designed to show problematic changes when using EMF Compare.

6.2.1. Example *Model Match Challenge*

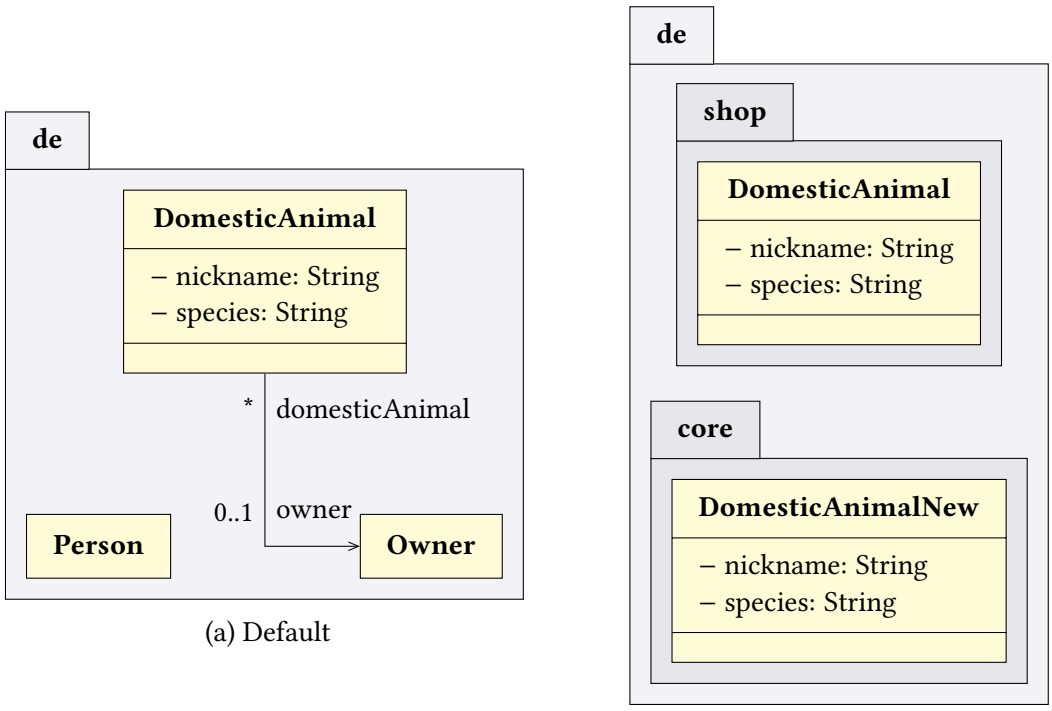
The *Model Match Challenge* (MMC+) is an example system presented by Pietsch, Müller, and Rumpe to showcase identified problems with model matching and was designed to serve as a benchmark for model comparison [46]. The initial UML models for the test cases are shown in Figure 6.3. The default model (Figure 6.3a) was altered in comparison to the original paper such that the `DomesticAnimal` class always has the properties `nickname`, `species`, and the association `owner`. This was done to enable a shared initial model for four of the five test cases. Since the change only adds additional elements to the model which increases its complexity, it can be assumed that matching errors occurring in the originally proposed model will also occur in the altered version. For the test case *Exchange Elements*, a custom initial model is used which is shown in Figure 6.3b. The list of test cases is shown in Table 6.3 and exactly mirrors the test cases described in [46].

To extend the target domain, the `setSpecies` method of the `DomesticAnimal` class is deleted. For the *Exchange Elements* model, additionally the `setNickname` method of the `DomesticAnimalNew` class is deleted. These adjustments to the target model are added to the original test cases proposed by Pietsch et al. as they did not target model consistency preservation but model matching.

As pointed out in subsection 5.4.2, associations are a source of error for change sequence derivation. To evaluate the impact of associations, a modified model match challenge without associations (MMC-) is created. To avoid associations, the `owner` association of class `DomesticAnimal` is replaced with an equally named property of type `Owner`. Accordingly, models based on the associations-avoiding default model are created for all test cases except for *Exchange Elements* (as this does not have associations in first place). Since both the association-based and association-free models correspond to the same Java representation, the target domain extension remains identical and all Java model validation files can be used for both models.

6.2.2. Example *Thesis System*

In their work for extending EMF Compare with semantic matching features, Addazi et al. present the *Thesis System* (TS) example as a motivating scenario [1]. The model is specifically designed to show problematic matching with the EMF Compare default match engine. More specifically, the authors state that EMF Compare is only able to match 25 elements out of 37 manually identified matches. While the executed tests support their



(a) Default

(b) Exchange Elements

Figure 6.3.: UML Models for Example System *Model Match Challenge* [46]

Name	Changes
Exchange Elements	Move class `DomesticAnimal` to package `core`
	Move class `DomesticAnimalNew` to package `shop`
Move	Create package `shop` as subpackage of `de`
	Move class `DomesticAnimal` to package `shop`
Move Renamed	Create package `shop` as subpackage of `de`
	Move class `DomesticAnimal` to package `shop`
	Rename class `DomesticAnimal` to `Pet`
	Rename attribute `nickname` of class `Pet` to `moniker`
Rename	Rename class `DomesticAnimal` to `Pet`
	Rename attribute `nickname` of class `Pet` to `moniker`
Update Reference Target	Change type of attribute `owner` of class `DomesticAnimal` to `Person`

Table 6.3.: Model Match Challenge System Test Description [46]

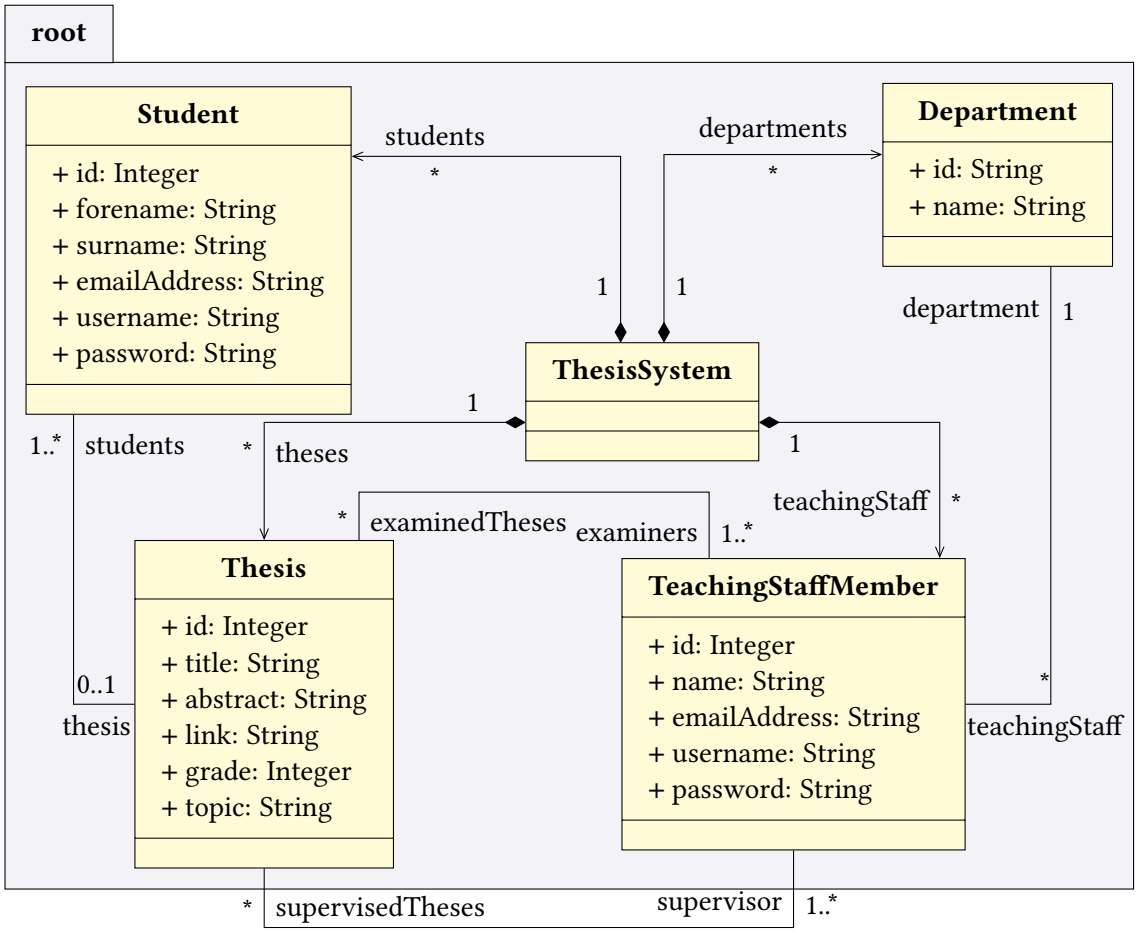


Figure 6.4.: UML Model for Example *Thesis System* [1]
All containment endpoints of `ThesisSystem` are named `thesisSystem`.

statement that the test suite poses problems to EMF Compare, the plain numbers may be misleading as for a perfect matching rate multiple elements from the original model need to be matched with one element from the changed model which is not intended in EMF Compare. The results of the performed tests are discussed in detail in chapter 8.

The initial UML model of the *Thesis System* example is shown in Figure 6.4. The performed test case is described in Table 6.4. Like in subsection 6.2.1, the authors did consider model matching but not model consistency preservation, therefore the target model extensions are added additionally. These include deleting the setter methods for the attributes `id` of class `Department`, `id` and `grade` of class `Thesis`, `departments` of class `ThesisSystems`, and all setter methods of classes `Student` and `TeachingStaffMember`.

6.3. Generated Large-Scale Systems

One common problem during model matching is the compromise between performance and accuracy. Naively, each element needs to be compared to each other element to find a possible matching partner, leading to a run-time of $O(n^2)$ based on the model's elements

Name	Changes
Default Test	Extract superclass `User` from `Student` and `TeachingStaffMember` Create class `User` Make `Student` a subclass of `User` Move all six attributes of `Student` to `User` Make `TeachingStaffMember` a subclass of `User` Delete all five attributes of `TeachingStaffMember` Rename attribute `topic` of class `Thesis` to `subject`

Table 6.4.: Thesis System Test Description [1]

number. To improve this performance, modern model matching algorithms use different heuristics like limiting the search to a certain neighborhood or only matching elements of identical metamodel classes [9]. However, the introduction of these heuristics may reduce the matching accuracy. As for small-scale systems, like the ones described in the previous sections, both performance and accuracy can be kept high with the processing power of modern computers, there may not occur any limitations imposed by the heuristics for performance improvement. Therefore, two large-scale test systems are generated which scale beyond the size of the previously described systems. Since the systems are generated, they are not built using the *Papyrus* editor but created programmatically.

Both generated systems consist of packages, classes, attributes, operations, and operation parameters. Each element is provided a generated name, and operation parameter types are set to one of the basic types `String`, `Boolean`, `Integer`, `Real`. All other attributes are left with their default values. Each class contains between four to seven attributes and three to five operations. Each operation has zero to four parameters and has a 50 % chance of returning `void`, otherwise one of the basic types is returned. The exact numbers are generated randomly using a fixed seed for reproducibility. The number of packages and their hierarchy as well as the number of classes per package is defined individually per system.

For the *Large System* (LS), there are four packages in total: two root packages, and each root package containing another package. Each package contains three classes, making it twelve classes in total. Overall, the system consists of 250 elements. The *Huge System* (HS) uses a package hierarchy of seven levels. There are three root packages, and each package of the first six hierarchy levels contains another two sub-packages. Each package contains ten classes. In total, the system contains 381 packages, 3810 classes, and 88 318 elements.

The test cases used with the generated systems are described in Table 6.5. While the test cases for the large system are close to the ones described in subsection 6.1.1 but include multiple atomic changes of the same kind in a single test scenario, the test cases for the huge system are designed with the explicit intention of breaking the assumptions made by the heuristics of the strategies. More specifically, they try to exploit the locality principle which performs the matching search only on elements within a certain distance. Therefore, in the test cases classes in different hierarchy levels are moved to packages which are *far away*, i. e. the closest shared ancestor between the initial and changed container of the class is multiple hierarchy levels above.

Name	Changes
Large System	
Delete Classes	Delete 4 classes
Delete Methods & Attributes	Delete 4 methods Delete 3 attributes
Move Classes	Move 5 classes
Rename Classes	Rename 5 classes
Rename Single Method	Rename 1 method
Rename Single Attribute	Rename 1 attribute
Rename Methods & Attributes	Rename 4 methods Rename 3 attributes
Huge System	
Move Shallowly Nested Classes	Move 4 classes within hierarchy levels 2-4
Move Deeply Nested Classes	Move 4 classes within hierarchy levels 4-7

Table 6.5.: Generated Large-Scale Systems Test Description

Since the generated test cases do not contain any semantics, the target model extension step of the pipeline and the target model validation are skipped for these systems. Instead, the tests are validated by manually assessing the derived change sequences. Another reason for skipping the mentioned steps is the performance with the Vitruvius framework. While for the large system the consistency preservation is slow but feasible, for the huge system the consistency preservation has to be disabled as it does not terminate after running even for several hours.

6.4. External Model Editor

To simulate a realistic application of state-providing views, we want to use an editor external to the consistency preservation framework. To select an appropriate editor, several requirements were defined. First and most importantly, the editor needs to support serializing the model in UML 2.x XML format. This is a crucial property as it allows to load the created model without modifications into the model consistency preservation tool and thus simplifies the pipeline drastically. If the exported model format was not supported, it either would need to be converted to UML 2.x or an adapter for the new format would have to be added. Secondary requirements are a public license, that it is open-source, and that the editor is still maintained.

The chosen editor that fulfills all stated requirements is *Papyrus* [21]. It supports up to UML 2.5.0 and provides editors for various UML diagram types, including UML class diagrams. It is licensed under the Eclipse Public License 2.0, is open-source, and is under active development. Considering its role as an external editor, it may be questionable that both Papyrus and Vitruvius are based on the Eclipse modeling framework. Theoretically, fine-grained changes could be observed in Papyrus. However, for our scenario

we do not use this feature and only provide state-based differences, making the editor a state-providing view. Furthermore, among the evaluated editors, Papyrus was the only editor to fulfill all requirements thus making it a reasonable trade-off to allow the shared foundations.

7. Findings

This chapter presents the findings obtained by integrating and evaluating state-providing views in delta-based consistency preservation. We first discuss three generic consistency preservation problems for critical cases and their causes. Second, reoccurring error patterns of the similarity-based derivation strategy are presented. Finally, we use our obtained findings to construct a custom similarity-based matching strategy. This strategy is extended with two new matching heuristics and is able to reduce the occurrence probability of the identified error patterns.

7.1. Classification of Consistency Preservation Problems

Since any critical case for model consistency preservation needs to exhibit certain properties, we can deduce problems that may arise due to these properties. As motivated in section 4.3, a view which covers elements from the SUM only partially forms a critical case. We identify two possible problems from this partial projection. Either the additional element information contained in the SUM is deleted though it should be preserved or it is preserved though it should be deleted. We call the former the *information deletion problem* and the latter the *information preservation problem*. For the concept of a V-SUM, we additionally identified that ambiguous representations can cause a critical case. In these cases, the *information ambiguity problem* may occur. This problem describes the case that a certain representation for some ambiguous concept is not preserved during consistency preservation but rather another representation of the multiple allowed ones is created. Since the state of some ambiguous representation can also be treated as some additional information of the target domain, we consider the information ambiguity problem to be a specialized version of the information deletion problem.

For all described problems, in either the correct or incorrect case information of the SUM stripped away in the changed view is deleted. As this additional information is not covered by the view, it cannot get directly modified or intentionally deleted by the consistency preservation. As a consequence, the only change subsequence that can trigger the deletion of this additional information is a deletion operation of the element containing the additional information. Therefore, the information deletion and ambiguity problems are triggered by an incorrect deletion of some element and the information preservation problem is triggered by a missing deletion of such an element. When transforming this problem to the V-SUM approach, the additional information translates to coupled elements of other domains. While any operation could cause a corresponding model element to get deleted, commonly element life cycles are coupled across domains. This means that elements are only deleted by the consistency preservation if their corresponding elements get deleted. This holds true for every element covered by our considered consistency

Problem	Domains Property	Cause	Severity
Information Preservation	Partial Overlap	Incorrect Match	low
Information Deletion	Partial Overlap	Missing Match	very high
Information Ambiguity	Ambiguous	Missing Match	high

Table 7.1.: Consistency Preservation Problems

specification. Thus, the causes of all three problems are in the V-SUM approach also triggered by the incorrect deletion respectively absence of a deletion in the changed view. It is worth noting that the constraint about element life cycle coupling does only affect the causes in a V-SUM of the identified problems. The problems itself remain valid even if the life cycles are not coupled.

In model matching, a deletion of some element is caused by an unmatched element. Whenever no match for an element of the original version could be found, the element is treated as not existent in the changed version anymore and thus deleted. Since for the information deletion and ambiguity problems to occur some element needs to get incorrectly deleted, this incorrect deletion is caused by the absence of a match for the affected element. For the information preservation problem it is the contrary case; the element is incorrectly not deleted thus the element is matched with a wrong element.

Although all problems can occur in the context of consistency preservation, they are of different severity. We assume that it is easier for a developer to manually remove accidentally preserved data than to restore it. This is mainly motivated by the fact that existing data can simply be deleted while the restoration of data requires an additional effort to search for the data to restore in the multitude of previous versions. This problem becomes even more difficult if the absence of information is only detected with delay such that intermediate changes are performed, making it harder to find and correctly restore the lost information. Therefore, the occurrence of the information deletion or information ambiguity problem is treated to be more critical than the occurrence of the information preservation problem. Since in the case of the information ambiguity problem only the ambiguous state information but not actual data is lost, we treat its occurrence as less critical than the occurrence of the information deletion problem. An overview of the problems, their causes in model matching, and their severity for consistency preservation is shown in Table 7.1.

It is important to note that the occurrence of these problems is not only depending on the change sequence but also on the information in the SUM. If the information projected away in the view is equal to its initial values, deleting or preserving it makes no difference as upon regeneration of the element the features would get initialized with the same values again. Therefore, a consistency problem might not appear even though in the derived change sequence an error exists that would lead in other cases to a problem. In our test suites, we try to avoid hidden consistency problems by extending the target domain (see subsection 4.4.1). However, we cannot cover every possible target domain extension. Therefore, to guarantee the correctness of the V-SUM independent of the target domain extension, the actual change sequence needs to be derived. Still, since it is rarely the

case that every element of the target domain has additional information, even a change sequence with potential consistency problems is admissible in most cases.

7.2. Limitations of Existing Derivation Strategies

Since we want to obtain results generalizable to any domain, we need to identify abstract scenarios that pose challenges to the existing strategies. To detect these challenges, common error patterns are identified by assessing the derived change sequences of the various test cases. Using the detected patterns, test cases can be synthesized to show the limitations of the existing strategies.

7.2.1. Identification of Error Patterns

To identify failure reasons on the fine-granular level of a single test case, the generated change sequence can be compared to the expected sequence. While this can be helpful to detect issues with the consistency specification or the used consistency preservation framework, the comparison is too specific to allow a general conclusions on the used change sequence derivation strategy. Therefore, reasons for the identified failures need to be grouped to reoccurring patterns which abstract away from underlying domain semantics or element-specific properties. In this thesis, we identify two generic patterns for the similarity-based matching strategy. Leaving out failures caused by domain particularities (see section 5.4), all incorrect change subsequences could be mapped to one of these two patterns. For the identity-based matching strategy, no reoccurring patterns are found. The absence of patterns with this strategy is caused by the high matching rate of the derived change sequence with the actual one. In chapter 8 this accuracy of the identity-based strategy is explained in detail.

The identified patterns are an *undetected movement* operation and an *undetected renaming* operation. Both patterns manifest themselves in a change sequence by an unexpected high number of movement operations. When assessing the change sequences in detail, the moved respectively renamed element gets deleted and a new element with the same attributes gets inserted at the new location or with the new name. Subsequently, all children of the affected and deleted element are attached to the newly created element which causes the high number of movement operations. A list of the pattern occurrences when running the test cases with the similarity-based matching strategy is shown in Table 7.2. As one would expect, the number of occurring error patterns increases with the complexity of the system. Furthermore, renaming operations pose a harder challenge to the derivation strategies than movement operations. This is indicated by undetected renaming cases for the simplest test suite of atomic change operations and no renaming detection at all for the *Large System* whereas movement operations are correctly identified for that system. One special case is produced by the *Thesis System* test. Here, a superclass is extracted from two classes which share the same attributes. For the superclass extracting, the attributes of class `Student` are moved to the superclass while the attributes of class `TeachingStaffMember` are deleted. However, in the derived change sequence the attributes of `TeachingStaffMember` are moved instead of those of class `Student`. Although a move-

ment is correctly identified, we still count those operations as undetected movements as they move the wrong elements.

It is important to note that an undetected operation in an element does not propagate matching errors down the hierarchy, i.e. descendants of an unmatched container can still get correctly matched. This can be seen in the change sequence by the performed movement operations on the children. If the unmatched container prevented its descendants from correctly matching, they would also get deleted and regenerated. The local boundedness of the matching error is a crucial property and reduces the severity of the identified patterns by far, as consistency problems can only occur for the scope of the unmatched element instead of its entire sub-hierarchy.

7.2.2. Synthesis of Failing Test Cases

Considering the consistency preservation problems described in section 7.1, both detected error patterns may cause an information deletion problem or information ambiguity problem. These problems may occur as in both patterns the affected element is incorrectly deleted instead of being modified. Due to the incorrect deletion, additional information inherent to the element of the target domain respectively the state of an ambiguous representation is lost. Upon recreation of the element, the additional information cannot be restored and the ambiguous representation is set to its default state — as defined by the consistency specification — which does not necessarily match its previous representation. Since the patterns can cause the information deletion problem, they cannot cause its contrary problem, the information preservation problem.

To show that the described problems may actually occur, test cases exploiting the described patterns can be synthesized with the intention to fail for the similarity-based matching strategy. To synthesize a failing test case, a change that triggers any of the error patterns is required. Additionally, the element affected by the pattern needs to have a non-default target domain representation. This can either be an extension with additional information to cause the information deletion problem, or a modified representation of an ambiguity to cause the information ambiguity problem. Using the described approach, consistency problems could theoretically get injected to any of the test cases in which an error pattern occurs. In practice however, the test cases are limited due to the target domain modification constraints. Due to the large overlap of UML class diagrams and Java source code, the possible modifications to the target domain are strictly limited (see subsection 5.1.1).

As a basis for concrete test cases, the *Rename* and *Move Renamed* scenarios of the *Model Match Challenge* system are reused. In both test cases, a UML attribute is affected by the error patterns which allows a target domain modification. To create this modification, its Java setter method is deleted. With only this modification to the target domain, for both test cases the consistency preservation regenerates the setter method when using the similarity-based matching strategy, thus causing the *information ambiguity problem.*

Test Case	Undetected Movement	Undetected Renaming
Atomic Change Operations		
Move Attribute	0 / 1	-
Move Class 1	0 / 1	-
Move Class 2	0 / 1	-
Rename Class	-	0 / 1
Rename Method	-	0 / 1
Rename Attribute	-	0 / 1
Rename Class	-	**1** / 1
Rename Method	-	0 / 1
Common Refactoring Operations		
Change Method Signature	-	0 / 1
Collapse Hierarchy	0 / 1	-
Extract Associated Class[1]	0 / 2	-
Extract Superclass[2]	0 / 2	0 / 1
Inline Class	0 / 1	-
Model Match Challenge[3]		
Exchange Elements	0 / 2	-
Move	0 / 1	-
Move Renamed[2]	**1** / 1	**2** / 2
Rename	-	**2** / 2
Thesis System		
Default Test	**4** / 6	**1** / 1
Large System		
Move Classes	0 / 5	-
Rename Classes	-	**5** / 5
Rename Single Method	-	**1** / 1
Rename Single Attribute	-	**1** / 1
Rename Methods & Attributes	-	**7** / 7
Huge System		
Move Shallowly Nested Classes	**4** / 4	-
Move Deeply Nested Classes	**4** / 4	-

Table 7.2.: Error Pattern Occurrences per Test Case for Similarity-Based Strategy
Test cases without any movement or renaming operation are not listed.

[1] Evaluation performed on a non-conservative change sequence.

[2] For these test cases, the movement and renaming operations are correlated.

[3] Pattern occurrence is identical for the Model Match Challenge system with and without associations.

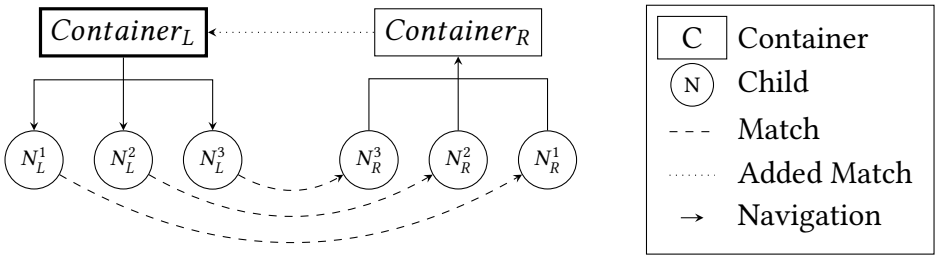


Figure 7.1.: Match Correction for Unmatched Containers

7.3. Custom Similarity-Based Matching Strategy

Using our findings for consistency preservation problems and error patterns in the current similarity-based derivation strategies, we want to provide a strategy that is optimized for model consistency preservation. We motivated the information deletion and the information ambiguity problems to be those problems with the highest severity for consistency preservation. Therefore, our custom similarity-based matching strategy should aim to reduce the occurrence probability of these problems. Since both problems are caused by incorrectly unmatched elements, an approach to avoiding them is to increase the number of matches identified by the strategy. While this could be achieved by lowering the threshold of the similarity distance function to consider two elements sooner as a match, the side effects introduced by modifying this already elaborate function are hard to estimate. Therefore, the threshold is not changed. Instead, the extended matching is applied after the matching of the default similarity-based strategy is completed. Another benefit of this post-processing approach is that most elements are already matched which reduces the search space. However, producing additional matches always comes with the risk of matching the wrong elements which can lead to the information preservation problem. Although the strategy is designed with this risk in mind, there can always be unconsidered cases where incorrect matches get created. Nevertheless, the severity of the information preservation problem is low compared to the problems aimed to be solved with the extended matching.

We know from the identified error patterns that it is common that a renamed or moved container is not matched but its children are. Furthermore, due to the hierarchical navigability of both the matches and the elements, the correct match for the container element can be reached by accessing the parent of a matched element of a child. Therefore, in the custom similarity-based matching strategy, the missing match is restored by traversing this path to retrieve the matching container (Figure 7.1). To limit the impact of incorrect child matches, all matched children of some unmatched container must be matched to an element with the same container. If there are multiple possible container matches, no match is created. A more aggressive approach to consider incorrect child matches would be to already treat a container as the correct match when it is referred to by only a certain percentage of the element's children. However, in the evaluated test cases there was no benefit of allowing this. Besides of its uniqueness among the children, the found container is also required to be unmatched and to be an instance of the same metaclass as the unmatched container. While it is theoretically possible that one element of one model

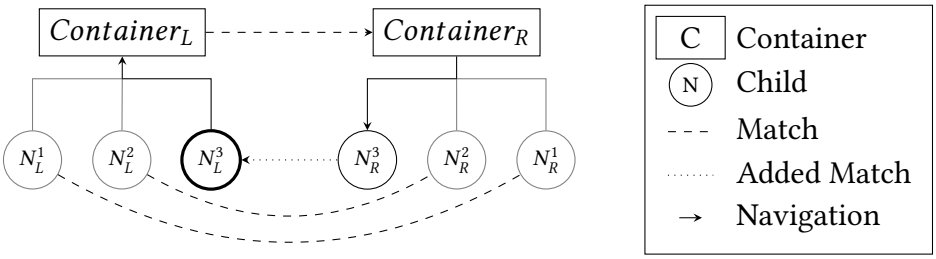


Figure 7.2.: Match Correction for Unmatched Leaves

version corresponds to multiple elements of a different model version, EMF Compare restricts this to at most one match per element. The metaclass constraint is required as an element's metaclass cannot change after instantiation, thus elements with different metaclasses cannot be changed versions of another.

One problem with restoring element matches based on its children occurs when the unmatched element does not have any children (a *leaf* element). Without any children to traverse, no corresponding element can be found. However, this scenario was detected in the test cases for different renaming scenarios. To deal with it, the custom matching strategy is extended with another matching heuristic. For any unmatched leaf, all unmatched children with the same metaclass as the unmatched element of both the leaf's container and its matched counterpart are collected. If for the leaf's container only the leaf is unmatched and for the counterpart there is also only one unmatched child with a matching metaclass, the leaf is matched to this child (Figure 7.2). This restriction is used to prevent multiple unmatched leaves of the same container to be matched to the same element. Since this matching does rely only on the content hierarchy and not on any distance metric inherent to the matched element, it contains a high risk of introducing the information preservation problem. To allow an optimized balance between accidental deletion and accidental preservation, the match correction for unmatched leaves can be disabled depending on the use case. However, for the considered test cases the evaluation (chapter 8) shows that with this aggressive merging technique enabled, the results improve further.

An additional customization point introduced with the custom matching strategy is to prevent the extended matching for certain elements. While per default every unmatched element is eligible for the extended matching, developers can limit this set to compensate for known matching issues in the particular use case or to prevent unintended side effects when extending the strategy with domain-specific matching logic. In the considered use case of UML class diagrams and Java classes, UML associations may get incorrectly matched (see section 5.4) already in the original matching. Therefore, associations or any of their descendants are excluded from the extended matching to prevent subsequent matching errors.

8. Evaluation

In this chapter we present the obtained results by evaluating the different strategies against our created test suites. First, we show how the identity- and similarity-based strategies perform in terms of deriving a conservative, admissible, or even the actual change sequence. Then, we compare the outcome of our extended similarity-based matching strategy to these results. Second, we analyze the performance overhead of state-providing views when compared to delta-providing views as well as the performance comparison of the three strategies. Finally, we present threats to the validity of our obtained findings.

8.1. Test Suite Results

Each of the considered strategies is run against all seven test suites. We defined a test case to succeed whenever the derived change sequence is admissible which is indicated by a correct source and target model. As an additional metric, for each derived change sequence it was annotated whether it matches the actual change sequence. Since the admissibility of a change sequence depends on the target domain extension, the test case could fail if the extension got changed. If the actual change sequence is derived, the test case will succeed in every case in which a delta-providing view would succeed, as they provide the same change sequence in this case. We consider a change sequence to match the actual change sequence if it consists of the same operations, independent of the order. Since we only consider a correct consistency specification, any order of the same operations produces the same outcome.

8.1.1. Existing Strategies Results

The test results for the identity- and basic similarity-based matching strategies are shown in Table 8.1. For the identity-based matching strategy one can see that in all but one case the actual change sequence was derived. This is reasonable as due to the immutability of the element identifiers the matching phase can produce perfect results. It is worth noting that the matching is not influenced by the size of the system, as even for the large-scale generated models the actual change sequences are derived. Since the actual change sequence is also conservative and admissible, all test cases except for the one mentioned are passed successfully with the identity-based matching strategy.

For the similarity-based strategy, the actual change sequence is only derived in less than half of all test cases. In UML models with associations, no change sequence matches the actual change sequence due to the known problems with deriving changes for UML associations (see section 5.4). To not distort the results by the UML associations matching problem, we added an additional comparison to check whether the actual change sequences

Test Suite	Identity-Based			Similarity-Based			
	conservative	admissible	actual	conservative	admissible	actual	actual*
Atomic Change Operations	13/13	13/13	13/13	13/13	13/13	12/13	-
Common Refactoring Operations	5 / 6	5 / 6	5 / 6	4 / 6	4 / 6	0 / 6	4 / 6
Model Match Challenge (with Associations)	5 / 5	5 / 5	5 / 5	3 / 5	3 / 5	1 / 5	3 / 5
Model Match Challenge (without Associations)	4 / 4	4 / 4	4 / 4	4 / 4	4 / 4	2 / 4	-
Thesis System	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	0 / 1	0 / 1
Large System	7 / 7	-	7 / 7	7 / 7	-	3 / 7	-
Huge System	2 / 2	-	2 / 2	2 / 2	-	0 / 2	-
Total	37/38	28/29	37/38	34/38	25/29	18/38	24/38

Table 8.1.: Change Sequence Properties Achieved by Existing Strategies
The *actual** column counts those cases where the actual change sequence was derived but extended with noise due to faulty UML associations matching.

would have been derived if associations had been excluded (column *actual**). This raises the actual change sequence quota to slightly above 60 %. The test cases with non-actual change sequences are exactly those where one of the two identified error patterns occur (see section 7.2). Although the actual change sequence quota is significantly lower compared to the identity-based strategy, the rate of successful test cases, i.e. derived admissible change sequences, is with above 86 % closer to the one of the identity-based strategy (96 %). Since in the test cases only parts of the target domain are extended, some occurring matching errors do not result in a consistency problem and therefore the consistency preservation can still produce correct results. However, it is worth noting that the test success rate is dependent on the target domain extension and could drop down to the actual change sequence quota if target domain extensions were constructed with the intention to fail the test cases.

Even though we required every change sequence to be conservative, the tested strategies fail to provide such sequences in some cases. For the identity-based matching strategy, one non-conservative change sequence is derived. For the similarity-based matching strategy, four non-conservative change sequences are derived. In all of these five cases, the failure is caused by UML associations. In the one case produced by the identity-based matching strategy and in three cases produced by the similarity-based strategy, children of an association are incorrectly managed and remain as dangling elements in the resource which is forbidden. In the last case of the similarity-based matching strategy, an association end is not correctly renamed to its new name. All these cases can be corrected to produce conservative (and in these cases also admissible) change sequences by replacing the problematic association with a UML property.

Test Suite	Similarity-Based		Custom (Conservative)		Custom (Aggressive)	
	actual	actual*	actual	actual*	actual	actual*
Atomic Change Operations	12 / 13	-	13 / 13	-	13 / 13	-
Common Refactoring Operations	0 / 6	4 / 6	0 / 6	4 / 6	0 / 6	4 / 6
Model Match Challenge (with Associations)	1 / 5	3 / 5	1 / 5	3 / 5	1 / 5	3 / 5
Model Match Challenge (without Associations)	2 / 4	-	2 / 4	-	4 / 4	-
Thesis System	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1	0 / 1
Large System	3 / 7	-	4 / 7	-	7 / 7	-
Huge System	0 / 2	-	0 / 2	-	0 / 2	-
Total	18 / 38	24 / 38	20 / 38	26 / 38	25 / 38	31 / 38

Table 8.2.: Change Sequence Properties Achieved by Custom Strategy
The *actual** column counts those cases where the actual change sequence was derived but extended with noise due to faulty UML associations matching.

8.1.2. Custom Similarity-Based Matching Strategy Results

For the similarity-based matching strategy with extended matching there exist two different configurations. We call these configurations *conservative* and *aggressive*, depending on whether the matching of single leaves is enabled (aggressive) or disabled (conservative). For both configurations, the conservative and admissible properties of the change sequences for each test case remain unchanged compared to the similarity-based matching strategy. It is expected that the results do not worsen as the custom strategy only extends the matches produced by the similarity-based strategy. Additionally, since all non-conservative change sequences are caused by UML associations which are not post-processed by the custom strategy, correcting the derived change sequences for these cases was not possible.

Even though the number of passed tests remains identical compared to the similarity-based strategy, an improvement is achieved when comparing the change sequence to the actual change sequence (Table 8.2). Here, the conservative configuration produces the actual change sequence in two and the aggressive configuration in seven additional cases. When excluding the four cases for which no conservative strategy could be derived, the aggressive strategy is able to derive the actual change sequence for all test cases of medium-sized systems except for one and for the *Large System*. By being closer to the actual change sequences, achieving a successful test result is less dependent on the target domain extension. As an example, the aggressive configuration is able to pass the test cases which were constructed to fail for the similarity-based strategy (see subsection 7.2.2).

To get a more fine-grained insight on the improvements of the custom strategy, we compared the occurring error patterns to those of the default similarity-based strategy (Table 8.3). Since the custom strategy only adds additional matches, no new unmatched movement or unmatched renaming errors can occur. For the conservative configuration, 11 out of 33 previously occurring error patterns got corrected. Especially the detection of renaming operations was improved. With the aggressive configuration, there are only

Test Case	Similarity -Based		Custom (Conservative)		Custom (Aggressive)	
	U. Move	U. Rename	U. Move	U. Rename	U. Move	U. Rename
Atomic Change Operations						
Rename Class	-	1 / 1	-	**0** / 1	-	**0** / 1
Model Match Challenge[1]						
Move Renamed[2]	1 / 1	2 / 2	**0** / 1	**1** / 2	**0** / 1	**0** / 2
Rename	-	2 / 2	-	**1** / 2	-	**0** / 2
Thesis System						
Default Test	4 / 6	1 / 1	4 / 6	1 / 1	4 / 6	**0** / 1
Large System						
Rename Classes	-	5 / 5	-	**0** / 5	-	**0** / 5
Rename Single Method	-	1 / 1	-	1 / 1	-	**0** / 1
Rename Single Attribute	-	1 / 1	-	1 / 1	-	**0** / 1
Rename Methods & Attributes	-	7 / 7	-	**5** / 7	-	**0** / 7
Huge System						
Move Shallowly Nested Classes	4 / 4	-	4 / 4	-	4 / 4	-
Move Deeply Nested Classes	4 / 4	-	4 / 4	-	4 / 4	-
Total	13 / 15	20 / 20	**12** / 15	**10** / 20	**12** / 15	**0** / 20

Table 8.3.: Error Pattern Occurrences per Test Case for Custom Strategy
Test cases with no occurring error patterns for the similarity-based strategy are not listed.

[1] Pattern occurrence is identical for the Model Match Challenge system with and without associations.
[2] For this test case, the movement and renaming operations are correlated.

three test cases left in which error patterns occur. In the *Default Test* of the *Thesis System* example, the default similarity-based matching strategy identifies the wrong elements to be moved which cannot get corrected by the extended matching. For both test cases of the *Huge System*, due to the system's size the accuracy of the local bound matching search of the default similarity-based strategy is strongly reduced. As the extended matching requires a certain level of matching accuracy which is not reached anymore, there are no benefits with the custom strategy for such large systems. Besides of correcting the results for several previously occurring error patterns, no new errors in the form of incorrect matches are produced. Interestingly, this applies to both strategy configurations, showing that the restrictions chosen for the aggressive configuration are sufficient. However, the considered test cases are not designed for revealing information preservation problems and thus the absence of incorrect matches may be biased on the considered test suites.

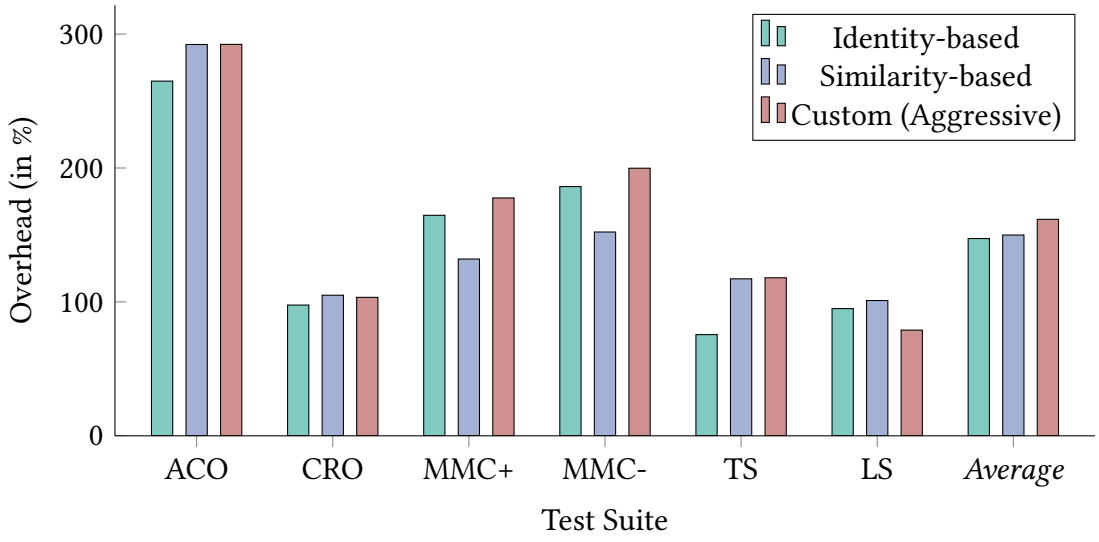


Figure 8.1.: Average Performance Overhead of State-Providing Views compared to Delta-Providing Views

8.2. Performance Overhead of State-Providing Views

In comparison to delta-providing views, state-providing views require an additional computation phase when used in delta-based consistency preservation. This phase is the derivation of the change sequence from the state differences. Since all other phases of the consistency preservation are identical for both view types, state-providing views introduce a performance overhead. As the availability of a change sequence is a requirement for delta-based consistency preservation, this overhead is unavoidable. To measure it, we compare the time to derive a change sequence T_{CS} to the overall execution time T_{V_D} of the consistency preservation for delta-providing views. Since the execution time of delta-providing views is the same as for state-providing views minus the additional time of the change sequence derivation, we can measure the overall execution time of the state-providing views T_{V_S} and compute T_{V_D} by $T_{V_D} = T_{V_S} - T_{CS}$. By measuring both values in the same test run, distortions in the measurements due to different caching or machine workload can be reduced. The performance overhead is computed by $T_{CS}/T_D = T_{CS}/(T_{V_S} - T_{CS})$ and is shown in Figure 8.1. Each test suite is run twenty times and for each test case the overhead average over these twenty runs is computed. For a test suite, the overhead is the average of the averaged overheads of each test case within that suite.

The results show that the averaged overhead is between 75 % and 300 %, depending on the test suite. Interestingly, the overhead reduces with the complexity of the system. This is reasonable as for larger systems and longer change sequences, the consistency preservation execution time also increases. Furthermore, the overhead across the strategies appears to remain stable. Since for each strategy the derived change sequence is taken as the measurement baseline, the execution time of the consistency preservation varies. If the strategy derives a change sequence with more operations, the execution time of the consistency preservation increases. Therefore, the relative overhead should not be compared across strategies as different baselines are involved.

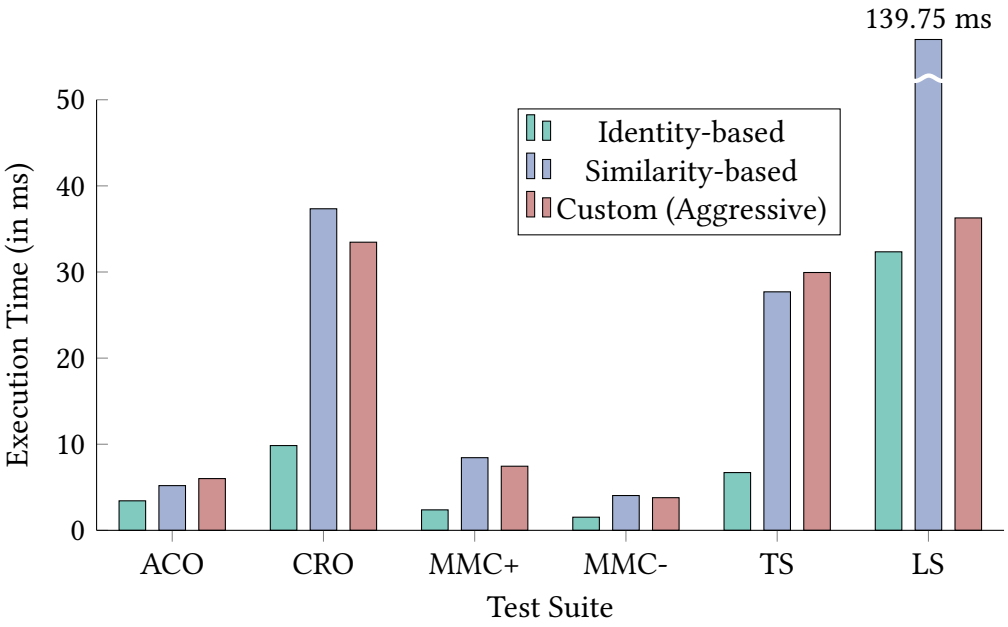


Figure 8.2.: Absolute Execution Time of Change Sequence Derivation

A better metric to compare the strategies is their absolute execution time to derive a change sequence which is shown in Figure 8.2. The execution times are the absolute values of the overhead computation and are averaged using the same approach. In general, the identity-based matching outperforms the other strategies in any test suite. This is reasonable as the matching logic for similarity-based strategies is more complex than an identifier comparison and therefore takes more times. We can also see that all strategies perform slower with growing complexity of the system. However, the identity-based matching strategy scales better than the similarity-based ones. While the similarity-based strategy is around 50 % slower for the Atomic Change Operations, this increases to around 280 % for the Common Refactoring Operations and even 330 % for the Large System. Although one reason for this is also the simpler matching of the identity-based strategy, it is additionally influenced by incorrect or missing matches. As for similarity-based strategies the number of matching errors increases with the system complexity, the effort of the differencing phase increases and thus also the execution time. The difference in scalability manifests itself even more when assessing the execution time for the Huge System tests (Figure 8.3). Here the similarity-based strategies are around 15 (*Move Shallow*) respectively 22 (*Move Deep*) times slower than the identity-based strategy. Interestingly, for the identity-based strategy there is no significant difference whether shallowly or deeply nested classes are moved. In contrast, the similarity-based strategies require almost double the time to derive a change sequence for the movement of the deeply nested classes.

Even though the custom matching strategy adds an additional matching step compared to the similarity-based strategy, its performance is similar or even better. While the basic matching for both strategies is identical and thus should last the same time, improvements in the matching phase reduce the execution time of the differencing phase. A significant improvement can be seen for the Large System, where the custom strategy achieves an

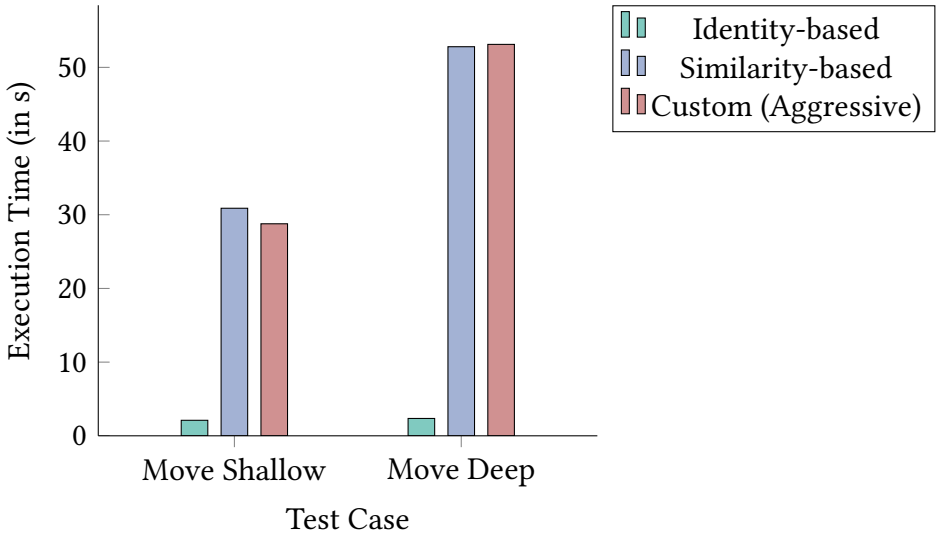


Figure 8.3.: Absolute Execution Time of Change Sequence Derivation for Huge System

almost four times speed-up. Since all occurring matching errors in this test suite are resolved by the custom strategy, the number of detected differences decreases in the most extreme case of the *Rename Classes* test suite from 57 to 5 change operations. Another reason for the only marginal overhead of the extended matching is that the existing strategy already matches the majority of elements. Therefore, the extended matching is only performed on a fraction of the entire model.

8.3. Threats to Validity

To conclude the evaluation, in this section we discuss the threats to validity of our results. We see the strongest threat in the choice and coverage of our considered test suites. Although it is tried to obtain a broad coverage by using multiple approaches to constructing a test suite, constraining the considered elements and features to a small subset of the domain, and reusing relevant scenarios from existing literature, certain cases may not be considered. Similarly, there may exist pairs or tuples of domains for which our generalization of the employed UML to Java case study do not apply. Since in these unconsidered cases new consistency problems may occur, further evaluation of state-providing views in different contexts has to be done in the future.

Regarding our extended similarity-based matching strategy, we identify two possible threats. First, as the strategy is extended with matching heuristics obtained exclusively from our own evaluation, it may be overfitted to our test suites. This could reduce its performance in other applications by creating wrong matches leading to the information preservation problem. However, due to the strict constraints on the added matches, we assume to just not generate any additional matches in the worst case instead of wrong ones. Second, the performance of the custom strategy is only evaluated for cases with an already high matching rate. Since we are looping over the unmatched elements of both models, the performance of the extended matching routine is $O(n^2)$. While this is

not problematic for a few elements, if the default similarity-based matching is not able to match a large number of elements, we expect the execution time to increase significantly. However, our tests show that even for models with over 80 000 elements the matching rate remains stable enough to not have strong influence on the execution time of the extended matching.

Finally, by using the model consistency framework Vitruvius our results are dependent on its correctness. This includes its in-memory model representation, the used consistency specification, as well as its management of the V-SUM state. Exemplary, if a deleted element is preserved too long in memory, the find-or-create pattern may reuse it to create a correspondence which could incorrectly preserve information. We minimized this threat by additionally assessing each derived change sequence manually.

9. Future Work

Besides of mostly providing only state-based information, there are other aspects of industry tooling that need to be considered to fully support these in automated model consistency preservation. Answering these questions is a task for future work. Furthermore, the validity of our results can be assessed by extending the considered domains or evaluating changes from industrial applications.

Besides of their state-providing trait, models from real world applications introduce additional challenges to automated model consistency preservation. It is possible that these models contain information inherent to the used editor that should not be included in the SUM, like layout information. When regenerating such a model from the SUM, solutions need to be found to cope with this information, either leaving them out or preserving them from the previous view state. Another challenge is dealing with simultaneous changes to the SUM and the view. Commonly, an engineer works on a local copy of a model. If the SUM is changed but the local model is not refreshed, subsequently propagated changes from the model are based on an outdated state and may reference deleted or changed elements. To start the consistency synchronization, in this thesis manual triggers are used. These need to be extended to automatic triggers which allow view synchronization without user input. Imaginable triggers are on every file save or on every model versioning commit. For the choice of triggers the rate of the synchronization needs to be weighed up against the size of the changes, as the derivation imprecision increases with larger changes.

As an early approach, we evaluate two existing strategies and our modified one in a UML to Java case study. In future work, the evaluation should be extended to further domains and to incorporate changes of real applications. Exemplary, this could be achieved by using the change history of existing projects. Additionally, in the context of a V-SUM the impact of state-providing views for a network of transformations can be assessed. As in this thesis only binary, non-transitive transformations are considered, there might occur new consistency problems in transitively linked domains or when having circular consistency dependencies.

Another interesting topic is whether the information in the SUM can be used to improve the change sequence derivation for a view. Although the change sequences are applied to the entire SUM, currently only the information of the view is used for the sequence derivation. Providing additional information to the change sequence derivation strategies may improve their matching accuracy. A challenge of this approach is that SUM information is only available for the initial view state but not for the changed one.

10. Conclusion

In this thesis, we showed how state-based differences can be integrated into delta-based model consistency preservation. This was done as in real world applications the delta-based changes are rarely available. For the evaluation, we created different test suites with consistent UML class diagrams and Java source code. A test pipeline was created which uses the Vitruvius framework to preserve model consistency and to propagate the derived change sequences. Change sequences were derived by using the EMF Compare framework. To obtain meaningful results, the existing consistency specification was extended to support all considered operations.

As an evaluation metric, we introduced the term of an *admissible* change sequence. This was necessary to identify cases in which an incorrect change sequence is derived which application still produces the correct SUM. We discussed properties of tuples of domains that pose additional challenges to consistency preservation, and showed that the chosen domains are a *critical* domains scenario. We presented steps taken in the design of the test suites to allow general conclusions for arbitrary domains from the findings. These span over the choice of a critical domains scenario, the XML serialization of UML class diagrams, and the restriction of considered elements and attributes in the individual test suites.

In the evaluation, we identified two reoccurring problems of similarity-based matching and one error pattern specific to the chosen domain. We showed that for the identity-based matching strategy almost always the actual change sequence is derived. The strategy only failed once due to the domain-specific error. For the similarity-based strategy, we achieved a test success rate of over 86 % while deriving the actual change sequences in three out of five cases.

To avoid the identified problems with similarity-based matching, we presented a customized similarity-based matching strategy that aims at reducing the occurrence probability of the error patterns. We showed that even though the passed tests remained unchanged, we were able to halve the number of occurring error patterns and derive the actual change sequence in six additional cases. In terms of performance, we were able to introduce only a marginal overhead compared to similarity-based matching and in some cases were even able to improve the overall performance.

In conclusion, our results indicate that using state-providing views with delta-based consistency preservation is feasible. Especially when identity-based matching is possible, there was almost no difference to using delta-providing views. For similarity-based matching, we proposed an extended strategy to reduce critical problems that may appear with the default strategy. Although the results appear promising, further case studies based on other domains or realistic change histories need to be executed. Additionally, there are other unresolved problems to integrate existing model editing tools into automated model consistency preservation.

Bibliography

[1] Lorenzo Addazi et al. “Semantic-based Model Matching with EMFCompare”. In: *10th Workshop on Models and Evolution*. Ed. by Tanja Mayerhofer et al. CEUR-WS, Oct. 2016, pp. 40–49. URL: http://www.es.mdh.se/publications/4468-.

[2] Thorsten Arendt et al. “Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations”. In: vol. 6394. Oct. 2010, pp. 121–135. ISBN: 978-3-642-16144-5. DOI: 10.1007/978-3-642-16145-2_9.

[3] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. “Orthographic Software Modeling: A Practical Approach to View-Based Development”. In: *Evaluation of Novel Approaches to Software Engineering*. Ed. by Leszek A. Maciaszek, César González-Pérez, and Stefan Jablonski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 206–219. ISBN: 978-3-642-14819-4. DOI: 10.1007/978-3-642-14819-4_15.

[4] Colin Atkinson and Christian Tunjic. “A Deep View-Point Language for Projective Modeling”. In: *2017 IEEE 21st International Enterprise Distributed Object Computing Conference (EDOC)*. 2017, pp. 133–142. DOI: 10.1109/EDOC.2017.26.

[5] Latifa Ben Arfa Rabai, Barry Cohen, and Ali Mili. “Programming Language Use in US Academia and Industry”. In: *Informatics in Education* 14 (Oct. 2015), pp. 143–160. DOI: 10.15388/infedu.2015.09.

[6] Benjamin Biegel et al. “Comparison of Similarity Metrics for Refactoring Detection”. In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. MSR ’11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 53–62. ISBN: 9781450305747. DOI: 10.1145/1985441.1985452. URL: https://doi.org/10.1145/1985441.1985452.

[7] Stefan Bodewig, Jeff Martin, and Tim Bacon. *XMLUnit - Unit Testing XML for Java and .NET*. URL: https://www.xmlunit.org (visited on 06/14/2021).

[8] Petra Brosch et al. “An Introduction to Model Versioning”. In: *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. Ed. by Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 336–398. ISBN: 978-3-642-30982-3. DOI: 10.1007/978-3-642-30982-3_10. URL: https://doi.org/10.1007/978-3-642-30982-3_10.

[9] Cédric Brun and Alfonso Pierantonio. “Model differences in the eclipse modeling framework”. In: *UPGRADE, The European Journal for the Informatics Professional* 9.2 (2008), pp. 29–34.

[10] Erik Burger and Oliver Schneider. "Translatability and Translation of Updated Views in ModelJoin". In: *9th International Conference on Theory and Practice of Model Transformations, ICMT 2016 Held as Part of Conference on Software Technologies: Applications and Foundations, STAF 2016; Vienna; Austria; 4 July 2016 through 5 July 2016. Ed.: P. Van Gorp*. Vol. 9765. Lecture Notes in Computer Science. 37.06.01; LK 01. Springer International Publishing, 2016, pp. 55–69. ISBN: 978-3-319-42063-9. DOI: 10.1007/978-3-319-42064-6_4.

[11] Erik Burger et al. "View-Based Model-Driven Software Development with ModelJoin". In: *Softw. Syst. Model.* 15.2 (May 2016), pp. 473–496. ISSN: 1619-1366. DOI: 10.1007/s10270-014-0413-5. URL: https://doi.org/10.1007/s10270-014-0413-5.

[12] Fei Chen. "Änderungsgetriebene Konsistenzhaltung zwischen UML-Klassenmodellen und Java-Code". Bachelor's Thesis. Karlsruhe, Germany: Karlsruher Institut für Technologie (KIT), May 24, 2017.

[13] Antonio Cicchetti, Federico Ciccozzi, and Thomas Leveque. "A hybrid approach for multi-view modeling". In: *ECEASST* 50 (Jan. 2011). DOI: 10.14279/tuj.eceasst.50.738.

[14] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. "A Metamodel Independent Approach to Difference Representation." In: *Journal of Object Technology* 6 (Oct. 2007), pp. 165–185. DOI: 10.5381/jot.2007.6.9.a9.

[15] Manuel Clavel et al. *All About Maude - A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Vol. 4350. Jan. 2007. ISBN: 978-3-540-71940-3. DOI: 10.1007/978-3-540-71999-1.

[16] Zinovy Diskin et al. "From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case". In: *Model Driven Engineering Languages and Systems*. Ed. by Jon Whittle, Tony Clark, and Thomas Kühne. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 304–318. ISBN: 978-3-642-24485-8. DOI: 10.1007/978-3-642-24485-8_22.

[17] Brian Dobing and Jeffrey Parsons. "Dimensions of UML Diagram Use". In: *Journal of Database Management* 19 (July 2010), pp. 1–18. DOI: 10.4018/jdm.2008010101.

[18] Anthony Finkelstein et al. "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development". In: *International Journal of Software Engineering and Knowledge Engineering* 02.01 (1992), pp. 31–57. DOI: 10.1142/S0218194092000038. URL: https://doi.org/10.1142/S0218194092000038.

[19] Eclipse Foundation. *Eclipse Modeling Framework*. URL: https://www.eclipse.org/modeling/emf/ (visited on 06/14/2021).

[20] Eclipse Foundation. *EMF Compare*. URL: https://www.eclipse.org/emf/compare (visited on 06/14/2021).

[21] Eclipse Foundation. *Papyrus*. URL: https://www.eclipse.org/papyrus/ (visited on 06/14/2021).

[22] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Ed. by Kent Beck. Second edition. The Addison-Wesley signature series - A Martin Fowler signature book. Boston, MA, USA: Addison-Wesley, 2019. ISBN: 9780134757698.

[23] Enterprise Big Data Framework. *Data Types: Structured vs. Unstructured Data*. Jan. 2019. URL: https://www.bigdataframework.org/data-types-structured-vs-unstructured-data/ (visited on 06/14/2021).

[24] Sinem Getir et al. "CoWolf – A Generic Framework for Multi-view Co-evolution and Evaluation of Models". In: vol. 9152. Springer. July 2015, pp. 34–40. ISBN: 978-3-319-21154-1. DOI: 10.1007/978-3-319-21155-8_3.

[25] Object Management Group. *Meta Object Facility*. Oct. 2016. URL: https://www.omg.org/spec/MOF/ (visited on 06/14/2021).

[26] Object Management Group. *Unified Modeling Language Specification*. URL: https://www.omg.org/spec/UML/ (visited on 06/14/2021).

[27] Florian Heidenreich et al. "Closing the Gap between Modelling and Java". In: *Software Language Engineering*. Ed. by Mark van den Brand, Dragan Gašević, and Jeff Gray. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 374–383. ISBN: 978-3-642-12107-4.

[28] Frédéric Jouault et al. "ATL: A QVT-like Transformation Language". In: *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 719–720. ISBN: 159593491X. DOI: 10.1145/1176617.1176691. URL: https://doi.org/10.1145/1176617.1176691.

[29] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. "A rule-based approach to the semantic lifting of model differences in the context of model versioning". In: Dec. 2011, pp. 163–172. DOI: 10.1109/ASE.2011.6100050.

[30] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. "Consistency-preserving edit scripts in model versioning". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2013, pp. 191–201. DOI: 10.1109/ASE.2013.6693079.

[31] Marouane Kessentini et al. "Search-based metamodel matching with structural and syntactic measures". In: *Journal of Systems and Software* 97 (2014), pp. 1–14. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2014.06.040. URL: http://www.sciencedirect.com/science/article/pii/S0164121214001484.

[32] Heiko Klare. *Vitruvius Wiki - The Reactions Language*. URL: https://github.com/vitruv-tools/Vitruv/wiki/The-Reactions-Language#Reactions (visited on 06/14/2021).

[33] Heiko Klare and Joshua Gleitze. "Commonalities for Preserving Consistency of Multiple Models". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Sept. 2019, pp. 371–378. ISBN: 978-1-7281-5125-0. DOI: 10.1109/MODELS-C.2019.00058. URL: http://dx.doi.org/10.1109/MODELS-C.2019.00058.

[34] Heiko Klare et al. “Enabling consistency in view-based system development – The Vitruvius approach”. In: *Journal of Systems and Software* 171 (2021). ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110815. URL: http://www.sciencedirect.com/science/article/pii/S0164121220302144.

[35] Dimitrios Kolovos et al. “Different Models for Model Matching: An analysis of approaches to support model differencing”. In: May 2009, pp. 1–6. ISBN: 978-1-4244-3714-6. DOI: 10.1109/CVSM.2009.5071714.

[36] Max Emanuel Kramer. “Specification Languages for Preserving Consistency between Models of Different Languages”. PhD thesis. Karlsruher Institut für Technologie (KIT), 2017. 278 pp. DOI: 10.5445/IR/1000069284.

[37] Thomas Kühn et al. “A Metamodel Family for Role-Based Modeling and Programming Languages”. In: *Software Language Engineering*. Ed. by Benoît Combemale et al. Cham: Springer International Publishing, 2014, pp. 141–160. ISBN: 978-3-319-11245-9.

[38] Sven Leonhardt et al. “Integration of Existing Software Artifacts into a View- and Change-Driven Development Approach”. In: *Proceedings of the 2015 Joint MORSE/-VAO Workshop on Model-Driven Robot Software Engineering and View-Based Software-Engineering*. MORSE/VAO ’15. L’Aquila, Italy: Association for Computing Machinery, 2015, pp. 17–24. ISBN: 9781450336147. DOI: 10.1145/2802059.2802061. URL: https://doi.org/10.1145/2802059.2802061.

[39] Yuehua Lin, Jeff Gray, and Frédéric Jouault. “DSMDiff: A differentiation tool for domain-specific models”. In: *European Journal of Information Systems - EUR J INFOR SYST* 16 (Aug. 2007), pp. 349–361. DOI: 10.1057/palgrave.ejis.3000685.

[40] Johannes Meier and Andreas Winter. “Model Consistency ensured by Metamodel Integration”. In: *6th International Workshop on The Globalization of Modeling Languages (GEMOC), co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018)*. Ed. by Regina Hebig and Thorsten Berger. Copenhagen: CEUR Proceedings of MODELS 2018 Workshops, Oct. 2018, pp. 408–415.

[41] Johannes Meier et al. “Classifying Approaches for Constructing Single Underlying Models”. In: *Model-Driven Engineering and Software Development : 7th International Conference, MODELSWARD 2019, Prague, Czech Republic, February 20–22, 2019. Revised Selected Papers. Ed.: S. Hammoudi*. 7th International Conference on Model-Driven Engineering and Software Development. MODELSWARD 2019 (Prag, Tschechien, Feb. 20–22, 2019). Vol. 1161. Communications in Computer and Information Science. Springer Nature, 2020, pp. 350–375. ISBN: 978-3-030-37872-1. DOI: 10.1007/978-3-030-37873-8_15.

[42] Tom Mens. “A state-of-the-art survey on software merging”. In: *IEEE Transactions on Software Engineering* 28.5 (2002), pp. 449–462. DOI: 10.1109/TSE.2002.1000449.

[43] Klaus Müller and Bernhard Rumpe. “User-Driven Adaptation of Model Differencing Results”. In: Feb. 2014. DOI: 10.13140/2.1.2796.7682.

[44] Oracle. *Java programming language*. URL: https://www.java.com (visited on 06/14/2021).

[45] Parul and Brahmaleen Kaur Sidhu. “Model Smells In Uml Class Diagrams”. In: *International Journal of Enhanced Research in Management & Computer Applications*. Vol. 5. May 2016, pp. 1–13.

[46] Pit Pietsch, Klaus Müller, and Bernhard Rumpe. “Model Matching Challenge: Benchmarks for Ecore and BPMN Diagrams”. In: *Softwaretechnik-Trends* 33.2 (2013), pp. 95–100. DOI: 10.1007/s40568-013-0061-x.

[47] Ronald C. Read and Derek G. Corneil. “The graph isomorphism disease”. In: *Journal of Graph Theory* 1.4 (1977), pp. 339–363. DOI: https://doi.org/10.1002/jgt.3190010410. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/jgt.3190010410.

[48] Michael Rys. “XML and Relational Database Management Systems: Inside Microsoft® SQL Server™ 2005”. In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’05. Baltimore, Maryland: Association for Computing Machinery, 2005, pp. 958–962. ISBN: 1595930604. DOI: 10.1145/1066157.1066301. URL: https://doi.org/10.1145/1066157.1066301.

[49] Timur Sağlam and Heiko Klare. “Classifying and Avoiding Compatibility Issues in Networks of Bidirectional Transformations”. In: *STAF 2021 Workshop Proceedings: 9th International Workshop on Bidirectional Transformations*. CEUR-WS, 2021.

[50] Petri Selonen. “A review of UML model comparison approaches”. In: *Workshop Proceedings of the 5th Nordic Workshop on Model Driven Engineering, 27-29 August 2007, Ronneby, Sweden. Blekinge Institute of Technology. Research report*. Ed. by M. Staron. 2007, pp. 37–51. ISBN: 978-91-7295-985-9.

[51] Brahmaleen K. Sidhu, Kawaljeet Singh, and Neeraj Sharma. “A Catalogue of Model Smells and Refactoring Operations for Object-Oriented Software”. In: *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*. 2018, pp. 313–319. DOI: 10.1109/ICICCT.2018.8473027.

[52] Dave Steinberg et al. *EMF: Eclipse Modeling Framework 2.0 (2nd Edition)*. Jan. 2008. ISBN: 9780321331885.

[53] Matthew Stephan and James R. Cordy. “A Survey of Model Comparison Approaches and Applications”. In: *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,* INSTICC. SciTePress, 2013, pp. 265–277. ISBN: 978-989-8565-42-6. DOI: 10.5220/0004311102650277.

[54] Misha Strittmatter and Amine Kechaou. *The Media Store 3 Case Study System*. Tech. rep. 1. Karlsruher Institut für Technologie (KIT), 2016. 35 pp. DOI: 10.5445/IR/1000052197.

[55] Nikolaos Tsantalis et al. “A Multidimensional Empirical Study on Refactoring Activity”. In: *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*. CASCON ’13. Ontario, Canada: IBM Corp., 2013, pp. 132–146.

[56] Christian Tunjic and Colin Atkinson. "Synchronization of Projective Views on a Single-Underlying-Model". Englisch. In: *Proceedings of the Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering : MORSE/VAO '15 : 21 July 2015, L'Aquila, Italy*. Ed. by Uwe Assmann. New York, NY: ACM, 2015, pp. 55–58. DOI: 10.1145/2802059.2802066. URL: https://madoc.bib.uni-mannheim.de/43503/.

[57] Christopher Werner and Uwe Aßmann. "Model Synchronization with the Role-oriented Single Underlying Model". In: 2018. DOI: http://ceur-ws.org/Vol-2245/mrt_paper_3.pdf.

[58] Manuel Wimmer, Nathalie Moreno, and Antonio Vallecillo. "Viewpoint Co-evolution through Coarse-Grained Changes and Coupled Transformations". In: May 2012, pp. 336–352. ISBN: 978-3-642-30560-3. DOI: 10.1007/978-3-642-30561-0_23.

[59] Manuel Wimmer et al. "A Catalogue of Refactorings for Model-to-Model Transformations". In: *Journal of Object Technology* 11.2 (Aug. 2012), 2:1–40. ISSN: 1660-1769. DOI: 10.5381/jot.2012.11.2.a2. URL: http://www.jot.fm/contents/issue_2012_08/article2.html.

[60] Zhenchang Xing and Eleni Stroulia. "UMLDiff: An Algorithm for Object-Oriented Design Differencing". In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ASE '05. Long Beach, CA, USA: Association for Computing Machinery, 2005, pp. 54–65. ISBN: 1581139934. DOI: 10.1145/1101908.1101919. URL: https://doi.org/10.1145/1101908.1101919.

A. Appendix

Test Case	ID	Similarity	Custom (Conservative)	Custom (Aggressive)
Atomic Change Operations				
Add Attribute	2	2	2	2
Add Class	1	1	1	1
Add Method	3	3	3	3
Move Attribute	1	1	1	1
Move Class 1	1	1	1	1
Move Class 2	1	1	1	1
Move Method	1	1	1	1
Remove Attribute	2	2	2	2
Remove Class	8	8	8	8
Remove Method	5	5	5	5
Rename Attribute	1	1	1	1
Rename Class	1	4	1	1
Rename Method	1	1	1	1
Common Refactoring Operations				
Change Method Signature	11	83	83	83
Collapse Hierarchy	12	84	84	84
Extract Associated Class	24	96	96	96
Extract Superclass	30	90	90	90
Inline Class	18	78	78	78
Remove Associated Class	21	81	81	81
Model Match Challenge (with Associations)				
Exchange Elements	2	2	2	2
Move	3	15	15	15
Move Renamed	6	29	27	24
Rename	4	28	25	22
Update Reference Target	2	14	14	14

Test Case	ID	Similarity	Custom (Conservative)	Custom (Aggressive)
Model Match Challenge (without Associations)				
Move	2	2	2	2
Move Renamed	4	9	7	4
Rename	2	8	5	2
Update Reference Target	1	1	1	1
Thesis System				
Default Test	29	119	119	116
Large System				
Delete Classes	167	167	167	167
Delete Methods & Attributes	18	18	18	18
Move Classes	5	5	5	5
Rename Classes	5	57	5	5
Rename Single Method	1	9	5	1
Rename Single Attribute	1	9	4	1
Rename Methods & Attributes	7	24	18	7
Huge System				
Move Shallowly Nested Classes	4	147	147	147
Move Deeply Nested Classes	4	218	218	218

Table A.2.: Change Sequence Size (in Number of Operations) per Test Case per Strategy

Test Case	ID	Similarity	Custom (Conservative)	Custom (Aggressive)
Atomic Change Operations				
Add Attribute	✓	✓	✓	✓
Add Class	✓	✓	✓	✓
Add Method	✓	✓	✓	✓
Move Attribute	✓	✓	✓	✓
Move Class 1	✓	✓	✓	✓
Move Class 2	✓	✓	✓	✓
Move Method	✓	✓	✓	✓
Remove Attribute	✓	✓	✓	✓
Remove Class	✓	✓	✓	✓
Remove Method	✓	✓	✓	✓
Rename Attribute	✓	✓	✓	✓
Rename Class	✓	✓	✓	✓
Rename Method	✓	✓	✓	✓
Common Refactoring Operations				
Change Method Signature	✓	✓	✓	✓
Collapse Hierarchy	✓	✓	✓	✓
Extract Associated Class	×	×	×	×
Extract Superclass	✓	×	×	×
Inline Class	✓	✓	✓	✓
Remove Associated Class	✓	✓	✓	✓
Model Match Challenge (with Associations)				
Exchange Elements	✓	✓	✓	✓
Move	✓	✓	✓	✓
Move Renamed	✓	×	×	×
Rename	✓	×	×	×
Update Reference Target	✓	✓	✓	✓
Model Match Challenge (without Associations)				
Move	✓	✓	✓	✓
Move Renamed	✓	✓	✓	✓
Rename	✓	✓	✓	✓
Update Reference Target	✓	✓	✓	✓
Thesis System				
Default Test	✓	✓	✓	✓

Table A.1.: Model Correctness per Test Case per Strategy