A History-based Approach for Model Repair Recommendations in Software Engineering

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Manuel Ohrndorf

Leiter der Arbeit:
Prof. Dr. Timo Kehrer
Universität Bern

A History-based Approach for Model Repair Recommendations in Software Engineering

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Manuel Ohrndorf

Leiter der Arbeit:
Prof. Dr. Timo Kehrer
Universität Bern

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 14.07.2023

Der Dekan
Prof. Dr. Marco Herwegh

Supervisor and First Reviewer
Prof. Dr. Timo Kehrer
Universität Bern

Second Reviewer
Prof. Dr. Lars Grunske
Humboldt-Universität zu Berlin

Supervisor
Prof. Dr. Udo Kelter
Universität Siegen

Abstract

Software is an everyday companion in today's technology society that need to be evolved and maintained over long time periods. To manage the complexity of software projects, it has always been an effort to increase the level of abstraction during software development. Model-Driven Engineering (MDE) has shown to be a suitable method to raise abstraction levels during software development. Models are primary development artifacts in MDE that describe complex software systems from different viewpoints.

In MDE software projects, models are heavily edited through all stages of the development process. During this editing process, the models can become inconsistent due to uncertainties in the software design or various editing mistakes. While most inconsistencies can be tolerated temporarily, they need to be resolved eventually. The resolution of an inconsistency affecting a model's design is typically a creative process that requires a developer's expertise. Model repair recommendation tools can guide the developer through this process and propose a ranked list of repairs to resolve the inconsistency. However, such tools will only be accepted in practice if the list of recommendations is plausible and understandable to a developer. Current approaches mainly focus on exhaustive search strategies to generate improved versions of an inconsistent model. Such resolutions might not be understandable to developers, may not reflect the original intentions of an editing process, or just undo former work. Moreover, those tools typically resolve multiple inconsistencies at a time, which might lead to an incomprehensible composition of repair proposals.

This thesis proposes a history-based approach for model repair recommendations. The approach focuses on the detection and complementation of incomplete edit steps, which can be located in the editing history of a model. Edit steps are defined by consistency-preserving edit operations (CPEOs), which formally capture complex and error-prone modifications of a specific modeling language. A recognized incomplete edit step can either be undone or extended to a full execution of a CPEO. The final inconsistency resolution depends on the developer's approval. The proposed recommendation approach is fully implemented and supported by our interactive repair tool called ReVision. The tool also includes configuration support to generate CPEOs by a semi-automated process.

The approach is evaluated using histories of real-world models obtained from popular open-source modeling projects hosted in the Eclipse Git repository. Our experimental results confirm our hypothesis that most of the inconsistencies, namely 93.4%, can be resolved by complementing incomplete edits. 92.6% of the generated repair proposals are relevant in the sense that their effect can be observed in the models' histories. 94.9% of the relevant repair proposals are ranked at the topmost position. Our empirical results show that the presented history-based model recommendation approach allows developers to repair model inconsistencies efficiently and effectively.

Kurzfassung

Software ist in unserer heutigen Gesellschaft ein alltäglicher Begleiter. Diese wird ständig weiterentwickelt und überarbeitet. Model-Driven Engineering (MDE) hat sich als geeignete Methode erwiesen, um bei der Entwicklung komplexer Software von technischen Details zu abstrahieren. Hierbei werden Modelle als primäre Entwicklungsartefakte verwendet, welche ein Softwaresystem aus verschiedenen Sichten beschreiben.

Modelle in MDE werden in allen Entwicklungsphasen einer Software fortlaufend überarbeitet. Während der Bearbeitung können die Modelle aufgrund von Unklarheiten im Design oder verschiedenen Bearbeitungsfehlern inkonsistent werden. Auch wenn Inkonsistenzen vorübergehend toleriert werden können, so müssen diese letztlich doch behoben werden. Die Behebung einer Inkonsistenz, welche sich auf das Design eines Modells auswirkt, ist meist ein kreativer Prozess, der das Fachwissen eines Entwicklers erfordert. Empfehlungswerkzeuge können den Entwickler mit Reparaturvorschlägen unterstützen. Damit solche Werkzeuge in der Praxis akzeptiert werden, müssen die Vorschläge plausible und nachvollziehbar sein. Die meisten aktuelle Ansätze verwenden Suchstrategien, welche Reparaturen durch systematisches Ausprobieren generieren. Die so generierten Reparaturen sind für Entwickler häufig schwer nachvollziehbar, da sie vorhergehende Bearbeitungsschritte nicht beachten oder diese einfach rückgängig machen. Darüber hinaus lösen diese Reparaturwerkzeuge in der Regel mehrere Inkonsistenzen gleichzeitig, was zu unverständlichen und umfangreichen Reparaturen führen kann.

Diese Arbeit beschreibt einen Ansatz zum Erkennen und Ergänzen unvollständiger Bearbeitungsschritte, basierend auf der Bearbeitungshistorie eines Modells. Dazu werden konsistenzerhaltende Bearbeitungsoperationen definiert, die komplexe und fehleranfällige Änderungen einer bestimmten Modellierungssprache formal erfassen. Ein unvollständiger Bearbeitungsschritt kann dann entweder rückgängig gemacht oder zu einer konsistenzerhaltenden Bearbeitungsoperationen erweitert werden. Die endgültige Reparatur der Inkonsistenz hängt von der Einschätzung des Entwicklers ab. Der vorgeschlagene Ansatz wurde in unserem interaktiven Reparaturwerkzeug ReVision implementiert. Darüber hinaus umfasst das Werkzeug Unterstützung zum Generieren von konsistenzerhaltenden Bearbeitungsoperationen.

Das Reparaturverfahren wurde anhand von Historien realer Modelle aus bekannten Open-Source-Modellierungsprojekten im Eclipse-Git-Repository bewertet. Die experimentellen Ergebnisse bestätigen unsere Hypothese, dass die meisten Inkonsistenzen, nämlich 93.4%, durch Ergänzung unvollständiger Bearbeitungen gelöst werden können. 92.6% der generierten Reparaturvorschläge könnten in der entsprechenden Modellhistorie beobachtet werden. Von diesen Reparaturvorschläge wurden 94.9% an erster Stelle vorgeschlagen. Unsere empirischen Ergebnisse zeigen, dass der vorgestellte historienbasierte Modellempfehlungsansatz es Entwicklern ermöglicht, Modellinkonsistenzen effizient und effektiv zu reparieren.

Contents

Thesis-related List of Publications

This thesis is related to the following international scientific publications that have been peer-reviewed according to scientific quality assurance standards (in chronological order):

[1] Timo Kehrer, Udo Kelter, **Manuel Ohrndorf**, and Tim Sollbach. Understanding Model Evolution through Semantically Lifting Model Differences with SiLift. In *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*, pages 638–641. IEEE Computer Society, 2012. 🔧.

[2] Gabriele Taentzer, **Manuel Ohrndorf**, Yngve Lamo, and Adrian Rutle. Change-Preserving Model Repair. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10202 of *Lecture Notes in Computer Science*, pages 283–299. Springer, 2017. 🗎.

[3] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, **Manuel Ohrndorf**, and Matthias Tichy. Henshin: A Usability-Focused Framework for EMF Model Transformation Development. In *Graph Transformation - 10th International Conference, ICGT 2017, Held as Part of STAF 2017, Marburg, Germany, July 18-19, 2017, Proceedings*, volume 10373 of *Lecture Notes in Computer Science*, pages 196–208. Springer, 2017. 🗎.

[4] Christopher Pietsch, **Manuel Ohrndorf**, Udo Kelter, and Timo Kehrer. Incrementally Slicing Editable Submodels. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 913–918. IEEE Computer Society, 2017. 🔧.

[5] **Manuel Ohrndorf**, Christopher Pietsch, Udo Kelter, and Timo Kehrer. ReVision: A Tool for History-based Model Repair Recommendations. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 105–108. ACM, 2018. 🔧.

[6] **Manuel Ohrndorf**, Christopher Pietsch, Udo Kelter, Lars Grunske, and Timo Kehrer. History-based Model Repair Recommendations. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–46, 2021. 📕.

[7] **Manuel Ohrndorf**, Christopher Pietsch, Udo Kelter, Lars Grunske, and Timo Kehrer. A Summary of ReVision: History-based Model Repair Recommendations. In *Software Engineering 2023*, pages 99–100. Gesellschaft für Informatik eV, 2023. 🗎.

📕 Journal Article 🗎 Conference Paper 🔧 Tool Demonstration Paper

1
Introduction

Software engineering raises the abstraction and modularity in the process of software development. In Model-Driven Engineering (MDE), models describe complex software systems from different modeling views abstracting from technical details. During their evolution, models are heavily edited through all stages of the software development process. Meanwhile, models can become inconsistent for various reasons. Model repair tools support developers by recommending a short list of repair alternatives. This thesis proposes a new approach to generate repair proposals based on the assumption that inconsistencies are mainly introduced by incomplete editing processes that can be located in the version history of a model. Such an incomplete editing process is either undone or extended to a complete consistency-preserving edit operation (CPEO) application.

Software systems are increasingly complex these days and often evolve over decades. However, the increasing size and complexity of a software project can delay its development or eventually lead to its abandonment. In order to address emerging development issues, software engineering has been established as a discipline of systematically developing software [73, 118]. As emphasized by Watts Humphrey, software engineering has to be approached as a multidisciplinary challenge of applying *"engineering, scientific, and mathematical principles and methods to the economical production of quality software"* [70]. From the dawn of software engineering during the 1960s [20, 195] till today, one of its major challenges and efforts has been to raise abstraction and modularity in the process of software development.

At the first conference on software engineering [20] in 1968, the question was raised of how software development can become a systematic process. As mentioned by Edsger Dijkstra [20], the process of implementation and documentation of a software system should go hand in hand to simplify software development and maintenance. As developers typically experience writing documentation as an additional burden, Dijkstra suggested starting the development by writing pre-documentation that can be mechanically processed to support the implementation of a system [20].

A methodology for combining a system's documentation and implementation by automation, researched in the 1980s and 1990s, was computer-aided software engineering

(CASE). Corresponding CASE tools offer general-purpose graphical languages like state machines, structure diagrams, and data flow diagrams to describe programs, which are then translated into implementation artifacts. The diagrams fulfill the purpose of a visual predocumentation of the system under development, and the generative process keeps this documentation synchronized with the implementation. Such diagrams can describe a system abstracting from technical details. Even though CASE has been extensively discussed in the research literature, it has not been widely accepted in practice [160]. According to Schmidt [160], the CASE method suffered from several problems: (1) The gap between the high-level graphical representation and the low-level target platform interfaces could not be compensated by the generation techniques at this time. (2) CASE projects were not designed for concurrently working in teams. (3) The general-purpose graphical languages were not customizable to efficiently express specific domain problems.

As mentioned by Niklaus Wirth [195], a computer system's complexity can only be mastered intellectually through abstractions, i.e., a development language must reflect the problem to be solved rather than the underlying machine [195]. Beyond general-purpose programming languages, modeling languages allow developers to describe a software system abstracting from technical details, resulting in a model specifying and documenting the system under development. Given the lessons learned from CASE, a modeling language should be expressive for the problems in a specific domain. A so-called domain-specific modeling language (DSML) can bridge the gap between the problem space in which domain experts work and the implementation space of a specific target platform [34]. A DSML can either be created from scratch or by adapting modeling languages such as the Unified Modeling Language (UML) [141], e.g., by domain-specific extensions, so-called profiles, or by adding restrictive design constraints. Graphical syntax notations are well suited to visualize and communicate relationships between the system components or to describe data and control flows of an application scenario. Modeling languages with textual syntax offer a more programming-like workflow to the developer. The choice of language and visualization depends on the domain, the problems to be solved, and the preferences of the development team [143]. In the following, we will generally refer to these different notations as DSMLs or modeling languages.

To create a general foundation and standards for software modeling, the Object Management Group (OMG) introduced the concept of Model-Driven Architecture (MDA) [137, 138, 170] in the early 2000s. The MDA Guide [138] highlights different applications in which a development process could derive value from software models. Starting with the initial design, models help to deal with the complexity of large systems defining their structure and semantics. During this phase, models can be used as communication vehicles to create a common understanding of the system under development, e.g., within a development team or with customers. Furthermore, the models can be used to analyze and monitor the system's development by generating statistics and metrics for quality assessment. The simulation and execution of models can help developers to understand the functionality and validate the software system's correctness. The practical benefit of models is their documentary nature and the exchangeability of technology-dependent parts of the software system. The methodology described by the MDA Guide [138] primarily focuses on the separation of models from platform-specific details. Starting from a technical and platform-independent model (PIM), which will then be stepwise transformed and enriched with technical detail until a platform-specific model (PSM) is derived. Finally, a model can either be executed by an interpreter or by generating code for a specific target platform.

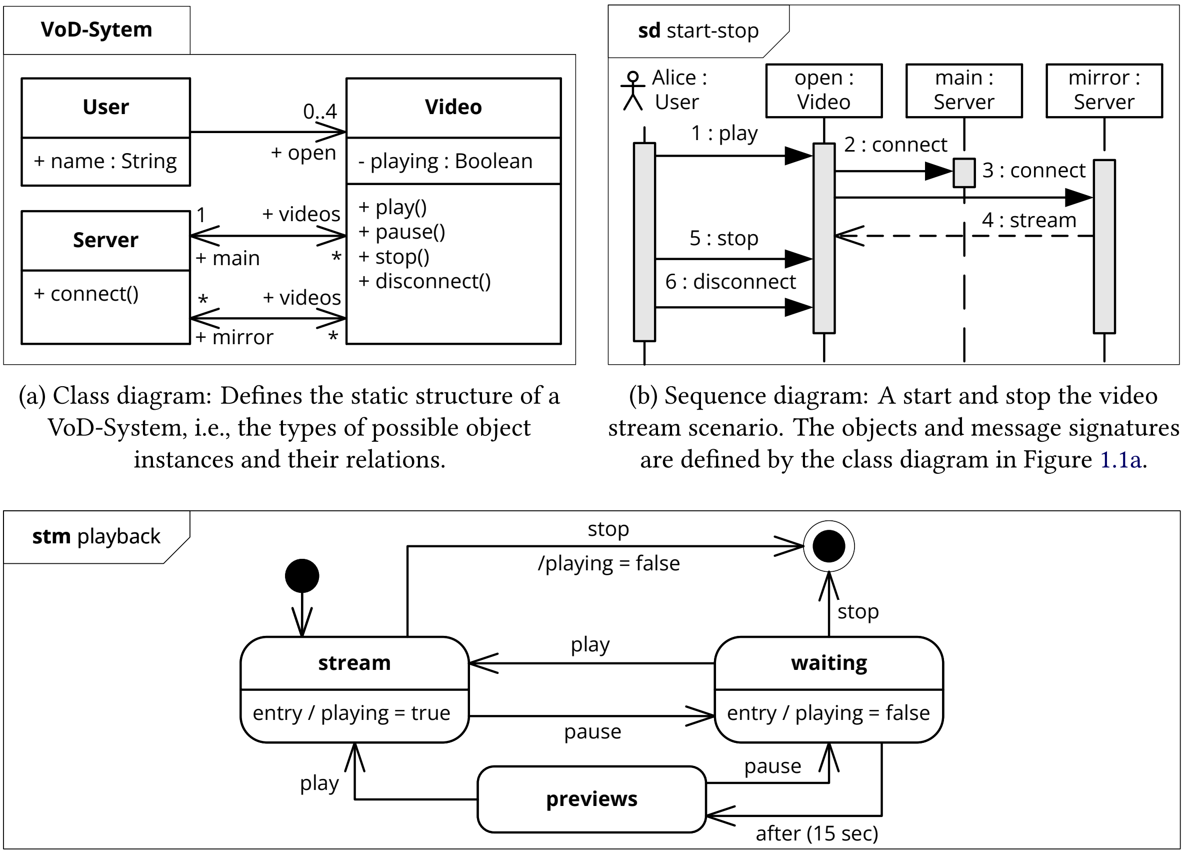
Models have shown to be a suitable method to raise abstraction levels and decouple software development from technical implementation details. Model-Driven Software Engineering (MDSE) or, for short, Model-Driven Engineering (MDE) can be considered as the discipline of applying MDA in practice [30]. This involves a variety of approaches, techniques, and tools. Models, which are formal descriptions of the software system, are primary development artifacts in MDE. Therefore, models need to be maintained with the same attention as source code in classical software development. Models are developed using modeling languages that take advantage of different viewpoints to describe complex software systems.

From the adoption of MDE by a development team to the maintenance of a long-living MDE software project, models are heavily edited through all stages of the development process. During this editing process, the models can become inconsistent for various reasons. For example, uncertainties in the software design, modeling views that need to be synchronized, incomplete edit steps, or misunderstandings of the modeling language's syntax. While most inconsistencies can be tolerated temporarily, they need to be resolved eventually. However, the resolution of an inconsistency is typically a creative process in which various factors need to be considered. Especially an inconsistency resolution that affects the design of a model should involve the expertise of a developer. Model recommendation tools can guide the developer through this process by recommending a ranked list of repairs to resolve the inconsistency.

This thesis proposes a history-based approach for model repair recommendations in software engineering. The approach focuses on recognizing and complementing incomplete edit steps that can be located in the editing history of a model. Complete edit steps are defined by consistency-preserving edit operations (CPEOs), which formally capture complex and error-prone modifications of a specific modeling language. A recognized incomplete edit step can either be undone or extended to a full execution of a CPEO. Finally, the decision to choose a concrete inconsistency resolution depends on the developer's approval. The proposed recommendation approach is fully implemented and supported by our interactive tool called ReVision [8]. Starting with a meta-model and set of consistency rules that formally define a modeling language's syntax, ReVision also includes configuration support to generate CPEOs by a semi-automated process.

1.1 Introduction of the Running Example

Let us take a look at the UML model of the *video-on-demand system* (VoD-System) in Figure 1.1. The VoD-System is described from three different modeling viewpoints. The first viewpoint in Figure 1.1a is a *class diagram*, which shows the static components and properties of the system and their structural relations. Class diagrams represent a central UML concept that is well suited to define a software system's data structure, e.g., the class definitions of an object-oriented language or a database schema. The VoD-System consists of the classes User, Video, and Server. In general, classes define the types of possible object instances and their structural connection, e.g., a User can open [0..4] up to four videos simultaneously.



(a) Class diagram: Defines the static structure of a VoD-System, i.e., the types of possible object instances and their relations.

(b) Sequence diagram: A start and stop the video stream scenario. The objects and message signatures are defined by the class diagram in Figure 1.1a.

(c) State machine diagram: States and transitions of a video playback. The transitions are triggered by operation calls of the class Video in Figure 1.1a.

Figure 1.1. A simple VoD-System that is modeled from three different UML diagram views.

A special case of a class diagram is an *object diagram.* While a class diagram models the general construction plans for objects, an object diagram shows a concrete snapshot of the system's objects and their attribute values and references at runtime. An object in UML is notated by an expression name:type, in which the type is specified by the corresponding class. Such an object instance notation is also used by the *sequence diagram* in the second viewpoint shown in Figure 1.1b. A sequence diagram illustrates an exemplary interaction between objects of a system. They show the chronological sequence of the messages sent between objects but do not describe their structural relations. The sequence diagram in our running example in Figure 1.1a illustrates a simple "start and stop the video stream" scenario.

The objects' names in the name:type notations of the sequence diagram in Figure 1.1b are defined by properties of the class diagram in Figure 1.1a. For example, the object open:Video is contained in the property named open [0..4] of an object of type User. The property open [0..4] is defined as an association between User and Video in the class diagram. A so-called lifeline depicted below an object in the sequence diagram represents the life cycle of this object. A lifeline can send and receive messages that represent operation calls on the receiving object. The operations are defined by the type of the receiving object. For example, the lifeline of open:Video receives the messages 1:play and 5:stop, which refer to calls of the operations defined in the Video class. Such interaction diagrams are typically used for the purpose of documentation or as a communication vehicle, e.g., between the developers of a client-server application to define and analyze the communication protocol.

The third diagram in Figure 1.1c shows a (behavioral) state machine that describes the behavior of the class Video. State machine diagrams can be used to supplement class diagrams to model the correct states in the life cycle of objects. A state machine can be owned by a class that specifies the state machine's context, e.g., accessible attributes. Every object created according to the definition of a class is always in a certain state that is defined by its attribute values. During its lifespan, an object can only have certain meaningful combinations of attribute values that are modeled as states. The transitions between the states are triggered by events in the system, e.g., by operation calls that are defined by the class owning the state machine. For example, the operation call pause() can only be called when the video is played (playing = true) and assumes that the playback is finally stopped (playing = false). Moreover, our exemplary state machine defines a state for movie previews that is entered by a so-called time event after pausing the video for 15 seconds.

As we can see, different modeling views can be used to define different aspects of the system. However, these views depend on each other, i.e., formally speaking, they need to be consistent. Technically, in our running example, changes in the class diagram need to be reflected in the sequence diagram and the state machine diagram. However, the actual editing process of a developer might temporarily violate the model's consistency. For example, a developer could add a new transition to a state machine before defining its trigger. To define the triggering event, the developer then adds a new operation to the class diagram. Moreover, changes in a single view can also break the consistency of a modeling view, e.g., deleting a state in the state machine diagram can lead to dangling transitions with missing source or target states.

The syntactic consistency of the overall model can be checked by formally defined *consistency rules*. Such rules statically check the integrity of a model, which is also referred to as static semantics or well-formedness. However, rules can also be infeasible to be captured by a formal consistency rule, e.g., if a call sequence in a sequence diagram is consistent with the possible sequences in the corresponding state machine diagrams. Such requirements are referred to as (dynamic or execution) semantics of a model [50, 103].

1.2 Problem Motivation

The maintenance and enhancements of long-living software projects make up a significant part of a software system's life cycle. Maintenance in the context of software engineering is defined as the *"process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment"* [73]. Typically, user requests for enhancements and extensions require most of the efforts during software maintenance [21, 59, 110].

Some researchers and practitioners use the term *software evolution* as a synonym or even a preferable substitute for software maintenance and enhancements [21, 32]. Moreover, the development of software is typically performed in multiple iterative and incremental life cycles [102], i.e., maintenance activities can take place at several stages of software development. According to Lehman [107, 108], *continuing change* is a law of software evolution that is required to retain the usefulness and manage the complexity of software systems. Technically, software evolution can be defined as the sequence of states and the transitions between them, starting from the initial creation of the software to its final retirement [32]. According to Mens et al. [130], software evolution comprises several challenges, including research on formalisms to analyze and understand software change, development of supporting tools and techniques, and real-world validation of long-lived software systems on an industrial scale. Moreover, such software evolution techniques should be raised to a higher level of abstraction to support the evolution of higher-level artifacts such as analysis and design models or software architectures [130].

Models are primary development artifacts in MDE. Thus, models are also primary targets of continuous changes during software evolution. As models are technically independent representations of the system, they are less likely to be affected by changes in the data or processing environment, performance improvements related to the technical platform, or fault corrections on the level of program code. However, models have to keep up with the ever-changing user requirements and will be maintained to improve the models' complexity, e.g., the system's architecture or modularity. Therefore, models are heavily edited during all stages of software development. As a consequence, models may get inconsistent for various reasons, e.g., due to misunderstandings or different interpretations of requirements when being edited collaboratively in teams [41]. From a more technical perspective, one main reason for consistency violations is the isolated editing of inter-related views [49, 52, 58, 60, 61, 149] or model fragments [2, 4].

While it is generally acknowledged that inconsistencies should be temporarily tolerated for the sake of flexibility [16, 52, 60], they must be resolved eventually. Therefore, consistency management is an essential discipline in MDE [115, 136, 172]. Engineers designing a modeling language need a way to specify precisely what consistency in a concrete modeling context actually means, and developers must be supported by techniques for detecting, diagnosing, and fixing inconsistencies. In MDE, the basic structure and typing of a modeling language, referred to as abstract syntax, is typically defined by a meta-model. More complex consistency and well-formedness rules are defined in addition to the meta-model using a language such as the Object Constraint Language (OCL) [189], and violations of these rules can be automatically obtained using inconsistency detection techniques [28, 43, 44, 61, 150]. While these techniques are widely established in practice, how to optimally support developers in resolving detected inconsistencies is still being actively discussed.

The primary origins of model evolution are changing requirements and related maintenance activities. Such changes are likely to violate the consistency of a model. Resolving inconsistencies that involve the design of the system is a creative process that requires the expertise of a developer. According to the classification of software repair approaches presented by Monperrus et al. [132], the most appropriate repair approach for this type of defect is a recommender system. Here, both a recommender system and a human developer contribute to the repair. The role of the recommender system is to generate a ranked list of suitable repair proposals from which developers can choose unless they create a handcrafted solution.

Repair recommender systems aim at the following quality goals and related evaluation criteria (see Table 2 in [132]). The proposed solutions should be understandable by humans, and they should be filtered and ranked. Partial solutions are acceptable and preferred over complete complex solutions that are hard to understand. More generally, the human workload to perform the repair should be as low as possible, and the quality and maintainability of the solution should be high [106, 132]. Thus, a tool that supports model repair will only be accepted in practice if it generates a limited set of plausible and understandable repair proposals [88, 105]. Moreover, it is impractical to redevelop the tooling for each different DSML in MDE, i.e., the repair tools should be adaptable to a specific modeling language.

Current approaches to automated model repair largely fail to meet these requirements. Exhaustive search strategies, as presented, e.g., in References [49,114,147,186], aim at a fully automated repair resolving all inconsistencies in a single step. If several solution candidates are found, the most appropriate one is selected. This is typically achieved through simple heuristics for assessing individual solution candidates, such as the minimal edit distance with respect to the inconsistent version. Such strategies are also known as *least-change principle* [115]. Turning a search-based approach into a recommender system by collecting and presenting *all* solution candidates typically produces a huge number of repair alternatives, which might overwhelm the developers [115]. Even ranked lists of repair proposals, e.g., based on the least-change principle, are often unlikely to rank the most suitable repairs in one of the topmost positions. Moreover, repairing all inconsistencies in a single step often leads to solutions whose rationale is hard to grasp for developers.

From a conducted user study, Marchezan et al. [121] concluded that repair recommendations help developers to fix inconsistencies more efficiently for complex repair tasks. According to Marchezan et al. [121], for simple tasks, however, repair recommendations may have the opposite effect. Therefore, repair tools that support the developer in resolving complex inconsistencies are crucial during modeling.

In this thesis, we will examine how developers can be effectively supported in dealing with complex inconsistencies during modeling. The approach focuses on complex consistency constraints that restrict the valid instances of a model, i.e., the static semantics of a model. In particular, the technique generates history-based repair recommendations that support the developer in resolving structural inconsistencies between modeling views and error-prone modifications of complex modeling fragments.

Most model repair approaches do not consider the origin of a defect; they are not aware of a model's change history (see Chapter 9). Therefore, they are likely to propose repairs that just undo former changes that have caused an inconsistency. The inconsistency resolutions analyzed in the approach's evaluation reveal that only a fraction of the inconsistencies have been repaired by undoing former changes. Thus, the history-based repair recommendations enable the developer to make an informed decision whether to undo former work or, if this work was just incomplete, retain and complete it.

1.3 Contributions

This thesis proposes a new approach to generate repair proposals based on the assumption that inconsistencies are introduced by incomplete editing processes that can be located in the version history of a model. Such an incomplete editing process is either undone or is extended to a full execution of a CPEO. These CPEOs are model transformations that preserve the consistency rule under consideration, i.e., their partial execution typically leads to an inconsistent model. The proposed recommendation technique is based on the concept of detecting and complementing partially executed edit operations with respect to the historical changes applied to a model. This technique is enhanced for model repair by detecting the origin of an inconsistency in the model's editing history. The approach described in this thesis has the following contributions and distinguishing features.

Contribution 1: Recognition of Partially Executed Edit Operations. The technique recognizes partially applied edit operations in a difference between two model versions. Given a CPEO and a set of historical changes applied to a model, the technique computes all subsets of these changes that correspond to a sub-rule of the CPEO. No external knowledge about the historical changes between revisions is required; these changes are calculated by using readily available model differencing tools (see, e.g., References [80, 86]). The technique for detecting partial executions of CPEOs in a given model difference is based on the approach presented by Kehrer et al. [1,78,82,83], which introduces a technique to recognize complete applications of edit rules from the changes of a difference. However, the challenge here is to detect all partial applications of an edit rule.

📘 See Reference [6] **Manuel Ohrndorf**, Christopher Pietsch, Udo Kelter, Lars Grunske, and Timo Kehrer. History-based Model Repair Recommendations. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–46, 2021.

Contribution 2: Complementation of Partially Executed Edit Operations. Given a partially recognized CPEO, a complement rule is computed by subtracting the already applied changes from the CPEO. In the next step, all valid applications of the complement rule on the current model version are calculated. Technically, the complement rule application must continue the recognized partial edit step. The complementations are then presented to the developer in a summarized way. Therefore, multiple applications of the same complement rule are combined into a parameterized rule. Finally, all ambiguous parameters have to be selected or bound by the developer.

📄 See Reference [2] Gabriele Taentzer and **Manuel Ohrndorf** and Yngve Lamo and Adrian Rutle. Change-Preserving Model Repair. *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10202 of *Lecture Notes in Computer Science*, pages 283–299. Springer, 2017.

Contribution 3: History-based Model Repair Recommendations. The approach to model repair builds upon the technique for detecting partially executed CPEOs. Basically, the detected incomplete edit step must have introduced the inconsistency under consideration, and the proposed complementation must improve this inconsistency. The approach introduced in this thesis generates two kinds of repair recommendations that are ranked and presented to the developer.

(1) *Complements* are repair proposals that extend a set of former model changes to a complete CPEO.

(2) *Rollbacks* remove the inconsistency in a controlled manner through a case-specific undo operation that reverts the inconsistency-inducing changes of a partially executed CPEO.

Based on our tool ReVision, we evaluated our approach using a selected set of real-world models extracted from 51 popular open-source Eclipse modeling projects [42]. The editing histories of the models have been retrieved from the corresponding public Eclipse Git repositories. For example, our evaluation dataset includes the evolution of the entire UML meta-model. The evaluation focuses on consistency violations in historical model versions, which have been resolved in a later revision of the same model. The experimental results confirm our hypothesis that most of the inconsistencies, namely 93.4%, can be resolved by complementing incomplete edits. 92.6% of the generated repair proposals are relevant in the sense that their effect can be observed in the models' histories. In this context, 94.9% of the relevant repair proposals are ranked at the topmost position. Our ReVision found at least one repair for 597 out of 638 inconsistencies. In detail, 510 inconsistencies have been resolved by complementing an edit step in the same way as it can be observed in a model's history. A minor number of 43 inconsistencies were resolved by undoing the invalid edit step, which our repair tool could also propose as a case-specific rollback operation. This is achieved at affordable costs and runtimes for the tool users. The number of repair operations generated by our approach is typically low, i.e., 10 or fewer in 92.5% of all supported cases. More importantly, the relevant proposal, i.e., the proposal observed in the model's history, was on the first two positions of the short list of repair proposals in virtually all cases.

📘 See Reference [6] **Manuel Ohrndorf**, Christopher Pietsch, Udo Kelter, Lars Grunske, and Timo Kehrer. History-based Model Repair Recommendations. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–46, 2021.

🗎 See Reference [7] **Manuel Ohrndorf**, Christopher Pietsch, Udo Kelter, Lars Grunske, and Timo Kehrer. A Summary of ReVision: History-based Model Repair Recommendations. In *Software Engineering 2023*, pages 99–100. Gesellschaft für Informatik eV, 2023.

Contribution 4: ReVision: A Tool for History-based Model Recommendations. According to studies by Staron [173] and Mohaghegi et al. [131], the lack of tool support is one of the major drawbacks regarding the adoption and acceptance of MDE in practice. The history-based analysis of inconsistencies as well as the generation and ranking of repair recommendations are fully implemented in our tool called Re(pair)Vision or ReVision for short.

ReVision interacts with the developer through an additional view that enhances the model editor. The view is placed beside the model editor and allows for inspecting a ranked list of repair proposals. Each proposal specifies a pair consisting of a sub-rule and a complement rule based on a partially applied CPEO with respect to the inconsistency to be repaired. A concrete complementation is applied by binding the parameters of the complement rule to model elements and concrete values. The generation of a case-specific undo operation based on the sub-rule can be triggered upon request at the discretion of the developer.

🔧 See Reference [5] **Manuel Ohrndorf**, Christopher Pietsch, Udo Kelter, and Timo Kehrer. ReVision: A Tool for History-based Model Repair Recommendations. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 105–108. ACM, 2018.

🔧 See Reference [1] Timo Kehrer, Udo Kelter, **Manuel Ohrndorf**, and Tim Sollbach. Understanding Model Evolution through Semantically Lifting Model Differences with SiLift. In *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*, pages 638–641. IEEE Computer Society, 2012.

🔧 See Reference [4] Christopher Pietsch, **Manuel Ohrndorf**, Udo Kelter, and Timo Kehrer. Incrementally Slicing Editable Submodels. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 913–918. IEEE Computer Society, 2017.

Contribution 5: Generation of Consistency-preserving Edit Operations. CPEOs are the primary configuration input that adapts the proposed approach to a corresponding modeling language. The specification of CPEOs is supported by a systematic process that captures typical complex edit steps of a modeling language. During the evolution of a model, such complex edit steps are likely to be applied only partially and consequently lead to model inconsistencies.

The CPEOs are composed based on a given set of graph patterns that describe valid model fragments with respect to a consistency rule. In this way, when designing a set of CPEOs for a particular modeling language, there is no need to consider all possible transformations between those patterns. The graph patterns are then automatically composed to different kinds of CPEOs.

The management of rules is supported by tooling that transforms examples of model fragments into graph patterns. An example-driven approach for the specification also allows specifying CPEOs without a detailed understanding of the models' abstract syntax, i.e., the examples can be specified using the developer's preferred model editor.

📘 See Section 5 in Reference [6] **Manuel Ohrndorf**, Christopher Pietsch, Udo Kelter, Lars Grunske, and Timo Kehrer. History-based Model Repair Recommendations. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–46, 2021.

📄 See Section 2.4 in Reference [3] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, **Manuel Ohrndorf**, and Matthias Tichy. Henshin: A Usability-Focused Framework for EMF Model Transformation Development. In *Graph Transformation - 10th International Conference, ICGT 2017, Held as Part of STAF 2017, Marburg, Germany, July 18-19, 2017, Proceedings, volume 10373 of Lecture Notes in Computer Science*, pages 196–208. Springer, 2017.

1.4 Thesis Outline

This section summarizes the main chapters of this thesis. In addition, each chapter is discussed with respect to our related papers that have been published in peer-reviewed scientific conferences and journals (see Thesis-related List of Publications). The chapters are structured in relation to the contributions discussed in the previous Section 1.3. The thesis is concluded by evaluating the approach and discussing related and future work.

Chapter 2: State of the Art in Model Repair. Initially, this chapter prepares some motivating running examples demonstrating representative inconsistency-inducing and corrective edit steps that typically occur during modeling. Subsequently, we discuss the state of the art and define some terminology for model repair. Finally, a brief outline of the concepts introduced in this thesis is given.

Chapter 3: Background and Preliminaries. This chapter introduces the technical foundation for the approach presented in this thesis, i.e., the relevant concepts of modeling in MDE. We start with the specification of modeling languages, i.e., the specification of metamodels and consistency rules. For expressing modification in models, we introduce a notation for concrete changes and partially initialized change actions. In this context, a structural difference is computed as a set of concrete changes that describes the modifications between two versions of a model. Furthermore, we formally describe edit operations based on graph transformation concepts. Finally, a modeling language is defined to represent model differences and edit rules using annotated graphs, which we will refer to as unified difference graphs and unified edit rule graphs. This chapter contains basic parts published in Reference [6].

Chapter 4: Complementation of Partially Executed Edit Operations. This chapter presents the recognition and complementation of partially executed edit operations in model histories. Basically, the change actions of the specifying edit rule must be detected in the changes described by a model difference between two versions of a model. This problem is solved by partially matching occurrences of the unified edit rule graph in the unified difference graph. In particular, the computational steps of the partial graph matching algorithm are documented on a pseudocode level. This chapter presents a detailed specification of the algorithm briefly outlined in Reference [6] and extends the algorithm concerning the matching of complement rule applications.

Chapter 5: History-based Model Repair. In this chapter, we adapt the technique for complementing partially executed edit operations from the previous Chapter 4 to repair inconsistencies in models. Initially, the developer selects a certain inconsistency that is to be repaired. The repair process starts by searching for the latest consistent version with respect to the selected inconsistency in the model history. In the next step, the structural model difference between the latest consistent model version and the inconsistent model version is computed. In addition, an analysis of the selected inconsistency is performed to determine the problematic portion of the model. After recognizing possible inconsistency-inducing edit steps, the corresponding consistency-improving complementations are turned into a ranked list of repair proposals. This chapter is based on the approach presented in Reference [6] and includes a detailed specification of the inconsistency validation analysis.

Chapter 6: Presentation of History-based Recommendations. In this chapter, our tool ReVision is presented as a proof of concept of our approach to history-based model repair recommendation. We will discuss some implementation insights and the technological background of the tool. The interactions of the user interface during model repair is demonstrated by applying them to our running example. In particular, the user interface has been presented in References [5,6].

Chapter 7: Generation of Consistency-preserving Edit Operations. Edit operations are the main configuration input of our tool ReVision. Defining a comprehensive catalog of complex edit operations by hand can be tedious. This chapter introduces our example-based generation approach for edit operation. The process generates edit operations based on graph transformation concepts that respect particular consistency rules. Initially, a consistent, minimal example is modeled by the developer using the (graphical) model editor. From a catalog of such minimal modeling examples, different CPEOs are generated. The basic edit rule generation algorithm has been published in Reference [6].

Chapter 8: Evaluation. This chapter evaluates the model repair approach proposed in this thesis. Our approach describes the effect of repair proposals in terms of user-level edit operations. Based on the recognition of incomplete edits, our approach gives a detailed description of the origin of an inconsistency. Those features are, first and foremost, qualitative advantages over existing approaches. To evaluate the helpfulness of our approach quantitatively, the approach is evaluated in an offline experiment. The examined dataset is obtained from real-world modeling projects, e.g., the entire history of the UML meta-model. Comparing the extracted real-world repairs with those proposed by ReVision, we assess the coverage, relevance, and efficiency with respect to the quality of the recommendation results. Moreover, we evaluated the tool's performance with respect to computational runtimes during the evaluation of the real-world repair samples. The evaluation results in this chapter have been published and peer-reviewed in Reference [6].

Chapter 9: Related Work. In this chapter, we discuss publications that are related to the presented approach. Therefore, the existing approaches are classified into fully automated, language-specific, and adaptable model repair techniques. While our approach fundamentally differs from fully automated and language-specific approaches, it falls into the category of adaptable model repair. Finally, the generation of edit operations presented in Chapter 7 is compared to related techniques. This chapter is a revised and extended version of the related work analysis published in Reference [6].

Chapter 10: Conclusions and Future Work. In the concluding chapter, we summarize the achieved contributions of this thesis. Finally, we discuss the lessons learned and possible directions for future work.

2

State of the Art in Model Repair

The most interesting repair scenario for ReVision are inconsistencies introduced by editing complex model fragments and isolated editing of depending modeling views. Furthermore, inconsistency resolution can be an iterative process in which negative side effects of a repair might have to be repaired subsequently. In general, we differentiate between fully automated techniques and recommender systems for model repair. A repair scenario that involves design changes to the system model typically requires the expertise of a developer. In such a scenario, a recommender system that supports the developer is preferable over fully automated approaches. The conceptional idea of ReVision is to determine complementations and rollbacks of incomplete, inconsistency-inducing edits, which are proposed as a ranked list of repairs.

The approach presented in this thesis deals with repairing inconsistencies in models. This chapter starts motivating model repair using some representative scenarios in Section 2.1 applied to our running example from Section 1.1. The following Section 2.2 examines the state of the art in model repair. Finally, Section 2.3 gives an overview of the concepts introduced in this thesis.

2.1 Motivating Examples

This section shows some modeling examples that illustrate the kinds of inconsistencies and development scenarios addressed in this thesis. A modeling scenario starts with an *incomplete, inconsistency-inducing edit step*, followed by a *complementing, corrective edit step* resolving the inconsistency. Moreover, we will show that repairing inconsistencies can be an iterative process, i.e., a corrective edit step might have a negative side effect that introduces new inconsistencies that must be resolved by a subsequent corrective edit step. In the following, we use numbered edit steps to describe a set of related changes in such a modeling scenario.

2.1.1 Scenario 1: Editing of Complex Model Fragments

In the first modeling scenario, we will only edit a single modeling view. In this scenario, we want to extend the state machine introduced in Version 1 of our exemplary model in Figure 2.1 with the functionality of fast-forwarding the video.

Initial edit steps (2.1) and (2.2)*:* We add a new state named fast-forwarding, as shown in Version 2 of the model in Figure 2.2. The new state is then connected to the streaming state by a new transition named play. The transition is triggered by the corresponding operation play() in the class Video.

Incomplete edit step (2.3): Next, we add the opposite transition fastForward from the state streaming to the new state fast-forwarding.

Unrelated edit steps (2.4) and (2.5)*:* Modifications between two model versions might include several unrelated edit steps. In addition to the new fast-forwarding state, the *edit step (2.4)* between Version 1 and Version 2 of the state machine removes the state for previews, including its connected transitions. Finally, the *edit step (2.5)* renames the state stream to streaming.

Complementing edit step (3.1): To create the trigger for the transition fastForward, we must create a new operation in the class Video, as shown in Version 3 in Figure 2.3. This modification is referred to as *edit step (3.1a)*. Technically, the *edit step (3.1)* also includes the creation of an operation call event and a trigger referred to as *edit steps (3.1b)* and *(3.1c)*. However, UML has no explicit notation for these structures in the concrete diagram syntax. We will show their representation in the abstract syntax of a model in Section 3.1.

This is a typical example of complementing a complex model fragment in a multi-view modeling environment. It can also be seen as a repair in terms of the well-formedness of the state machine since transitions without triggers are only allowed in exceptional cases, e.g., transitions of initial states. Such edit steps generally require tooling that integrates the different modeling views. However, incomplete edit steps often occur alongside other modifications, making it more challenging to identify the origin of the resulting inconsistencies.

2.1.2 Scenario 2: Isolated Editing of Model Views

In this scenario, we demonstrate how isolated modifications in a modeling view can introduce inconsistencies in another view. Modifications in a class diagram are typical examples of isolated edits that can introduce inconsistencies into the overall system model.

Inconsistency-inducing edit step (4.1): In comparison to Version 3 in Figure 2.3, the Version 4 in Figure 2.4 shows that the operation disconnect() was moved from the class Video to the class Server. The class diagram in Figure 2.4a is still consistent at this point. However, in the sequence diagram in Figure 2.4b, the operation disconnect() is now invoked by message 6:disconnect on the object open:Video. This will be detected as an inconsistency since the required operation no longer exists in the corresponding class Video. This relation between operation calls and their definition is checked by the consistency rule *message_signature(m:Message)*, which we will formally introduce in Chapter 5.

Corrective edit step (5.1): However, it is not obvious how to resolve this inconsistency. In general, the message 6:disconnect could be removed, or we can choose a different operation of class Video as a signature for the message. Ultimately, this decision depends on the intended semantics of the model. As illustrated by Version 5 in Figure 2.5, we will resolve the inconsistency by changing the message end of 6:disconnect from the lifeline open:Video to the lifeline mirror:Server. Syntactically, choosing the lifeline main:Server would also resolve the inconsistency.

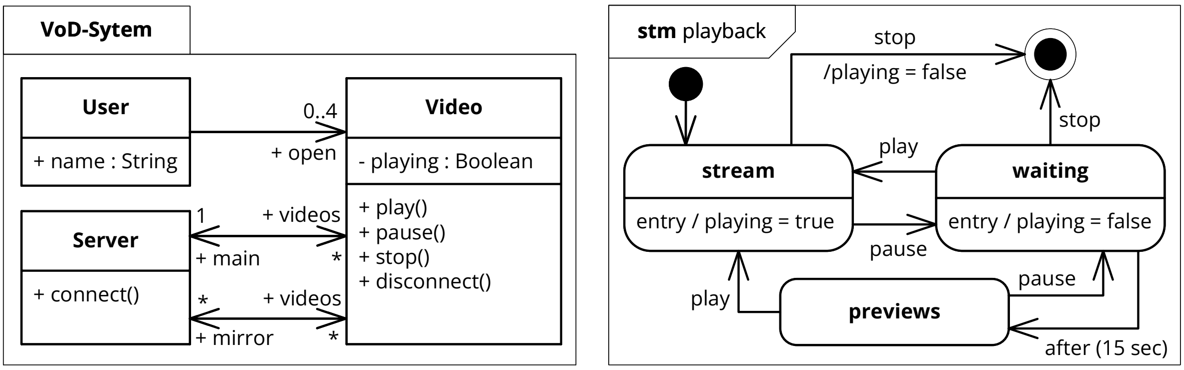
2.1.3 Scenario 3: Iterative Repair of Model Inconsistencies

The repair of the inconsistency after moving the operation disconnect() results in the Version 5 of our VoD-System model shown in Figure 2.5. The message 6:disconnect is now sent between the lifeline Alice:User and mirror:Server. However, this repair has a negative side effect which introduces a new inconsistency that has to be repaired subsequently.

The consistency rule *message_property(m:Message)* only allows sending messages between object instances that can reference each other. The consistency rule *message_property(m:Message)* is formally defined in Chapter 5. It is necessary that the class that defines the type of the sender has at least one property that refers to the type of the receiver. In the class diagram, such properties are typically illustrated by a navigable association. The consistency rule *message_property(m:Message)* is not fulfilled for the message 6:disconnect in Figure 2.5b and the defining classes User and Server in Figure 2.5a.

Inconsistency-inducing edit step (5.1): The inconsistency has been introduced by the last corrective *edit step (5.1)* which changed the receiver of the message 6:disconnect from lifeline open:Video to lifeline mirror:Server.

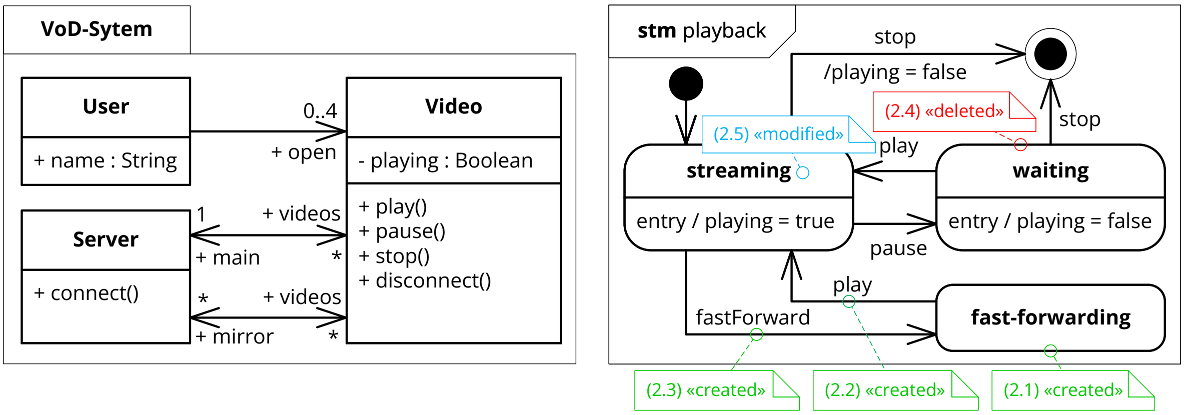
Corrective edit step (6.1): Again, there are different possibilities to repair this inconsistency, e.g., creating a new association or removing the message 6:disconnect. The latter seems unreasonable since we just started modifying the message to resolve the previous inconsistency. As illustrated in Figure 2.6 showing Version 6 of our exemplary model, we will correct the inconsistency by changing the sending lifeline of the message 6:disconnect. Changing the sender from Alice:User to open:Video will resolve the inconsistency, as the navigable associations main [1] and mirror [*] connect the corresponding defining classes.



(a) The unmodified initial class diagram.

(b) The unmodified initial state machine diagram.

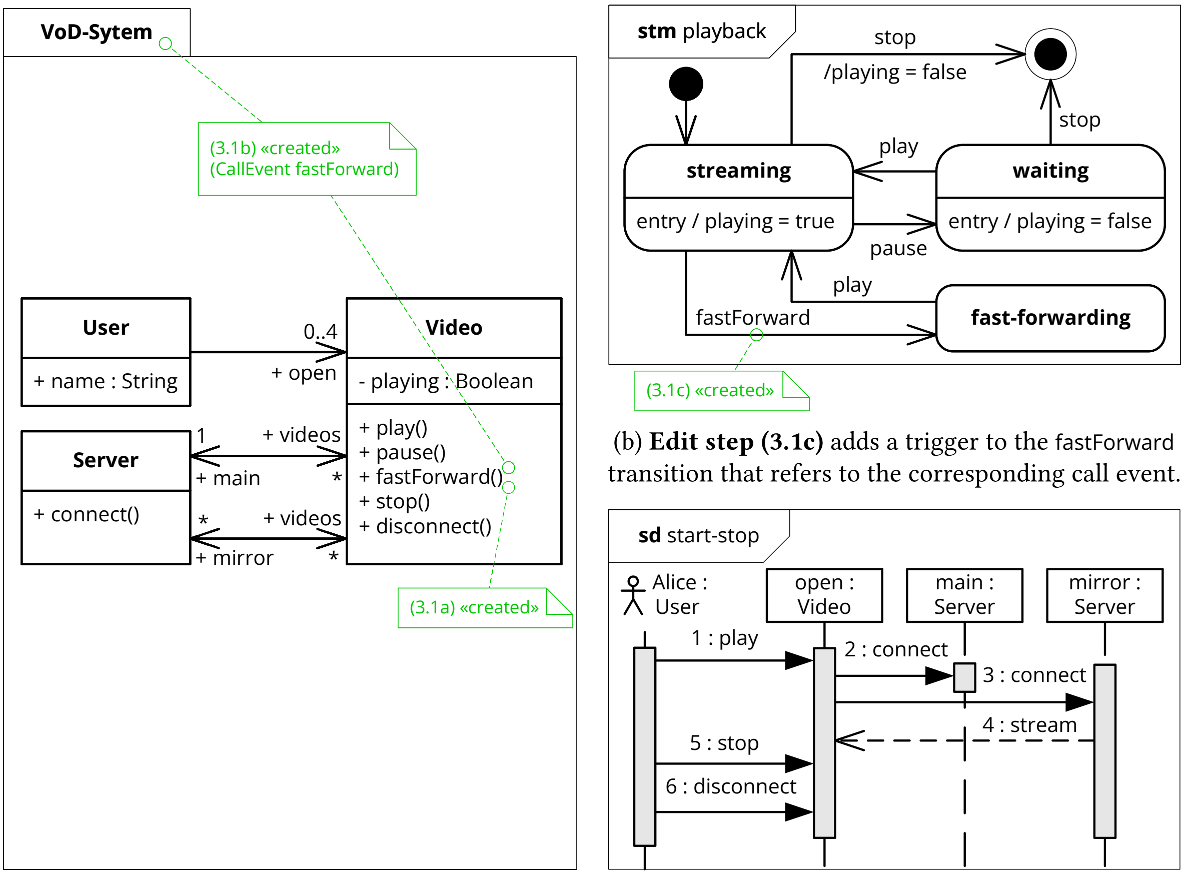
Figure 2.1. **Version 1:** The initial version of the class and state machine diagram from Figure 1.1.



(a) The unmodified class diagram.

(b) **Edit steps (2.1) - (2.3)** are adding state fast-forwarding. **Edit step (2.4)** removes state previews. **Edit step (2.5)** renames state stream.

Figure 2.2. **Version 2:** Adding the functionality of fast-forwarding to the state machine.

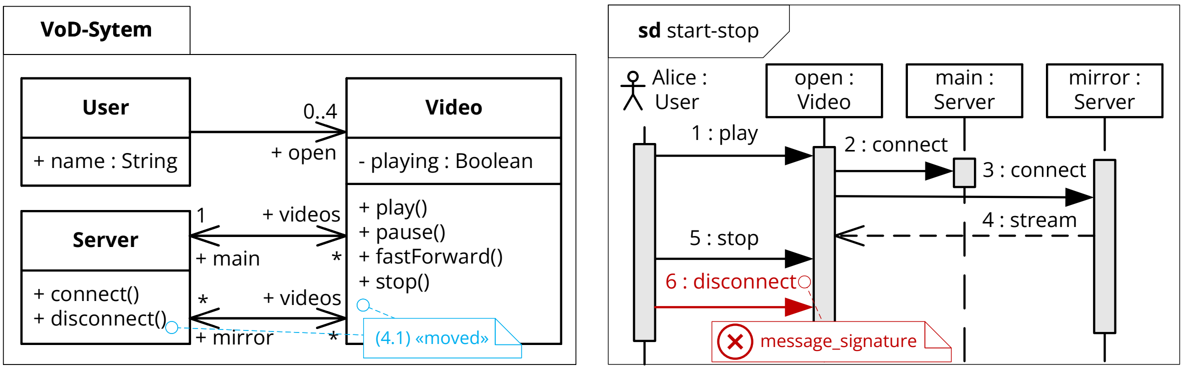


(a) **Edit steps (3.1a)** and **(3.1b)** are adding the operation fastForward() with a call event.

(b) **Edit step (3.1c)** adds a trigger to the fastForward transition that refers to the corresponding call event.

(c) The unmodified sequence diagram.

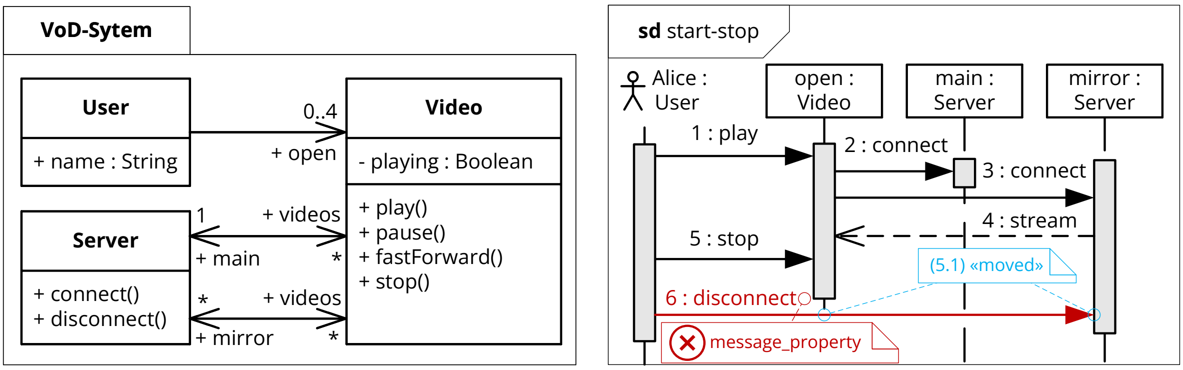
Figure 2.3. **Version 3:** Adding an operation for fast-forwarding to the class diagram.



(a) **Edit step (4.1)** moves the operation disconnect() from class Video to class Server.

(b) Violation of the *message_signature(m:Message)* consistency rule on message 6:disconnect.

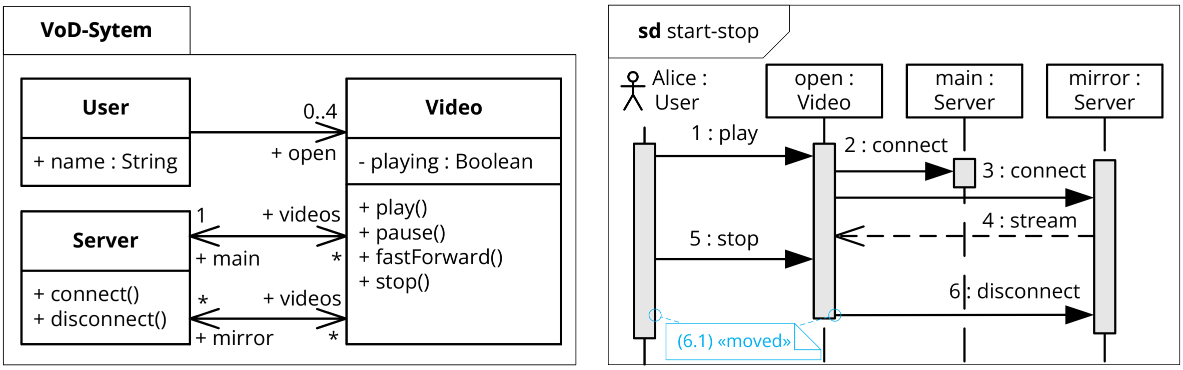
Figure 2.4. **Version 4:** Inconsistency-inducing edit step that moves the operation disconnect().



(a) The unmodified class diagram.

(b) **Edit step (5.1)** changes the receiver of message 6:disconnect from lifeline open:Video to mirror:Server, which violates the *message_property(m:Message)* consistency rule on message 6:disconnect.

Figure 2.5. **Version 5:** First corrective edit step that changes the receiver of message 6:disconnect.



(a) The unmodified class diagram.

(b) **Edit step (6.1)** changes the sender of message 6:disconnect from lifeline Alice:User to open:Video.

Figure 2.6. **Version 6:** Second corrective edit step that changes the sender of message 6:disconnect.

This modeling example shows that repairing inconsistencies can also have negative side effects, introducing new inconsistencies. Therefore, repairing models can be considered an iterative process in which multiple corrective edit steps are executed successively. This allows the developer to focus on a single inconsistency at a time, which reduces the complexity of a recommended corrective edit step and increases its understandability.

2.2 State of the Art

Large software systems are developed in teams where individual developers work on different tasks and parts of the project. Therefore, the team must develop a common understanding of the requirements and coordinate the interfaces between different software components. Especially in the early design phases, this can lead to contradictions in the design and implementation of the system. Such inconsistencies should be identified and communicated as early as possible; otherwise, more severe problems will likely arise in later development phases. However, under certain circumstances, e.g., if some design decisions are yet unclear, an inconsistency cannot be corrected immediately. In such cases, postponing an inconsistency's resolution can be helpful until more specific design decisions have been made.

The likelihood of inconsistencies increases as a project grows in size and complexity. If individual parts of a project are revised, overlapping or dependent parts in the system model must always be adapted accordingly. However, newly introduced inconsistencies can quickly be overlooked. Automated tooling should support the developers in detecting, documenting, and resolving inconsistencies. Since inconsistencies can occur in any development phase, the consistency of the project must be continuously monitored. Therefore, all tasks associated with the life cycle of an inconsistency are referred to as *consistency management* (also known as *inconsistency management*). Consistency management covers all activities, from the evolution of consistency rules and the detection of inconsistencies to analyzing their impact and developing a final resolution [41, 136]. The main focus of this thesis is the resolution of inconsistencies by recommending modifications that resolve a specific inconsistency in the latest model version. In the sequel, we will refer to this process as *model repair*.

In MDE, models are enriched with technical details over several phases, i.e., a model is developed from an abstract to a more refined model. In general, a model on a less abstract level must also be consistent with its predecessors on the more abstract levels. We refer to this kind of consistency as *inter-model consistency* or *vertical consistency* [71,112]. For example, the derived platform-specific implementation and the corresponding database schema must be inter-model consistent with the system's class diagram. The refinements between models at different abstraction levels are typically defined by incremental and bidirectional model transformation techniques [68], including triple graph grammars (TGGs) [11,54,166]. In this case, inter-model consistency is defined through these well-defined model transformation rules. Corresponding model repair techniques are typically fully automated, non-interactive approaches (see Table 2 in Reference [68]) with the objective of resolving all inconsistencies at once. Generally, an interactive human-in-the-loop transformation system could ask the developer to resolve ambiguous correspondences between transformed model fragments before resolving an inconsistency [68]. However, if the transformation between two abstraction levels is well-defined, an incremental transformation technique does not necessarily have to choose between multiple repair alternatives.

The modeling views of our running example in Figure 2.1 are on the same level of abstraction with respect to the MDE development process, i.e., a platform-independent description of a VoD-System. Conversely to inter-model consistency or vertical consistency, the consistency of models on the same level of abstraction is referred to as *intra-model consistency* or *horizontal consistency* [71, 112]. The approach in this thesis focuses on intra-model consistency between different views or within a single viewpoint on the same abstraction level.

Considering our running example from Figure 2.1, the views are defined by a single meta-model, i.e., the meta-model of the UML. In general, the models of a system can also be based on multiple meta-models, e.g., a meta-model for each viewpoint with connections between the meta-models. Conceptually, we can combine multiple meta-models to form a single unified meta-model [115]. An instance of such a (unified) meta-model can be conceptionally considered as an *abstract syntax graph (ASG)*, basically, a typed attributed graph with a containment structure in which the types of nodes and edges are drawn from the meta-model serving as a type graph (see, e.g., Reference [27]). The containment structure describes the *abstract syntax tree (AST)* within the ASG.

In the following sections, we will examine the state of the art in model repair approaches for intra-model consistency. The feature-based classification system proposed by Macedo et al. [115] distinguishes model repair approaches by several technical features. The main technical features will be discussed in Section 2.2.1 to Section 2.2.3. In Section 2.2.1, we will examine the adaptability and limitations of the repair approaches with respect to the different modeling domains or languages that can be handled. Section 2.2.2 compares the formalisms used to specify the consistency of a modeling language and its effect on the repair calculations. In Section 2.2.3, we will discuss techniques that include information on user modifications during the model repair.

Beyond the technical classifications of model repair, we will consider the *repair scenario* as an additional criterion. Therefore, we will look at the underlying engine of the repair generation procedure and the representation of the proposed repairs. In Section 2.2.4, as discussed in Reference [132], we will distinguish between the two repair scenarios of fully automated approaches and recommendation systems.

CPEOs are the primary configuration input of ReVision to define the consistent modifications of a modeling language. Finally, we will give an overview of the state of the art in generating edit operations in Section 2.2.5.

2.2.1 Modeling Domain

The modeling domain refers to a specific area of problems that should be represented by a modeling language. Special kinds of model repair techniques are *language-specific* approaches that cannot be adapted to different modeling languages, e.g., specific repair techniques for UML models [45, 129, 184, 185], architectural models [38, 48], or business process models [51, 117]. Language-specific approaches have limited applicability, especially when it comes to domain-specific modeling with a variety of languages in MDE.

A modeling language is typically defined by a meta-model or sometimes by similar (translatable) formalisms like graph grammars [9, 194]. All models defined for the specified modeling language must conform to the structure and type definitions of this meta-model. Therefore, other kinds of model repair techniques are *meta-model independent* approaches, i.e., approaches that can be applied or adapted to any given meta-model. In the context of modeling languages, we can also refer to meta-model independent approaches as *language-independent* repair approach. A language-independent approach is presented by the generic syntactic technique in References [134, 148, 149] that analyzes the logical constraints defining a consistency rule. A generic approach does not require language-specific adaptions. In contrast, a rule-based technique, as described in References [2,22,28,133,199], relies on language-specific rules that are used to adapt the technique to a given modeling language. Finally, some search-based repair techniques can handle arbitrary meta-models [64, 114, 147, 186]. Conclusively, meta-model independent model repair approaches have a wide range of possible applications, but this can come at the cost of additional required configurations.

A model repair technique can also be limited with respect to the supported model instances, e.g., only a preconfigured set of textual values or a fixed range of numbers might be supported [36, 45, 114, 186]. A *bounded universe* of possible model instances might help the approach to limit the search space for repair alternatives [115]. However, some possible repairs may not be found due to such limitations.

To work around this problem, some approaches introduce abstract placeholders, which must be instantiated by the developer, e.g., as parameters of a proposed repair plan. In general, parameterizable repair plans are also referred to as *abstract repair plans*. On the other hand, a repair plan with fixed values and repair actions is referred to as a *concrete repair plan*. Fully automated repair techniques naturally require concrete repair plans, while abstract repair plans are more suitable for recommendation approaches. Also, a combination of both approaches is possible so that several concrete instantiations are suggested for an abstract repair plan. However, the generation of attribute values is a challenging task. In some cases, even the number of correct alternatives can be practically unlimited, e.g., the name of a new element. Some approaches try to synthesize attribute values from existing values in the model [95].

Similar to auto-completion tools of programming languages, auto-completion in MDE can improve the quality of models and efficiency of modeling activities by supporting a developer with context-specific recommendations to extend a model. Several approaches have been proposed for generic [123, 168, 174] and language-specific [29, 69, 91, 98, 99, 124] auto-completion of models. In general, auto-completion techniques can also help to preserve the consistency of models. However, such approaches are not intended to repair specific inconsistencies already existing in a model.

Compared to model repair approaches, automated program repair approaches, as presented in References [74, 105, 126, 127, 135, 190], aim at repairing a program's semantics. Typically, the program must be repaired to pass a set of predefined test cases. However, the approach presented in this thesis aims at resolving syntactical inconsistencies instead of fixing a model's behavior.

2.2.2 Consistency Constraints

A meta-model defines the basic structure and types of a model's ASG. More complex consistency constraints require additional rules that restrict the allowed model instances. Some model repair techniques are designed to support only a fixed set of consistency rules, which is also typical for language-specific approaches. In contrast, an adaptable model repair approach allows the definition of new consistency rules. This is required for language-independent approaches to adapt the technique to a new modeling domain. Moreover, user-defined consistency rules can also constrain a modeling language in terms of domain-specific design rules.

We assume that a consistency rule is described using a specific formalism so that it can be checked automatically on a given model instance. The validation of a consistency rule at least indicates whether it succeeded or failed. Typically, a constraint is defined and checked on a specific type of element in the model that is referred to as the *context element* of the validation. The evaluation of the consistency rule can be analyzed (see, e.g., References [148, 151]) to reason about potential faulty locations in the model, e.g., missing edges or wrong attribute values. Most approaches use logical constraint languages, like the Object Constraint Language (OCL), which is based on the theory of first-order logic. Some graph-based approaches use pattern matching techniques to formulate consistency rules. Those graph constraints typically define positive and negative graph patterns that are combined using basic logical operators. Generally, a model repair technique can depend on a specific formalism or require a particular type of information as the result of the validation.

If multiple inconsistencies occur in a model, some repair approaches handle all inconsistencies at the same time [35, 39, 114, 146, 186], which is also typical for model synchronization and inter-model consistency approaches [12,65,65–67,104,113,116,142,171,200]. Other model repair tools allow the developer to select a specific inconsistency [22, 45, 64, 100, 117, 129, 134, 149, 185], which is referred to as *violation selection* [115].

Approaches have different characteristics with respect to their completeness and correctness regarding a given inconsistency to be repaired. An approach is considered to be *complete* if it can enumerate all possible repair alternatives for a concrete inconsistency [149]. A search-based approach can theoretically cover the entire search space of a model instance. However, in practice, the search engine is typically limited by some kind of time or space threshold. Moreover, we have to deal with those cases that can lead to practically unlimited repair alternatives. In such cases, it is only possible to achieve completeness by utilizing abstract repair plans.

2.2.3 Change-based Approaches

We can basically distinguish between *state-based* and *delta-based* model repair approaches [115]. A state-based approach evaluates and repairs the current version of a model. A delta-based technique involves the user's actions to propose model repairs. Delta-based repairing is mostly based on some kind of online logging of the modifications applied in the model editor. Some early human-centric consistency management approaches use change logging to help the developer to understand the cause of an inconsistency [41, 52, 61]. A few approaches log user actions to increase the efficiency of the inconsistency detection by incrementally checking the changed parts of the model [45, 149, 188]. However, the calculation of model repairs remains basically independent of the user's actions. Logging user actions can also replace the need to manually select specific inconsistencies, enabling an immediate resolution of conflicting actions [117, 199]. However, if user interactions directly trigger an automated inconsistency resolution, the approach is not intended to tolerate inconsistencies temporarily.

To determine the impact of user actions, some approaches define guard or detection rules that trigger predefined repair plans [2, 22, 36, 147]. However, the pairs of detection and resolution rules must be defined a priori at the configuration time of the repair technique. Thereby, only a few repair approaches can prevent undoing former work, which is referred to as *change-preserving* [2] or *delta-preserving* [163, 164].

The approach proposed by Puissant et al. [147] uses meta-information of the modeling history to guide search-based repair approaches, e.g., preferring to modify newer elements and keeping the older ones.

2.2.4 Fully Automated vs. Recommendation Approaches

Existing model repair techniques can mainly be categorized into *syntactic*, *rule-based* and *search-based approaches*. A syntactic approach analyzes a violated consistency rule to derive repair plans. Rule-based techniques are configured with rules for repairing and sometimes also with rules for detecting inconsistencies. Search-based model repair approaches [49, 90, 114, 186] heuristically explore the state space of a model to find suitable consistent states of a given model. Search-based approaches are typically used for fully automated repair processes. Similarly, rule-based approaches that exclusively aim at synchronizing multi-view models [58, 92, 199] are mostly fully automated repair techniques.

Tools for model repair recommendation typically allow violation selection, which helps developers to focus on a single inconsistency at a time. Inconsistencies can be resolved in incremental steps, i.e., negative side effects that introduce new inconsistencies are acceptable as long as the model converges to a consistent state. Moreover, model repair recommendation tools should be adaptable to the developers' preferences.

We can basically distinguish between recommendation approaches that aim at elementary meta-model consistency [133] and those that support arbitrary complex consistency rules [28, 147, 149]. The elementary consistency defines the basic structural constraints of a model's ASG, e.g., multiplicities of edges or containment relations between model elements [27].

Correspondingly, the modifications of edges, nodes, or attributes on the level of the ASG are referred to as elementary changes. In the context of model repair, multiple elementary changes can be composed into a *repair plan* that resolves a complex inconsistency. Assuming a repair plan addresses a specific inconsistency, we will refer to a repair plan as *inconsistency-improving* if it addresses the inconsistency only partially, i.e., the repair plan potentially requires additional changes to actually resolve the inconsistency. In particular, the additional changes do not roll back any already applied changes of the inconsistency-improving repair plan. A repair plan is called *inconsistency-resolving* if it effectively resolves the addressed inconsistency. In general, an inconsistency-resolving repair plan is also inconsistency-improving, i.e., in this case, no additional changes are required.

Model repair recommendation techniques should present the set of repair proposals in a comprehensible way. The most basic representation is a ranked list of repair plans. Most approaches rank repairs according to some kind of least-change criterion, i.e., proposing the repair plan with the minimum number of changes on the top positions. This is typically defined by a distance function comparing two alternative repair plans according to their modifications, e.g., by the number of contained elementary ASG changes or custom edit operations. However, this does not always mean that the topmost repair plan is actually the smallest possible update. Finding the actual smallest repair would require considering every existing repair [115]. Moreover, certain approaches also enable the developer to customize the repair ranking, e.g., by adjusting the distance function [114] or by weighting additional information like the creation time of a model element [147].

The repair plans can be described as state-based or delta-based proposals, i.e., by the corrected model state or by the modification to be applied, respectively. State-based repair plans are typically computed by search-based techniques that enumerate alternatives of model instances derived from the inconsistent state of a model. In contrast, delta-based repair plans can be proposed as parameterizable operations, i.e., proposing abstract repair plans that are initialized by the developer. Moreover, since elementary changes can be challenging to understand for developers who primarily work on the concrete (diagram) syntax of a model, some repair approaches utilize user-defined edit operations to describe the proposed repair plans [2, 22, 64, 92, 111, 188].

Reder et al. [149] propose repair trees from which a developer selects different combinations of repair actions to form a repair plan. The repair tree restricts the possible combinations of abstract repair actions that eventually need to be initialized with concrete parameter values.

2.2.5 Generation of Consistency-preserving Edit Operations

Edit operations are essential descriptions in many modeling tools [1,4,25,78,128,145,152,180] in MDE. Edit operations can be used to define the correct transition between two valid model instances. An edit operation combines multiple elementary changes, i.e., modifications of nodes, edges, or attributes in the ASG. CPEOs may be specified in a declarative way using in-place model transformations based on graph transformation concepts. In the context of our model repair approach, CPEOs are required that prevent typical inconsistencies when models are edited in standard editors (see Section 3.5). However, the manual definition of edit operations that pay attention to the complex consistency rules of a modeling language can be tedious and error-prone. Therefore, techniques that support or fully automate the specification of complex edit operations are required.

Some existing approaches for generating edit operations are used in the context of model instance generation, e.g., for generating test cases. Similar to context-free grammars for textual languages, such approaches derive graph grammars from the meta-model of a modeling language [47, 55, 178]. The derived rules are typically constructive, i.e., no rules for modifying existing elements are generated. However, modifications to existing model fragments may require synchronization during model evolution, especially when working with multiple modeling views. Other approaches like DeltaEcore [167] or SERGe [84, 85, 153] also generate deleting or modifying edit operations that respect the multiplicity constraints in a given meta-model. However, these approaches only consider the elementary consistency constraints specified by the meta-model. Such elementary edit operations can hardly serve as CPEOs for repairing advanced consistency constraints.

Model transformations represent a central concept in MDE. Those transformations are usually defined using the abstract syntax of a modeling language. As developers typically work on a model's concrete (diagram) syntax, some approaches suggest that it would be preferable to define transformations using concrete modeling examples. This allows the definition of transformations using the usual model editor. Approaches for model transformation by-example can basically be divided into *demonstration-based* and *correspondence-based* techniques [76]. A demonstration-based approach [31, 177] does a live recording of the changes applied in a model editor. Next, these recorded concrete modifications must be generalized into an edit operation. In contrast, a correspondence-based approach [3, 158] derives edit operation from a pair of model examples. These approaches compute correspondences between the model elements of both examples to derive the actual modifications between both models. A correspondence-based approach can work offline, i.e., no logging integration into the modeling environment is required. However, the derived modification can differ from the actual ones the developer performed to produce the example depending on the computed correspondences. Creating transformations by-example can lower the entry barrier for a formal edit operation specification. This task would otherwise require knowledge about the meta-model of a modeling language.

Generally, example-based approaches allow the creation of arbitrary complex edit operations. However, providing a comprehensive set of edit operations by modeling all possible scenarios using examples can still be challenging. In general, it is difficult to make assumptions about the completeness and correctness of a collected set of edit operations for a given meta-model (see Reference [85]) and its consistency rules.

2.3 Approach Outline

The approach introduced by this thesis addresses the issues of existing approaches to model repair analyzed in Section 1.2 and Section 2.2. It can be considered as analytic approach to developer-centric model repair recommendations, in contrast to exhaustive search strategies used for the automated repairing of model inconsistencies. Following the generally accepted debugging strategy of fixing a single defect at a time [201], the approach is designed to iteratively analyze and repair the violations of consistency rules in a model. According to the technical features discussed in Section 2.2, the approach can be classified as a hybrid syntactic rule-based model repair technique based on a violation selection repair process. Based on an impact analysis of the proposed repair plan changes with respect to the inconsistency, we determine whether a complementation is at least inconsistency-improving. As the repair plan describes a complex consistency-preserving edit step based on a CPEO, this typically implies that the repair plan is inconsistency-resolving. As shown by the modeling examples in Section 1.1 and Section 2.1, the approach particularly addresses inconsistencies in models introduced by isolated editing of single views or modifications of complex model fragments.

As we have analyzed in Section 1.2, system models in MDE have to keep up with the ever-changing requirements of the software evolution process. The most appropriate approaches to the repair of design-related model inconsistencies are recommender systems that assist a human developer. In general, recommender systems are software tools that support users in their decision-making while interacting with large information spaces by suggesting items that are likely to meet their needs and preferences [154]. According to the definition of Robillard et al. [154], a recommendation system for software engineering provides valuable information for a software engineering task in a specific context. Generally, recommender systems support developers in various activities such as searching codebases and software documentation. Auto-completions, code templates, or refactoring proposals are well-known tools that aim at speeding up a developer's programming workflow. More sophisticated recommendation systems for software engineering can aid developers during program repair, e.g., bug localization [89, 101], program debugging [105] or adapting existing code bases to a new library [37].

<table>
<tr><th>Nature of the context</th><th>Recommendation engine</th><th>Output mode</th></tr>
<tr><td rowspan="3">**Input:**
explicit | implicit | hybrid</td><td>**Data:**
source | history | interaction | ...</td><td>**Mode:**
push | pull</td></tr>
<tr><td>**Ranking:**
yes | no</td><td>**Presentation:**
batch | inline</td></tr>
<tr><td colspan="2">**Explanations:**
from none to detailed</td></tr>
</table>

Table 2.1. Design dimensions of recommendation systems for software engineering tasks from Table I in Reference [154].

Table 2.1 shows the classification schema for recommendation systems in software engineering as proposed by Robillard et al. [154] in Table I of Reference [154]. This schema classifies recommendation systems for software engineering by three general design dimensions: *nature of context*, *recommendation engine*, and *output mode*. Initially, the *context of a recommendation* needs to be established. The more precise the context is determined, the more accurate a recommendation can be. The context of a recommendation can be given explicitly by a user or is implicitly extracted for the corresponding task. The context of ReVision is explicitly given by a developer in terms of the inconsistency to be resolved.

In general, the *recommendation engine* is responsible for the computation and ranking of the recommendations. Its design mainly depends on the task-specific input data, which is analyzed in addition to the recommendation context [154]. In addition to the inconsistency, ReVision analyzes the modeling history for incomplete edit steps to discover corresponding complementations and rollbacks that are finally presented as a ranked list. The objective of our ranking is to find edit steps that are likely continuations of incomplete edit steps and which, at the same time, keep the number of proposed modifications low.

The third design dimension of recommendation systems in Table 2.1 is the output mode. The recommendation system can either interact with the developer by actively pushing recommendations to the developer or the developer can actively pull recommendations. Most recommendation systems operate in pull mode to avoid unnecessary distractions for the developer. In our repair tool ReVision, the developer must actively pull the recommendations, which will be presented in a batch mode, i.e., a ranked list of recommendations. A pull design is also more appropriate to tolerate inconsistencies temporarily during modeling than a push design (see Section 2.2.3).

Another essential task for recommender systems is to explain the proposals to the developer. Explaining the resulting recommendations can increase confidence in the system. However, the amount of explanations should not overwhelm the developer [154]. Re(pair)Vision explains a recommended repair by describing it as a *pair* of an incomplete, inconsistency-inducing edit step and a complementing, corrective edit step. Technically, the computations of ReVision are deterministic, which can also increase the developers' confidence in the recommendations.

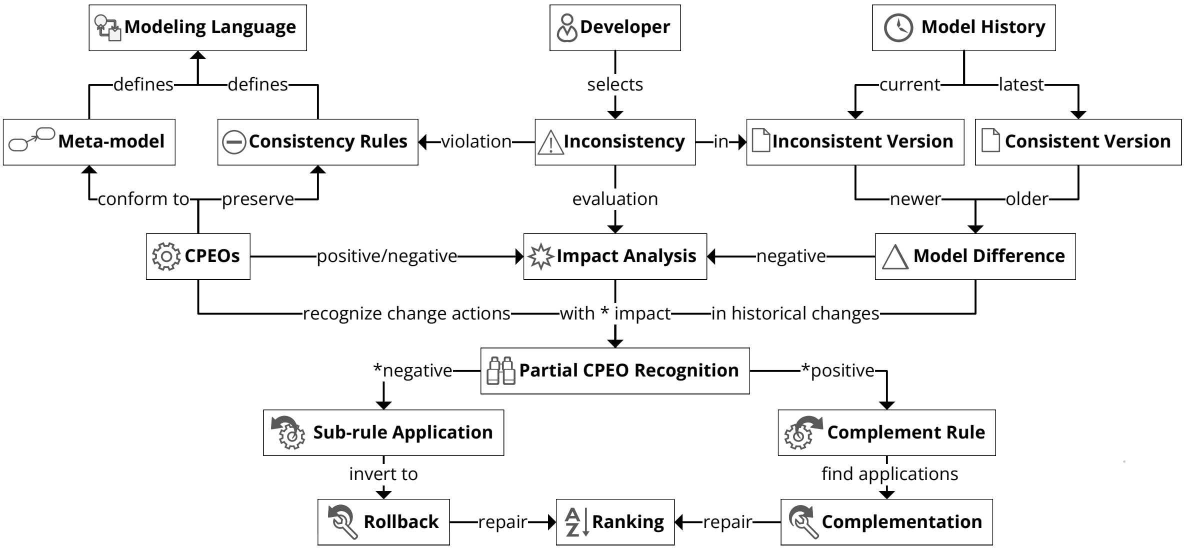


Figure 2.7. Approach Outline: Basic concepts of the model repair technique and their relations. The edges indicate the reading direction of the relations.

Figure 2.7 shows the basic concepts of our approach and their relations. We assume that the correct instances of a modeling language are defined by a meta-model and additional restrictive consistency rules. Transitions between correct instances are specified by CPEOs. Inconsistencies can be automatically detected by evaluating the consistency rules on the current model instance. If violations are detected, the developer can select the inconsistency to be repaired.

To find the cause of an inconsistency, the model difference between the current inconsistent model version and the latest known consistent version with respect to the selected inconsistency is computed. In the next step, potential inconsistency-inducing changes from the model difference are detected by analyzing the negative impact of changes with respect to the violated consistency rule. In contrast, a planned change action that can contribute to a corrective edit step positively impacts the violation. Therefore, we also analyze the potential negative and positive impact of the change actions specified by the CPEOs.

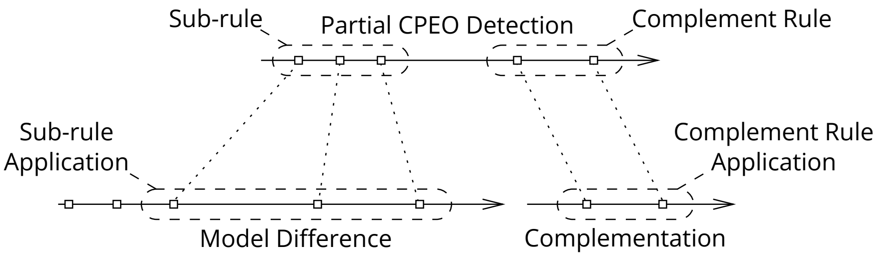


Figure 2.8. Complementation of partially executed CPEO detected in a model difference. The involved modifications are illustrated as sequences of changes.

In the next step, partial applications of the CPEOs are detected in the model difference. This process is illustrated in Figure 2.8 by showing the mappings between the involved changes in the model difference and the CPEO. Notably, our approach for detecting partial executions of CPEOs does not depend on the temporal order in which the changes have been applied, and they may be intertwined with other changes. The detection results in possible sub-rule applications of the CPEOs recognized in the model difference. For each sub-rule, a corresponding complement rule can be constructed comprising the remaining change actions of the CPEO. In the following processing step, parameterizable complementations are calculated that have an improving effect on the inconsistency. Conversely, a developer can also apply the inverted changes of the sub-rule application to roll back the inconsistency-inducing changes. Finally, the complementations and rollbacks will be ranked and presented to the developer as a list of possible repairs.

3
Background and Preliminaries

The abstract syntax of a modeling language, e.g., class, sequence, and state machine diagrams of the UML, is defined by a meta-model, including the types, attributes, references, and containment structure of the ASG. The abstract syntax of the meta-model itself is, in turn, defined by a meta-metamodel. A meta-model can be further constrained by the specification of complex consistency rules. Change actions during modeling can be generically expressed as elementary changes on the basis of the abstract syntax of a model. Similarly, changes between successive versions of a model can be represented as a model difference. Complex edit operations can be expressed using model transformations based on graph transformation rules. As a compact notation, change actions and changes can be represented using annotated graphs, which we will refer to as edit rule graphs and difference graphs.

In this chapter, we discuss the fundamental concepts that define a modeling language in MDE. We start with the basic definition of the types and structure of a model's ASG by defining a meta-model in Section 3.1. The formal specifications of advanced consistency rules that further restrict possible instances of a meta-model are introduced in Section 3.2.

In addition to the specification of modeling languages, we define a generic notation for structural changes in models in Section 3.3. Such changes can represent change actions with formal unbound parameters, abstract changes with partially bound parameters or concrete changes. Based on the notation of concrete changes, the historical changes between two versions of a model are represented as a model difference in Section 3.4. Section 3.5 defines complex edit operations of modeling languages using model transformations based on graph transformation concepts. Furthermore, Section 3.3, Section 3.4, and Section 3.5 introduce the concept of unified graphs to represent the different structural changes through a common notation.

3.1 Specification of Modeling Languages

Modeling languages use constructs that are expressive for specific problems in a particular domain. Developers define a model using its *concrete syntax*, i.e., a graphical diagram or a textual notation. However, tools in MDE typically process a model based on its *abstract syntax*. This allows the tools to be decoupled from the concrete syntax and their implementing technologies.

In MDE, specialized frameworks deal with the implementation of graphical editors [156] or grammars for textual DSMLs [23]. Such frameworks handle the synchronization between the concrete and abstract syntax. The specification of relations between the abstract and concrete syntax is out of the scope of this thesis. In the following, we assume that modifications or selected elements in the concrete syntax representation can be synchronized to the ASG and vice versa.

As usual in MDE, we assume that a meta-model specifies the abstract syntax of a modeling language, i.e., the allowed element types, their properties, and structural relations. Other approaches for defining modeling languages are sometimes used due to formalization reasons, e.g., graph grammars define modeling languages by graph transformation rules. Nevertheless, even for a grammar-based approach, a meta-model can be derived from the grammar in an automated way (see, e.g., [9,194]). In the following, we will discuss the definition of the abstract syntax by the concept of meta-modeling.

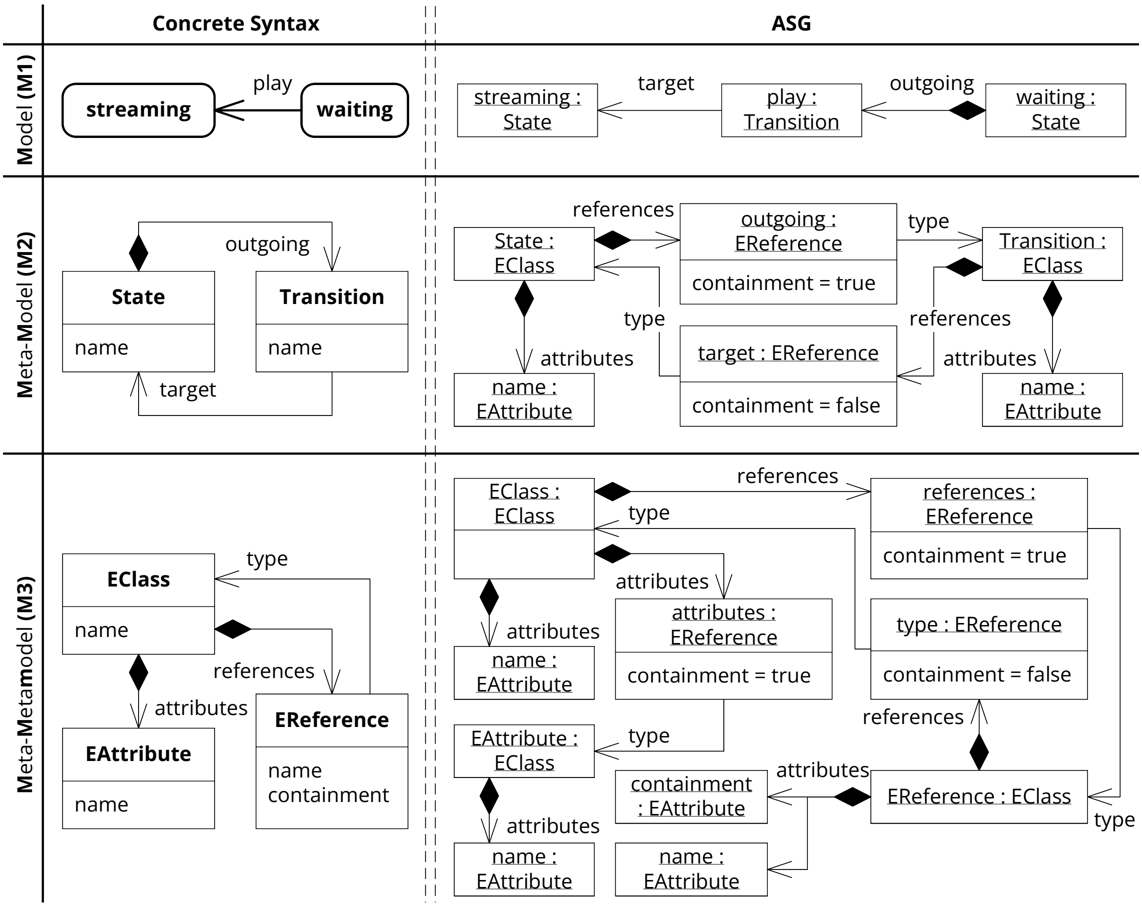


Figure 3.1. The definition of a simple modeling language for state machines by the concept of meta-modeling. The table illustrates the concrete syntax of each layer in the first column and the ASG representation in the second column.

Figure 3.1 (see column *Concrete Syntax* in row *Model (M1)*) shows a simple state machine diagram consisting of two states streaming and waiting connected by single transition play.

The corresponding ASG of the model in Figure 3.1 (see column *ASG* in row *Model (M1)*) is illustrated by an object diagram notation, i.e., an object name:Type is noted by the value of its name attribute and its type. Assignments of attributes are optionally shown in the body of the object notation. A reference is illustrated as a typed edge between two objects. The containment references specify the AST structure within an ASG. A reference is either directed, indicated by an arrow, or undirected. In the ASG in Figure 3.1, the transition of the state machine is represented by the object play:Transition connecting the objects streaming:State and waiting:State by the references of type outgoing and target.

On the next layer in Figure 3.1 (see row *Meta-Model (M2)*), the corresponding meta-model (also known as M2-layer) of the state machine model (also known as M1-layer) is shown. Technically, the state machine model on the first layer is an instance of the state machine meta-model on the second layer. The ASG of the model must conform with the definitions in the meta-model, i.e., only types and structures defined on the meta-model can be instantiated in the model. As a meta-model specifies the data structure of the ASG, the concrete syntax of the meta-model in Figure 3.1 (see column *Concrete Syntax* in row *Meta-Model (M2)*) uses a simplified class diagram notation. The meta-model shows that a State contains its outgoing Transitions, a transition specifies a target state, and states and transitions are named. As illustrated in Figure 3.1 (see column *ASG* in row *Meta-Model (M2)*), a meta-model can also be viewed by its ASG.

The types and structure of a meta-model's ASG are, in turn, defined by a so-called meta-metamodel (also known as M3-layer), i.e., the meta-model is an instance of the meta-metamodel. For example, as illustrated in Figure 3.1 (see column *Concrete Syntax* in row *Meta-Metamodel (M3)*), a reference of the meta-model is specified by the meta-metaclass EReference. A containment reference is indicated by setting the corresponding attribute.

The meta-metamodel's ASG in Figure 3.1 (see column *ASG* in row *Meta-Metamodel (M3)*) only uses types and structures that are defined by the meta-metamodel itself. Such meta-metamodels are referred to as self-describing or self-referencing definitions [33, 53], i.e., the meta-metamodel is an instance of itself.

3.1.1 Meta-Model (UML) of the Running Example

Consider again our motivating example introduced in Section 1.1. The diagrams shown in Figure 1.1 represent different views of the same system in concrete syntax. Considered as an ASG, they are one integrated model that conforms to the UML meta-model. For brevity, we introduce a simplified excerpt of the UML meta-model [141] that is relevant to our running example. For readability, the integrated UML meta-model is presented in separate excerpts for each diagram type, namely, the specification of class diagrams (see Figure 3.2), sequence diagrams (see Figure 3.3), and state machine diagrams (see Figure 3.4). The structural interrelations between the diagrams are indicated by marking the overlapping meta-classes with an ⇲ arrow icon.

Class diagram meta-model. Figure 3.2 shows the meta-model that defines the abstract syntax of class diagrams. Class diagrams can be structured by ModelPackages. A Class consists of Properties (also referred to as attributes) and parameterizable Operations. Classes can

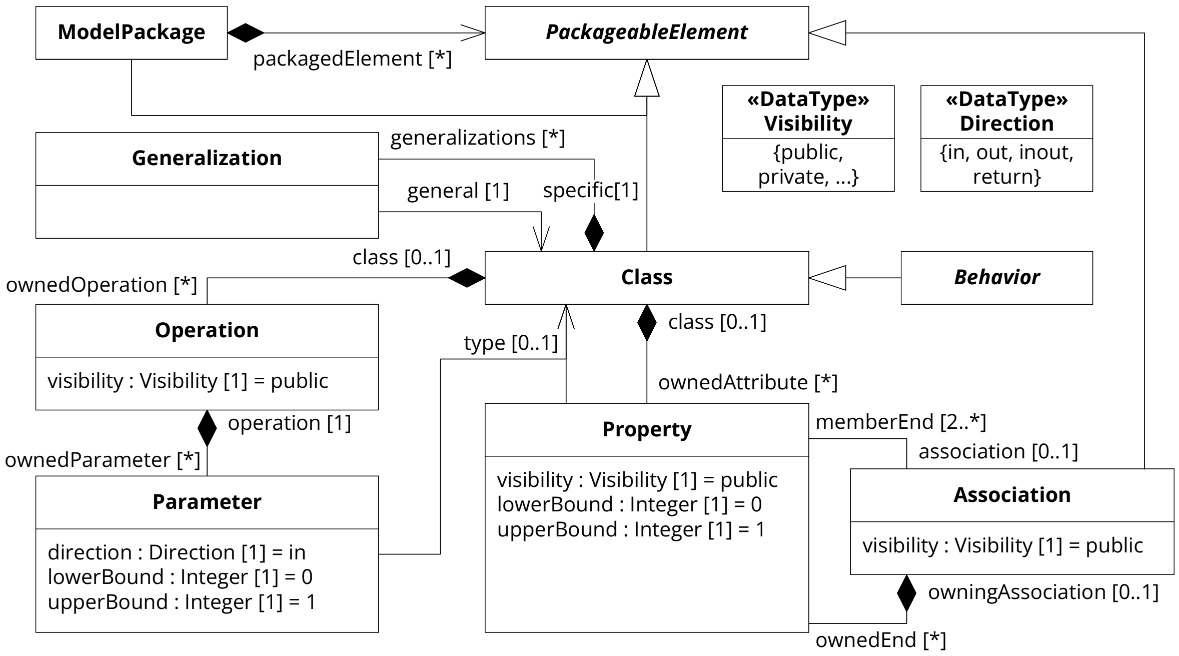


Figure 3.2. Simplified UML meta-model (M2-layer) for specifying class diagrams.

define a type hierarchy by specifying Generalizations between classes. Two or more properties (memberEnd) can be combined into an Association. An association is navigable in a specific direction if the property is contained by a class (ownedAttribute) and not by the association itself (ownedEnd). In particular, properties and parameters have multiplicities that define a lower-bound and upper-bound number of assignable entities.

Generally, the specification of the data types of attributes is represented as classes annotated with the «DataType» stereotype. We informally define the corresponding domain in the body of the data type. For example, the Direction of operation parameters is specified by enumerating the values of the domain.

Sequence diagram meta-model. Figure 3.3 extends the simplified UML meta-model from Figure 3.2 with the abstract syntax of sequence diagrams. A sequence diagram is contained by an Interaction, depicted as an outer frame in the concrete diagram syntax. A Lifeline represents a property of a class and can send and receive Messages. A MessageEnd defines the sending and receiving lifeline of a message. A message references an operation of a class that serves as the message signature.

State machine diagram meta-model. Figure 3.4 shows the simplified excerpt of the state machine meta-model. A state machine defines the behavior of a class. A StateMachine is basically a composition of Regions and States. For simplicity, we note the kind of state, e.g., a start or a final state, as an attribute value. Optionally, the entry and exit behavior of a regular state can be specified by an interpretable expression. A Transition connects a source and target state. Transitions can be annotated with a trigger [guard] /effect expression in the

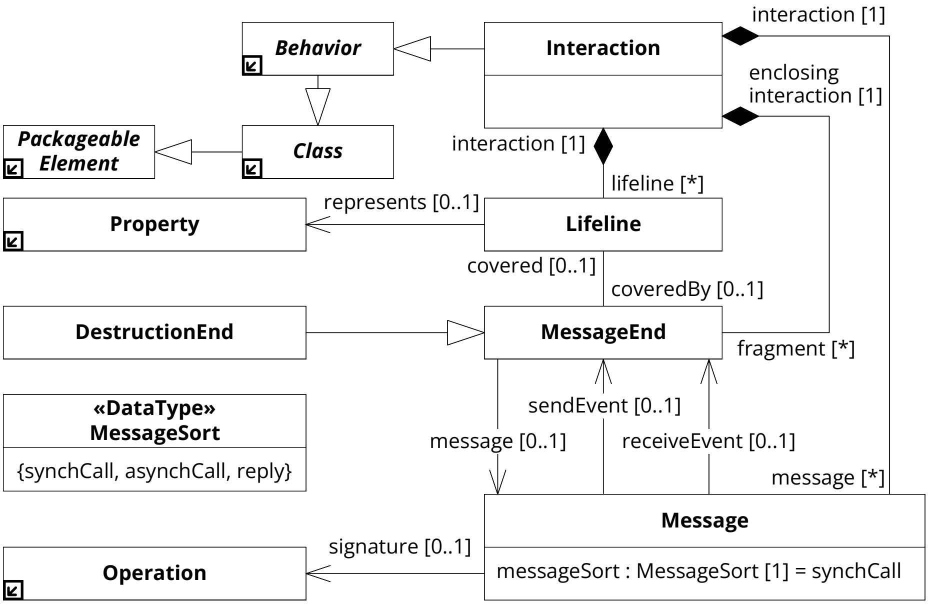


Figure 3.3. Simplified UML meta-model (M2-layer) for specifying sequence diagrams. Extends the ↙ UML class diagram meta-model from Figure 3.2.

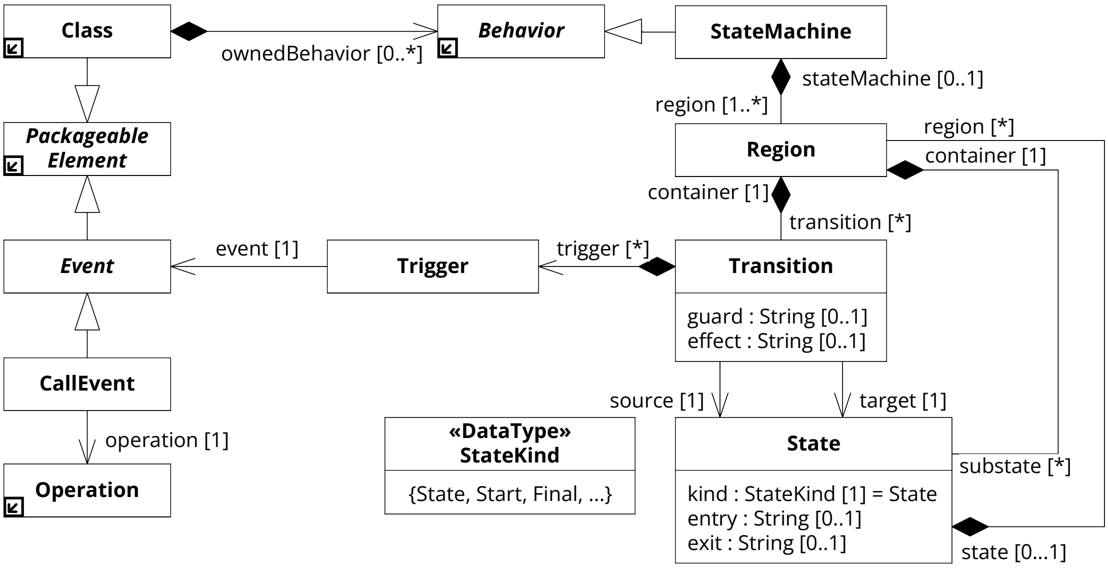


Figure 3.4. Simplified UML meta-model (M2-layer) for specifying state machine diagrams. Extends the ↙ UML class diagram meta-model from Figure 3.2.

concrete syntax of a state machine diagram. In the abstract syntax, we specify the guard and effect by interpretable expressions. The UML [141] actually specifies five different kinds of Events that can activate the Trigger of a transition. The simplified meta-model in Figure 3.4 includes the CallEvent that is triggered by an operation of a class.

3.1.2 Meta-Metamodel (EMOF/Ecore) of the Running Example

The meta-metamodel defined in Figure 3.5 is a simplified version of the Ecore [175] meta-metamodel (see Section 6.1). Ecore is an implementation of the Essential Meta Object Facility (EMOF) [140] that defines the meta-metamodel of the UML [141]. The meta-metamodel in Figure 3.5 allows the definition of a type hierarchy of meta-classes by the meta-metaclass named EClass. The type hierarchy must not contain any cycles. Each meta-class specifies a set of direct supertypes, i.e., the directly inherited meta-classes. For example, the direct supertype of CallEvent in the state machine meta-model in Figure 3.4 is the class Event. Moreover, a meta-class can be an abstract type that cannot be instantiated, e.g., Event is an abstract meta-class.

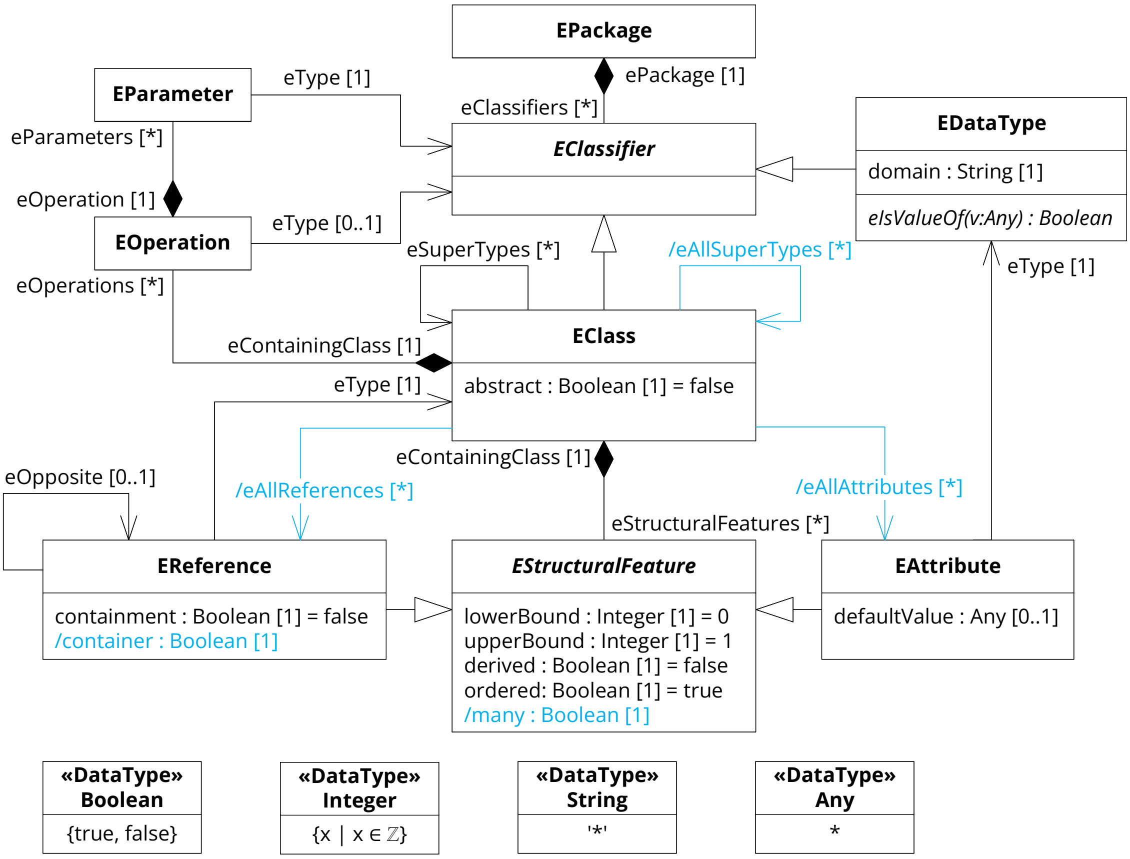


Figure 3.5. Simplified EMOF/Ecore meta-metamodel (M3-layer) for specifying meta-models of a modeling language.

A meta-class contains the specifications of its attributes (EAttributes) that have a primitive data type. A meta-class may also contain parameterizable operations (EOperation). Moreover, the meta-class that contains a reference (EReference) is the source of that reference, and the target meta-class is given by its type. Bidirectional references are represented as opposite references between two adjacent classes. Containment references (containment = true) define the AST structure of model instances. The opposite of a containment reference is referred to as a container reference (/container = true).

Attributes, operations, and parameters of meta-classes are typed. Primitive data types and their (informally defined) domains are specified by EDataTypes. In this context, let the data type Any be the base type that can be assigned with any kind of value or object instance. The method *eIsValueOf(v:Any)* checks if a given value *v* is in the domain of the data type.

Structural features, namely, references and attributes, can have multiplicities that are defined by an upper and lower bound value. An attribute with an upper bound greater than one is referred to as a multivalued attribute, i.e., the attribute can be assigned with a list of primitive values. Contrarily, an attribute with an upper bound of one is called a single-valued attribute.

As specified by EStructuralFeature in Figure 3.5, references and attributes can be derived features, i.e., the values or references are computed from the current state of the model's ASG. We assume that the expression which computes a derived feature is defined by a side-effect-free query. Such a query can be formulated using the same query language that is introduced in the following Section 3.2 for defining consistency rules for a meta-model. For example, in Figure 3.5, the /many attribute is derived by the query

$$\text{EStructuralFeature::many} \coloneqq \text{self.upperBound} > 1 \tag{3.1}$$

for a specific reference or attribute (self), the meta-attribute /many indicates a multivalued attribute declaration. Moreover, the derived /container attribute value is *true* if the opposite reference is a containment reference. In terms of meta-classes, the direct and indirect inherited supertypes (/eAllSuperTypes), references (/eAllReferences), and attributes (/eAllAttributes) can be derived. For example, for the meta-class CallEvent in the state machine meta-model in Figure 3.4, we derive Event and PackageableElement as all super types.

3.1.3 Elementary ASG Consistency

The meta-metamodel defines the syntactical constructs that have to be supported by generic modeling tools. A generic modeling tool can be applied to all modeling languages defined by meta-models based on the same meta-metamodel. In this context, the meta-metaclass EObject defined in Figure 3.6 is a central concept. This meta-metaclass is basically bootstrapping the meta-modeling process [33]. Technically, all objects in the ASGs on all meta-modeling layers are instances of EObject, i.e., all meta-classes and meta-metaclasses (implicitly) inherit EObject as their base class.

Figure 3.6 extends the meta-metamodel from Figure 3.5. For the sake of simplicity, we assume that EObject defines an attribute name:String, i.e., each object in any ASG can define a name. Particularly, the meta-metaclass EObject defines *reflective access* to read and modify the entire ASG of a model. The (unmodifiable) reference eClass [1] gives access to the meta-class that defines the type of an object. The meta-class contains all structural features that can be read from the object.

We can access the object's data by calling *eGet()* with the specific attribute or reference type. For the sake of simplicity, we assume that all referenced objects and attribute values are returned as a list, regardless of their upper bound multiplicity. The references can be modified by adding or removing references from the returned list. Notably, exceeding an upper bound of 1 is not allowed. In this context, the operation *eInvoke()* in Figure 3.6 gives

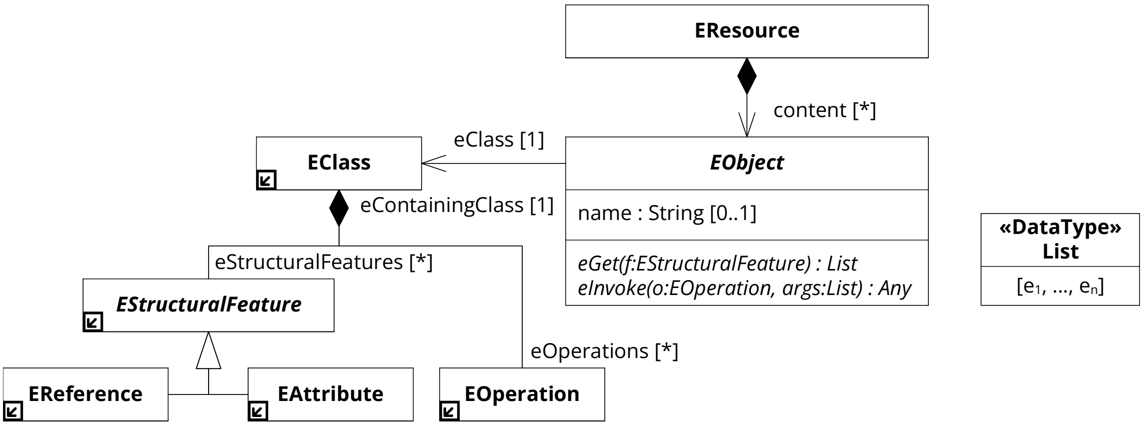


Figure 3.6. Simplified EMOF/Ecore meta-metamodel (M3-layer) for specifying meta-models of a modeling language. Extends the ⇙ EMOF/Ecore meta-model from Figure 3.5.

reflective access to a parameterized operations of an object. For example, using reflective access the object diagram representation of ASGs introduced in Figure 3.1 could be drawn generically for any model.

As already introduced, for (non-reflective) model access, we use the objs.typeA.typeB notation. Similar to the reflective model access, we assume that such a navigation expression always returns a set of objects or values. In particular, we allow such navigation expressions also on sets of objects, i.e., for a set of objects objs all referenced objects or values for typeA are collected in a new flat set. Therefore, we can also chain multiple navigation expressions, i.e., starting from the resulting set objects objs.typeA, all referenced objects or values of typeB are collected. Notably, an unset reference or value with an upper bound of 1 is returned as an empty set $\emptyset$.

For simplicity, in comparison expression ($\equiv$, $\leq$, etc.), if a value or object is compared to a singleton set, the singleton set is compared by its contained object or value. In this context, we will note the size of set c by the expression |c|. As a convention, we will use the $\equiv$ operator for equality checks of objects, values, or sets and the = operator for assignments. In the context of equality checks ($\equiv$, $\not\equiv$) of objects, we assume objects are compared by their identity, not their content.

For all models, we assume some basic structural characteristics of a model's ASG. In the following, we refer to these characteristics as *elementary consistency*, i.e., a minimal level of consistency of the ASG:

Basic well-formedness: In this context, we ignore cases in which a model is "physically damaged", e.g., a reference that can not be resolved to an object after loading the model from a file. We require proper typing of the objects, attribute values, and references in an ASG with respect to the corresponding meta-model.

AST structure: We assume the objects of an ASG are structured in a containment hierarchy referred to as AST. Thus, an object *exists* in a model if the object is contained in the AST. The common root model element of all models is defined in Figure 3.6 by the class EResource. In particular, we represent the *empty model* by an empty instance of EResource with no content.

References: An ASG cannot contain parallel references of the same reference type, i.e., two references are parallel if they share the same source and target object. Bidirectional references that are represented as pairs of opposite references are handled as atomic fragments that are always created or removed together.

Multiplicities: Regarding multiplicity constraints, we will only require that an upper bound that is exactly one is not succeeded by references or attributes in an ASG. In particular, lower bounds and upper bounds greater than one will *not* be required as elementary consistency constraints. In general, multiplicities can be formulated as consistency rules, as shown in the following Section 3.2.

3.2 Specification of Consistency Rules

Most modeling languages require more advanced constraints, which further restrict the valid instance structures of an ASG. Such consistency constraints are typically expressed in a rule-based manner. Consistency rules are formulated using a dedicated constraint language, typically based on first-order logic such as the Object Constraint Language (OCL) [189]. A consistency rule consists of a *context type* and a logical expression that must be valid on all ASG elements of this type or any of its subtypes. We refer to such an element as the *context element of a validation.* A model that satisfies all consistency rules is called syntactically valid or consistent. The violation of a consistency rule indicates an inconsistency with respect to the specified context.

As an example, Definition (3.2) shows a formal specification of a consistency rule named *dangling_transition(t:Transition)* that validates transitions of state machines. This consistency rule explicitly implements the multiplicity constraints defined in the state machine metamodel in Figure 3.4, i.e., a transition must always have a source and target state. The context element of the validation is an object *t* of type *Transition*. The Boolean expression that follows the context definition must be evaluated to *true* to pass the validation. An inconsistency is present if the expression is evaluated to *false.* The expression to be validated in Definition (3.2) is a conjunction that checks the source and the target of a transition. In general, a constraint language defines a set of functions for evaluating the structure and attribute values of a model's ASG. The validation expression of *dangling_transition(t:Transition)* first navigates from the transition *t* to find the source and target state. In both cases, the number of existing source and target references is determined, which must be exactly one.

$$\textit{dangling\_transition(t:Transition)} \coloneqq |\mathsf{t.source}| \equiv 1 \land |\mathsf{t.target}| \equiv 1 \tag{3.2}$$

Figure 3.7 shows an excerpt of our exemplary VoD-System model, i.e., Version 2 shown in Figure 2.2. The excerpt of the concrete state machine diagram is shown alongside the excerpt of the ASG. In this version, the new transition play is validated against the consistency rule *dangling_transition(t:Transition)* defined in Figure 3.2. As illustrated in the ASG, the transition can be successfully validated by checking the existence of the source and target reference.

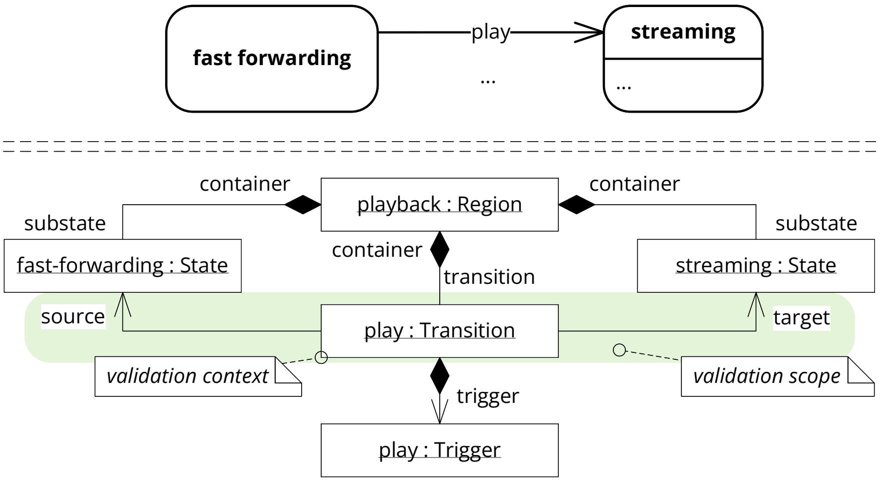


Figure 3.7. Excerpt of the VoD-System shown in Version 2 in Figure 2.2. The excerpt of the concrete state machine diagram (on the top) is shown alongside the excerpt of the ASG (on the bottom).

Generally, the origin of the inconsistency that may cause an evaluation of a consistency rule to fail is not necessarily the context element itself, i.e., a consistency rule can evaluate a larger model fragment starting from the context element. The set of model elements, references, and attributes of the ASG that are inspected during the evaluation of the consistency rule is referred to as *validation scope* [149]. The area in Figure 3.7 highlighted in green indicates the scope of the validation on context element play for the consistency rule *dangling_transition(t:Transition)*, i.e., the elements accessed during the evaluation of the consistency rule's Boolean expression. The scope of a validation can help the developer to understand the inconsistency detected by a violated consistency rule. As shown by Reder et al. [150], only modifications of the ASG that are applied within the validation scope can *potentially impact* the result of the consistency rule. For example, modifying the trigger of the play transition cannot impact the result of the validation of consistency rule *dangling_transition(t:Transition)*. Thus, a change in the scope of a validated consistency rule can potentially lead to a violation of this rule. Conversely, a change in the validation scope of a violated consistency rule may potentially resolve an inconsistency.

$$\begin{aligned}&\textit{references\_multiplicities(o:EObject)} := \\ &\quad \text{o.eClass.eAllReferences} \rightarrow \forall\, \text{r} \mid \text{r.upperBound} \geq |\text{o.eGet(r)}|\end{aligned} \tag{3.3}$$

In general, multiplicities defined in a meta-model can be handled as a constraint of the meta-metaclass EObject shown in Figure 3.6. As a result, the constraint is (implicitly) inherited by all model elements. As defined by the consistency rule *references_multiplicities(o:EObject)* in Figure 3.3, such a generic multiplicity constraint can be implemented by using reflective access (see Section 3.1.3). The consistency rule checks all types of references of an object by comparing the actual number of references with the upper bound specified in the corresponding meta-class.

In Chapter 5, we will introduce additional functions of the constraint language and further consistency rules, e.g., the rules *message_signature(m:Message)* and *message_property(m:Mes-*

sage) of our modeling scenarios from Section 2.1. In this context, Section 5.1 and Section 5.2 discuss potential positive and negative impacts with respect to changes in the validation scope of consistency rule's evaluation in detail.

3.3 Specification of Model Changes

During modeling, e.g., in a model's diagram editor, we need to specify possible modifications that can be instantiated and applied to the model. Such modifications can be generically specified for all modeling languages using the abstract syntax of a model. In this section, we will define different kinds of change actions as the most elementary way of expressing modifications in models, i.e., they cannot be split into smaller actions. Therefore, we will also refer to such change actions and their concrete instantiations as *elementary changes* in models.

3.3.1 Instantiation of Change Actions

ASGs of models basically consist of objects, references, and attributes. Objects and references can be created or deleted; attributes can be modified. These modifications lead to five kinds of elementary changes that can be applied on an ASG. Such elementary changes must conform to the types defined by the meta-model of the modeling language.

As illustrated in Figure 3.8, a *change action* specifies a template of a modification that can be initialized with specific model elements to form a *concrete change*. In this context, a partially instantiated change action is referred to as *abstract change*. For binding model elements to typed parameters, we use the following assignability test to check if the first given type is equal to or a more concrete subtype of the second given type.

$$\begin{aligned}\mathit{isAssignableTo(st{:}EClass,\ t{:}EClass)} \coloneqq\ & \mathrm{st} \equiv \mathrm{t} \\ & \vee\ \mathrm{st.eAllSuperTypes} \rightarrow \exists\ \mathrm{iht} \mid \mathrm{iht} \equiv \mathrm{t}\end{aligned} \tag{3.4}$$

Change actions. Change actions specify the parameters of a modification without binding those parameters to concrete model elements. The parameters defined by a change action must conform to the meta-model of a modeling language. An *object change action* specifies the creation or deletion (ChangeAction::action) of an object of a specific type (ChangeAction::contextType).

A *reference change action* specifies the creation or deletion of a reference of a specific type (ReferenceChangeAction::type). In general, a reference type is specified by an EReference in the meta-model, including its source (EReference::eContainingClass) and target (EReference::eType type (see Figure 3.5). Moreover, the type of the source (ChangeAction::contextType) and target (ReferenceChangeAction::targetType) object of a reference change action can be further restricted to corresponding subtypes.

$$\begin{aligned}&\mathit{restrict\_source\_and\_target(rca{:}ReferenceChangeAction)} \coloneqq \\ &\quad \mathrm{isAssignableTo(rca.contextType,\ rca.type.eContainingClass)} \\ &\quad \wedge\ \mathrm{isAssignableTo(rca.targetType,\ rca.type.eType)}\end{aligned} \tag{3.5}$$

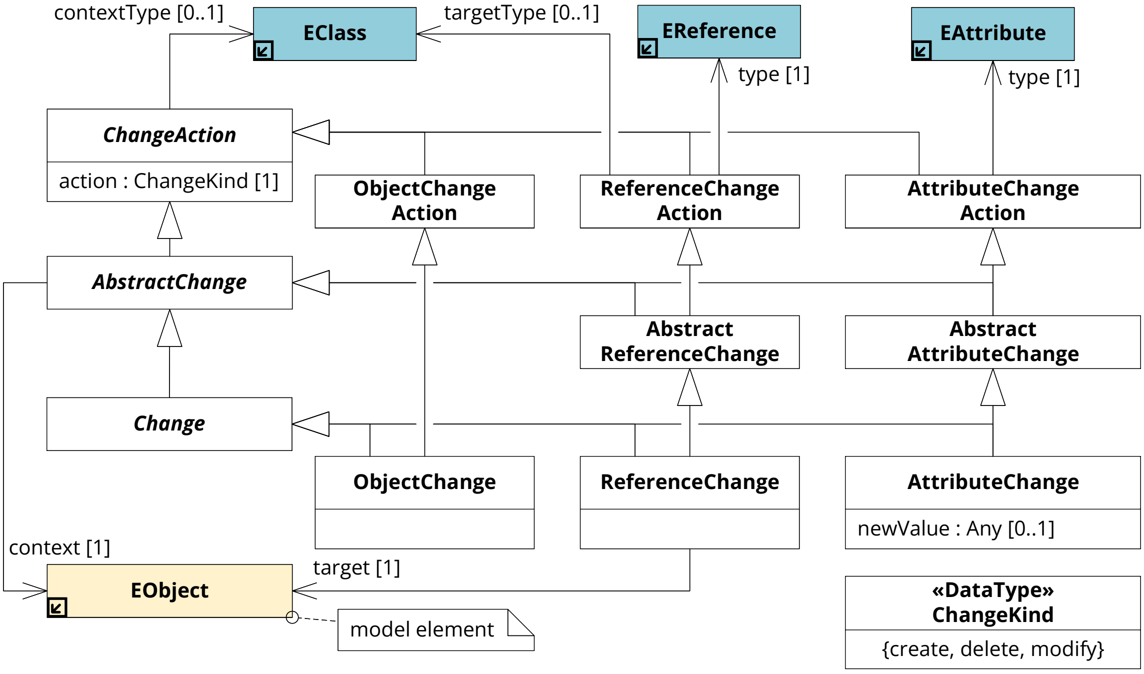


Figure 3.8. Meta-model that defines the refinement from change actions and abstract changes to concrete changes. Uses the ↙ EMOF/Ecore meta-model in Figure 3.5 and Figure 3.6.

Similarly, an *attribute change action* is specified by an attribute type (AttributeChangeAction::type) of the meta-model. In general, we will consider the changing or initializing of an attribute value as modifying change action (action = modify). A more concrete subtype of the containing object can be specified by the context type (ChangeAction::contextType).

$$\begin{aligned}&\mathit{restrict\_container(aca{:}AttributeChangeAction)} \ := \\ &\quad \mathsf{isAssignableTo(aca.contextType,\ aca.type.eContainingClass)}\end{aligned} \tag{3.6}$$

Abstract changes. Abstract changes are partially bound change actions, i.e., some parameters of the abstract change still need to be defined. An abstract change binds the context (AbstractChange::context) of a change action to a concrete object of the model's ASG. In terms of *abstract reference changes*, the source object of the reference is specified by the context model element. Similarly, the context of an *abstract attribute change* specifies the model element that contains the attribute to be modified. A bound model element must be assignable to the specified context type.

$$\mathit{context\_binding(ac{:}AbstractChange)} \ := \ \mathsf{isAssignableTo(ac.context.eClass,\ ac.contextType)} \tag{3.7}$$

Concrete changes. A concrete change, or *change* for short, binds all parameters of a change action or of an abstract change to concrete model elements. Therefore, the context model element (AbstractChange::context) of a concrete *object change* refers to the concrete element being created or deleted.

Regarding a concrete attribute change, the concrete value to be set for the context model element is specified. The assigned value must match the data type specified by the attribute in the meta-model.

$$\textit{value\_binding(ac:AttributeChange)} := \text{ac.type.eType.eIsValueOf(ac.newValue)} \tag{3.8}$$

Similarly, a concrete *reference change* binds a model element that specifies the target (ReferenceChange::target) of the reference. A bound model element must be assignable to the specified target type.

$$\textit{target\_binding(rc:ReferenceChange)} := \text{isAssignableTo(rc.target.eClass, rc.targetType)} \tag{3.9}$$

3.3.2 Unified Graph Representation

This thesis will discuss structural changes in models utilizing a compact representation referred to as unified graphs. A unified graph illustrates elementary changes using an annotated variant of an object diagram, similar to the notation of the model transformation tool Henshin [3] (see Section 3.5.3). Structurally, the unified graph is the union of two ASGs that must be typed according to a specific meta-model. The elements of a unified graph can represent either concrete elements of an ASG or unbound parameters of change actions. Therefore, a unified graph can express change actions, abstract changes, and concrete changes. Moreover, the structural relations between change actions are declared by the graph.

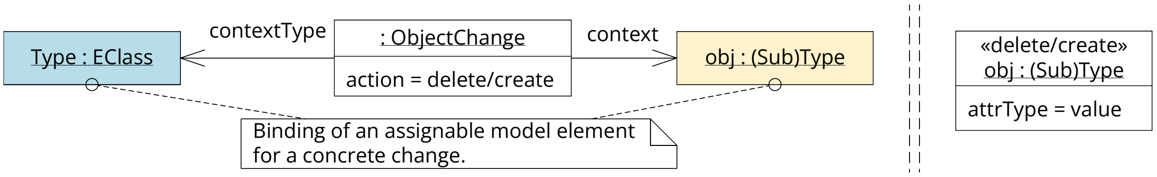


Figure 3.9. The mapping between an object change (on the left side) and the corresponding unified graph representation (on the right side).

Figure 3.9 illustrates a concrete object change (on the left side) and the corresponding unified graph notation (on the right side). The action kind «delete» or «create» of the object change is annotated accordingly on the model element in the unified graph. Initializations of attribute values for a newly created object are not explicitly annotated with actions, i.e., the attributes are initialized as part of the creation of an object.

In the case that no context object is bound, the node of the unified graph represents an object change action. In the diagram notation of the unified graph, concrete elements of an ASG are depicted by underlining the object's name and type, i.e., otherwise, the node can represent unbound parameters of (abstract) change actions.

In Figure 3.10, a concrete reference change (on the left side) is illustrated by an accordingly annotated («create» or «delete») edge in the unified graph (on the right side). The context or target of the reference change may be unbound to represent an abstract change or a change action, respectively.

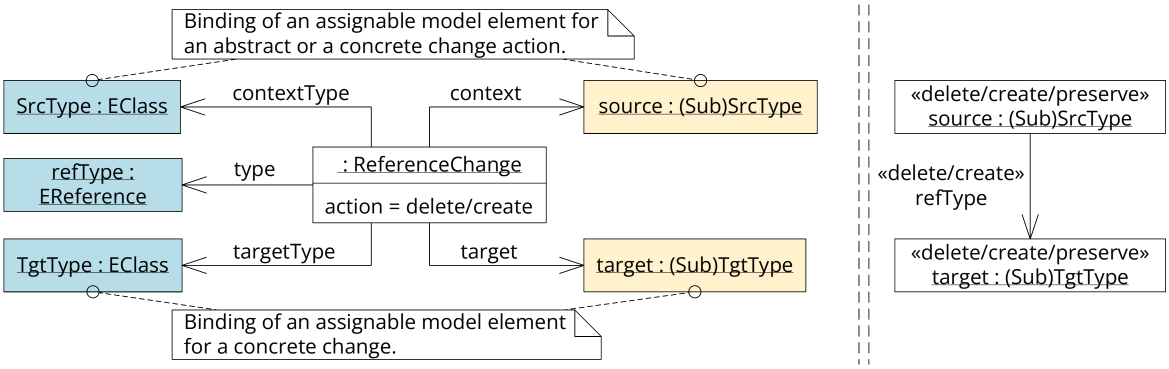

Binding of an assignable model element for an abstract or a concrete change action.
SrcType : EClass
contextType
context
source : (Sub)SrcType
refType : EReference
type
: ReferenceChange
action = delete/create
TgtType : EClass
targetType
target
target : (Sub)TgtType
Binding of an assignable model element for a concrete change.
«delete/create/preserve» source : (Sub)SrcType
«delete/create» refType
«delete/create/preserve» target : (Sub)TgtType

Figure 3.10. The mapping between a reference change (on the left side) and the corresponding representation (on the right side).

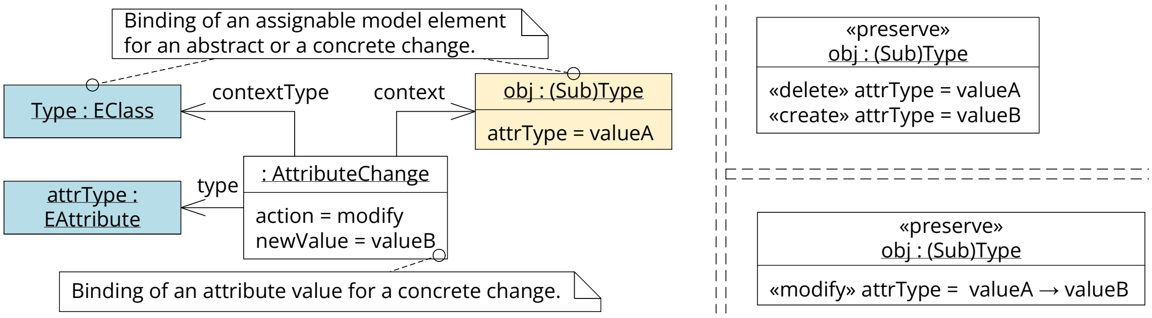

Binding of an assignable model element for an abstract or a concrete change.
Type : EClass
contextType
context
obj : (Sub)Type
attrType = valueA
attrType : EAttribute
type
: AttributeChange
action = modify
newValue = valueB
Binding of an attribute value for a concrete change.
«preserve» obj : (Sub)Type
«delete» attrType = valueA
«create» attrType = valueB
«preserve» obj : (Sub)Type
«modify» attrType = valueA → valueB

Figure 3.11. The mapping between an attribute change (on the left side) and the corresponding representation (on the right side).

Figure 3.11 illustrates the notation of an attribute change in the unified graph (on the right side). The attribute change (on the left side) is noted in the unified graph by two annotated attributes of the same type. The first attribute showing the old value is annotated with a «delete» action. The attribute of the new value is annotated with a «create» action. Technically, such an attribute value change is overwriting the value valueA with valueB. Therefore, an attribute value change is handled as a single atomic modification, which can also be noted as an expression «modify» attrType = valueA → valueB. In the case of an attribute change action, the new valueB and, optionally, the old valueA represent variables that must be bound to concrete values.

Technically, the approach implemented in our tool ReVision handles structural changes without an explicit transformation into the unified graph notation, i.e., the unified graph can practically be handled as a data view representation. However, within the following definitions of our repair approach, we will represent unified graphs as instances of the meta-model defined in Figure 3.12. Basically, this meta-model defines a data structure for typed, attributed graphs with annotations. In general, we will refer to nodes, attributes, and edges in such a unified graph as graph elements.

The types of graph elements are defined by the corresponding meta-model of the modeling language. This allows us to represent a complete ASG of a model, i.e., also unchanged model elements, as a unified graph. Each object, reference, and attribute of an ASG is rep-

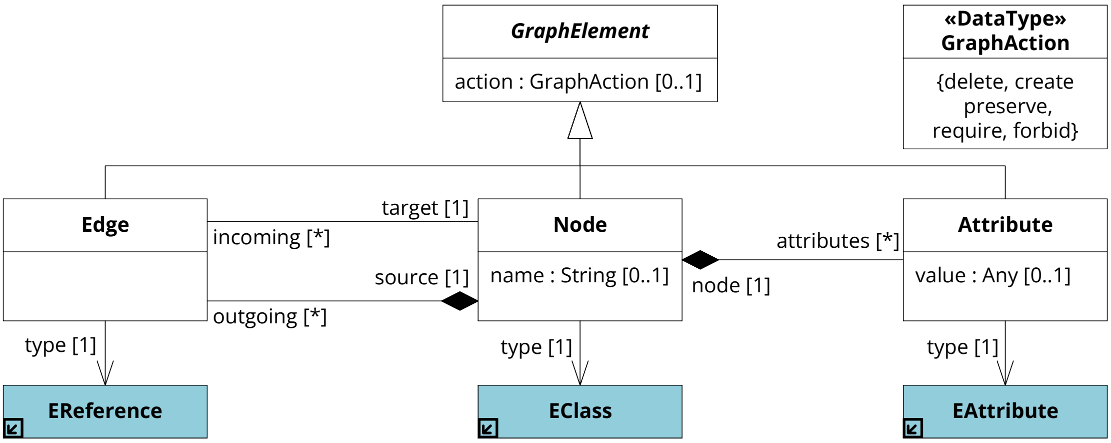


Figure 3.12. Meta-model that defines the abstract syntax of unified graphs. Uses the ↙ EMOF/Ecore meta-metamodel from Figure 3.5.

resented by a corresponding node, edge, and attribute in the unified graph.

All graph elements of a unified graph can be annotated with graph actions. Change actions are annotated accordingly with «create» and «delete» actions. In the following, speaking of changes or change actions in a unified graph, we refer to nodes, edges, or attributes annotated with «delete» or «create» actions. Model elements that remain unchanged are annotated with the «preserve» action. In addition, as we will discuss in Section 3.5, a unified graph can also define structural restrictions by adding graph constraints that forbid («forbid») or require («require») certain elements.

3.3.3 Elementary Change (Action) Dependencies

Given a set C consisting of elementary change actions, some change actions $ca_n \in C$ can have mutual dependencies with respect to the elementary ASG consistency defined in Section 3.1.3, i.e., these change actions have to be applied in a single edit step to not introduce intermediate elementary inconsistencies. We will refer to such a minimal edit step as *atomic change set* A. In general, dependencies of change actions also apply to all their possible concrete change initializations. In the following dependency definitions, we only refer to change actions for the sake of simplicity. A set of change actions C can be partitioned according to the following types of non-overlapping atomic change sets $A_i \subseteq C$:

Atomic opposite reference: An undirected reference in a model's ASG is declared by a pair of references with opposite directions. Such a pair of opposite references must be treated as an atomic fragment, i.e., the references have to be created or deleted together. Two opposite references of type association and memberEnd defining association ends in a class diagram's ASG (see meta-model in Figure 3.2) are an example of such a bidirectional reference.

Atomic containment: As defined in Section 3.1.3, model elements must be structured in a containment hierarchy referred to as AST. Model elements that are created/deleted from the model's AST have to be created/deleted, including their containment references.

It follows that a container reference, which is the opposite reference of a containment reference, is also included in the atomic change set creating/deleting a model element.

Atomic move: If a model element is removed from its containing model element and added to another container, then such a relocation in the model's AST must be handled as an atomic change set. The moving of the operation disconnect() in Version 4 in Figure 2.4a from class Video to Server is an example of such an atomic move.

Atomic model element attribute initializations: The creation of a model element in an ASG including the initialization of its attribute values is treated as an atomic change set.

Assuming that all non-dependent changes in C are contained in singleton atomic change sets $A_i \subseteq C$, a *dependency graph* can be computed in which nodes represent atomic change sets, and edges are pointing from atomic change set A_i to all dependent atomic change sets A_k that can only be executed subsequently. The edges of the dependency graph are constructed by checking for the following kinds of dependencies between the change actions $ca_n \in A_i$ and $ca_m \in A_k$ in the atomic change sets:

Elementary creation dependency: Generally, an elementary creation dependency exists if a model element e should be created together with an incident reference r, i.e., the model element has to be created before the incoming and outgoing references. Thus, if the atomic change set A_i creates the reference r, then it depends on the creation of the model element e in atomic change set A_k.

Following the elementary creation dependencies of containment references in the ASG of a model, the corresponding AST is created from the root element to the child elements. For example, according to state-machine meta-model in Figure 3.4, a new triggered transition is created by first creating the transition and second the trigger as a contained child model element.

Elementary deletion dependency: An elementary deletion dependency is the opposite of an elementary creation dependency. If an atomic change set A_i that deletes a model element e, then it depends on all change actions $ca \in A_k$ that specify the deletion of references in an ASG being incident to model element e. The deletion of a model element from an ASG without its incident references would cause so-called dangling references, i.e., references without connected source or target ends are not allowed in the ASG of a model.

Elementary replacement dependency: The replacement of a single-valued reference (with an upper bound multiplicity of 1) has to be executed in a distinct order. Such a reference can only be replaced with a new one by first removing the old one. For example, we can assume such a case in a modification that changes the target end of transitions from one state to another (see Transition in meta-model shown in Figure 3.4).

3.4 Structural Model Differences

An ASG represents the state of a model at a specific time, e.g., the current state in a model editor. During a model's evolution, it can have a variety of states, some of which need to be shared between the developers. We will refer to such ASG states as model versions. Similar to source code, those model versions are typically maintained and shared in a version control system. Conceptionally, a model history can be understood as a sequence of model versions in which the model is changed gradually from one version to the next. Typically, this is a sequence of model versions maintained in a version control system, including the currently edited model version in the local workspace of a developer. Picking any two model versions of such a model history, we will refer to the older model version as V_A and the newer model version as V_B.

Our repair approach uses the historical model changes that can be observed in a model's development history to detect and complement incomplete edits. To extract the changes from the model history, we compare two successive versions V_A and V_B and compute a structural difference.

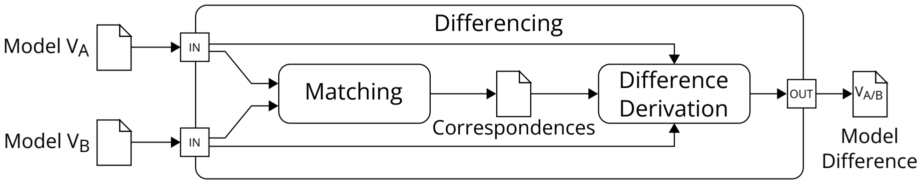


Figure 3.13. Process for calculating a model difference from two input model versions V_A and V_B.

Figure 3.13 shows the typical processing steps of a model differencing calculation. As it is customary for state-based model differencing, a structural model difference is calculated in a pipeline architecture [78]. First, the corresponding model elements in V_A and V_B are determined, then all historical changes are derived based on these correspondences.

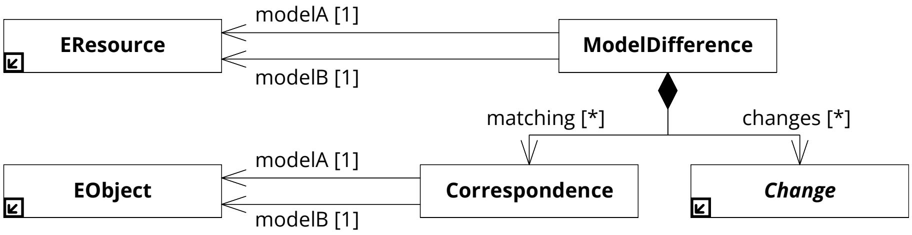


Figure 3.14. A meta-model for model differences. Extends the ↙ change meta-model from Figure 3.8. Therefore, uses the ↙ EMOF/Ecore meta-metamodel from Figure 3.6.

A structural model difference is represented on the basis of a model's ASG. As defined by the meta-model in Figure 3.14, a ModelDifference $\Delta(V_A, V_B)$ is computed between two model version V_A and V_B. Each model version is represented by a resource containing the ASG of the model version V_A (ModelDifference::modelA) and V_B (ModelDifference::modelB). A model difference contains the corresponding model elements and the changes between the model versions. A Correspondence represents a pair of objects which originate from the

ASG of model V_A (Correspondence::modelA) and V_B (Correspondence::modelB), respectively. A historical Change is an instance of the five kinds of concrete changes defined by the metamodel in Figure 3.8, i.e., the deletion/creation of objects/references and the modification of attribute values. The model difference in Figure 3.15 shows an excerpt of the changes between Version 1 in Figure 2.1 and Version 2 in Figure 2.2 of our exemplary VoD-System model.

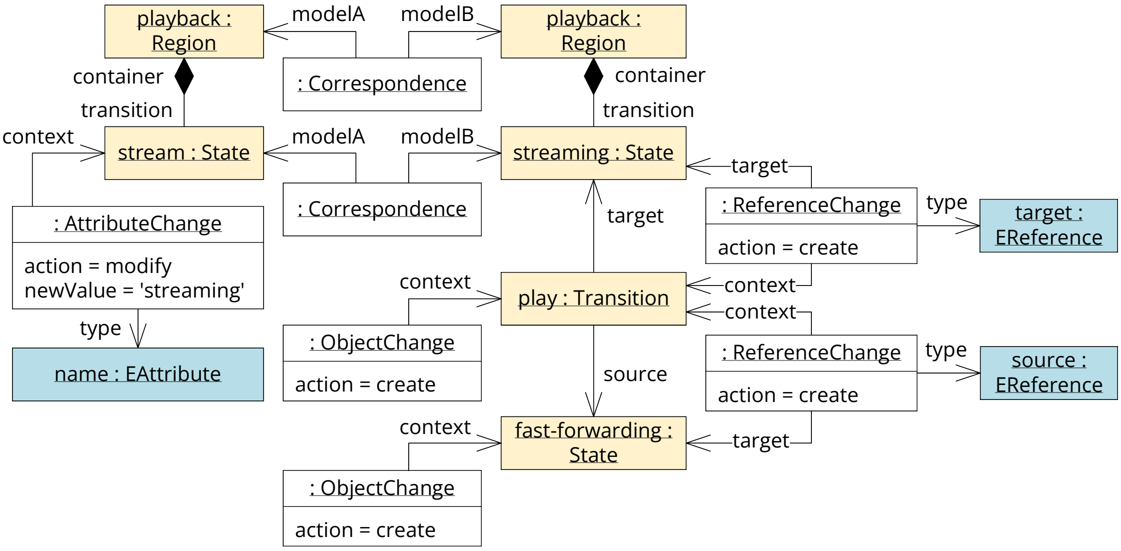


Figure 3.15. Excerpt of the difference between the models shown in Figure 2.1 and Figure 2.2. For the sake of brevity, the correspondences and historical changes are only illustrated partially. For example, the creations of containment references of transitions play and fastForward are not shown.

3.4.1 Matching

During the matching phase, shown in Figure 3.13, the corresponding model elements between the model versions will be determined, i.e., those model elements that have not been changed from version V_A to version V_B. A model element of V_A can only correspond to exactly one model element in V_B. As we do not allow the conversion of an object's type in an ASG, two corresponding model elements have to be of the same type. For example, in the model difference in Figure 3.15 model elements of type Region and State are corresponding.

As described in the adaptable matching pipeline of SiDiff (see Reference [80]), to reduce the complexity of the matching calculation, the first step is to determine corresponding model elements by computing unique signatures for each element in the ASG. Such signatures might be computed specifically for a modeling language, e.g., by determining a kind of unique namespace for model elements. If available, a signature can also be determined by unique identifiers assigned and persisted, e.g., by the model editor, for each model element. The remaining unmatched model elements can be processed by computing similarities between the model elements, e.g., by comparing and weighting the similarity of their attribute values and references. For instance, the correspondence between the states in Figure 3.15 might not be found by a name-based signature, but the name stream and streaming can be matched by measuring the textual similarity. The similarity-based comparison typically depends on a domain-specific configuration for the modeling language.

For each unmatched model element in V_A, a similarity value $[0, 1]$ for all compatible, unmatched model elements in V_B is calculated. The similarity-based comparison results in a ranked list of candidates for the model elements of version V_A. Model elements from version V_B that have a similarity value below a certain threshold are filtered. Finally, a matching strategy (see Reference [80]) must be applied to select one model element from each candidate list to form a correspondence. During the matching, a model element that is selected for a correspondence must be removed from all other candidate lists.

3.4.2 Differencing

During the difference derivation phase shown in Figure 3.13, the changes of the model differences are derived based on the correspondences between V_A and V_B computed by the matching pipeline step. The attributes of all corresponding model elements are compared.

Attribute changes: If the attribute values of two corresponding model elements are not equal, an *attribute change* is generated for the object in V_A to set the new value in V_B. In this context, changes in multivalued attributes are treated as an atomic value change of that attribute. However, initializations of attributes for new objects are handled implicitly as part of an object's creation, i.e., no explicit changes are created.

Object changes: Next, for all unmatched model elements in V_A/V_B an *object change* with a delete/create action is generated.

Reference changes: Finally, the deleted/created references between V_A/V_B need to be derived. Therefore, a reference in V_A must not have a corresponding reference in V_B. References correspond if they have the same type, and their source and target objects correspond. This matching of references is unique as we do not allow duplicated references of the same type between objects (see Section 3.1). For each unmatched reference, a *reference change* is generated accordingly for the model difference.

As illustrated in the model difference in Figure 3.15, the changes annotate the model elements from version V_A and V_B. The new state fast-forwarding and the connecting transition play result in two object creations. In addition, the two reference creations are shown in Figure 3.15, namely, the definition of the source and the target of the play transition. For the sake of brevity, the model difference excerpt in Figure 3.15 does not show the creations of the containment references of the transitions play and fastForward, which includes the new model elements in the AST. Finally, the renaming of the state from "stream" to "streaming" is annotated as an attribute change.

A state-based model differentiation calculation makes it unnecessary to log changes during modeling, e.g., in a model editor. Notably, the derived sequence of changes does not necessarily reflect the original editing sequence performed by a developer or tool.

Deviating correspondences: The matching and its resulting correspondences might deviate from those correspondences in the original editing sequence, leading to another sequence of changes. In particular, these changes applied to model version V_A still lead to the same result, namely, model version V_B. Moreover, similarity-based matching might also discover new correspondences in comparison to the original editing sequence, e.g., if the developer removes and similarly recreates some model elements, leading to a smaller number of changes. Likewise, similarity-based matching may reasonably remove correspondences in comparison to the original editing sequence, e.g., if a developer reuses some model element for purely technical reasons but completely modifies it from a contextual perspective.

Transient effects: A model difference describes the "direct transition" between two model versions V_A and V_B. Changes to objects or references that have been applied and rolled back between V_A and V_B are not included in $\Delta(V_A, V_B)$. Likewise, if attribute values have been overwritten multiple times, $\Delta(V_A, V_B)$ only includes the final attribute value change (assuming the final value in V_B differs from the value in V_A). Such editing sequences are also referred to as *transient effects* [78]. However, missing such transient effects has the advantage that the model difference does not involve unnecessary changes.

3.4.3 Unified Difference Graph

For a compact notation of model differences, we can apply the concept of the unified graph introduced in Section 3.3.2. A structural difference $\Delta(V_A, V_B)$ between an original model V_A and its revised version V_B is conceptually treated as a unified graph over $V_A \cup V_B$, where $V_A \cap V_B$ is defined by pairs of corresponding elements in V_A and V_B. In such a unified graph, in the following referred to as *difference graph* $G_{\Delta(V_A,V_B)}$, corresponding elements just appear once, while all other elements and attribute values that are unique to model V_A and V_B are marked as deleted and created, respectively.

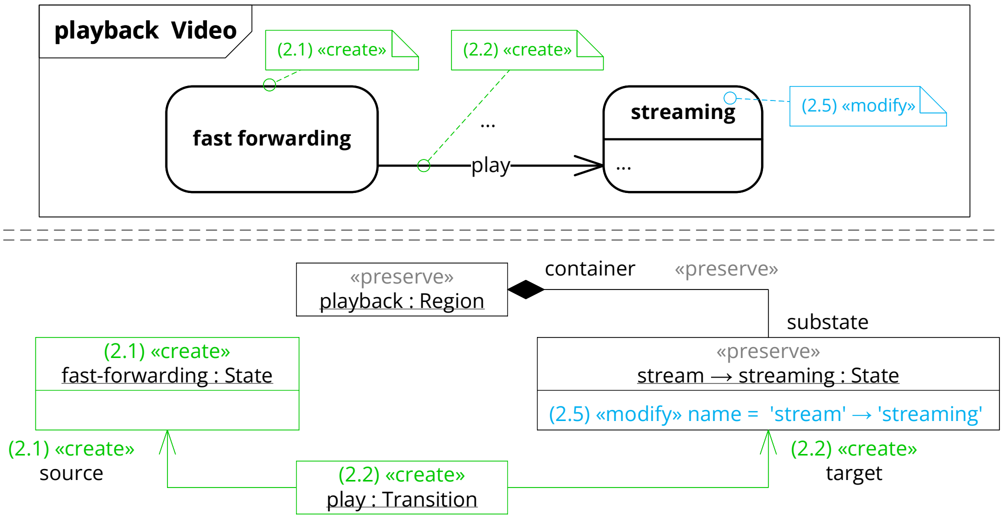


Figure 3.16. Excerpt of the unified graph representing the model difference illustrated in Figure 3.15, i.e., the difference between the model versions shown in Figure 2.2 and Figure 2.3.

The historical changes in a difference graph are represented according to the transformation patterns defined in Section 3.3.2. Figure 3.16 shows an excerpt of the difference graph between the model versions in Figure 2.1 and Figure 2.2 of our running example, i.e., the difference graph $G_{\Delta(V_A,V_B)}$ of the model difference $\Delta(V_A, V_B)$ in Figure 3.15. The upper part of Figure 3.16 depicts the excerpt of our running example in concrete diagram syntax, i.e., the resulting version of the state machine diagram. The lower part of Figure 3.16 illustrates the corresponding difference graph. As discussed in Section 3.4, a new state fast-forwarding is created, including a new transition play connected to the streaming state. The renaming of the state is illustrated using the compact notation stream → streaming instead of «create» and «delete» annotated attributes. For the sake of readability, the changes are numbered and color-coded by their kind of action.

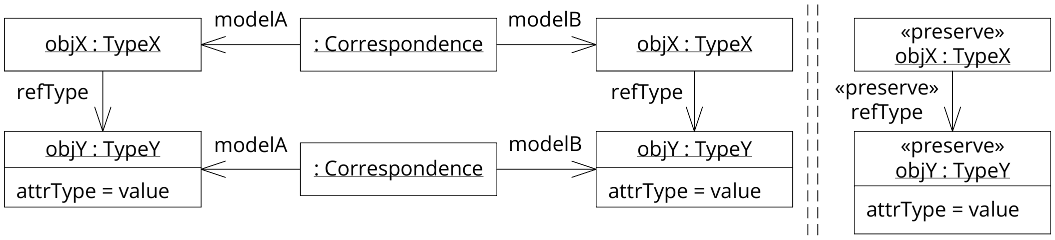


Figure 3.17. The mapping between two corresponding objects (on the left side) and the difference graph representation (on the right side).

In addition to the representation of changes in Section 3.3.2, the unchanged elements in the difference graph $G_{\Delta(V_A,V_B)}$ are derived from the model elements that correspond in the model difference $\Delta(V_A, V_B)$. The left side of the transformation pattern in Figure 3.17 shows two correspondences of objects in a model difference. The objects are connected by the same type of reference in both model versions, i.e., these references implicitly correspond to each other. On the right side of Figure 3.17 the resulting unified difference graph is illustrated. The nodes and edges of the corresponding model elements are annotated with «preserve» action. Moreover, the unchanged attribute and its value is shown once (without «preserve» annotation) for the corresponding objects. As a result, in terms of our running example, the difference graph in Figure 3.3.2 shows the corresponding state machine region and the renamed stream → streaming state as annotated «preserve» nodes.

3.5 Specification of Edit Operations

Change actions, as defined in Section 3.3.1, are the most elementary way of defining modifications in models. More complex modifications of a modeling language can be defined by *edit operations* combining multiple elementary change actions, e.g., edit operations can define the modifications that are allowed in a model editor. For example, an edit operation can specify the creation of a transition in a state machine diagram, including their source and target state. The modifications described by an edit operation can be specified as a set of change actions $EO = \{ca_0, \dots, ca_m\}$. As defined in Section 3.3.1, the change actions ca_n specify the elementary modification in the ASG of a model. This set-based specification of EOs does not restrict the structural relations of the contained change actions.

Initially, in the following Section 3.5.1, we define our notion of edit operations with respect to the consistency of a modeling language. The formal and structural specification of edit operations as edit rules will be discussed in the Section 3.5.2 and Section 3.5.3.

3.5.1 Consistency-Preserving Edit Operations (CPEOs)

Specialized edit operations can support the developer while editing complex model fragments. For example, the transition fastForward in the state machine and its triggering operation in the class diagram in Figure 2.2 could be created by a single complex edit operation. Although not typically offered by standard model editors, an additional recommender tool can offer such complex edit operations to speed up the developer's workflow. Thus, the developer does not have to create and connect several model fragments in different views, with the risk of missing some parts and possibly violating the model's consistency. Inconsistencies caused by incomplete edit steps, as in our motivating examples in Figure 2.4, can be avoided by using more complex edit operations that may synchronize isolated editings from one modeling view to another. For instance, moving the operation disconnect() and changing the target end of message 6:disconnect could be performed in a single edit step without violating the model's consistency. In general, we assume that a complex edit operation prevents the violation of a subset of the consistency rules of a modeling language. Therefore, we refer to these edit operations as *consistency-preserving edit operations* (CPEOs).

A CPEO may also preserve all consistency rules of a modeling language. However, allowing only fully consistent editing steps forces the developer to execute edit rules in a specific order. Editors implementing such a so-called syntax-directed editing process suffer from several usability issues in practice. As noted by Khwaja et al. [87], the edit operations of a syntax-directed editor that strictly operates according to the underlying syntactic structure of language can be in conflict with the edit operations expected by a developer, leading to confusion and frustration during editing. According to Welsh et al. [191], an editor should not require that the syntax of a document be correct before or after applying a certain edit operation. However, it might be useful to impose some practical constraints on edit operations to benefit from editors designed for a specific language [191].

In terms of CPEOs, such practical constraints might limit possible parameter binding of the edit operation for a simpler application of the operation by the developer. Thus, the CPEO should be designed carefully with respect to the preserved consistency rules. Basically, we assume that a CPEO preserves a specific consistency rule of the modeling language and the elementary consistency that is implied by the model fragment to be edited.

CPEOs are the main configuration input of our recommendation tool ReVision in order to adapt its behavior to a given modeling language. Notably, ReVision does not require an explicit mapping between CPEOs and consistency rules, i.e., the relevance of a CPEO during a recommendation is determined on case-specific criteria. Moreover, we do not make explicit assumptions about the CPEOs given as configuration input for a specific modeling language. In principle, one can construct arbitrarily many complex CPEOs. The most important criterion for selecting a CPEO is that it avoids typical inconsistencies which occur when models are edited in standard editors of that language. If available, project- or language-specific editing style guides, as well as interviews with domain experts, provide a promising starting point for the development of a set of useful CPEOs. In general, the most interesting kinds of CPEOs for our approach are creations, deletions, and modifications of complex model fragments, potentially ranging over different modeling views. Chapter 7 presents a systematic process for deriving a set of CPEOs from a given meta-model of a modeling language.

3.5.2 Model Transformation Rules

Complex edit operations of a modeling language can be specified by parameterized in-place model transformation rules. Such transformation rules have shown to be well-suited to configure tools in MDE tool suites. Examples of this are modern refactoring tools [14, 128], merge tools [81, 161, 180], evolution analysis tools [56], or slicing tools [4]. Due to their declarativeness and well-defined formal semantics, our tool ReVision uses in-place model transformations [3, 13, 27] based on graph transformation concepts [40, 46] for the specification of edit operations. In the following, we will refer to the specifications of in-place model transformations as *transformation rules*. In particular, such transformation rules can be parameterized. Input parameters can supply model elements and additional values, such as the names or other properties of newly created elements.

Rule Specification

More formally, an edit operation EO is declaratively defined by a parameterized edit rule $ER(p_0, \dots, p_n)$ through a transformation rule r specified by a precondition and postcondition graphs L and R, called the *left-hand side* (LHS) and *right-hand side* (RHS) of the rule. Input parameters can be mapped to nodes and attributes occurring in the LHS L.

$$EO = \{ca_0, \dots, ca_m\} = ER(p_0, \dots, p_n) = r : L \nrightarrow R \tag{3.10}$$

In particular, all graphs that specify the transformation rule are typed attributed graphs, i.e., each node, edge, and attribute of the rule refer to a type that is defined in the meta-model of the modeling language. The graphs are not allowed to contain parallel edges of the same type, i.e., directed edges with the same type, source, and target node.

The transformation is specified by a partial graph morphism $r : L \nrightarrow R$, i.e., a subgraph of L is mapped to a subgraph of R. This partial graph mapping is injective, structure-compatible, and type-preserving, i.e., the mapping is unique, the mapped element must have the same type, and the mapping preserves the graph structure with respect to nodes and incident edges. The mapping is explicitly specified by a set of mapped nodes. Consequently, two edges with the same type are mapped if their source and target nodes are mapped. Similarly, two equally typed attributes are mapped if their containing node is mapped, and they share the same attribute value. In this context, we assume an attribute value is either a concrete value or a variable mapped to a parameter.

Based on the mapping of graph elements, we can specify the semantics of the transformation. The resulting graph of the intersection $L \cap R$ specifies the graph elements to be preserved by the transformation rule. The graph fragments $L \setminus R$ and $R \setminus L$ specify the model elements and references to be deleted and created by the rule, respectively. However, as attributes can not be removed directly, the attributes in $L \setminus R$ are considered as additional preconditions. Finally, attributes in $R \setminus L$ specify the attribute values to be modified or initialized.

Figure 3.18 illustrates the transformation rule *createTransition* using an object diagram notation for the LHS and RHS. The input parameters are mapped to nodes (name:Type) of the LHS by their names, namely, the parameters specifying the *region*, *source*, and *target* of the transition. Besides that, the names of the nodes do not have any effect on the model

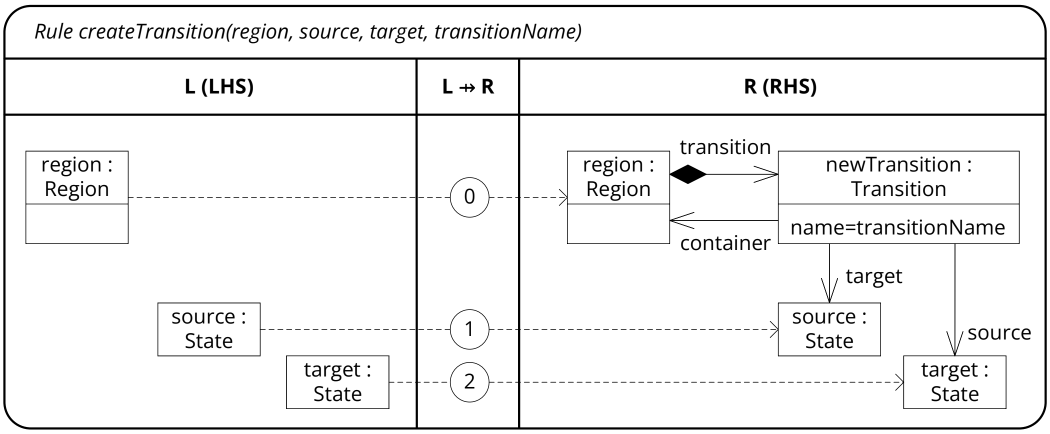


Figure 3.18. Model transformation rule based on graph transformation concepts. Creating a new transition, including its source and target state in the ASG of a state machine diagram.

transformation. In Figure 3.18, the name of the new transition is specified by the *transitionName* parameter and the corresponding variable in the attribute. Thus, the precondition graph L is fully specified by the input parameters. In this rule, the LHS is fully mapped to the RHS. Computing the graph fragment $R \setminus L$ to be created, we get the node newTransition, its incident edges, and the name attribute. The graph fragment $L \setminus R$ to be deleted is empty for the *createTransition* transformation rule.

Application condition. The specification of the transformation rule can be extended with an *application condition*. In general, the application condition of a rule is a Boolean formula *ac* that must be fulfilled in order to apply the transformation to an ASG. A literal in *ac* is an atomic graph constraint C_i. The graph C_i is mapped to the LHS graph L by a partial graph morphism $e_i : L \nrightarrow C_i$. An atomic graph constraint that is checked as a positive literal in the Boolean formula *ac* is referred to as *positive application condition* (PAC). Conversely, an atomic graph constraint that is a negative literal in *ac* is referred to as *negative application condition* (NAC). Let B_i be the intersection $B_i = C_i \cap L$ of the graph constraint with the LHS, which is also referred to as the boundary of the embedded graph. Thus, B_i defines the context with respect to the LHS in which the graph constraint is to be checked. Therefore, the graph fragment $C_i \setminus B_i$ is said to be required by the PAC and forbidden by the NAC.

Nodes of the LHS of a rule or graph conditions can also involve checks on attribute value. Therefore, variables v, which are defined as parameters of the rule, can be assigned to attributes $a = v$. For other attributes $b = v$ assigning the same variable, the equality of the mapped attribute values is required. The value of the variable can be given as input parameter binding or is determined during the application of the rule.

The *dangling condition* is an optional application condition of a transformation rule. Assuming G is the graph to be transformed by the rule r. The dangling condition states that a node of G can only be removed during the execution of a transformation rule if also all incident edges are removed. Otherwise, the deletion of $L \setminus R$ from G would lead to dangling edges without a source or target node. The dangling condition is a consequence of the graph transformation concept defined in the double-pushout (DPO) approach (see References [46,155]).

In contrast, the single-pushout (SPO) graph transformation approach allows dangling edges after the deletion of $L \setminus R$ from G. In the SPO approach, the dangling edges are also removed from G as a side effect of the graph transformation (see References [46, 155]). We assume that a model transformation rule checks the dangling condition if not further specified. Conversely, if the dangling condition is to be ignored, the dangling edges are removed implicitly during the execution of the transformation.

Multi-rules. In addition, a transformation rule can contain nested rules $r_i : L_i \nrightarrow R_i$ that are applied with a universally quantified execution semantic [26, 63]. In general, the rules can be nested in an arbitrary depth. The parent rule to be executed is referred to as *kernel rule*. In particular, the topmost kernel rule $r_k = r$ is the rule itself. Nested rules of a kernel rule are referred to as *multi-rules*. A multi-rule is executed as often as possible for each execution of its kernel rule. Multi-rules are embedded into their kernel rule by a partial graph morphism $e_i : L_k \nrightarrow L_i$ of the LHS L_k of the kernel rule and the LHS L_i of the multi-rule. Similarly, the RHS R_k of the kernel and the RHS R_i of the multi-rule can be mapped $e_i' : R_k \nrightarrow R_i$.

Rule Application

The execution of a model transformation $r : L \nrightarrow R$ is referred to as *rule application*. Given a so-called *working graph* G to be transformed, the first step is to find an occurrence of L in G by mapping the node, attributes, and edges of L to G. The initial context of a rule application can be determined by passing model elements and attribute values from the input parameters to form a partial pre-match $m_0 : L \nrightarrow G$ of the LHS. This partial mapping m_0 is extended to a complete match m by *matching* the graph L as a subgraph in G. In general, the matching of L can result in several matches in G.

A transformation rule $r : L \nrightarrow R$ is *applicable* to G if there is a match $m : L \rightarrow G$, an injective, type-compatible, and structure-compatible mapping that denotes an occurrence of L in G. In an injective mapping, each node of G is mapped at most once to a node in L. A mapping of a node is type-compatible if the type of the object in G is the same or a subtype of the node in L. For edges and attributes, the types must be matched exactly. A mapping is structure-compatible if, for a given mapping of nodes, all edges in L are mapped with respect to their source and target nodes to references in G. In addition, the rule's application condition, if defined, must be fulfilled.

The matching of possible applications of the model transformation rule $r : L \nrightarrow R$ with a given input parameter binding can be computed by the following steps:

(1) Derivation of the partial pre-match $m_0 : L \nrightarrow G$ from the input parameters.

(2) Extending pre-match m_0 to a complete match m by matching L in the ASG G.

(3) Evaluation of the application condition $acc \overset{!}{=} true$ with the graph constraints C_i:

- (3.1) Derivation of the pre-match $m_i : B_i \rightarrow m(B_i)$ with $B_i = C_i \cap L$.
- (3.2) Check if m_i can be extended to a complete match by matching C_i in G.
- (3.3) Evaluation of the graph constraint's existence as NAC or PAC in *ac*.

(4) Collect all possible applications of multi-rules $r_i : L \nrightarrow L_i$ with kernel rule r:

(4.1) Derivation of the pre-match $m_i : B_i \rightarrow m(B_i)$ with $B_i = L_i \cap L$

(4.2) Starting at step (2) with $r = r_i$ and $m_0 = m_i$ and $L = L_i$.

(5) *Optional:* Checking the dangling condition with respect to all node deletions.

For a computed matching, the transformation is finally *executed*, i.e., the changes are applied to the model's ASG. First, if the rule r contains multi-rules r_i, a so-called *amalgamated rule* is constructed by merging a copy of the multi-rule r_i for each match m_i in the kernel rule r and including the corresponding matchings m_i in the matching m of the kernel rule [63]. The initial step of the transformation execution with match m removes the model elements and references $m(L \setminus R)$ from G. In the next step, an instance of $R \setminus L$ is created in G within the context of m. In order to create such an instance, the graph fragment $R \setminus L$ must only contain nodes that refer to concrete (non-abstract) types of the meta-model (see Section 3.1.2). Finally, the attribute values specified by $R \setminus L$ are set in the preserved and newly created model elements in G. In this context, new attribute values of the RHS are inserted that are specified by input parameters $p_0, \dots, p_n$ of the rule.

Model Transformation based on Graph Transformation Concepts

Graph transformation concepts consider G as a graph, and the mapping from L to G is a (total) graph morphism. In contrast, references and attributes in an ASG of a model are parts of objects (see Section 3.1.2). Given an ASG G of a model, an occurrence of L in G is described by mapping the node, attributes, and edges of L to objects, attributes, and references in G, respectively. In particular, references and attributes do not have their own identity and cannot be referenced directly. Assuming the graph of L is an instance of the graph meta-model in Figure 3.12 (without action annotations). Technically, we define $m : L \rightarrow G$ by an explicit, type-compatible mapping of a node $n \in L$ to an object $m(n) \in G$.

$$\begin{aligned} \textit{type\_compatible(n:Node)} \; &:= \; \text{n.type} \equiv \text{m(n).eClass()} \\ &\quad \vee \text{n.type} \in \text{m(n).eClass().eAllSuperTypes()} \end{aligned} \tag{3.11}$$

The edges $e \in L$ are mapped to type- and structure-compatible references with respect to objects matched in m. In particular, the ordering of matched edges and references is not checked, i.e., their position with respect to the containing list in the rule's graph and the model's ASG must not match.

$$\textit{structure\_compatible(e:Edge)} \; := \; \text{m(e.target)} \in \text{m(e.source).eGet(e.type)} \tag{3.12}$$

Similarly to references, the mapping of attributes can be checked. First, all unbound parameters p_i must be determined with respect to the mapping $m(n)$. Therefore, the value of a parameter named x is determined by an attribute a = x with x = m(a.node).eGet(a.type) from the model. As mentioned before, if multiple attributes are assigned to the same variable x, then the determined values must be equal. Generally, assuming the function eval() substitutes variables in attribute values with the bound (input) parameter value, the attribute values must match the values in the model.

$$\textit{value\_compatible(a:Attribute)} \; := \; \text{eval(a.value)} \equiv \text{m(a.node).eGet(a.type)} \tag{3.13}$$

In terms of an edit operation $EO = \{ca_0, \dots, ca_m\}$ defined for a modeling language, we assume that the input parameters of the edit rule $ER(p_0, \dots, p_n) = r : L \to R$ uniquely define the change actions ca_i to be executed (see Definition (3.10)). Therefore, to execute an edit operation, it is sufficient to find the first applicable match of L.

3.5.3 Unified Edit Rule Graph

In the remainder of this thesis, transformation rules based on graph transformation concepts are represented in an integrated form. Figure 3.19 shows the corresponding integrated graph of the transformation rule *createTransition* illustrated in Figure 3.18. The LHS and RHS are merged into a single *edit rule graph*, referred to as G_r, following the visual syntax of the model transformation language HENSHIN [3, 13, 24]. Basically, G_r is the unified graph over $L \cup R$, where mapped elements that are to be preserved just appear once while all other elements are annotated with change actions. The LHS comprises all model elements annotated with «delete» and «preserve» actions, while the RHS contains all model elements annotated with «preserve» and «create» actions. Similar to the transformation rule in Figure 3.18, the rule definition in Figure 3.19 starts with the declaration of its edit rule signature, i.e., the operation name and the required input parameters. In the following discussions about edit operations, we refer to the operation's specification as an *edit rule ER* defined by an edit rule graph G_r.

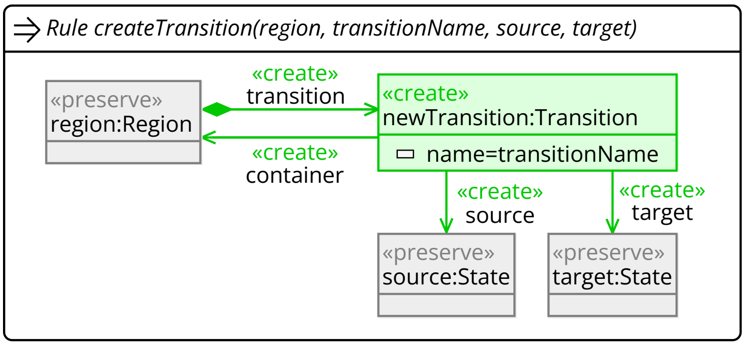


Figure 3.19. Example of a unified edit rule graph that creates a new transition in a state machine and connects it to a source and a target state.

Figure 3.20 shows an extension of the edit operation *createTransition*, namely, the CPEO *createTriggeredTransition*. In addition, this edit rule also creates a new trigger that is connected to an existing event. The edit rule graph of *createTriggeredTransition* in Figure 3.20 also shows the integrated graphs of a NAC and a PAC. The graph elements of the NAC graph constraint are annotated with «forbid» actions. In this case, the NAC forbids that a transition with the same name already exists for the given source state. Similarly, the PAC graph constraint is annotated with «require» actions. In Figure 3.20, the PAC requires that the source state and the new transition are contained in the same region of the state machine. In this case, the PAC graph constraint determines the region in which the transition is created by selecting the source state as an input parameter. Similar to the LHS and RHS, the graph constraints C_i are integrated into the edit rule graph by their mapping to the LHS L, i.e., the extension of the unified edit rule graph with $L \cup C_i$. If not further specified, we assume that the application condition *ac* of the edit rule graph is a simple conjunction of all contained graph constraints.

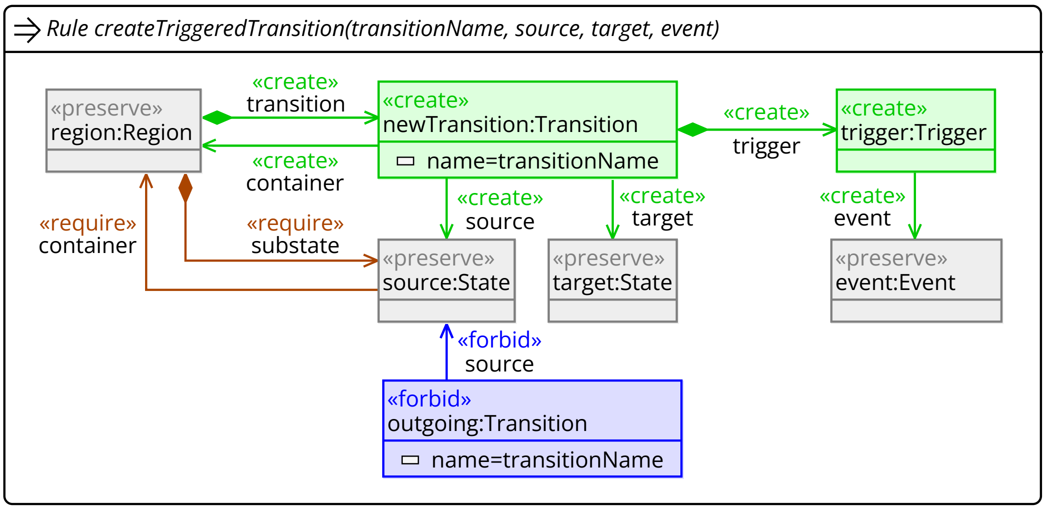


Figure 3.20. Example of a CPEO that creates a complex fragment consisting of a transition between a source and a target state triggered by a specific event.

More formally, we can derive the action of a graph element *e* from a transformation rule by the *action()* function defined in Definition (3.14). In this context, the functions *positive()* and *negative()* determine if a given graph constraint C_i is a positive or negative literal in the application condition *ac* of the transformation rule (see Section 3.5.2).

$$\mathit{action(e{:}GraphElement)} \;:=\; \begin{cases} \text{«preserve»} & \text{if } \mathsf{e} \in \mathsf{L} \cap \mathsf{R} \\ \text{«delete»} & \text{if } \mathsf{e} \in \mathsf{L} \setminus \mathsf{R} \\ \text{«create»} & \text{if } \mathsf{e} \in \mathsf{R} \setminus \mathsf{L} \\ \text{«require»} & \text{if } \mathsf{e} \in \mathsf{C_i} \setminus \mathsf{L} \wedge \mathsf{positive(C_i, ac)} \\ \text{«forbid»} & \text{if } \mathsf{e} \in \mathsf{C_i} \setminus \mathsf{L} \wedge \mathsf{negative(C_i, ac)} \end{cases} \tag{3.14}$$

Figure 3.21 depicts an example of a transformation rule with multi-rules. The edit rule *deleteStateWithTransitions* consists of a kernel rule with two multi-rule, namely, *in* and *out*. The kernel rule removes a state from a state machine diagram, and the multi-rules remove all incoming and outgoing transitions of that state. Annotations of multi-rules are marked by an asterisk (*) added to the actions. For example, the «delete*» and «preserve*» actions in Figure 3.21 are indicating the multi-rules. Nodes of a multi-rule are illustrated as multi-objects known from UML object diagrams [63]. Multi-rules r_i are integrated recursively into the corresponding edit rule graph G_r by the mappings $e_i : L_i \to L_k$ and $e_i' : R_k \nrightarrow R_i$, i.e., they are integrated by their LHS and RHS mapping into the parent kernel rule r_k. The name of a multi-rule is indicated as part of the action annotation, e.g., annotation «preserve*/out» for the multi-rule named *out*. In the rule's header, we can note additional configurations as injective matching, checking for dangling edges and the application condition *ac* as set *{injective_matching = true/false, check_dangling = true/false, ac = ...}*. The edit rule *deleteStateWithTransitions* is configured for a non-injective matching of the LHS. This allows matching the

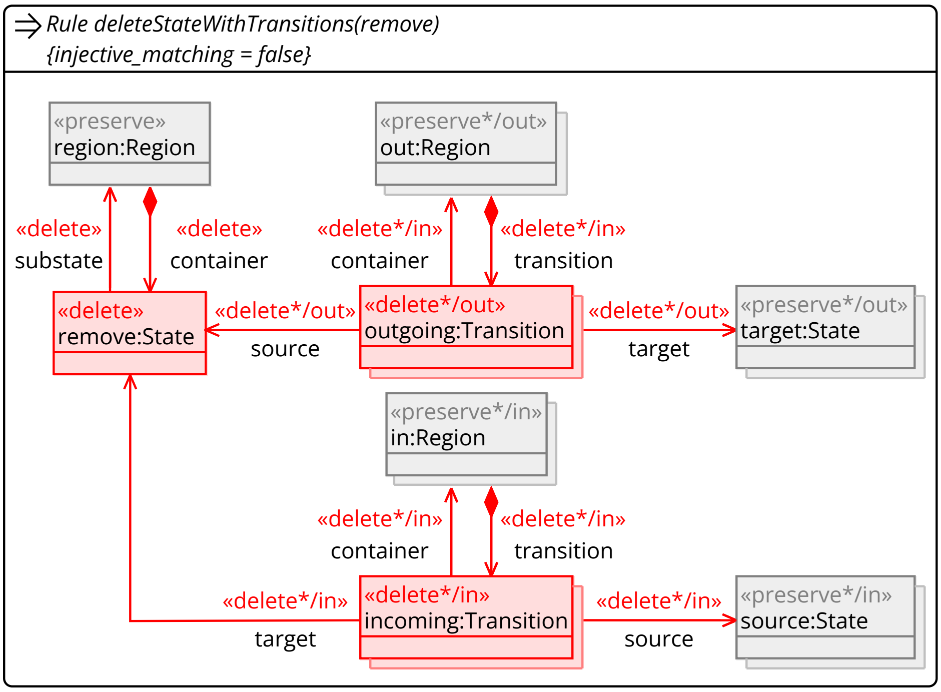


Figure 3.21. CPEO *deleteStateWithTransitions* with multi-rules that delete a state, including all incoming and outgoing transitions. The rule is configured for a non-injective matching.

states and transitions regardless of the containing region, i.e., the region may be the same or a different one in the kernel and multi-rules.

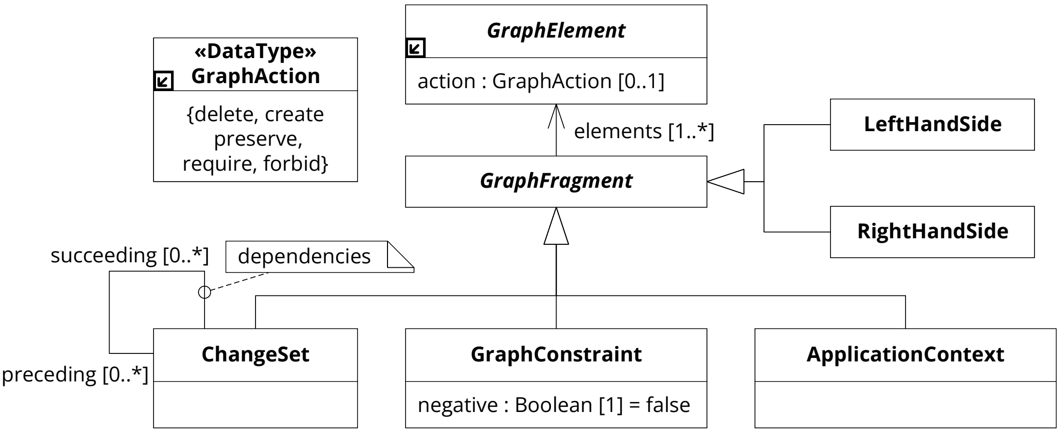


Figure 3.22. Extends the ↙ unified graph meta-model from Figure 3.12 with graph fragments.

We describe instances of edit rule graphs by the meta-model for unified graphs in Figure 3.12. The extension of the graph meta-model in Figure 3.22 allows the edit rule graph to be structured into GraphFragments. Technically, the graph fragments are allowed to overlap with respect to the contained graph elements. The *left-hand side* graph fragment contains the graph elements with «preserve» and «delete» actions. Similarly, the *right-hand side* graph fragment contains the graph elements with «preserve» and «create» actions. A *change set* contains graph elements annotated with «create» or «delete» actions. In addition, a change set can define execution dependencies (see Section 3.3.3) by referencing the preceding change sets it depends on and the dependent succeeding change sets. NACs or PACs of the application condition are represented by *graph constraint* graph fragments. We will refer to the graph annotated with «preserve» actions as *application context* of the edit rule, i.e., the context in the model for applying the transformation.

4

Complementation of Partially Executed Edit Operations

In this chapter, we discuss the algorithm for the detection and complementation of partially executed edit operations in a model difference. The algorithm is introduced in terms of a constraint satisfaction problem (CSP) and the implementation of its solver. Partially executed edit operations are detected by mapping the change actions of an edit operation to historical changes of a model difference. For each detection, the edit operation is split into the already executed sub-rule and a complement rule containing the remaining changes. Finally, possible parameter bindings are computed for the complement rule. As an alternative to complementing partially executed edits, the changes of a recognized sub-rule can also be inverted to compute a case-specific rollback operation.

Models are the main subject within the continuous evolutionary process of MDE. As new features are requested or requirements change, the models must be extended, modified, or enhanced. Modeling is a creative process that requires the expertise and efforts of a developer. However, the developer must follow the syntactic rules of the modeling language and must pay attention to the overall model consistency, e.g., while describing a system from different modeling views. Therefore, the modeling process should be supported by tools that speed up the workflow and assist the developer during each edit step. Such recommendation tools are known from source-code editors, such as auto-completions, quick fixes, refactorings, or code templates.

Recommendation tools for modeling languages must take into account MDE-specific requirements. Approaches that implement hard-coded, domain-specific recommendation strategies have only limited applicability with respect to the various DSMLs in MDE. Therefore, a recommendation approach should be adaptable to different modeling languages. Our tool ReVision uses edit operations for the adaption to a DSML. ReVision proposes history-based model recommendations by recognizing partially executed edit operations from a model's editing history. In contrast, most recommendation tools implement a state-based approach, suggesting recommendations by analyzing only the model's current state, ignoring the developer's intention with respect to the actual editing process. Taking into account the historical changes of a model reduces the number of possible complementation alternatives and prevents accidentally undoing former work.

The process of modeling can be described as developing and applying a partially-ordered sequence of edit steps on a model version V_A to achieve a certain editing goal in a new model version V_B. Such edit steps can be described by an edit operation applied to the model. However, in some cases, an edit operation may have only been executed partially and requires a complementing edit step to finally achieve the editing goal. In the following, we will examine how such incomplete editing steps can be detected and complemented.

Difference graph. In practice, models are typically stored in a version control system comprising only state-based information of the model's evolution. However, the application of an edit operation can be recognized from the state-based model history. The first step is to compute the model difference $\Delta(V_A, V_B)$ between the older version V_A and the newer model V_B. As described in Section 3.4, the model difference $\Delta(V_A, V_B)$ describes the modifications as a set of historical changes between the ASGs of version V_A and version V_B of the model. This model difference can be represented by a difference graph $G_{\Delta(V_A,V_B)}$ as introduced in Section 3.4.3. Figure 4.1 shows an excerpt of the difference graph between the model versions in Figure 2.1 and Figure 2.2 of our running example in which two new transitions, namely, fastForward and play, are created.

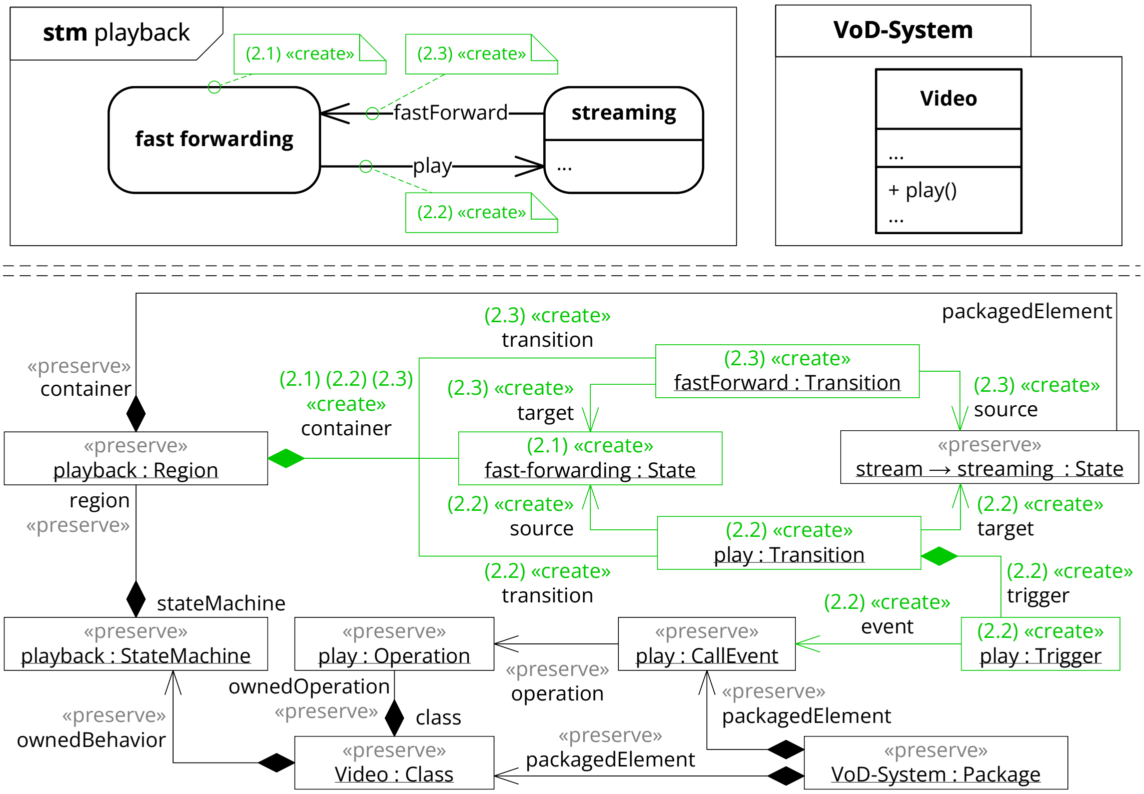


Figure 4.1. Excerpt of the unified difference graph $G_{\Delta(V_A,V_B)}$ between the model Version 1 and Version 2 illustrated in Figure 2.1 and Figure 2.2, respectively.

Edit rule graph. As discussed in Section 3.5, an edit operation $EO = \{ca_0, \dots, ca_m\}$ comprises a set of change actions ca_i defining possible modifications of the ASG of a model. The change actions have structural connections that are formally defined by a parameterized edit rule $ER(p_0, \dots, p_n) = r : L \twoheadrightarrow R$ (see Definition (3.10)). As defined in Section 3.5.3, such an edit rule based on graph transformation concepts can be represented by a corresponding edit rule graph G_{ER}. Figure 4.2 shows the edit rule graph of the edit operation *createOperationTriggeredTransition*. This edit rule graph describes the creation of a transition in a region of a state machine, including the creation of a new operation in a class diagram, which serves as a trigger for the transition.

The application condition of the edit rule comprises some NAC and PAC graph constraints. For example, assuming a domain-specific constraint for class diagrams that do not allow overloading of operations, the NAC forbids the creation of a new operation if another operation with the same name already exists in the specified class. The PAC graph constraints restrict the application context of the edit rule. The edit rule assumes that the transition and its source state are contained in the same region of the state machine. Moreover, the call event of the transition is created in the model's package containing the class.

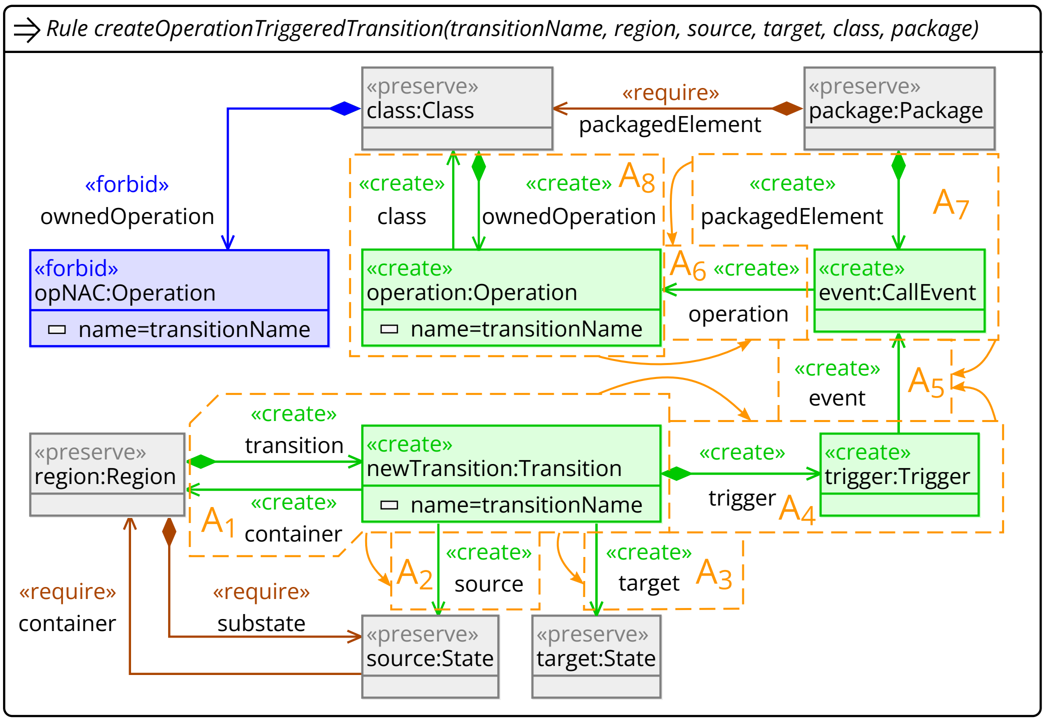


Figure 4.2. Edit rule *createOperationTriggeredTransition* creating a transition and an operation connected by a trigger.

Complementation of partially executed edit operations. Let us again have a look at our VoD-System running example in Figure 4.1, which is an excerpt of the model Version 2 shown in Figure 2.2. In comparison to the previous Version 1, the *edit step (2.3)* in Version 2 of the state machine diagram introduces the new transition fastForward. The transition connects the state streaming with the in *edit step (2.1)* newly created fast-forwarding state.

These edit steps are incomplete with respect to a missing trigger of the newly created transition fastForward. In the successive Version 3 of the class diagram in Figure 2.3, the operation fastForward() is created in *edit step (3.1a)*. Finally, the transition fastForward is connected to the operation by a call event, namely the *edit steps (3.1b)* and *(3.1c)* also indicated in Figure 2.3.

The edit operation *createOperationTriggeredTransition* in Figure 4.2 describes the compound effect of the discussed *edit steps (2.3)* and *(3.1)*, i.e., the creation of a transition triggered by a new operation. However, it does not include the *edit step (2.1)* that creates the fast-forwarding state. In the following, we will discuss the detection of the edit operation *createOperationTriggeredTransition* as a partially executed edit operation, i.e., the recognition of the sub-rule creating the transition fastForward in the model difference shown in Figure 4.1. Finally, we will compute the complementation of this partial execution of *createOperationTriggeredTransition*, which produces model Version 3 in Figure 2.3 of our running example, i.e., the recommendation proposal that inserts the operation fastForward() in the class Video and connects it with the fastForward transition by a new trigger.

As described by Kehrer et al. [1,78,82,83], the *complete execution* of an edit operation can be recognized by detecting the effect of its application in the model difference. Therefore, we need to find the historical changes in $\Delta(V_A, V_B)$ that correspond to all change actions in the edit rule ER. Consequently, to find a *partially executed* edit operation, we need to recognize a change set included in $\Delta(V_A, V_B)$ that corresponds to a *sub-rule* $ER_{sub} \subset ER$ of the edit rule ER. In this context, we will refer to the remaining parts of the edit rule $ER \setminus ER_{sub}$ as *complement rule* $\overline{ER}$.

$$\begin{aligned} complementation\,(\Delta(V_A, V_B)) &= complementation\,(V_A \overset{a}{\Longrightarrow}\overset{sr}{\Longrightarrow}\overset{b}{\Longrightarrow} V_B) \\ &= V_A \overset{a}{\Longrightarrow}\overset{sr}{\Longrightarrow}\overset{cr}{\Longrightarrow}\overset{b}{\Longrightarrow} V_B' \\ &= V_A \overset{a}{\Longrightarrow}\overset{er}{\Longrightarrow}\overset{b}{\Longrightarrow} V_B' \\ &\overset{!}{=} V_A \overset{a}{\Longrightarrow} V_A' \overset{sr}{\Longrightarrow}\overset{b}{\Longrightarrow} V_B \overset{cr}{\Longrightarrow} V_B' \end{aligned} \tag{4.1}$$

As shown in Definition (4.1), the problem of complementing a partially executed edit step can be described as the complementation of a sequence of edit steps a, sr, b applied on a model version V_A that results in a new model version V_B. In this sequence, the edit step sr refers to the changes of the sub-rule ER_{sub}. The previous edit step a has to be executed before ER_{sub}, i.e., the minimal set of changes that create the application context of ER_{sub}. The last edit step b contains all remaining changes of the model difference. The edit step b can also include changes that depend on the changes in edit step a and sr. In general, the edit steps a and b can also be empty.

The edit step cr refers to the changes of the complement rule $\overline{ER}$ that takes place after the edit step sr. In our complementation scenario, the edit step cr must be postponed to be applied to the current model version V_B, producing the complemented model version V_B'.

However, we have to consider that parts of the application context of the edit rule might have been removed in edit steps *sr* and *b*. Moreover, the application condition of the edit rule might not be fulfilled anymore on V_B due to elements created or deleted by the edit steps *sr* and *b*.

A complement rule application *cr* can be constructed from a sub-rule application *sr* if the effect of applying the complement rule to model version V_B is the same as replacing *sr* and *cr* with a full execution *er* of the edit rule *ER*. The application context of *ER* is determined after edit step *a*. Therefore, the application context and condition of *ER* and ER_{sub} need to be computed for an *intermediate model version* V'_A. The intermediate model version V'_A can be constructed by applying a subset of the changes in the model difference to V_A. In particular, we will not consider any transiently applied changes between V_A and V_B for the construction of V'_A (see Section 3.4.2). In general, V'_A can also be equal to V_A or V_B by applying no or all changes of the model difference. In Section 4.5, we will further discuss possible approximations of V'_A for the recognition of sub-rules.

The final step of generating complementation proposals is to determine all possible parameter bindings yielding applications of the complement rule $\overline{ER}$ such that the applied ER_{sub} is extended to a full execution of *ER*. Therefore, the application contexts regarding the model version V'_A is transferred to parameter bindings of the complement rule with respect to model version V_B. In particular, the elements in the application context of the complement rule must not be deleted in edit step *b*. Moreover, as the application condition is already checked by the sub-rule, the complement rule can ignore such checks.

The detected partial execution of an edit operation can help the developer to understand the intention of the historical changes and to make an informed decision to complement or rollback an incomplete edit step. In particular, a rollback of the edit step *sr* can only be applied without further side effects if the edit step *b* does not depend on *sr*.

CSP Algorithm. The recognition and complementation of partially executed edit operations is implemented by translating this problem into an equivalent constraint satisfaction problem (CSP) [157, 181]. The data structure in Figure 4.3 describes the representation of an edit rule graph G_{ER} in the CSP. Basically, every graph element annotated with a change action in G_{ER} is considered as a variable in the CSP. Figure 4.4 extends the data structure from Figure 4.3 to describe the mapping of change actions from G_{ER} to the difference graph $G_{\Delta(V_A,V_B)}$. For each variable, a domain slot is created comprising the changed graph elements in the model difference graph $G_{\Delta(V_A,V_B)}$ serving as possible variable values. The CSP solver assigns values to variables by a backtracking algorithm [97, 125] that removes impossible values for the variables by imposing restrictions on their domain. A solution found by the CSP solver defines a partial mapping from change actions of the edit rule *ER* to historical changes of the model difference $\Delta(V_A, V_B)$. The partial mapping represents an application of a sub-rule ER_{sub} of *ER*. Finally, possible parameter bindings of the complement rule $\overline{ER}$ are computed as recommendation proposals.

In the following Section 4.1, the derivation of a sub-rule and complement rule based on a given set of change actions of an edit rule is introduced. In Section 4.2, we define the syntactical criteria that must be fulfilled by all possible sub-rules that can be derived from an edit rule. Following the syntactical definition of sub-rules, in Section 4.3, the application

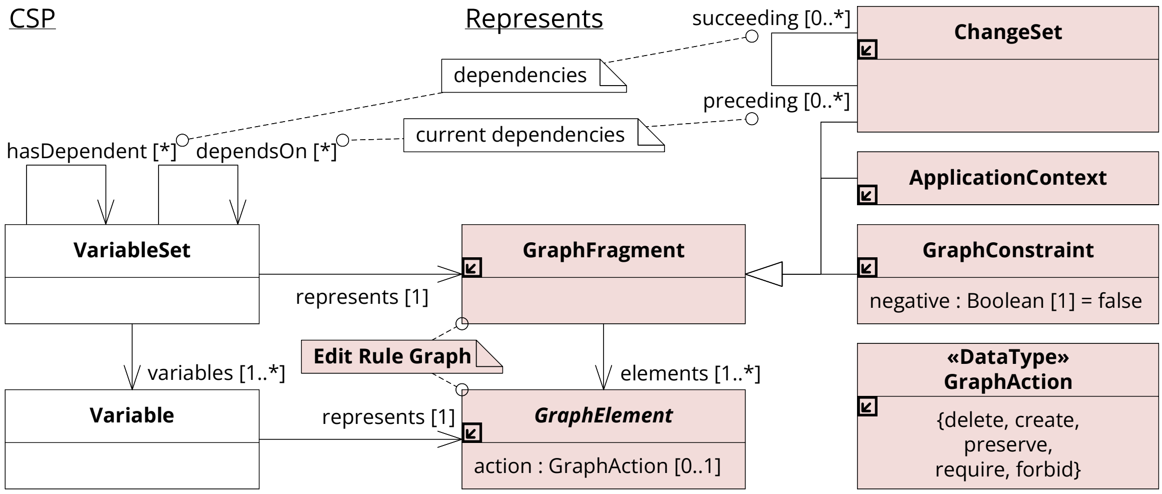


Figure 4.3. Meta-model of the CSP data structure. The CSP variables represent graph elements of the ↙ edit rule graph meta-model from Figure 3.22.

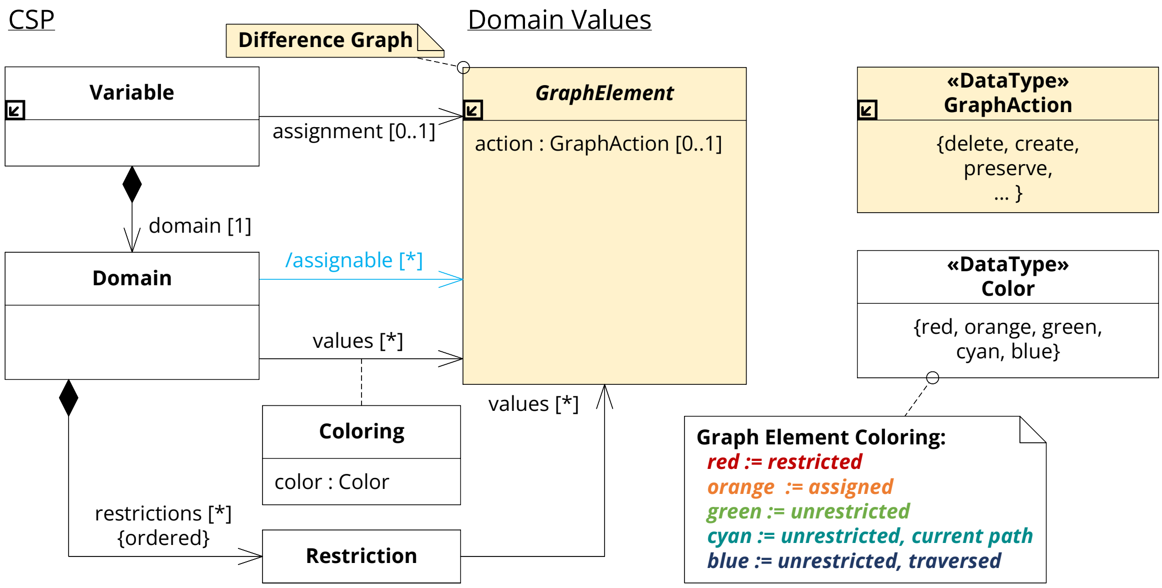


Figure 4.4. Defines the domain of a variable by extending the ↙ CSP meta-model in Figure 4.3. The values of a domain refer to the ↙ unified graph meta-model from Figure 3.12 for representing elements of a difference graph.

of a sub-rule is recognized in a model difference. Based on a recognized sub-rule application, in Section 4.4, we determine all possible complement rule applications. In addition, in Section 4.5, the application condition of the edit rule is considered for the sub-rule recognition. In Section 4.6, we will discuss possible adaptations of the approach in the context of a recommendation scenario, e.g., by specifying an editing context. In the context of user interactions, Section 4.7 introduces possible refinements of the proposed complementations.

Finally, Section 4.8 extends the approach to multi-rules (see Section 3.5.2). As an alternative to a complementation, in Section 4.9, a rollback of the sub-rule's changes is computed to undo an incomplete edit step.

4.1 Derivation of Sub-Rules and Complement Rules

An edit rule graph G_{ER} can be split into a sub-rule graph $G_{ER_{sub}}$ and a complement rule graph $G_{\overline{ER}}$ based on the corresponding sets of change actions $ER_{sub} \subset ER$ and $\overline{ER} = ER \setminus ER_{sub}$. Notably, during the detection of sub-rules in Section 4.3, the (potential) sub-rules are not constructed explicitly. Conceptually, a sub-rule $G_{ER_{sub}}$ can be constructed by removing the complementing change actions $\overline{ER}$ from G_{ER}.

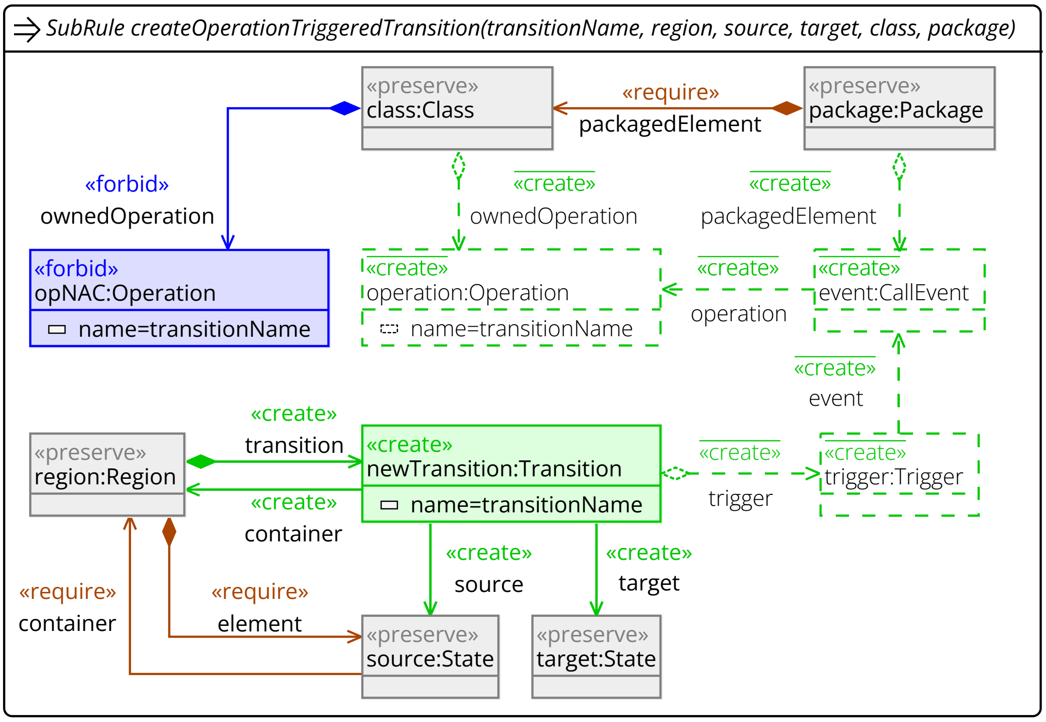


Figure 4.5. Sub-rule of the edit rule *createOperationTriggeredTransition* in Figure 4.2.

The sub-rule graph shown in Figure 4.5 is an example of a syntactically correct sub-rule of the edit rule graph shown in Figure 4.2. The sub-rule only creates the transition between two given states of a state machine. In Figure 4.5, the complementing change actions are illustrated as dashed nodes and edges with overlined graph actions. In general, the sub-rule construction can be described by applying the following transformation steps on the edit rule graph:

(1) All graph elements (nodes, edges, and attributes) that represent change actions of $\overline{ER}$ annotated with «create» are removed from G_{ER}.

(2) As a result, attribute value changes «modify» attrType = valueA → valueB (≡ {«delete» attrType = valueA, «create» attrType = valueB}) in G_{ER} might be reduced to «delete» valueA attributes. Therefore, we change their graph actions to «require» valueA to interpret them as part of the sub-rule's application condition.

(3) Complement rule nodes with «delete» actions that have incident edges to be deleted by the sub-rule are converted to context nodes annotated with «preserve» actions.

(4) The remaining «delete» actions of all graph elements that represent changes of $\overline{ER}$ are replaced with «require» actions.

Based on a given sub-rule ER_{sub} of an edit rule ER a corresponding complement rule $\overline{ER}$ can be derived. Basically, the complement rule graph $G_{\overline{ER}}$ is constructed by removing the change actions of the sub-rule from the edit rule graph, i.e., there is exactly one complement rule for each sub-rule. A formal treatment of the complement rule construction can be found in Reference [2].

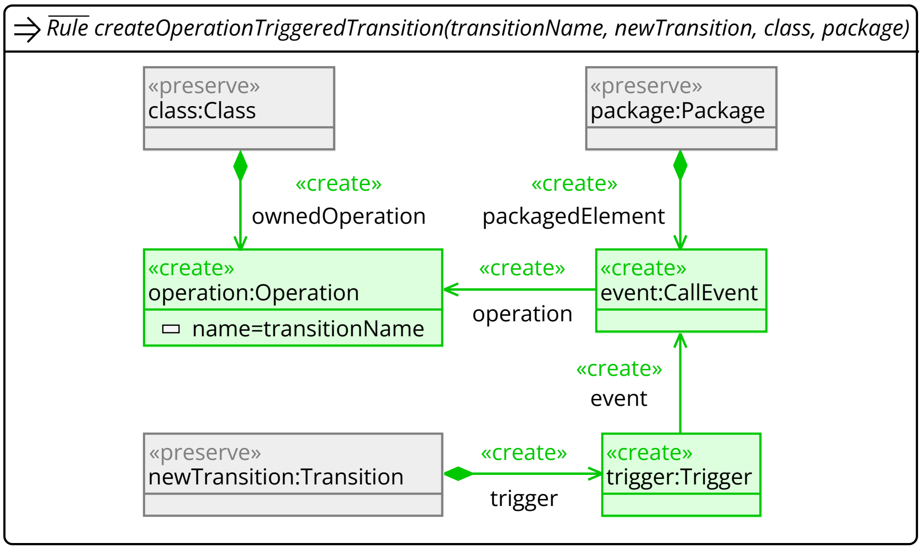


Figure 4.6. Complement rule of the edit rule *createOperationTriggeredTransition* in Figure 4.2.

Consider again the edit rule specified in Figure 4.2 and its sub-rule that creates a transition between two states. The complement rule constructed from this sub-rule is shown in Figure 4.6. In comparison to the sub-rule in Figure 4.5, the complement rule does not include the application conditions. The application context in the lower part of Figure 4.6 is adapted and reduced to the newly created transition. Moreover, the attribute *transitionName* is given as predefined input of the complement rule. Generally, the complement rule $\overline{ER}$ is obtained as follows:

(1) All creation change actions comprised by ER_{sub} are turned into application context that is to be preserved by $\overline{ER}$.

(2) Next, all graph elements that specify deletion change actions comprised by ER_{sub} are removed from $\overline{ER}$.

(3) As the application condition of the edit rule is already checked by the sub-rule, the application context of the complement rule is reduced to preserve only the boundary nodes of the remaining change actions.

(4) The input parameters are derived from the remaining application context and attribute values of the resulting complement rule.

4.2 Generation of Sub-Rules

To detect partial executions of an edit rule, we will first define how to enumerate all syntactically correct sub-rules without considering their occurrence in the model difference. Basically, a sub-rule $ER_{sub} \subset ER$ can be specified as a nonempty, proper subset of the change actions contained in the edit rule ER. While an edit rule may have many sub-rules, we are only interested in those sub-rules meeting the specific conditions specified in Section 4.2.1 to Section 4.2.3.

Basically, a complex edit operation can be composed of multiple atomic change sets that specify the minimal edit steps that can be partially executed without violating the elementary consistency of a model (see Section 3.1.3). Moreover, the change actions of an edit rule are partially ordered by their dependencies, i.e., the changes cannot be applied in an arbitrary order. A sub-rule is an initially executable part of an edit rule, e.g., the node newTransition:Transition in the edit rule *createOperationTriggeredTransition* in Figure 4.2 must be created before the edges specifying its source and target state.

The conditions from Section 4.2.1 to Section 4.2.3 are implemented as constraints in the CSP solver shown in the pseudocode in Listing 4.1. In particular, the algorithm in Listing 4.1 specifies the initial part of the CSP solver for enumerating all possible sub-rules.

4.2.1 Preservation of Elementary ASG Consistency

Since a sub-rule is expected to be a valid edit rule, which does not cause elementary defects in a model, it must adhere to the elementary ASG consistency, as defined in Section 3.1.3. According to Section 3.3.3, we will first identify all atomic change sets in G_{ER}. An atomic change set consists of graph elements of the edit rule graph G_{ER}, i.e., nodes, edges, or attributes annotated with a «create» or «delete» action. The dashed boxes in Figure 4.2 are showing the atomic change sets A_i of the edit rule *createOperationTriggeredTransition*. For example, the atomic change set A_1 contains the creation of a new transition, including its containment edge. Moreover, the container edge is also included in A_1 as the opposite edge of the transition containment edge.

Considering an edit rule with m change actions, the number of all possible sub-rules (without considering further conditions) is $(2^m - 2)$, i.e., the power set of all change actions excluding the empty set and the complete edit rule. Combining the change actions into atomic change sets that cannot be split, the number of all considered sub-rules will be reduced accordingly.

The pseudocodes in Listing 4.1 and Listing 4.2 describe the basic algorithm to enumerate all sub-rules of an edit rule. In particular, all subsets of change actions of the corresponding edit rule graph G_{ER} are enumerated, taking into account that the atomic change sets are not split.

Initialization (Line 1): The atomic change sets of an edit rule are represented by corresponding variable sets in the CSP. As illustrated in the CSP's data structure in Figure 4.3, a VariableSet consisting of Variables can represent a ChangeSet consisting of corresponding GraphElements. The CSP in Listing 4.1 is initialized by creating such variable sets from the edit rule graph G_{ER}. The function call changeSets(G_{ER}) inserts the initial variable sets corresponding to the atomic change sets into the stack named remaining of

```
remaining: Stack<VariableSet> = changeSets($G_{ER}$)
removed: Stack<VariableSet> = []
solution: Stack<VariableSet> = applicationContext($G_{ER}$) ∪ graphConstraints($G_{ER}$)
bounds: Bounds = Bounds{lower = |solution| + 1, upper = |solution| + |remaining| − 1}

function expandSolution():
  if constraintSolution() then                          ⟩ Is the solution within the solution space?
    if remaining ≢ ∅ then                               ⟩ Is the partial solution finished?
      vars: VariableSet = selectVariableSet()           ⟩ (A) Expand Solution: remaining → solution
      findSolution(vars)                                ⟩ ↪ solutions with selected variables
      removeVariableSet()                               ⟩ (B) Reduce Solution: solution → removed
      expandSolution()                                  ⟩ ↪ solutions without selected variables
      returnVariableSet()                               ⟩ (B, A) Backtracking: removed → remaining
    else
      solutionFound(solution, removed)                  ⟩ next (partial) solution found
    endif
  endif
-----------------------------------------------------------------------------------------------
function constraintSolution(): Boolean
  return |solution| ≤ bounds.upper                                  ⟩ is below!
         ∧ |solution| + |remaining| ≥ bounds.lower                  ⟩ is above!
         ∧ (solution ≢ ∅ ⟹ solution.peek().dependsOn ≡ ∅)          ⟩ is independent!
         ...                                                        ⟩ see Listing 4.5 and Listing 4.9
-----------------------------------------------------------------------------------------------
function selectVariableSet(): VariableSet
  if hasIndependent(remaining) then                     ⟩ (A) Expand Solution: with ...
    independent: VariableSet = selectIndependent(remaining)
    remaining.remove(independent)
    solution.push(independent)                          ⟩ ...independent node
  else
    solution.push(remaining.pop())                      ⟩ ...dependent node
  endif
  selected: VariableSet = solution.peek()
  removeFromDependencyGraph(selected)                   ⟩ (C) Consider the change as executed.
  return selected
-----------------------------------------------------------------------------------------------
function removeVariableSet():
  removed.push(solution.pop())                          ⟩ (B) Reduce Solution
  reinsertIntoDependencyGraph(removed.peek())           ⟩ (C) Backtracking
-----------------------------------------------------------------------------------------------
function returnVariableSet():
  remaining.push(removed.pop())                         ⟩ (B, A) Backtracking
```

Listing 4.1. Initial part of the CSP solver algorithm to generate all sub-rules of an edit rule.

```
function findSolution(vars:VariableSet):
  expandSolution()                                      ⟩ ↪ solutions with selected variable
```

Listing 4.2. Stub of the assignment function for generating sub-rules without assignments.

the CSP solver. For our example edit rule in Figure 4.2 the variable sets representing the atomic change sets A_0 to A_7 are created.

CSP solution (Line 2 - 3): In order to enumerate all sub-rules of an edit rule, the CSP solver implements a recursive algorithm. The result of the CSP solver is finally contained in the solution stack, i.e., a stack of variable sets representing the atomic change sets of a recognized sub-rule ER_{sub}. In contrast, the variable sets in the removed stack represent the atomic change sets of the complement rule $\overline{ER}$. As the application context and graph constraints of the edit rule are always contained in the sub-rule, the solution is initialized by the applicationContext(G_{ER}) and graphConstraints(G_{ER}) function calls with the corresponding variables.

Main function (Line 5 - 6): The recursive computation is implemented in the expandSolution() function. The recursion continues as long as the constraintSolution() function evaluates to *true*. This function checks the actual constraints of the CSP, which we will introduce in the following sections. In particular, if we would ignore all constraints and dependencies between atomic change sets, then the algorithm would simply enumerate all possible 2^m subsets of the m given variable sets in the remaining stack.

Step (A) expand solution (Line 7 - 8): In order to compute a solution, the variable sets are systematically moved from the remaining stack to the solution or removed stack as long as the remaining stack is not empty (remaining $\not\equiv \emptyset$). In this process, an atomic change set must always be fully contained in a sub-rule, i.e., a variable set must also be fully included in the CSP's solution. Therefore, calling the function selectVariableSet() always moves a complete variable set from the remaining variables to the solution variables (Line 28).

Solutions with selected variables (Line 9): In this section, we only discuss the derivation of syntactically correct sub-rules. Therefore, we will skip the assignment of variables by replacing the responsible function findSolution() with a simplified function stub, as shown in Listing 4.2. In order to enumerate all sub-rules that include the just selected atomic change set, the function expandSolution() is recursively called (via findSolution()) until no more variables are contained in the remaining stack.

Next (partial) solution found (Line 14): As soon as the remaining stack is empty, a new solution has been found and can be processed further by the solutionFound() function (Line 14). For example, storing the sub-rule and the complement rule by processing the solution and removed stack, respectively.

Step (B) reduce solution (Line 10 - 11): After handling the new solution, the function expandSolution() returns to its last recursive call (Line 9). In the next step, calling removeVariableSet() moves the last variable set from the solution stack to the removed stack (Line 34). The following recursive call of expandSolution() (Line 11) computes all remaining partial solutions that exclude the removed variable set.

Backtracking step (B, A) (Line 12): In order to search the entire solution space, the function expandSolution() implements a backtracking algorithm. Therefore, the function returnVariableSet() always moves the last variable set from the removed stack back onto the remaining stack. (Moving the variable set to the solution stack can be skipped in the backtracking for efficiency.) In terms of the sub-rule enumeration, this backtracking step allows an atomic change set to be reselected as part of a sub-rule while another atomic change set is temporarily removed from the solution space.

4.2.2 Minimal/Maximal Size of Sub-rules

In general, we are only interested in partial edit rules, i.e., the sub-rule without change actions and ER itself can be excluded. Moreover, the developer might want to specify the sub-rule size in terms of a lower and an upper bound number of atomic change sets depending on task-specific preferences, e.g., specified as a percentage value. For example, a partial execution should only be detected if 50% of the edit rule has already been executed. In general, a large sub-rule with respect to the edit rule leads to a small complement rule minimizing the proposed edit step, i.e., small modifications are more likely recommendations than large ones.

Moreover, increasing the lower bound or decreasing the upper bound also reduces the number of sub-rules to be checked by the CSP solver. The bounds can be computed initially for each ER and specified as constraints of the CSP in Listing 4.1 (Line 4). The bounds need to be checked after modifying the solution. Checking the predefined upper bound in the function constraintSolution() will skip the subsequent computation of all larger sub-rules. If the solution is outside of the solution space with respect to the predefined lower bound, the subsequent smaller solutions will be skipped. Therefore, the predefined lower bound is compared with the potential maximal solution size (|solution| + |remaining|) during the computation.

4.2.3 Dependencies Between Change Actions

Change actions defined by a sub-rule ER_{sub} must not depend on change actions within the complement rule $\overline{ER} = ER \setminus ER_{sub}$, i.e., ER_{sub} is self-contained with respect to all dependencies between change actions defined by ER. According to Section 3.3.3, a dependency graph can be created in which nodes represent atomic change sets of the edit rule. As illustrated for edit rule *createOperationTriggeredTransition* in Figure 4.2, the dependencies are drawn as edges between atomic change sets A_i indicating their execution order. The edit rule *createOperationTriggeredTransition* contains several elementary creation dependencies.

The dependencies between atomic change sets form a directed acyclic graph (DAG). The nodes of a DAG can be sorted in a topological order [75] representing a correct execution order of the atomic change sets. Sub-rules of ER that do not represent an initial sequence of any topological ordering of atomic change sets of ER can be excluded. This constraint further reduces the number of potential sub-rules to be considered. For example, the edit rule in Figure 4.2 has 55 possible sub-rules considering the topological ordering of its atomic change sets.

As shown by the data structure in Figure 4.3, the dependencies of the atomic change sets

(ChangeSet::preceding, ChangeSet::succeeding) are correspondingly stored by the variable sets (VariableSet::dependsOn, VariableSet::hasDependent). For example, in the edit rule in Figure 4.2, the atomic change set A_2 is a succeeding change set of A_1, i.e., the corresponding variable set A_2 depends on A_1.

The topological ordering is maintained by the selectVariableSet() function of the CSP solver in Listing 4.1 (Line 22). Basically, the function selects the variable sets from the dependency graph in any valid topological ordering. The function hasIndependent() (Line 23) checks the remaining stack for variable sets without current dependencies. A variable set $v_i \in$ remaining is independent if it currently does not depend on other variables sets (v_i.dependsOn $\equiv \emptyset$). Finally, one of the independent variable sets is selected and moved from the remaining set to the solution stack (Line 25 - 26).

After the selection of a variable set v_i, the function removeFromDependencyGraph() removes v_i from the dependency graph by removing v_i from all of its dependent variable sets. For example, selecting the variable set corresponding to A_1 from the edit rule in Figure 4.2, A_1 is removed as dependency from A_2, A_3, and A_4. Technically, we only modify the dependsOn references for each VariableSet, i.e., the preceding atomic change sets. The VariableSet::hasDependent references containing the succeeding atomic change sets are preserved and will be used to reinsert a variable set into the dependency graph during the backtracking.

If no independent variable set exists in the remaining stack (Line 23), which can occur if some required variable set is already pushed to the removed stack, the solution cannot be extended any further. Therefore, all remaining variable sets are removed from the solution space. Technically, the selectVariableSet() function temporarily pushes any of the remaining dependent variable sets to the solution stack (Line 28). A dependent variable set in the solution will cause the function constraintSolution() to discard the current solution (Line 20). Subsequently, the dependent variable set will be removed from the solution space by the removeVariableSet() function (Line 10).

Whenever a variable set is temporarily removed from the solution space, the removeVariableSet() function will reinsert the variable set into the dependency graph (Line 35). Technically, the removed variable set is added back as a dependency (VariableSet::dependsOn) to all of its dependent (VariableSet::hasDependent) variable sets. Backtracking the removal of A_1 from the dependency graph with respect to our previous example, we add A_1 back as dependency to A_2, A_3, and A_4. This procedure prevents the generation of sub-rules that do not contain an initial sequence of any topological ordering of the change actions in the processed edit rule.

4.3 Recognition of Sub-Rules

In this section, we will extend the sub-rule generation in order to detect partially executed edit rules in a model difference $\Delta(V_A, V_B)$. The difference $\Delta(V_A, V_B)$ contains all concrete, historical changes between two versions V_A and V_B of a model. The task is to recognize corresponding historical changes in $\Delta(V_A, V_B)$ that match the definitions of the change actions of a sub-rule of the edit rule. Therefore, we can take advantage of the approach introduced by Kehrer et al. [1, 78, 82, 83], which reduces the edit rule recognition problem to a graph matching problem. The basic idea is that each edit rule can be transformed into a well-defined graph pattern consisting of the change actions, structural relations, and application conditions that can be matched in a model difference. However, instead of recognizing a complete edit rule ER, the task is to do such a recognition for all proper sub-rules $ER_{sub} \subset ER$ of the edit rule.

Similar to the edit rule graph G_{ER} the model difference $\Delta(V_A, V_B)$ can be viewed as unified graph $G_{\Delta(V_A,V_B)}$ (see Section 3.4). Figure 4.1 shows the difference graph $G_{\Delta(V_A,V_B)}$ for the VoD-System between the model version in Figure 2.1 and the version in Figure 2.2. The difference graph shows the creation of the transition fastForward between the states streaming and fastforwarding on the level of the model's ASG. To recognize the partial execution of an edit rule, we need to find a partial matching between the edit rule graph G_{ER} and the difference graph $G_{\Delta(V_A,V_B)}$.

For example, the edit rule graph shown in Figure 4.2 can be mapped partially to the difference graph in Figure 4.1. The partial mapping matches the change actions of the edit rule and the historical changes creating the transition between the two states. The unmatched change actions of the edit rule will create an operation connected to the transition by a new event and a trigger. Basically, the partially matched change actions represent the sub-rule, and the unmatched change actions represent the complement rule.

The partial matching of changes must be connected by the structure imposed by the edit rule graph G_{ER}. Moreover, we will only consider maximal sub-rule matches, i.e., a sub-rule must not be contained in another larger sub-rule match. In particular, the partial graph matching between the edit rule graph G_{ER} and the difference graph $G_{\Delta(V_A,V_B)}$ must meet the conditions defined in the following Section 4.3.1 to Section 4.3.3. The sub-rule recognition and the condition from Section 4.3.1 to Section 4.3.3 are implemented by extending the CSP solver from Listing 4.1 in Listing 4.3.

4.3.1 Compatibility of Sub-rule Matchings

We can describe an edit rule by its edit rule graph G_{ER}. Therefore, a sub-rule leads to a subgraph $G_{ER_{sub}} \subset G_{ER}$ of the edit rule graph, which has to be matched in the difference graph $G_{\Delta(V_A,V_B)}$. Given a sub-rule graph $G_{ER_{sub}}$ and a difference graph $G_{\Delta(V_A,V_B)}$, the matching problem is to find an injective, yet type-, structure- and action-compatible mapping $m : G_{ER_{sub}} \to G_{\Delta(V_A,V_B)}$. Specifically, the following compatibility constraints must be fulfilled for all mapped graph elements in m:

type-compatible: Basically, the type of a node, edge, or attribute in the sub-rule graph $G_{ER_{sub}}$ must match the type in the difference graph $G_{\Delta(V_A,V_B)}$. Excluding node creations («create»), nodes in $G_{ER_{sub}}$ can also be mapped to model elements in $G_{\Delta(V_A,V_B)}$ with a compatible subtype, i.e., the type of the node in $G_{\Delta(V_A,V_B)}$ can be a more concrete subtype compared to the $G_{ER_{sub}}$ node's type.

action-compatible: Table 4.1 gives an overview of the action compatibility rules between the edit rule graph and the difference graph. Depending on the mapping between the graphs, we can derive the corresponding action of the sub-rule (see Section 4.1).

```
function findSolution(vars: VariableSet):
  if  hasUnassignedVariables(vars) then                   ⟩ select variable and compute assignments...
    v:  Variable = selectUnassignedVariable(vars)
    foreach assignment ∈ v.domain.assignable do
      assignVariable(v, assignment)                       ⟩ (D) assign selected variable
      if  constraintAssignment(v, vars) then              ⟩ Is the assignment within the solution space?
        findSolution(vars)                                ⟩ ↪ assign next variable in given set
      endif
      freeVariable(v)                                     ⟩ (D) backtracking assignment
    endforeach
  else
    expandSolution()                                      ⟩ ↪ solutions with assigned variables
  endif
--------------------------------------------------------------------------------------------------
function constraintAssignment(v: Variable, vars: VariableSet): Boolean
  return checkAssignmentInjectivity(v, solution)
          ∧ checkStructureCompatibility(v, solution)
          ∧ checkAttributeConstraints(v, solution)
          ...                                             ⟩ see Listing 4.4
--------------------------------------------------------------------------------------------------
function assignVariable(v: Variable, assignment: GraphElement):
  v.assignment = assignment                               ⟩ (D) assign selected variable
  restrictDomains(v, assignment)                          ⟩ (E) structural restrictions starting with v
--------------------------------------------------------------------------------------------------
function freeVariable(v: Variable):
  v.assignment = undefined                                ⟩ (D) backtracking assignment
  unrestrictDomains(v)                                    ⟩ (E) backtracking structural restrictions
```

Listing 4.3. CSP algorithm to compute the assignments of a given variable set.

	Edit Rule	**Difference**	**Sub-Rule**	**Comment**
(1)	«create»	«create»	«create»	sub-rule creations
(2)	«create»	∅	∅	to be created by the complement rule (see Section 4.4)
(3)	«delete»	«delete»	«delete»	sub-rule deletions
(4)	«delete»	«preserve»	«preserve»	to be deleted by the complement rule (see Section 4.4)
(5)	«preserve»	«preserve» / «delete» / «create»	«preserve»	including preceding/succeeding changes (see Section 4.4)

Table 4.1. Action compatibility of the sub-rule matching.

In general, graph elements in the edit rule graph G_{ER} that are annotated with «create» or «delete» actions must be mapped to difference graph elements that are annotated with the same kind of action (Row 1 and 3). Graph elements with «create» actions that cannot be mapped to the difference graph (∅) are to be created by the complement rule (Row 2). Conversely, graph elements to be deleted by the complement rule must be mapped to graph elements preserved in the difference graph (Row 4). Notably, mapping the graph elements to be deleted to elements created with respect to the difference graph would result in changes being undone instead of complementing an editing step.

A graph element in the edit rule annotated with a «preserve» action can be mapped to graph elements annotated with «preserve» actions in the difference graph (Row 5). Moreover, such a graph element can also be mapped to a difference graph element that is annotated with a «create» or «delete» change action. In this latter case, the graph element is created or deleted by another edit step before or after the recognized sub-rule.

In general, the graph constraints («require» /«forbid») of an edit rule's application condition are checked subsequently (see Section 4.5) after determining all possible application context matchings (see Section 4.4) for a partial matching of change actions.

structure-compatible: The matching of nodes in the mapping $m : G_{ER_{sub}} \rightarrow G_{\Delta(V_A,V_B)}$ must be injective, i.e., all nodes, edges, and attributes of $G_{ER_{sub}}$ are mapped, and a graph element of $G_{\Delta(V_A,V_B)}$ is only mapped once to a graph element in $G_{ER_{sub}}$. The edges in the sub-rule graph $G_{ER_{sub}}$ must be mapped to edges of the same type incident to the mapped nodes in difference graph $G_{\Delta(V_A,V_B)}$.

As a preliminary step, the domains of all variables have to be initialized with elements from the difference graph $G_{\Delta(V_A,V_B)}$ that can be potential assignments for the variables of any sub-rule. As shown in the data structure in Figure 4.4, a variable is associated with a domain that comprises the compatible graph elements in model difference graph $G_{\Delta(V_A,V_B)}$ serving as possible variable assignments. Initially, the type- and action-compatible graph elements from the difference graph are collected for the domains. Moreover, attributes in the edit rule graph that are bound to constant values are checked during the domain initialization, i.e., non-matching graph elements can already be filtered from the domains.

Regarding our running example, the corresponding domains of the creation of the transition in the edit rule in Figure 4.2 can be initialized with the new transitions play and fastForward from the difference graph in Figure 4.1. Regarding the application context, for example, the corresponding domain of the source:State node can be initialized with all states from the model Version 2 in Figure 2.2. In particular, the domain of this application context node also contains the newly created state fast-forwarding.

All CSP constraints that cannot be checked during the initialization of the domains need to be checked dynamically during the assignment of the solution. Generally, the CSP solver assigns variables with values from their corresponding domains, whereby the variable assignment must fulfill certain constraints. A solution to the CSP is found when all currently selected variables that represent the change actions of a sub-rule are assigned to changes in the model difference graph $G_{\Delta(V_A,V_B)}$.

Computing a matching from scratch for each sub-rule generated by the algorithm in Listing 4.1 would be very inefficient. In fact, the matchings are computed incrementally by the CSP solver in Listing 4.3 while the variables are added and removed from the solution. The findSolution() function in Listing 4.3 is called by the sub-rule generation algorithm in Listing 4.1 (Line 9) for each selected variable set. The CSP solver in Listing 4.3 determines a concrete solution by the findSolution() function, which takes an unassigned set of variables and computes all possible valid assignments.

Select next unassigned variable (Line 41 - 42): The findSolution() function is called with the variable set vars to be assigned. Initially, as long as not all variables are assigned, an unassigned variable v is selected from the given variable set vars. Technically, the algorithm can be optimized by preferring variables with a small number of remaining assignable elements in their domain. In particular, if the domain of a variable in the given set is empty (v.domain.assignable $\equiv \emptyset$), then no complete assignment of the variable set is possible and the current assignment is not explored further. Therefore, the selectUnassignedVariable() (see Listing 4.3) always selects the unassigned variable with the currently smallest domain. In this context, the selectVariableSet() (see Listing 4.1) selects the variable set with the currently smallest product of their domain sizes, i.e., the maximum number (possibly 0) of assignment combinations. Notably, after each assignment, the size of the domains of the unassigned variables might change due to structural restrictions, which we will discuss in the following Section 4.3.2.

Compute assignments (Line 43 - 49): For the selected variable v, all values of its domain are processed. To generate all possible assignments, the findSolution function implements a recursive backtracking algorithm. After assigning a variable by the assignVariable() function (Line 44), the findSolution() function is called recursively (Line 46) until all variables of the given set are assigned. In a backtracking step, the variable is unassigned by the freeVariable() function (Line 48), and the next value of the domain is assigned until all possible combinations of valid assignments are found.

Constraint assignment (Line 45): The constraintAssignment() function (Line 53) checks the latest assignment incrementally. In this context, the injectivity of the overall assignment and the local structure compatibility to assignments of adjacent nodes is checked. In general, if we would ignore all constraints and restrictions during the assignment, then all possible combinations of values in the variable's domains would be generated.

Based on the mapping of a node of the edit rule, the contained attributes attributeName = parameterName mapped to a parameter of the transformation rule need to be checked, e.g., the attribute name = transitionName in the edit rule in Figure 4.2 defining the name of the new transition and operation. In general, if multiple attributes are assigned to the same parameter, the values of the mapped attributes in the difference graph must be equal. In particular, for a given mapping of a node, a contained attribute of an edit rule with a «preserve» or «require» action can be assigned to attributes of a difference graph with «delete» action at the same time (see Table 4.1). Therefore, the checkAttributeConstraints() (Line 56) checks incrementally for each parameter that at least one valid assignment exists.

Expand solution (Line 51): If a complete assignment is found for all given variables, the solution can be further expanded with the remaining variables by (recursively) calling the expandSolution() function in Listing 4.1.

4.3.2 Connectivity of Sub-rule Matchings

Referred to as the "principle of locality" by Heckel et al. [63], an edit rule with a connected edit rule graph G_{ER} applied to a model has a local effect in terms of the performed modifications. To limit possible partial matching, we also assume such local effects for the sub-rules $ER_{sub} \subseteq ER$. Searching for locally connected changes drastically reduces the search space when searching for sub-rule occurrences in a large model difference graph $G_{\Delta(V_A,V_B)}$. It follows that change actions in the edit rule graph $G_{ER_{sub}}$ of the sub-rule must be connected by at least one path retained from the complete edit rule graph G_{ER}. As described for the mapping $m : G_{ER_{sub}} \rightarrow G_{\Delta(V_A,V_B)}$, such a path requires an injective, type-, structure-, and action-compatible mapping to a path in the difference graph $G_{\Delta(V_A,V_B)}$.

To give an illustration, consider again the edit rule *createOperationTriggeredTransition* in Figure 4.2. All changes specified by the edit rule form a connected graph, including the context elements that are to be preserved by the edit rule. The edit rule graph has to be mapped partially to the difference graph shown in Figure 4.1. We can recognize two corresponding sub-rule executions in the model difference graph. The creation of the transition can be mapped to the transition fastForward or play.

We are only interested in those assignments that are also structure-compatible with respect to the edit rule graph and the difference graph. As defined in Listing 4.3, starting with the assignment (Line 59) in function assignVariable() (Line 44), the domains of the other variables in the CSP are restricted by the restrictDomains() function (Line 60) to contain only those elements that are structurally reachable from the currently assigned element.

During the structural restriction, we consider the complete edit rule graph G_{ER}, i.e., the variables representing the change actions, application context, and graph constraints (see Section 4.3.1) contained in the remaining and removed stack. Conceptually, the CSP solver in Listing 4.1 only assigns variables representing change actions of the edit rule with changes of the difference graph. Finally, the domains of the variables that represent the application context and graph constraints contain all partial graph fragments of the difference graph connecting the assigned changes.

As shown in the example graph Figure 4.7, the computation of the structural restriction can be described as a graph coloring algorithm. An edit rule graph is illustrated by the gray boxes in the background of Figure 4.7. The nodes and edges from a difference graph are shown on top. In this example, for brevity, we will not further specify the concrete edit rule and model difference. Basically, this process can be described by a depth-first traversal on the edit rule graph G_{ER}. During the traversal, the reachable elements in the corresponding domains are identified. The traversal follows all non-cyclic paths between adjacent nodes of G_{ER}, i.e., the direction of edges in G_{ER} is not considered, and parallel edges are combined into one evaluation step.

In the CSP, each node and edge of the edit rule graph is represented by a variable with a domain comprising the nodes and edges of the difference rule graph. As depicted in the CSP data structure in Figure 4.4 the coloring is stored for each domain of a variable. Moreover, for each domain, the computed restrictions are stored for later backtracking.

Step 1: Initially, we assume all elements of the difference graph are colored in *green*. First, the variable representing the node named $N1$, with its domain colored in *orange*, is

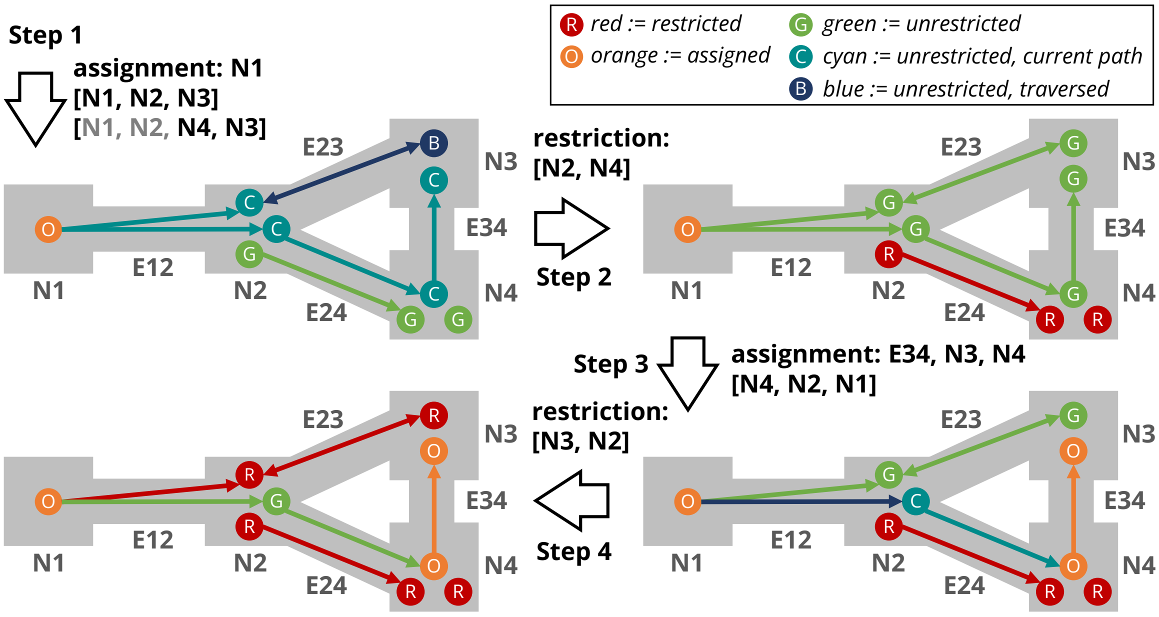


Figure 4.7. Structural restriction of the variable's domains representing the edit rule graph G_{ER} and the difference graph $G_{\Delta(V_A,V_B)}$.

assigned. Following all paths of the edit rule graph from node $N1$, the nodes and edges of the difference graph in the corresponding domains are colored in *blue*. We use the colors *cyan* and *blue* to mark the current state of the depth-first traversal, i.e., the current path is colored with *cyan*, and the previously traversed nodes are colored in *blue*. The result of the first traversal is shown in Step 1 of Figure 4.7.

Step 2: In computation Step 2 in Figure 4.7, all nodes and edges that are finally still colored in *green* are restricted by coloring them in *red*, and all *blue* nodes are reset to *green*. Moreover, the restricted elements of the domains are stored for later backtracking. As defined in the CSP data structure inFigure 4.4, the restrictDomains() function (Line 60) adds a Restriction to all variables of the CSP. Notably, the number of restricted values can also be empty, e.g., node $N3$ in Step 2.

Step 3: Now we assign the next variable representing the edge $E34$ of the edit rule graph. An edge is always selected, including its source and target node. After Step 3 in Figure 4.7, the resulting *blue* colored paths within the remaining unrestricted, *green* colored difference graph is shown.

Step 4: The restriction that results from the selected paths in Step 3 are shown in Step 4 in Figure 4.7. In particular, the domain corresponding to edge $E23$ is now empty, i.e., the current assignment is a partial matching of the edit rule and the difference graph. The subsequent assignments in the example can be done without further restrictions, namely, the variables representing the nodes and edges $N2$, $E12$, and $E24$.

Backtracking: After finishing a partial match, a backtracking step is performed to the initial state shown in Figure 4.7. Therefore, the stored restrictions of each domain are removed by coloring the elements *green*. This is done by first calling (Line 48) the freeVariable() function (Line 61) to reset the assignment of the variable. Next, the unrestrictDomains() function (Line 63) removes the latest restriction from all variables of the CSP. Subsequently, another path can be selected starting from node $N3$ of the edit rule graph.

4.3.3 Maximality of Sub-rule Occurrences

A sub-rule occurrence must cover a maximal number of historical changes in the difference $\Delta(V_A, V_B)$. That is, for a sub-rule ER_{sub} and its corresponding edit rule graph $G_{ER_{sub}}$ having a match m in the difference graph $G_{\Delta(V_A,V_B)}$, there must not be a larger sub-rule ER'_{sub} with $ER_{sub} \subset ER'_{sub} \subseteq ER$ such that the match m can be extended to become a match m' of the edit rule graph $G_{ER'_{sub}}$ in $G_{\Delta(V_A,V_B)}$. Otherwise, the complementation of a partially executed edit operation could accidentally repeat or overwrite historical changes already performed between the model versions V_A and V_B.

For instance, regarding our running example in Figure 2.2, a sub-rule of the edit rule in Figure 4.2 that comprises only the historical changes of the atomic change set A_1 (that creates a new transition) is not a maximal sub-rule. There is a larger sub-rule which, in addition, covers also the change sets A_2 and A_3, namely, the source and target of the transition.

However, there is a minor number of 5 out of 596 observable cases in the evaluation of the approach (see Chapter 8) in which a sub-rule is slightly too large, namely, a newly created text-based annotation of a model element is modified instead of creating a new additional annotation. Nevertheless, it might be confusing for the developer to find the required sub-rule in a recommendation list that also includes all sub-rules of sub-rules.

```
function constraintAssignment(v: Variable, vars: VariableSet): Boolean
  return ...    〉extends Listing 4.3
          ∧ (vars.represents.eClass ≡ ChangeSet  ⟹  v.assignment.action ≢ «preserve» )
```

Listing 4.4. Maximality of sub-rule with respect to the contained change actions.

Graph elements to be deleted by the complement rule have to be matched to elements preserved in the difference graph (see Table 4.1). Those preserved elements are also required to form a connected subgraph $G_{ER_{sub}}$ between the change actions of the sub-rule (see Section 4.3.2). However, during the sub-rule recognition, we must first maximize the set of recognized deletions of the sub-rule, i.e., we only allow the matching of sub-rule deletion actions to deletions in the difference graph. As shown in Listing 4.4, this can be enforced by a constraint during the assignment of variables. Technically, we can also optimize the assignment by already restricting elements with «preserve» actions from the variables' domains at the time of their selection. The selectVariableSet() function in Listing 4.1 can apply such a restriction on the domains. Accordingly, the backtracking of this restriction can be done by the removeVariableSet() function.

In the partial CSP solver, the maximality of a sub-rule matching simply means that the solver always does a full exploration up to the maximum number of variables that can be

```
function constraintSolution(): Boolean                ⟩ extends Listing 4.1
  return ...                                          ⟩ partial solution finished
         ∧ (remaining ≡ ∅  ⟹  isMaximalPartialSolution())
```

Listing 4.5. Maximality of sub-rule with respect to the contained change actions.

successfully assigned. A partial solution is not accepted if the domain of an unassigned variable is not empty, i.e., the variable representing a change action of the edit rule could be assigned with a change from the model difference.

Within the algorithm in Listing 4.1, the maximality of the assignment is checked by the constraint isMaximalPartialSolution() extending the function constraintSolution() in Listing 4.5. After all variables from the remaining set have been selected and assigned, the isMaximalPartialSolution() constraint checks the domains of all independent variables (see Section 4.2.3) currently contained in the removed stack. Those domains must be empty or contained values must not be assignable, i.e., the solution can not be extended with any of the variable sets in the removed stack.

Notably, the structural restrictions (see Section 4.3.2) computed during the assignment of a variable are always computed with respect to the full edit rule graph, i.e., the restrictions are also computed for variables in the removed stack. In addition, incomplete assignments of atomic change sets can already be filtered during the initialization of their corresponding variable domains. In this case, for actual maximal solutions, the domains are typically just checked as being empty. For actual non-maximal solutions, a variable set can typically be tested as being assignable, starting with any value in its non-empty domains.

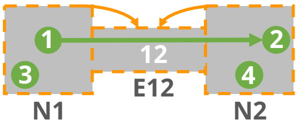


Figure 4.8. Maximality of sub-rule occurrence during backtracking.

Such non-maximal assignments can occur during the backtracking computation and can occur if a maximal solution comprising the set of variables S_m overlaps with another maximal solution $S_n \subset S_m$. For example, the graph in Figure 4.8 consists of two nodes $N1$ and $N2$ and an edge $E12$ that depends on its source and target nodes. The subgraph matching problem has three maximal, structurally-connected solutions: $S_0 = \{N1 = 1,\ E12 = 12,\ N2 = 2\}$, $S_1 = \{N1 = 3\}$, and $S_2 = \{N2 = 4\}$. To compute such solutions, we initialize the CSP solver in Listing 4.1 with the remaining = {N1, E12, N2} variables. Assuming $N1$ is selected before $N2$ (Line 8), the backtracking algorithm first determines the solutions for $N1,\ E12,\ N2$, before the partial solution $N1$ and $N2$ are computed. However, during the assignments starting from $N2$, we will first find the non-maximal solution $\{N2 = 2\}$.

During the backtracking from the solution $S_0 = \{N1 = 1,\ E12 = 12,\ N2 = 2\}$, the CSP solver also computes the partial solutions $S_{0.1} = \{N1 = 1\}$ and $S_{0.2} = \{N1 = 1,\ N2 = 2\}$. Both solutions are non-maximal solutions, i.e., the solutions $S_{0.1}$ and $S_{0.2}$ are subsets of the solution S_0. As an optimization, the isMaximalPartialSolution() function should first check the variable on top of the removed stack, i.e., the latest variable that has been removed during the backtracking search to identify such cases directly.

4.4 Matching of Application Contexts

As discussed in Section 4.1 and Definition (4.1), the sub-rule must be applicable in an intermediate model version V'_A considering the model difference $\Delta(V_A, V_B)$. To compute the application context of a sub-rule in V'_A, the CSP solver in Listing 4.6 extends the algorithm from Listing 4.1 and Listing 4.3. Therefore, the partial solution that represents the changes of a sub-rule application in $G_{\Delta(V_A,V_B)}$ has to be extended.

Create sub-rule context (Line 72 - 73): Initially, the variables of the sub-rule's application context that are to be matched in the intermediate model version V'_A are collected. The application context consists of all graph elements of the edit rule annotated with «preserve» actions (Line 72). In addition, all graph elements that are to be deleted by the complement rule are included as elements to be preserved by the sub-rule application context (Line 73).

Prepare domains (Line 75), (Line 84): The matching of connected subgraphs (see Section 4.3.2) can lead to a sub-rule graph $G_{ER_{sub}}$ consisting of unmatched disconnected subgraphs $G_c \subset G_{ER_{sub}}$. For example, the sub-rule graph illustrated in Figure 4.5 consists of two disconnected subgraphs. However, only the subgraph related to the creation of the transition is mapped to the difference graph. In this context, let $G_{sr} \subset G_{ER_{sub}}$ be the subgraph that is mapped to the difference graph $G_{\Delta(V_A,V_B)}$. Therefore, the domains of the variables representing the disconnected subgraphs $G_c \not\equiv G_{sr}$ are completely restricted, i.e., their domains do not contain any assignable values. To match the application context, we ignore the restrictions of such variables (Line 75), i.e., we allow all values of the variables' domains to be assigned. Finally, such domain restrictions are restored (Line 84) in a backtracking step.

Matching the sub-rule's application context (Line 78 - 79): The findSolution() function in Listing 4.6 generalizes the findSolution() function from Listing 4.3. To compute a complete application context matching, we call the findSolution() function with the prepared sub-rule application context (Line 79). The found matchings are combined (Line 78 - 79) with respect to each graph element (Line 88) of the application context.

Edit Rule	Difference	Sub-Rule Context	Comment
«delete»	«preserve»	«preserve»	to be deleted by the complement rule
«preserve»	«preserve» / «create» / «delete»	«preserve»	including preceding/succeeding changes
«preserve»	«preserve» /«create»	«preserve»	complement rule context nodes

Table 4.2. Compatibility of the complement rule matching.

```
function solutionFound(solution: Stack<VariableSet>, removed: Stack<VariableSet>):
  〉initialize application context:
  context: VariableSet = { v ∈ solution.variables | v.represents.action ≡ «preserve» }
  context = context ∪ { v ∈ removed.variables | v.represents.action ≡ «delete» }
  solution.push(context)
  fullyRestricted : VariableSet = ignoreDomainRestrictions({ v ∈ context | v.assignable ≡ ∅ })
  restrictions : VariableSet = restrictContextDomains(context)
  〉match application context:
  matches: Map<GraphElement, Set> = { context.variables.represents ↦ ∅ }
  if findSolution(context, test=false, vars → contextFound(vars, matches)) then
    deriveComplementRule(subChanges=solution, contexts=matches)
  endif
  〉backtracking step...
  unrestrictContextDomains(restrictions)
  restoreIgnoredDomainRestrictions(fullyRestricted)
  solution.pop()
-----------------------------------------------------------------------------------
function contextFound(context: VariableSet, matches: Map<GraphElement, Set>): Boolean
  if evaluateApplicationCondition(context) then
    return addMatchFromAssignment(context, matches)
  endif
  return false
-----------------------------------------------------------------------------------
function findSolution(vars: VariableSet, test: Boolean, yield: VariableSet→Boolean): Boolean
  exists : Boolean = false                          〉Does at least one assignment exists?
  if hasUnassignedVariables(vars) then              〉select variable and compute assignments...
    v: Variable = selectUnassignedVariable(vars)
    foreach assignment ∈ v.domain.assignable do
      assignVariable(v, assignment)                 〉(D) assign selected variable
      if constraintAssignment(v, vars) then         〉Is the assignment within the solution space?
        exists = findSolution(vars, test, yield) ∨ exists 〉↪ assign next variable in given set
      endif
      freeVariable(v)                               〉(D) backtracking assignment
      test ∧ exists ⇒ break                         〉Only find the first solution?
    endforeach
  else
    exists = yield(vars)                            〉process complete assignment...
  endif
  return exists
```

Listing 4.6. Matching of the application context in the model difference graph.

```
107 recommendations: List<Proposal> = []
108 function restrictContextDomains(context: VariableSet): VariableSet
109   complementChanges: Set<GraphElement> = removed.represents.elements
110   complementContext = { v ∈ context.variables | isContextOf(v.represents, complementChanges) }
111   restrictDomainsByAction(complementContext, «delete» )
112   return complementContext
113 function deriveComplementRule(subChanges: VariableSet, contexts: Map<GraphElement, Set>): Boolean
114   complement: Rule = deriveComplementRule(ER, subChanges.represents.elements)
115   parameters: Map<String, Set> = deriveParameterInput(complement, subChanges, contexts)
116   recommendations.add(Proposal{rule = complement, input = parameters})
```

Listing 4.7. Matching and derivation of the complement rule.

In addition, in the deriveComplementRule() function in Listing 4.7, the actual complement rule is derived, and the matching of the application context is translated into possible parameter bindings.

Application context domain restrictions (Line 108 - 112): Except for the sub-rule's application context, the complement rule must be applicable in model version V_B. Considering the corresponding graph transformation rule, the left-hand side of the complement rule must be mapped to nodes in the difference graph with «preserve» or «create» actions. In particular, the matching of the sub-rule application context and the complement application context can be performed in the one matching step. Therefore, based on the variables representing the sub-rule's application context in Listing 4.7, the restrictContextDomains() function call (Line 76) first determines the variables that represent the complement rule's application context. Next, the domains of those variables are restricted to not contain any graph elements of the difference graph annotated with «delete» actions.

Derive the complement rule (Line 113 - 116): As defined in Section 4.1, we can derive the complement rule from the edit rule by a given set of change actions that represent the sub-rule (Line 114). In addition, the parameter bindings are derived from the application context matching (Line 115). Finally, we store the complement rule and parameter bindings as a new complementation recommendation (Line 116).

As shown in the model excerpt in Figure 4.1 of the model version in Figure 2.2, the sub-rule depicted in Figure 4.5 can be recognized, which creates the transition fastForward in the state machine diagram. As shown in the corresponding complement rule in Figure 4.6, the parameter *newTransition* is bound to the transition fastForward as container of the new trigger. The name of the transition fastForward is set as an input parameter *transitionName* for the operation to be created. The parameter of the *class* can be bound to one of the classes User, Video, or Server in the class diagram in Figure 2.2. The parameter defining the *package* is bound to the VoD-System package containing those classes.

4.4.1 Approximation of Parameter Bindings

The application context to be recognized can consist of multiple disconnected graphs $G_c \subset G_{ER_{sub}}$. During their matching, all possible combinations of matches of the disconnected graphs are computed. For example, for the edit rule *createOperationTriggeredTransition* in Figure 4.2 the source and target state of the new transition can be selected independently. Searching all applications of the complete edit rule in a model, we must compute the product of all states in the model for the application context. In general, this combinatorial computation can lead to runtime problems during the matching of the application context.

For the recommendation proposals, we combine the matchings into lists of input parameter bindings with respect to the complement rule. In the case of the edit rule *createOperationTriggeredTransition*, this simply means we show all states of the model as possible parameter binding of the source and the target state. In general, as an approximation of the parameter bindings, the combinatorial problem can be avoided by separately computing the matchings for each disconnected subgraph $G_c \subset G_{ER_{sub}}$ of the application context. In particular, the overall matching of the application context fails if no matching can be found for one of the subgraphs G_c.

The matching of the disconnected subgraphs G_c is an over-approximation of the parameter bindings if application conditions between the subgraphs have to be checked. If the edit rule only allows injective matchings of its LHS (see Section 3.5.2) this constraint has to be checked for a concrete binding selected by the developer. For example, requiring an injective matching for edit rule *createOperationTriggeredTransition* would mean that no self-transitions can be created.

A similar problem appears for the testing of disjunctive connected graph constraints $a \vee b$ of the edit rule's application condition. Basically, one graph constraint can be considered optional if the other graph constraint is fulfilled. As an over-approximation of the possible parameter bindings, we ignore such constraints and check the complete application condition only for a concrete parameter binding. In general, we could also leave it up to the developer whether to filter such parameter bindings or to make an informed decision to (temporarily) ignore certain edit rule application conditions.

4.5 Validation of Application Conditions

According to Section 3.5.2, an edit rule can define additional application conditions that have to be checked in order to determine the applicability of the rule in a model's ASG G. Given a matching of the application context, the application condition is a logical formula that is checked over atomic graph constraints, i.e., PAC and NAC graph constraints that require or forbid specific graph patterns embedded into the application context. Moreover, the edit rule can require an injective matching, i.e., two nodes of G_{ER} are not allowed to be mapped to the same node in G. Optionally, the dangling condition can be required for the edit rule, i.e., nodes in G can only be deleted if their incident edges are also deleted by the edit rule.

During the sub-rule recognition, for each matching of the application context and graph constraints, the injectivity of the mapping from $G_{ER_{sub}}$ with respect to $G_{\Delta(V_A,V_B)}$ must be checked. The dangling constraint only has to be checked for the nodes to be deleted by the complement rule. By definition, the dangling condition for the nodes deleted by the sub-rule is always fulfilled, i.e., if a node is deleted according to the model difference $\Delta(V_A, V_B)$, then $\Delta(V_A, V_B)$ also contains the deletions of incident edges of this node (see Section 3.4.2).

Let us consider again the recognized sub-rule in Figure 4.5 of the edit rule *createOperationTriggeredTransition* in Figure 4.2. With respect to the model difference in Figure 4.1, the creation of the transition fastForward is detected as a partial edit step. In this application context, the NAC is fulfilled for all classes in the class diagram, i.e., none of the classes contain an operation named fastForward(). The PAC of the class matches the VoD-System package of the class diagram. The PAC of the state is fulfilled for the already matched application context nodes, i.e., the source state streaming is contained in the same region playback as the fastForward transition.

As already discussed in Listing 4.6, for each partial solution that is computed, the function evaluateApplicationCondition() checks the application condition for the recognized sub-rule application. The application condition ac (Line 117) is a logical formula over atomic graph constraints (see solution stack (Line 3) in Listing 4.1). Each graph constraint is represented as a set of variables in the CSP. We first check the dangling condition for the nodes to be deleted by the complement rule (Line 119). Finally, the evaluate() function in Listing 4.8 determines the result of the logical formula ac by checking the graph constraints using the findSolution() function (Line 120). We apply the generalized findSolution() function from Listing 4.3 to check if at least one (test=true) assignment exists for a given variable set representing a graph constraint.

```
117 ac: Formula<VariableSet> = applicationCondition(G_ER, solution)                ⟩ application condition
118 function evaluateApplicationCondition(context: VariableSet): Boolean
119   return checkDangling({ v ∈ context.variables | v.represents.action ≡ «delete» })  ⟩ of complement rule
120          ∧ ac.evaluate(context, graphConstraint → findSolution(graphConstraint, test=true, vars → true))
```

Listing 4.8. Validation of the edit rule's application condition (optionally) including the dangling condition of the matching.

In the extension of the CSP in Listing 4.8, each atomic graph constraint in the application condition formula ac is represented as individual VariableSet. As shown in the CSP data structure in Figure 4.3, a variable set can represent a GraphConstraint, which is a special kind of GraphFragment. A GraphConstraint that contains a NAC is marked as negative = true.

4.5.1 Precondition Graph Constraints

The application context and the PAC graph constraints define graph fragments that must exist in a model's ASG. Similar to the recognition of the sub-rule's application context, the PACs of an edit rule are checked on the union $V_A \cup V_B$ of the old and the new model version. This results in the action compatibility for PAC graph matchings as defined in Table 4.3, i.e., the elements in the edit rule annotated with «require» actions can be mapped to model elements in the difference graph $G_{\Delta(V_A,V_B)}$ annotated with «preserve», «delete», or «create» actions.

In contrast, a NAC graph constraint checks for the absence of a certain graph fragment. During the recognition of a sub-rule according to Definition (4.1), we have to check if the application condition has been fulfilled prior to the execution of the sub-rule. This has to be checked on the intermediate model version V'_A. However, finding V'_A that fulfills all PAC and NAC graph constraints of an edit rule simultaneously requires considering all possible intermediate versions between version V_A and V_B. Therefore, the absence of NAC graph constraints during the sub-rule recognition is approximated based on the minimal intermediate model version V'_A, i.e., the intersection of the model versions $V_A \cap V_B$. As defined in Table 4.3, the graph elements of a NAC can be mapped to the model elements in the difference graph annotated with «preserve» actions.

Edit Rule	Difference	Sub-Rule AC	Comment
«require»	«preserve» / «create» / «delete»	«require»	PAC precondition graph constraint
«forbid»	«preserve»	«forbid»	NAC precondition graph constraint

Table 4.3. The action compatibility of the variables' domains that represent graph constraints that define *preconditions* of the edit rule.

4.5.2 Invariant Graph Constraints

Alternatively, we assume a graph constraint can be configured as an invariant instead of a precondition. In this case, from the perspective of recognizing an edit rule *ER* in a model difference $\Delta(V_A, V_B)$, the application condition of *ER* must be valid for the intermediate model version V'_A and must stay valid for all succeeding model versions. Therefore, the application condition can be checked on model version V_B. In the CSP solver, the variables' domains representing invariant PAC or NAC graph constraints are initialized with the action-compatible element from the difference graph $G_{\Delta(V_A,V_B)}$ as defined in Table 4.4, i.e., the graph elements in $G_{\Delta(V_A,V_B)}$ annotated with «preserve» and «create» actions.

Edit Rule	Difference	Sub-Rule AC	Comment
«require»	«preserve» / «create»	«require»	PAC invariant graph constraint
«forbid»	«preserve» / «create»	«forbid»	NAC invariant graph constraint

Table 4.4. Action compatibility of the variables' domains representing *invariant* graph constraints.

For example, the PAC graph constraints of the edit rule *createOperationTriggeredTransition* in Figure 4.2 can also be interpreted as invariant conditions. As a precondition, the PAC defines that the new transition is created in the region of its source state. As an invariant, it states that a transition must *always* be contained in the region of its source state, i.e., the state cannot be moved without moving its outgoing transitions to another region. Assuming we apply the complete edit rule, the NAC graph constraint, however, is not fulfilled anymore. The NAC checks that an operation with the name of the operation to be created not already exists in the target class. After creating the operation, the NAC would detect that operation. Therefore, the NAC can only be interpreted as a precondition graph constraint of the edit rule.

4.6 Impact of Change Actions

In the context of a recommendation tool, the developer needs to define the context of the recommendation with respect to the task to be performed (see Table 2.1 in Section 2.3). Generally, the context of our approach can be restricted by specifying the model elements that can be potentially modified by the sub-rule and complement rule. As shown in Listing 4.9 extending the CSP solver, this can be achieved by additional constraints with respect to the sub-rule and complement rule.

```
function constraintSolution(): Boolean
  return ...                                                ⟩ extends Listing 4.1
        ∧ hasHistoricalImpact(solution ∪ remaining)         ⟩ in sub-rule!
        ∧ hasComplementaryImpact(removed ∪ remaining)       ⟩ in complement rule!
```

Listing 4.9. Constraint impact of change actions.

4.6.1 Historical Impact of Change Actions

Generally, the recommendation approach of ReVision focuses on the complementations of incomplete edits. The editing history is represented as a model difference that contains concrete, historical changes between two versions of a model V_A and V_B. Let $\Delta_\mu(V_A, V_B) \subset \Delta(V_A, V_B)$ be the set of changes with a historical impact within the current recommendation context, i.e., historical changes that indicate potential incomplete edits. A sub-rule ER_{sub} is considered to have a historical impact if at least one change action of ER_{sub} matches a concrete change in $\Delta_\mu(V_A, V_B)$. Therefore, we first compute the subset of the change actions $ER_\mu \subset ER$ which have a *potential historical impact*, i.e., the change actions that can be initialized to form a concrete change included in $\Delta_\mu(V_A, V_B)$.

For example, let us assume the newly created transition fastForward of *edit step (2.3)* in our running example in Figure 4.1 is selected by the developer as the context for detecting incomplete edits. By collecting the historical changes related to the selected model fragment, the changes of *edit step (2.3)* define $\Delta_\mu(V_A, V_B)$. The change actions $ER_\mu = A_1 \cup A_2 \cup A_3$ of the edit rule *createOperationTriggeredTransition* shown in Figure 4.2 are matching the historical changes of the *edit step (2.3)* as potential historical impact. Therefore, only those 51 sub-rules that contain at least one of the change actions in ER_μ are considered. Moreover, if a larger set of edit rules is processed, all edit rules which have no historical impact will be ignored immediately.

The historical impact is tested as part of the constraintSolution() function in Listing 4.9. The function hasHistoricalImpact() first checks the *potential impact* of the sub-rule. Therefore, we can check if at least one atomic change set with historical impact is contained in the solution or the remaining set, i.e., the maximal possible sub-rule that can be constructed at the current computational state of the CSP solver.

Given a sub-rule and a matching $m : G_{ER_{sub}} \rightarrow G_{\Delta(V_A,V_B)}$ in the difference graph, we can also check for the *actual historical impact* of change actions. Let $\Delta_\mu(V_A, V_B) \subset \Delta(V_A, V_B)$ be a set of concrete historical changes from the model difference $\Delta(V_A, V_B)$. A sub-rule matching m has an actual historic impact if m contains at least one changed graph element of $G_{\Delta(V_A,V_B)}$ that corresponds to change in $\Delta_\mu(V_A, V_B)$.

For example, we assume that the developer has selected the transition fastForward from our running example in Figure 4.1. The historical impact $\Delta_\mu(V_A, V_B)$ contain the historical changes from the *edit step (2.3)*. The changes of $\Delta_\mu(V_A, V_B)$ can be found in the domains of the variables representing the creation of the transition $ER_\alpha = \{A_1, A_2, A_3\}$. In contrast, the sub-rule that creates the transition play in *edit step (2.2)* is filtered by the historical impact.

The function hasHistoricalImpact() returns *true* if at least one of the variables with a potential historical impact also contain elements in their domain with an actual historical impact. Moreover, to reduce the number of backtracking steps of the computation, the selectVariableSet() function can be optimized to prefer change sets with historical impact.

4.6.2 Complementary Impact of Change Actions

Similar to the potential historical impact, the computed applications of a complement rule $\overline{ER}$ may require additional restrictions depending on a specific recommendation task. For example, the recommendation context might be restricted to the currently selected model fragment or a particular inconsistency to be fixed. Let $ER_\alpha \subset ER$ be a subset of the change actions in ER that have a *potential complementary impact.* A complement rule $\overline{ER}$ is considered to have a complementary impact if at least one change action of $\overline{ER}$ matches a change action in ER_α.

Let us again consider the example of assigning a trigger to the newly created transition fastForward in Figure 4.1. Assuming that the developer has selected the transition as a context for the detection and complementation of incomplete edits. Therefore, ER_α is computed by collecting all change actions from ER that are adjacent to the creation of a transition. In the case of the edit rule *createOperationTriggeredTransition* in Figure 4.2, the creation of the transition's trigger is adjacent to the creation of the transition. In terms of the atomic change sets in Figure 4.2, we can denote the complementary impact as $ER_\alpha = A_4$. In this case, 20 sub-rules exist that have a potential historical and complementary impact. In general, considering a larger set of edit rules, an edit rule that has no potential complementary impact can be filtered instantly.

As shown by the CSP solver in Listing 4.1, the complementary impact is checked similarly to the historical impact. The function hasComplementaryImpact() of the constraintSolution() function checks if at least one atomic change set of the complement rule has a complementary impact. The function checks the change actions of the change sets currently represented by the variable sets in the removed and remaining stacks. The change sets that are currently removed from the solution space correspond to the complement rule, and the remaining change sets are not yet decided.

Let Δ_α be a set of abstract change actions (see Section 3.3) that can be initialized and applied to the current model version V_B. Similarly to the historical impact in Section 4.6.1, we can check the change actions of the complement rule $\overline{ER} = ER \setminus ER_{sub}$ for an *actual complementary impact.* A complement rule has an actual complementary impact if the matching $m' : L_{\overline{ER}} \rightarrow V_B$ initializes at least one change action of $\overline{ER}$ that matches an abstract change contained in Δ_α.

We have determined the change actions $ER_\alpha = \{A_4\}$ of the edit rule in Figure 4.2 as potential complementary impact, i.e., the change actions adjacent to the potential historical impact $ER_\mu = \{A_1, A_2, A_3\}$. The complementary impact Δ_α can constraint those change actions to be bound as abstract changes in the context of the selected fastForward transition. In other words, the complement of the partially executed edit step should be performed at the boundary of the selection in the model editor. This condition is fulfilled for the creation of the trigger in the edit rule *createOperationTriggeredTransition*.

The actual complementary impact is also checked by the function call hasComplementaryImpact() function in Listing 4.1. The change actions of the complement rule are represented by the variables in the removed or remaining stack of the CSP solver. The function checks if at least on change action represented by those variables has a complementary impact. Therefore, for each change action with a potential historical impact, a set of impacted model elements can be collected from the context elements of the matching abstract changes in Δ_α. During the assignments in the CSP solver, the domains of the variables that represent the context of a change action with a potential historical impact must be monitored. For example, the context of the creation of the trigger and, in particular, its containment edge in Figure 4.2 is the node newTransition. For a historical impact, at least one of those domains must contain an impacted model element.

4.7 Refinement of Sub-Rules

The described algorithm in Section 4.3 for recognizing sub-rules deals with some limitations to reduce the complexity and numbers of resulting complementation alternatives. However, as discussed below, those limitations can be mitigated by allowing the developer to request omitted sub-rules interactively.

4.7.1 Historically Preserved Elements as Creations

Based on the action compatibility of the sub-rule matching described in Table 4.1 of Section 4.3.1, a «create» action in the edit rule cannot be mapped to a «preserve» action in the difference graph. For example in Figure 4.2, to use an existing operation as a trigger in our running example, another edit rule would be required that preserves that operation. In general, the action compatibility could be extended to also allow such mappings. However, this can result in a vast number of additional sub-rules, possibly overwhelming the developer.

As an interactive solution, we allow the developer to refine an already selected complement rule by computing additional sub-rules. During this refinement step, the action compatibility is relaxed to allow the mapping of «create» actions to «preserve» actions. Moreover, the developer might already select some of the parameters of the complement rule to refine the recommendation context. In our running example, the Video class could be selected to recommend only the contained operations.

4.7.2 Combine Disconnected Sub-rules

As described for the structural graph matching in Section 4.3.2, the structural restriction allows only connected matchings of the sub-rule graph $G_{ER_{sub}}$ matchings in the difference graph $G_{\Delta(V_A,V_B)}$. This prevents the combination of structurally independent changes in the sub-rule that can potentially lead to a combinatorial explosion of the possible sub-rule matchings in the model difference. However, such disconnected sub-rule graphs can be combined in an interactive refinement step. For a selected complement rule and its sub-rule, we search for additional sub-rules within the proposed complement rule.

For example, if the developer selects the new transition fastForward in Figure 4.1 as the context for a complementation, the proposal based on edit rule *createOperationTriggeredTransition* in Figure 4.2 is to create a new operation as a trigger. Assuming the operation fastForward() has already been created manually, the new operation can be recognized from the model difference in a subsequent refinement step. To further restrict the application context of the refined sub-rule, the developer can also select some parameters of the complement rule, e.g., *class* or *package* of the edit rule *createOperationTriggeredTransition*.

4.7.3 Reduce Sub-rules

As described in Section 4.3.3, the matching of a sub-rule graph in a model difference graph must cover the maximal possible number of historical changes. Showing all non-maximal sub-rules might overwhelm the developer with recommendation proposals. Instead, such cases in which the sub-rule is too large might be handled as an interactive refinement of the sub-rule. In the recommendation tool, we allow the developer to shift recognized change actions from the sub-rule to the complement rule. Notably, in terms of deletion actions, this requires determining the elements to be deleted by the complement rule in the current model version V_B.

4.8 Complementation of Multi-rules

As defined in Section 3.5.2, a graph transformation rule that specifies an edit rule can contain nested multi-rules, which implements a "*for all*" semantic to apply recurring transformations within a single rule. Basically, partially executed multi-rules can be recognized in the model difference by the same technique as for regular edit rules.

The nesting of multi-rules can be considered as a tree of embedded rules. For each path in this tree, starting from the topmost kernel rule, an edit rule graph G_{ER} of the nested multi-rules is created that contains the kernel rule and a single instance of all nested multi-rules on that path, i.e., the multi-rules are flattened into a single edit rule graph. Based on the flattened edit rule graphs, possible sub-rules can be recognized in the model difference graph.

For example, consider again Figure 3.21, which shows a multi-rule that deletes a state and all of its incoming and outgoing transitions. From this multi-rule, we derive two flattened edit rule graphs for the incoming and outgoing transitions, respectively. In this case, the multi-rules have to be considered separately to not recognize the product of all sub-rules with respect to partially removed incoming and outgoing transitions.

Assuming that from Version 1 to Version 2, shown in Figure 2.1 and Figure 2.2, the state named previews is deleted without removing the outgoing transitions play and pause. Based on the edit operation *deleteStateWithTransitions* in Figure 3.21, we can recognize a partial execution of the multi-rule *deleteStateWithTransitions/out* in the model Version 2 illustrated in Figure 2.2. In this case, the kernel rule and the multi-rule deleting the incoming transition *deleteStateWithTransitions/in* are fully executed, i.e., the deletion of the state previews, including the incoming transition with the trigger after(15 sec). The multi-rule that deletes the outgoing transitions is only executed partially, i.e., only the edges in the ASG determining the source state of the transitions are already deleted from the objects play and pause. In this case, one partially executed sub-rule is found for the flattened edit rule graph G_{ER} of multi-rule *deleteStateWithTransitions/out*. The resulting complement rule deletes the remaining transitions play or pause by selecting them one after another as input parameters, i.e., the same complement rule is executed twice with different parameter bindings.

For a multi-rule with a tree-like embedding structure, this approach results in a single complementation proposal for each partial execution with respect to different paths in that tree. In general, multiple recognized sub-rules contribute to the same partially executed edit rule if they share the same recognized changes with respect to their kernel rule(s). For future work, a recommendation tool could also group such complement rules with a common sub-rule into units to execute all complementations in one step.

4.9 Rollback of Partially Executed Edit Operations

In contrast to the complementation of partially executed edits, the changes of a partially recognized edit step can also be rolled back. The rollback might be done at the developer's discretion, e.g., in a situation where no reasonable complementation exists. In order to construct a rollback, the change actions of the sub-rule must be inverted. Therefore, in the sub-rule ER_{sub}, the «create» actions are replaced by «delete» actions, and vise versa. This includes undoing attribute changes performed by the sub-rule.

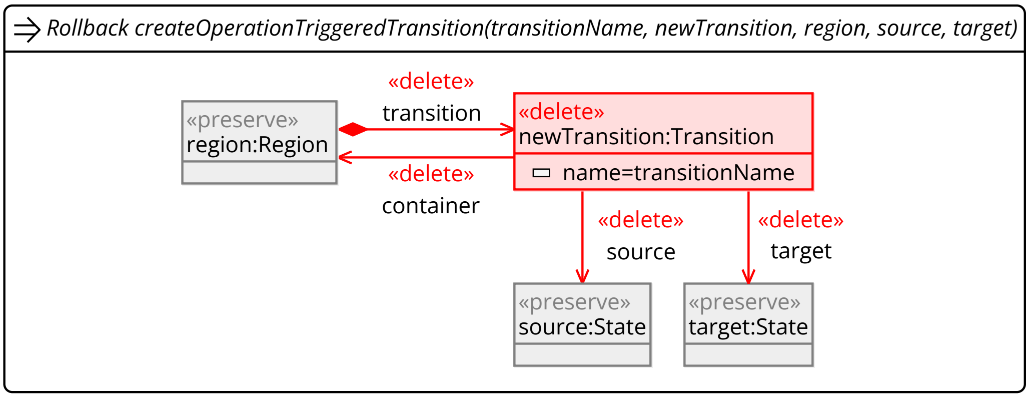


Figure 4.9. Rollback rule of the edit rule *createOperationTriggeredTransition* in Figure 4.2.

As discussed for the reconstruction of edit sequences in Definition (4.1), a sub-rule can have preceding and succeeding edit steps performed between the versions V_A and V_B of a model. If changes in a succeeding edit step b in the model difference depend on changes in ER_{sub}, those changes must also be rolled back. For a given set of changes of the difference, the dependent changes can be computed according to the elementary dependencies defined in Section 3.3.3. Those dependent changes lead to side effects of the rollback, e.g., a model element deleted by a sub-rule must be inserted in V_B, including its attribute values defined in V_A. In such cases, the developer should be informed about the dependent changes and side effects of the rollback.

Figure 4.9 shows the rollback rule of the newly created transition fastForward. The rollback contains only the boundary nodes of the change actions. In this case, the rollback has no side effects. Moreover, the parameter *newTransition* of the rollback rule is bound to the transition fastForward, including the parameters of the corresponding application context.

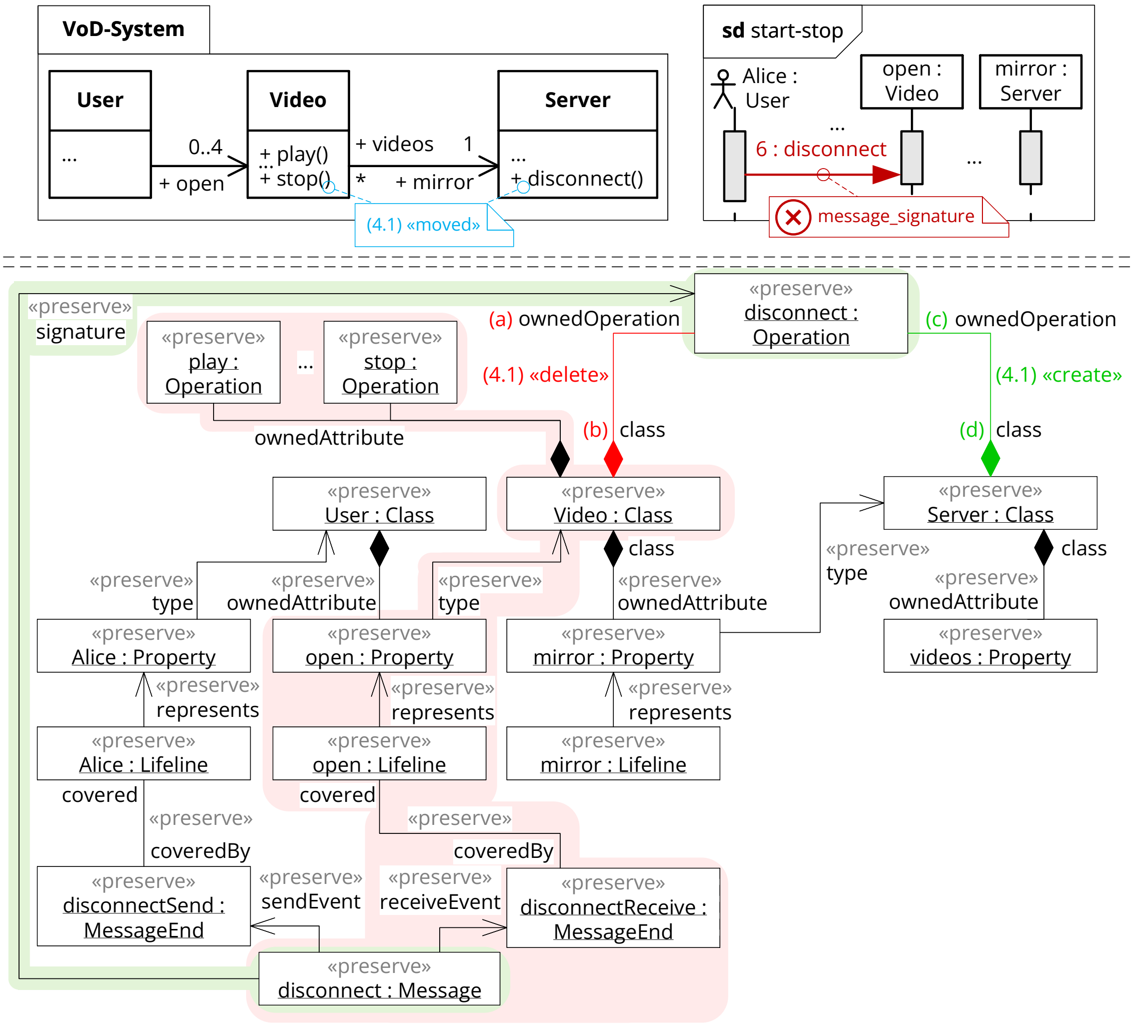
5

History-based Model Repair

Models in MDE are primary development artifacts that are heavily edited in all software development stages and can become temporarily inconsistent during editing. Model repair tools can support developers by proposing a list of the most promising repairs. Such repair recommendations will only be accepted in practice if the generated proposals are plausible and understandable and the set as a whole is manageable. The main idea of our approach is to consider CPEOs as ideal edit operations and to recommend the "gap" between ideal edits and the edits which have caused an inconsistency as model repairs. Based on the approach for detecting and complementing partial executions of edit operations described in the previous Chapter 4, incomplete edit steps are detected in the model history and can be either undone or extended to the full execution of a CPEO. Technically, we recognize sub-rules that could have caused the inconsistency. At the same time, the resulting complement rule must repair the inconsistency.

This chapter presents our interactive repair approach for models, which exploits information about the editing history of a model to generate meaningful repair recommendations. In MDE, models are the primary development artifacts that are frequently edited. During this development process, a model can become inconsistent by temporarily violating the consistency rules of the modeling language. Inconsistencies are particularly likely to happen in systems that are modeled from several viewpoints, which is a widespread paradigm in managing the complexity of large-scale software-intensive systems. Such large system models are often edited in distributed teams. Moreover, in a development team that covers multiple disciplines, different developers can be responsible for designing different system views. During its evolution, a model is exposed to various potential sources of inconsistency. Especially in the early design phases of a software project, some requirements may be undecided or still need to be understood [129]. Furthermore, during the ongoing evolution of a software project, multiple versions or variants of a model can be developed in parallel branches. Such branches must eventually be merged, and changes must be propagated to local workspaces [81].

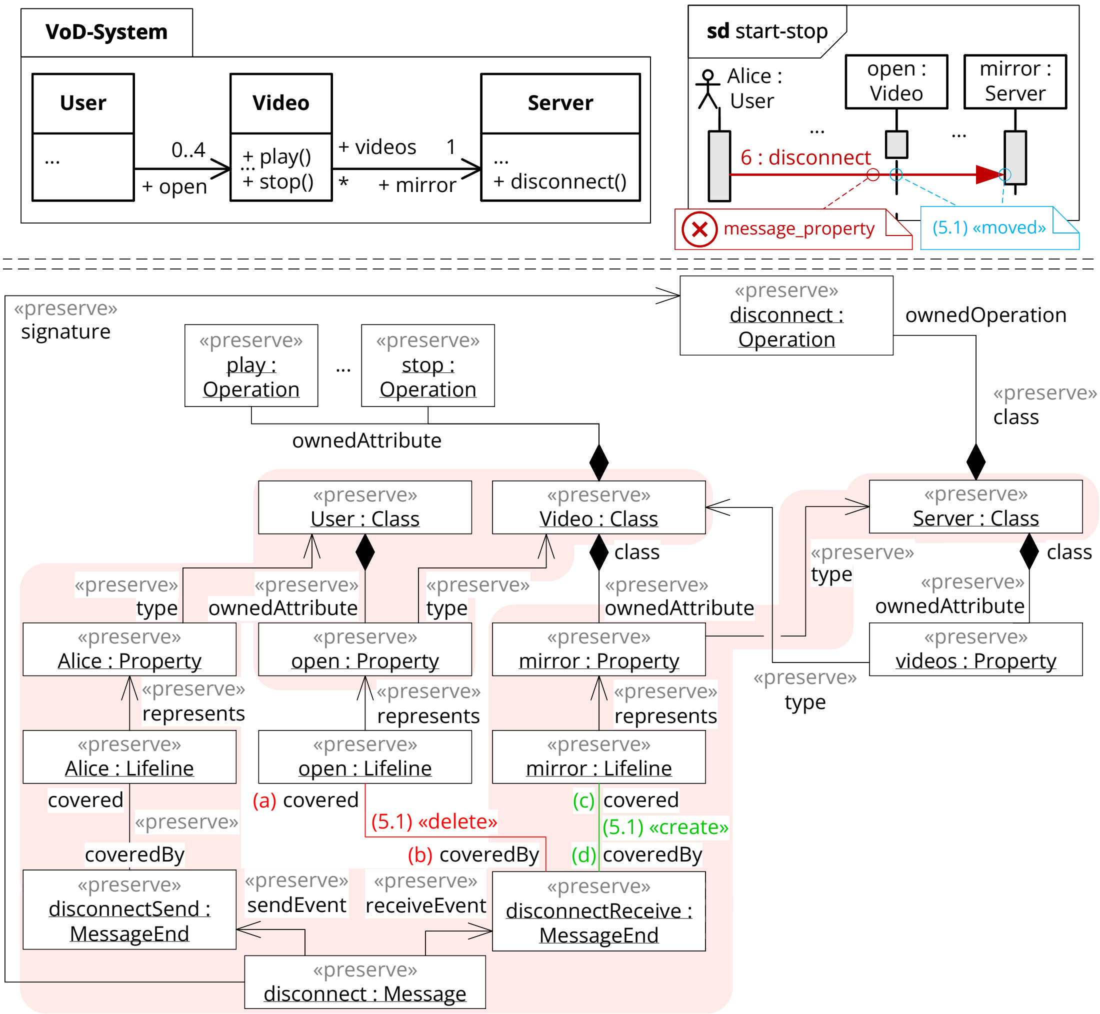
The VoD-System introduced in Section 2.1 is an example of a software system that is modeled from different viewpoints. The system's static structure is defined by the class diagram in Figure 2.3a. In addition, the sequence diagram in Figure 2.3c depicts a "start



message_signature(m:Message): The signature of a message in a sequence diagram must be identical to an operation of the class that defines the type of the receiving lifeline.

$$
\begin{aligned}
&\textit{message\_signature(m:Message)} \;:= \\
&\quad \text{m.signature.name} \equiv \text{m.name} \;\rangle\,\text{(a)} \\
&\quad \wedge \\
&\quad \text{m.receiveEvent.covered.represents.type.ownedOperation} \;\rangle\,\text{(b)} \\
&\quad\quad \rightarrow \exists\, \text{o} \mid \text{m.signature} \equiv \text{o} \;\rangle\,\text{(c)}
\end{aligned}
\tag{5.1}
$$

Figure 5.1. **Version 4 with difference to Version 3:** Excerpt of the unified difference graph $G_{\Delta(V_A,V_B)}$ between the model Version 3 in Figure 2.3 and Version 4 in Figure 2.4 of the VoD-System.



message_property(m:Message): For a message in a sequence diagram, the sender's class must be able to reference objects of the receiver's class by a property.

$$
\begin{aligned}
&\textit{message\_property(m:Message)} \;:= \\
&\quad \text{m.receiveEvent.covered} \rightarrow \exists\, \text{rl} \mid \text{m.sendEvent.covered} \\
&\quad\quad \rightarrow \exists\, \text{sl} \mid \text{sl.represents.type.ownedAttribute} \rightarrow \exists\, \text{p} \mid \text{p.type} \equiv \text{rl.represents.type}
\end{aligned}
\tag{5.2}
$$

Figure 5.2. **Version 5 with difference to Version 4:** Excerpt of the unified difference graph $G_{\Delta(V_A,V_B)}$ between the model Version 4 in Figure 2.4 and Version 5 in Figure 2.5 of the VoD-System.

and stop the video stream" use case scenario. In particular, the class diagram describes the possible operations that can be invoked by messages between objects in the sequence diagram. In this context, the sequence diagram must conform to the following exemplary consistency rules.

message_signature(m:Message): The formal description of the consistency rule *message_signature(m:Message)* is expressed in Definition (5.1). The type of its context is a Message of a sequence diagram (see Figure 3.3). The logical expression is a conjunction of two parts. The first part (see Definition (5.1a)) checks that there is an operation serving as the signature for the message. The operation must have the same name as the message. In the second part (see Definition (5.1a, 5.1b)), the consistency rule checks the lifelines that receive the message (m.receiveEvent.covered). The lifeline represents an instance of a class (represents.type) that is located in a property. This class must contain the operation (o) that is specified as the signature of the message (m.signature).

message_property(m:Message): The formal specification of a second exemplary consistency rule *message_property(m:Message)* is shown in Definition (5.2). This rule states that for a message between two lifelines (rl and sl), the classes of the sender (sl.represents.type) must contain a property (p) with the type (p.type) of the receiver (rl.represents.type). In other words, sending a message can also be read as an operation call. In this context, it must be possible for the calling object to store a reference of the called object. As shown in the class diagram in Figure 2.3a, such a referencing property can be included and illustrated by a navigable association between two classes.

As defined in Section 3.2, the consistency rules are evaluated for all elements on a model's ASG that match the type of the rule's context. Validating the consistency rules *message_signature(m:Message)* and *message_property(m:Message)* on Version 3 in Figure 2.3, no such inconsistencies are found with respect to the sequence diagram. Now let us consider again the modeling scenarios described in Section 2.1.2 and Section 2.1.3, which introduce and resolve two inconsistencies between the sequence and class diagram. The scenario comprises three subsequent edit steps on the VoD-System model.

Inconsistency-inducing edit step (4.1): First, we edit only the class diagram by moving the operation disconnect() from class Video to class Server, producing Version 4 shown in Figure 2.4. The validation of the consistency rule *message_signature(m:Message)* will fail now for the message 6:disconnect as illustrated in Figure 2.4b. Figure 5.1 shows an excerpt of Version 4 and the corresponding difference graph between Version 3 and Version 4 of the model. In the ASG of the difference graph, moving the operation disconnect() is described by the deletion and creation of the containment references between the disconnect() operation and the Video and Server class.

First corrective edit step (5.1): Next, a complementary modification in the sequence diagram is required. The message 6:disconnect, sent from the lifeline Alice:User to the lifeline open:Video, must now be received by an object of class Server that owns the operation disconnect(), e.g., the objects represented by the lifelines main:Server and mirror:Server.

To resolve the inconsistency, we perform a *corrective edit step* in the sequence diagram. We change the target of the message 6:disconnect to the lifeline mirror:Server as shown in Figure 2.5, which we will refer to as Version 5. The changes of the edit step are illustrated in the excerpt of the difference graph in Figure 5.2, i.e., the changing of the target end of the message 6:disconnect.

Second corrective edit step (6.1): The first consistency rule is now satisfied. However, our first corrective edit step has introduced a new defect. As illustrated in Figure 2.5b, the consistency rule Definition (5.2) is violated now. In fact, the sending and receiving lifelines, namely, Alice:User and mirror:Server, are not connected by an association or property in the class diagram. This inconsistency can be fixed in various ways, one option is to change the source of the message from lifeline Alice:User to lifeline open:Video as shown in Figure 2.6, which we will refer to as model Version 6.

The described evolution scenario is a typical example of how the editing of dependent views in isolation causes inconsistencies. According to the requirements on repair recommendations analyzed in Section 1.2, the most viable approaches to resolve such inconsistencies are recommender systems, which interactively support the developer in resolving the inconsistencies.

In general, a recommender system determines and suggests a ranked list of repair proposals from which the developer can choose. For example, the violation of consistency rule *message_signature(m:Message)* on context element 6:disconnect could be alternatively resolved by (i) moving the operation disconnect() from class Server to class Video, (ii) creating a new operation disconnect() in class Video serving as the signature for message 6:disconnect, or (iii) deleting the message 6:disconnect. In general, each of these repair alternatives above would be considered highly relevant by a recommender system since, starting from the inconsistent model, they employ only a single change. However, considering the development history of our example, these alternatives are unlikely to meet a developer's intention. Alternative (i) can be immediately spotted as undo operation. Alternative (ii) seems bizarre since operation disconnect() is recreated in class Video after having been removed from this class in the previous step. Finally, alternative (iii) is not very likely, too. We know that the functionality offered by operation disconnect() still exists, though offered by another class, so why should we delete the message disconnect(), ending up in an incomplete "start and stop the video" scenario?

By having a look at the initial violation of the consistency rule *message_signature(m:Message)* at message 6:disconnect, developers may quickly understand the correction of the target of that message, while they are likely to wonder about the rationale of changing the message's source at the same time. The second change is better explained in a two-step process as described by the corrective *edit steps (5.1)* and *(6.1)*. This is a typical example of resolving only one inconsistency at a time and dealing with the potential side effects of a fix in a separate step.

Our model repair tool ReVision considers a model's editing history and supports the developer to iteratively repair each individual violation of a consistency rule. The approach utilizes CPEOs as ideal edit operations and recommends the "gap" between ideal edits and

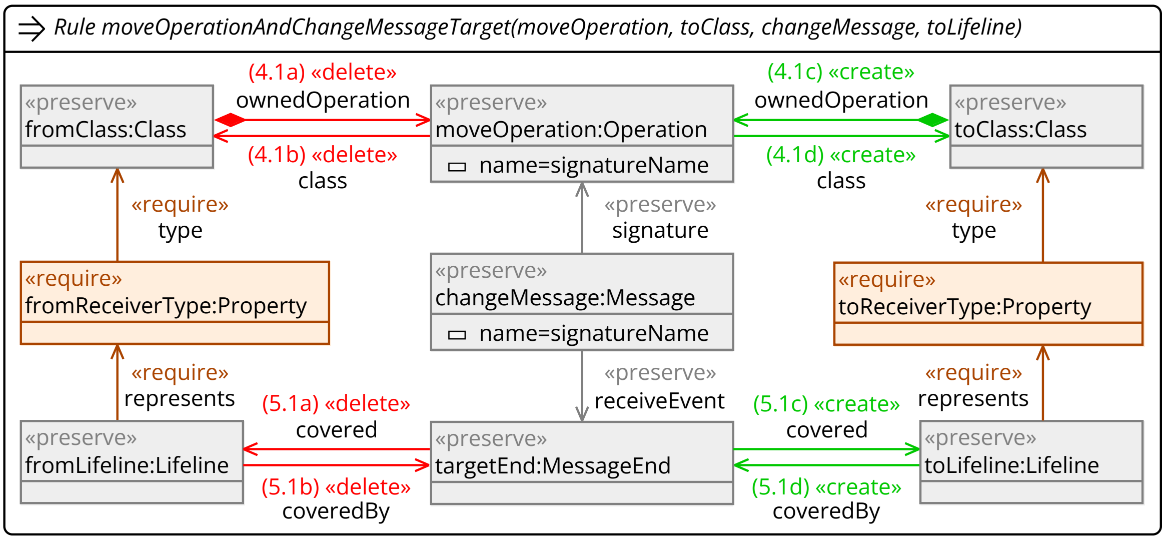


Figure 5.3. CPEO *moveOperationAndChangeMessageTarget*: Moving an operation in a UML class diagram from one class to another and simultaneously changing the receiver of a corresponding message in a sequence diagram.

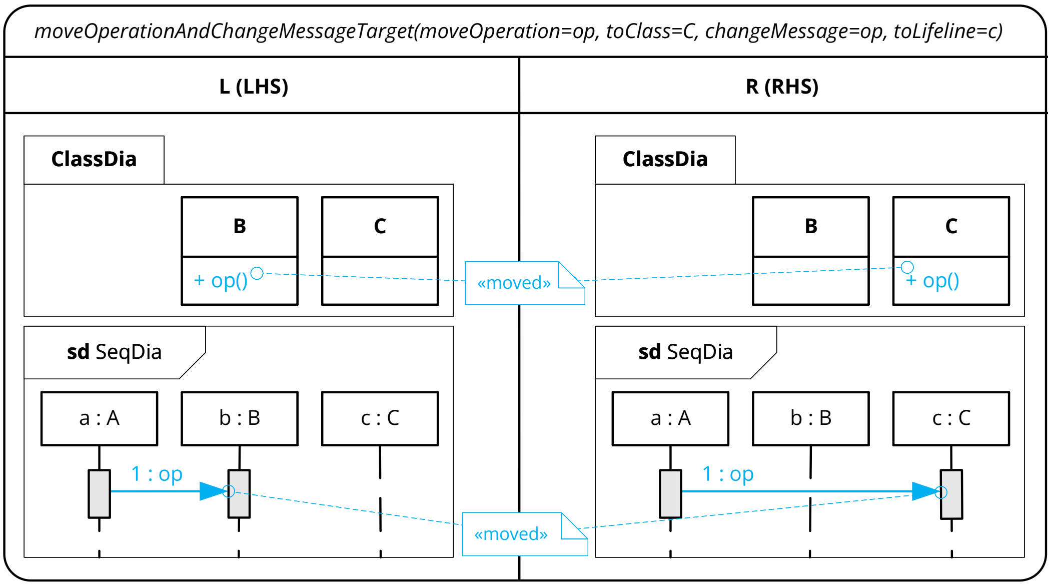


Figure 5.4. Exemplary illustration of the changes from CPEO *moveOperationAndChangeMessageTarget* in Figure 5.3 using the concrete UML diagram syntax.

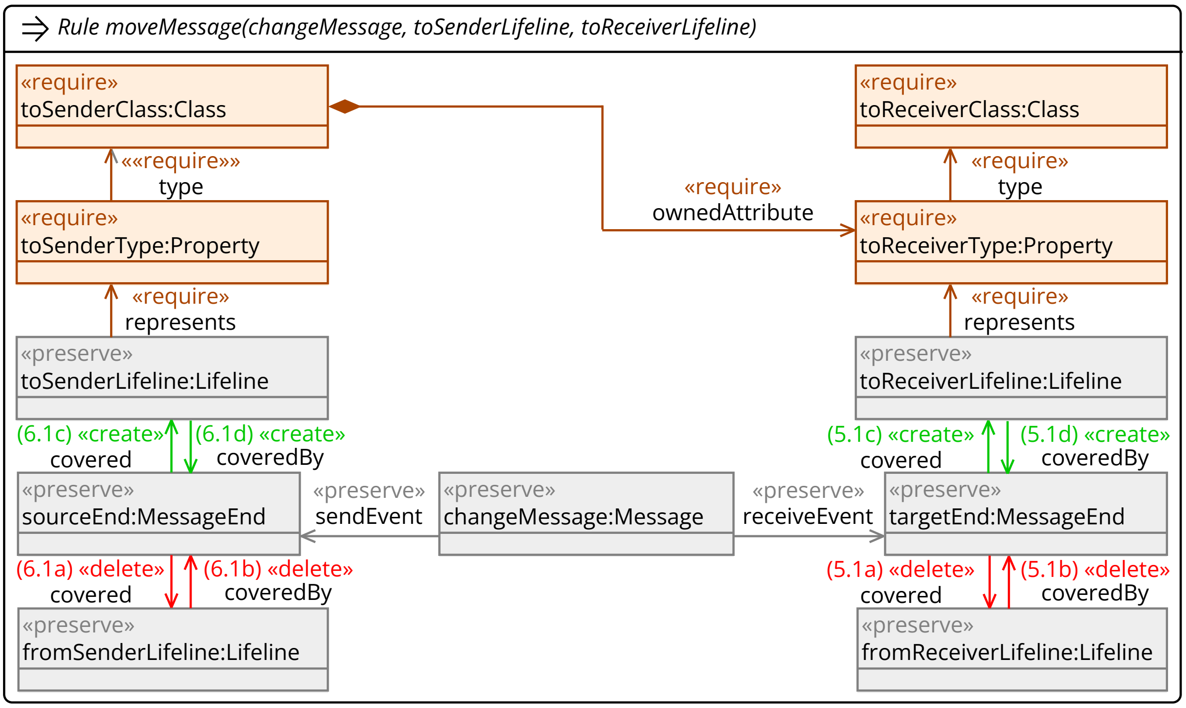

Rule moveMessage(changeMessage, toSenderLifeline, toReceiverLifeline)
«require» toSenderClass:Class
««require»» type
«require» toSenderType:Property
«require» represents
«preserve» toSenderLifeline:Lifeline
(6.1c) «create» covered
(6.1d) «create» coveredBy
«preserve» sourceEnd:MessageEnd
«preserve» sendEvent
«preserve» changeMessage:Message
«preserve» receiveEvent
«preserve» targetEnd:MessageEnd
(6.1a) «delete» covered
(6.1b) «delete» coveredBy
«preserve» fromSenderLifeline:Lifeline
«require» ownedAttribute
«require» toReceiverClass:Class
«require» type
«require» toReceiverType:Property
«require» represents
«preserve» toReceiverLifeline:Lifeline
(5.1c) «create» covered
(5.1d) «create» coveredBy
(5.1a) «delete» covered
(5.1b) «delete» coveredBy
«preserve» fromReceiverLifeline:Lifeline

Figure 5.5. CPEO *moveMessage*: Moving a message's source and target end in a UML sequences diagram between lifelines.

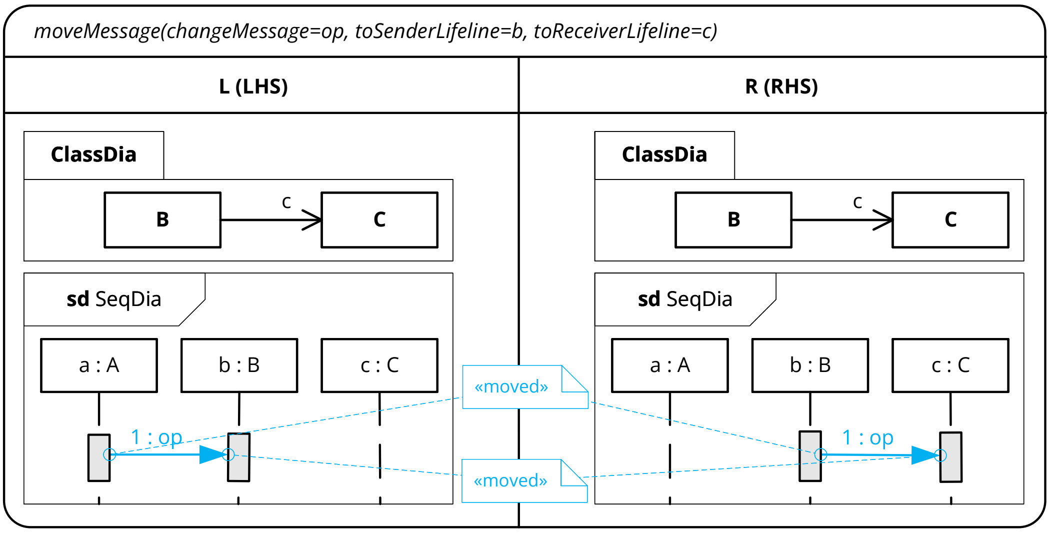


Figure 5.6. Exemplary illustration of the changes from CPEO *moveMessage* in Figure 5.5 using the concrete UML diagram syntax.

the edits which have caused an inconsistency as model repairs. Based on the approach described in the previous Chapter 4, partial executions of CPEOs are detected in the model history and can be either undone or extended to the full execution of a CPEO. Our exemplary evolution scenario, namely, the *edit steps (4.1)*, *(5.1)*, and *(6.1)*, can be described by two CPEOs.

moveOperationAndChangeMessageTarget: Figure 5.3 shows the CPEO *moveOperationAndChangeMessageTarget*. Its edit rule graph specifies the move of an operation from one class to another (upper part of the rule). At the same time, the receiver lifeline of a message using this operation as a message signature is changed accordingly (lower part of the rule). Binding the parameters *moveOperation* and *toClass* applies the CPEO onto a concrete operation in a class diagram. The parameters *changeMessage* and *toLifeline* specify a concrete message in a sequence diagram. Notably, the targetEnd:MessageEnd node is uniquely determined during the matching of the edit rule's application context. To give an idea of the specified transformation, Figure 5.4 illustrates the CPEO's changes by an exemplary LHS and RHS of the edit rule (see Section 3.5.2) using the concrete UML diagram syntax.

This CPEO describes the compound effect of the inconsistency-inducing *edit step (4.1)* and the first corrective *edit step (5.1)* in our modeling scenario, i.e., the moving of the operation disconnect() from the class Video to the class Server, and the changing of the target of message 6:disconnect to the lifeline mirror:Server. Therefore, applying the CPEO with the input parameter binding *moveOperation* = disconnect(), *toClass* = Server, *changeMessage* = 6:disconnect, and *toLifeline* = mirror:Server to model Version 3 would produce the revised model Version 5 without violating the consistency rule *message_signature(m:Message)*.

moveMessage: The CPEO *moveMessage* in Figure 5.5 moves a message in a UML sequence diagram between two lifelines. According to consistency rule *message_property(m:Message)*, the edit rule requires that the type of the sending lifeline contains an attribute referencing objects of the receiving lifeline. As illustrated in Figure 5.6, such an attribute is typically defined as an association between the classes defining the lifeline's types.

Applying the CPEO *moveMessage* with the input parameter binding *changeMessage* = 6:disconnect, *toSenderLifeline* = open:Video, and *toReceiverLifeline* = mirror:Server to model Version 3, the inconsistency with respect to the consistency rule *message_signature(m:Message)* would be avoided, i.e., the execution of the *edit steps (5.1)* and *(6.1)* of our modeling scenario. Notably, to produce the model version Version 6 without temporary inconsistencies, the CPEOs *moveOperationAndChangeMessageTarget* and *moveMessage* must be combined with respect to the changing of the message's target end.

Initially, ReVision expects the developer to select the inconsistency to be resolved. After generating a ranked list of repair proposals, the developer selects and applies the repair. It may be necessary that the developer binds certain input parameters that cannot be uniquely determined by ReVision. Figure 5.7 depicts an iteration of the overall repair process, focusing on the involvement of developers by indicating the different options for manual intervention. The successive computation steps of repairing a single inconsistency, including their intermediate results, are illustrated in Figure 5.8 and Figure 5.9.

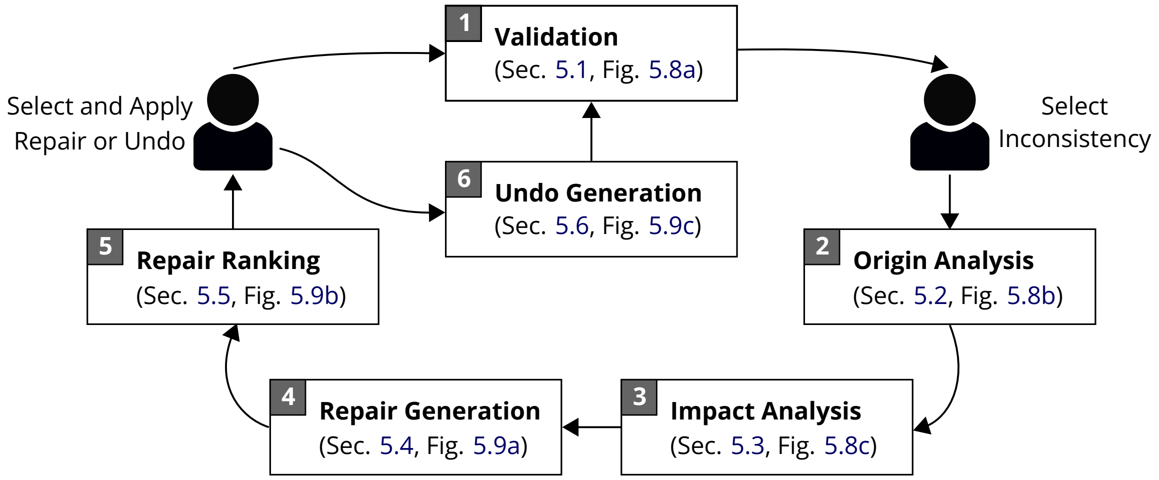

1 Validation (Sec. 5.1, Fig. 5.8a)
Select Inconsistency
2 Origin Analysis (Sec. 5.2, Fig. 5.8b)
3 Impact Analysis (Sec. 5.3, Fig. 5.8c)
4 Repair Generation (Sec. 5.4, Fig. 5.9a)
5 Repair Ranking (Sec. 5.5, Fig. 5.9b)
Select and Apply Repair or Undo
6 Undo Generation (Sec. 5.6, Fig. 5.9c)

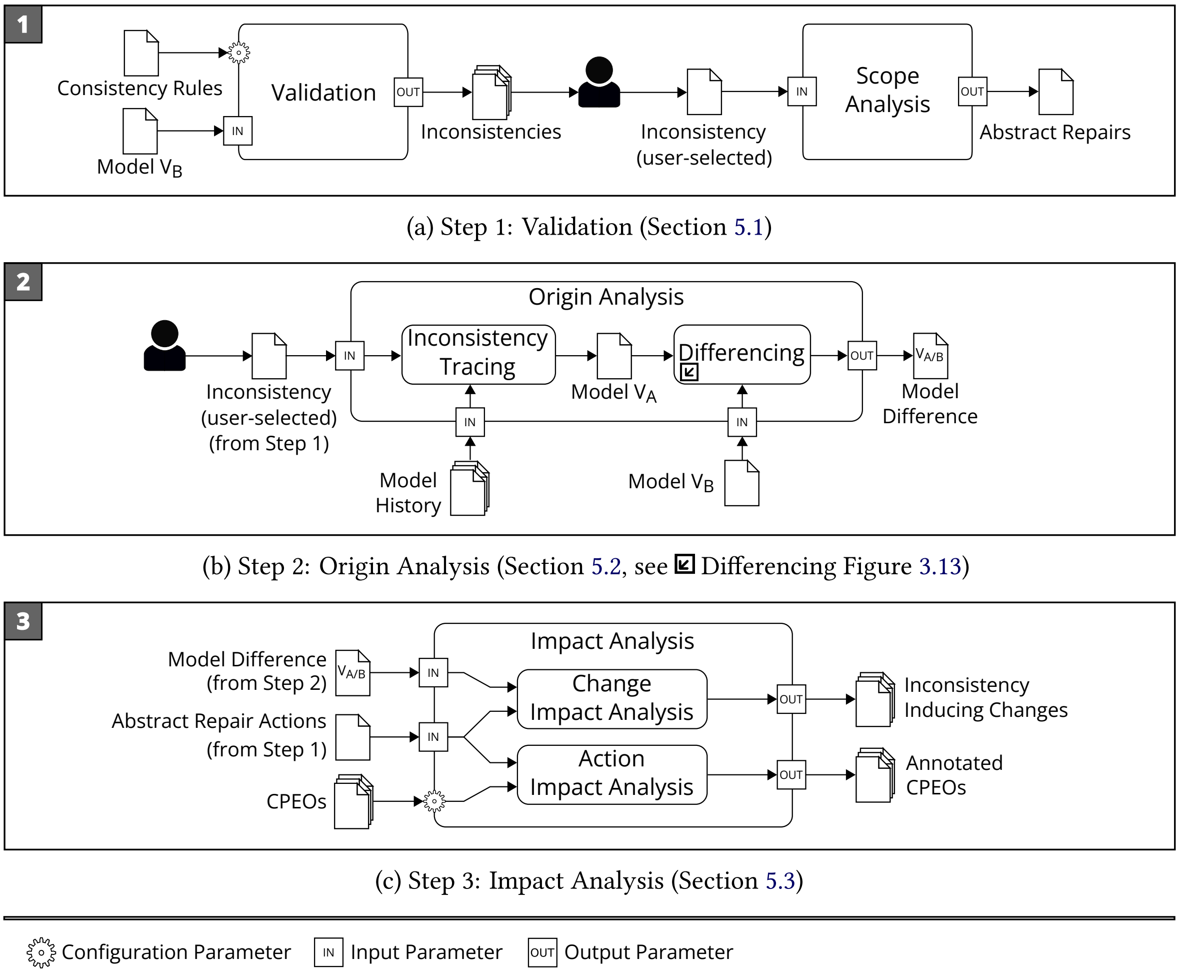
Figure 5.7. Overview of the approach: Overall, iterative process and options for manual intervention.

In Step 1 (see Figure 5.8a), the potentially inconsistent version V_B of a model, typically the latest version, is validated against the set of predefined consistency rules that are to be evaluated on that model. As described in Section 3.2, a validation tool delivers a set of all detected inconsistencies. Initially, the developer selects the concrete inconsistency that is to be fixed in the current iteration. As an initial step, the evaluation of the selected inconsistency is analyzed to determine possible starting points for repairing the version V_B of the model (details see Section 5.1).

Step 2, referred to as *origin analysis* (see Figure 5.8b), traces the selected inconsistency in the model history back to the latest version V_A in which this inconsistency does not occur. It also calculates the model difference $\Delta(V_A, V_B)$ comprising all the changes from V_A to V_B (details see Section 5.2).

Step 3 (see Figure 5.8c) takes the selected inconsistency from Step 1, the difference $\Delta(V_A, V_B)$ calculated in Step 2 and the set of predefined CPEOs as input, and carries out two kinds of *impact analyses* (details see Section 5.3). The *change impact analysis* determines those changes in $\Delta(V_A, V_B)$ that may have caused the inconsistency. They are referred to as the set of inconsistency-inducing changes. The *action impact analysis* determines those change actions in the predefined CPEOs that may have a positive impact on the inconsistency under consideration. As a result, CPEOs are annotated with the information on potentially consistency-improving change actions.

Next, the Step 4 and Step 5 are dedicated to the generation of repair recommendations (see Figure 5.9a and Figure 5.9c). Given the selected inconsistency, the difference $\Delta(V_A, V_B)$ calculated in Step 2 as well as the inconsistency-inducing changes and annotated CPEOs determined in Step 3 as input, we first generate a set of repair proposals that complement partial edit steps. This is achieved by searching for partial executions of CPEOs where (i) the partial execution leads to inconsistency-inducing changes, and (ii) complementing the partial execution to a full CPEO has a positive impact on the inconsistency that is to be fixed



(a) Step 1: Validation (Section 5.1)

(b) Step 2: Origin Analysis (Section 5.2, see ↙ Differencing Figure 3.13)

(c) Step 3: Impact Analysis (Section 5.3)

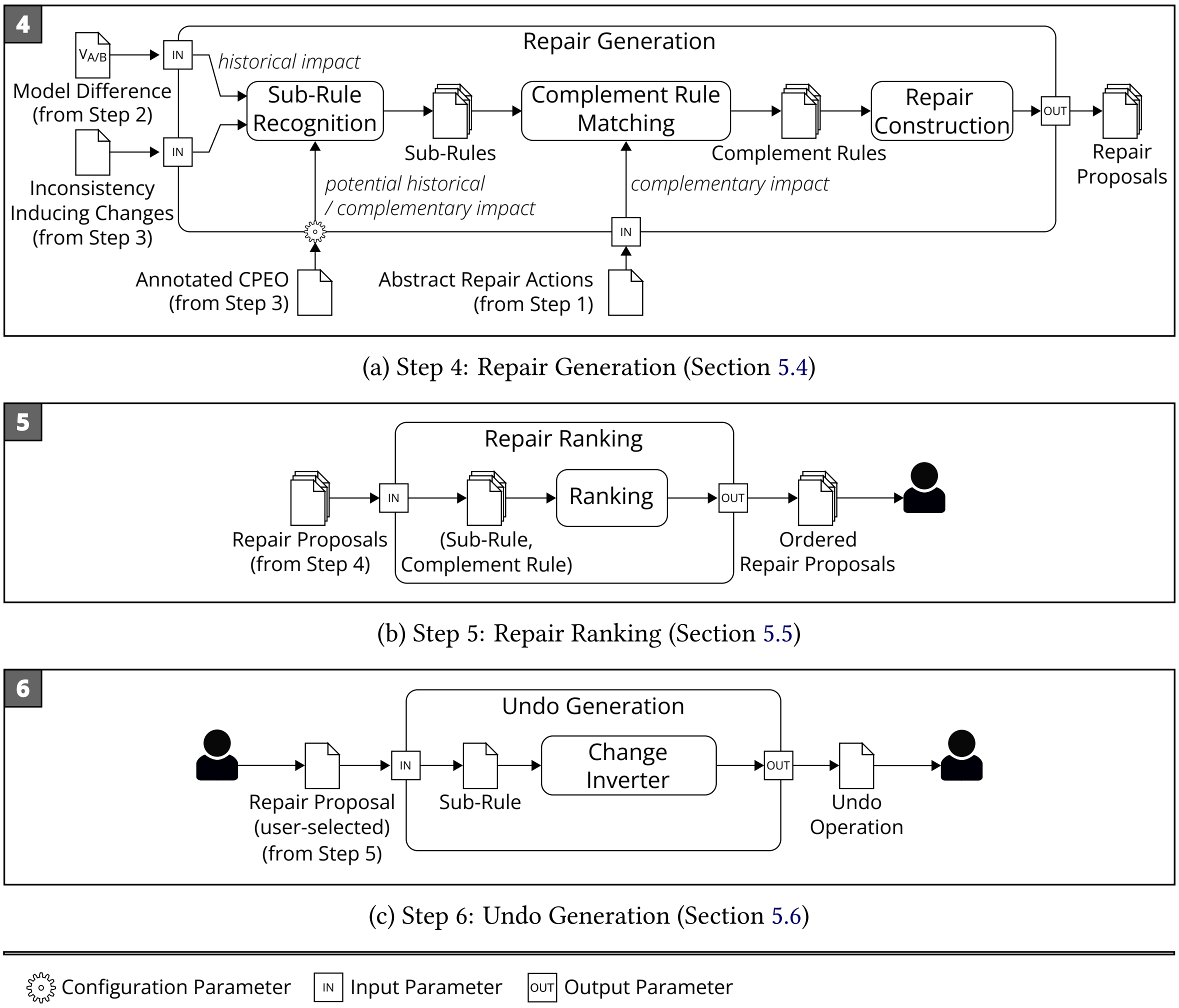
Figure 5.8. Overview of the approach: Successive, automated steps 1 - 3 of a single iteration, including their producer/consumer dependencies.

(details see Section 5.4). Finally, the set of repair proposals is further processed in Step 5 (see Figure 5.9b) to produce a ranked list of repair recommendations, sorted by relevance (details see Section 5.5).

Instead of complementing an incomplete edit step, a developer might want to undo the inconsistency-inducing changes. Re(pair)Vision proposes repairs as pairs of a sub-rule and a complement rule corresponding to the inconsistency-inducing and complementing changes, respectively. Step 6 (see Figure 5.9c) can optionally be triggered at the discretion of the developer to generate a case-specific undo operation. The developer can choose one of the recognized partial executions of a CPEO from the list of repair proposals (details see Section 5.6).

5.1 Validation

The first step of our repair process illustrated in Figure 5.7 and Figure 5.8a is to validate the model against a predefined set of consistency rules. As defined in Section 3.2, a consistency



(a) Step 4: Repair Generation (Section 5.4)

(b) Step 5: Repair Ranking (Section 5.5)

(c) Step 6: Undo Generation (Section 5.6)

Figure 5.9. Overview of the approach: Successive, automated steps 4 - 6 of a single iteration, including their producer/consumer dependencies.

rule is defined for a specific context type of the modeling language. The consistency rule validates a logical expression for all model elements of the specified context type and compatible subtypes. The result is a list of inconsistencies from which the developer selects the one to be repaired.

Let us consider again the excerpt of the class and sequence diagram illustrated in Figure 5.1. It refers to Version 4 of the evolution of the VoD-System shown in Figure 2.4, after moving the operation disconnect() from class Video to Server and before resolving the inconsistency by a complementary edit step in the sequence diagram. This state of the model violates the consistency rule *message_signature(m:Message)* from Definition (5.1) for the message 6:disconnect.

The changes that introduced the inconsistency are annotated in the difference graph in Figure 5.1. Starting from the message 6:disconnect, Figure 5.1 shows the validation scope (see Section 3.2) of the consistency rule *message_signature(m:Message)*, i.e., the set of nodes, edges, and attributes accessed during the validation (see Section 3.2). The green box shown in the background indicates the validation scope of the first part of the consistency rule in Definition (5.1a) with respect to the conjunction of checking the message's names and the receiving lifeline. The red box indicates the validation scope of the second part of the rule in Definition (5.1b, 5.1c). While the first part evaluates to true, the second part evaluates to false since the required operation disconnect() has been removed from the receiving Video class for message 6:disconnect.

In the following sections, we will analyze the violation of a consistency rule on a specific model element to identify starting points for the repair of the indicated inconsistency. This processing step is referred to as *scope analysis* in Figure 5.8a.

5.1.1 Abstract Repairs

A *concrete repair plan* is a sequence of concrete changes (see Section 3.3.1) that describes a transition between an inconsistent and a consistent version of a model with respect to a specific inconsistency. The repair plan is *minimal* if none of the changes can be removed from the plan without resulting in an intermediate, still inconsistent model state, i.e., no subset of the repair plan's changes resolve the inconsistency.

When generating repair proposals, in some cases, it is not possible to provide concrete repair plans without producing a huge number of alternatives. For example, if the negative value of a numerical attribute must be repaired by setting it to any positive value. A method to avoid such countless repairs are so-called *abstract repairs*, as, e.g., utilized in References [95, 134, 149], that have unbound parameters.

Such abstract repairs can be understood as starting point for concrete repair plans. Our approach takes advantage of abstract repairs for determining the relation between partially executed edit operations detected from a model's history and the specific inconsistency to be repaired. Generally, not all partially executed edit operations may be related to the selected inconsistency. Moreover, the abstract repairs allow the recognition of sub-rules, as described in Chapter 4, to focus on a specific part of the model, significantly lowering the complexity of the graph matching problem in terms of large models. Assuming a complete and minimal set of abstract repairs with respect to a specific inconsistency, we use the abstract repairs to determine whether a given a set of changes has an improving effect related to that inconsistency.

Technically, we define abstract repairs as abstract changes (see Section 3.3.1), which are elementary modifications of model elements, attributes, or references in the model's ASG. An abstract change only binds the context of the change action to a concrete model element. For example, an abstract change may specify the model element of an attribute to be modified without providing a specific value for the attribute. In this context, an *abstract repair plan* can consist of concrete changes and abstract changes at the same time.

We specify abstract repairs according to the meta-model in Figure 3.8 of Section 3.3 by the functions defined in Listing 5.1. Those functions instantiate an abstract change with a given *context* model element and the reference or attribute type to be created, deleted or modified, respectively. Furthermore, Listing 5.1 defines the initialization of (abstract) object changes. In particular, an object and reference modification is split into a corresponding creation and deletion change.

The scope analysis illustrated in Figure 5.8a takes the inconsistency to be repaired and

```
function change($\rho$:RepairHint, context:EObject, type:EStructuralFeature): Set<AbstractChange>}
  changes:Set<AbstractChange> = $\emptyset$
  context $\in$ meta-model $\Rightarrow$ return changes          〉 The meta-model is considered to be constant.
  if type.eClass $\equiv$ EReference then                        〉 Create an abstract reference change.
    if $\rho \equiv$ create $\vee$ $\rho \equiv$ modify then
      changes $\cup$ AbstractReferenceChange{action = ChangeKind::create, context = context, type = type}
    endif
    if ($\rho \equiv$ delete $\vee$ $\rho \equiv$ modify) $\wedge$ context.eGet(type) $\not\equiv \emptyset$ then
      changes $\cup$ AbstractReferenceChange{action = ChangeKind::delete, context = context, type = type}
    endif
  else if type.eClass $\equiv$ EAttribute then 〉 attribute change (create, delete, modify, increase, decrease, exp(v))
    changes $\cup$ AbstractAttributeChange{action = ChangeKind::modify, context = context, type = type}
  endif
  return changes
```

```
function change($\rho$:RepairHint, type:EClass): Set<ChangeAction>
  if $\rho \equiv$ create $\vee$ $\rho \equiv$ modify then
    return { ObjectChangeAction{action = ChangeKind::create} }
  endif
  if ($\rho \equiv$ delete $\vee$ $\rho \equiv$ modify) then
    return { ObjectChangeAction{action = ChangeKind::delete} }
  endif
```

```
function delete(context:EObject): Set<Change>
  return { ObjectChange{action = ChangeKind::delete, context = context} }
```

Listing 5.1. Initialization of model changes as defined in Figure 3.8.

outputs a set of *abstract repairs*. The goal is to approximate a complete and minimal set of abstract repairs *R*. A set *R* is *complete* if all possible concrete repair plans for a given inconsistency can be derived from *R*. In other words, every concrete repair plan instantiates at least one abstract repair. Finally, *R* is *minimal* if it contains only abstract changes that have the potential to repair the inconsistency [149], i.e., each abstract repair can be instantiated as a change of at least one possible concrete repair plan.

In some cases, if the repair of an inconsistency requires to create/delete objects in a model's ASG, there is only an abstract repair for creating/deleting the corresponding containment reference in *R*. The abstract repair does not specify whether the object (or a complete subtree of the AST) should be created/deleted or moved from/to another location in the model. However, in addition to the approach in Reference [149], a single object change is included in *R* that removes the context element of a validation to resolve an inconsistency.

Notably, the instantiation of an abstract repair in a concrete repair plan that also maintains the elementary consistency of a model (see Section 3.1.3) can involve additional dependent changes not included in *R*. For example, an actual removal of a model element from the AST requires to also delete all references pointing to that element. As *R* only indicates starting points for repairs within the validation scope of an inconsistency, we do not include such dependent changes in *R*.

A complete abstract repair set R_{scope} can be specified by defining an abstract change modifying each reference and attribute in a validation scope. However, the derived abstract repair set $R_{scope} \subseteq R$ is not necessarily minimal. For example, only modifications in the validation scope of the second part of *message_signature(m:Message)* in Figure 5.1 can resolve the inconsistency. As the first part of the consistency rule is already fulfilled, neither changing the operation name nor the message name would improve the inconsistency. The basic idea of the scope analysis is to ignore the already correct parts of the validation in order to narrow down the real cause of an inconsistency.

5.1.2 Validation Trace

To analyze the scope of a consistency rule's validation, Reder et al. [149] describes the concept of validation trees, which record the results of each logical operation and the accesses of model elements during the validation. Similarly, we will record the validation of a consistency rule in a structure which we will refer to as *validation trace*. Figure 5.10 shows an excerpt of the validation trace of the second part of the consistency rule *message_signature(m:Message)* in Definition (5.1b, 5.1c). In comparison to a validation tree, the trace also illustrates the data flow of all variable assignments and intermediate results of non-Boolean function calls. In practice, the recording of the validation trace is only performed for the inconsistency to be repaired, i.e., when the developer selects the inconsistency, the validation is performed again and recorded.

Basically, the validation trace can be read as a kind of data flow notation. A rounded node in the validation trace represents a function call, e.g., a logical operator or a predicate function. Similarly, constant values and the context element are also represented as functions returning the corresponding value or object, e.g., the message 6:disconnect assigned to the context variable *m* of the consistency rule *message_signature(m:Message)* in Figure 5.10.

Functions can declare variables that can be used in expressions computing parameters of that function, e.g., the universal and existential quantification specifies an iteration variable. The values assigned to a variable are represented by rectangular nodes in Figure 5.10. Containment edges (noted with a diamond) specify a single variable or a group of variables owned by a function. For example, consider the iterate function in Figure 5.10 with the assignment traces of the variables acc[i]. As illustrated in Figure 5.10, the iteration variable o[i] has no explicit assignment trace. The assignment is represented by the c={o[i] | 0 ≤ i ≤ 3} edge of the iterate function.

Directed edges in the validation trace show the flow of data values, i.e., the input and output of functions and the writing and reading of variable values. Notably, variable values are only written once but can be read multiple times. If a variable x is overwritten, e.g., in an iteration over a set of model elements, we enumerate each assigned value by x[i].

5.1.3 Scope Analysis

A consistency rule is formulated using a specific constraint language. The constraint language defines the functions that can be evaluated on a model. To generate the abstract repairs for a given validation trace, we define a repair operation repairOp(e:CallTrace, ρ:RepairHint) for each function of a constraint language. A repair operation is invoked for a function

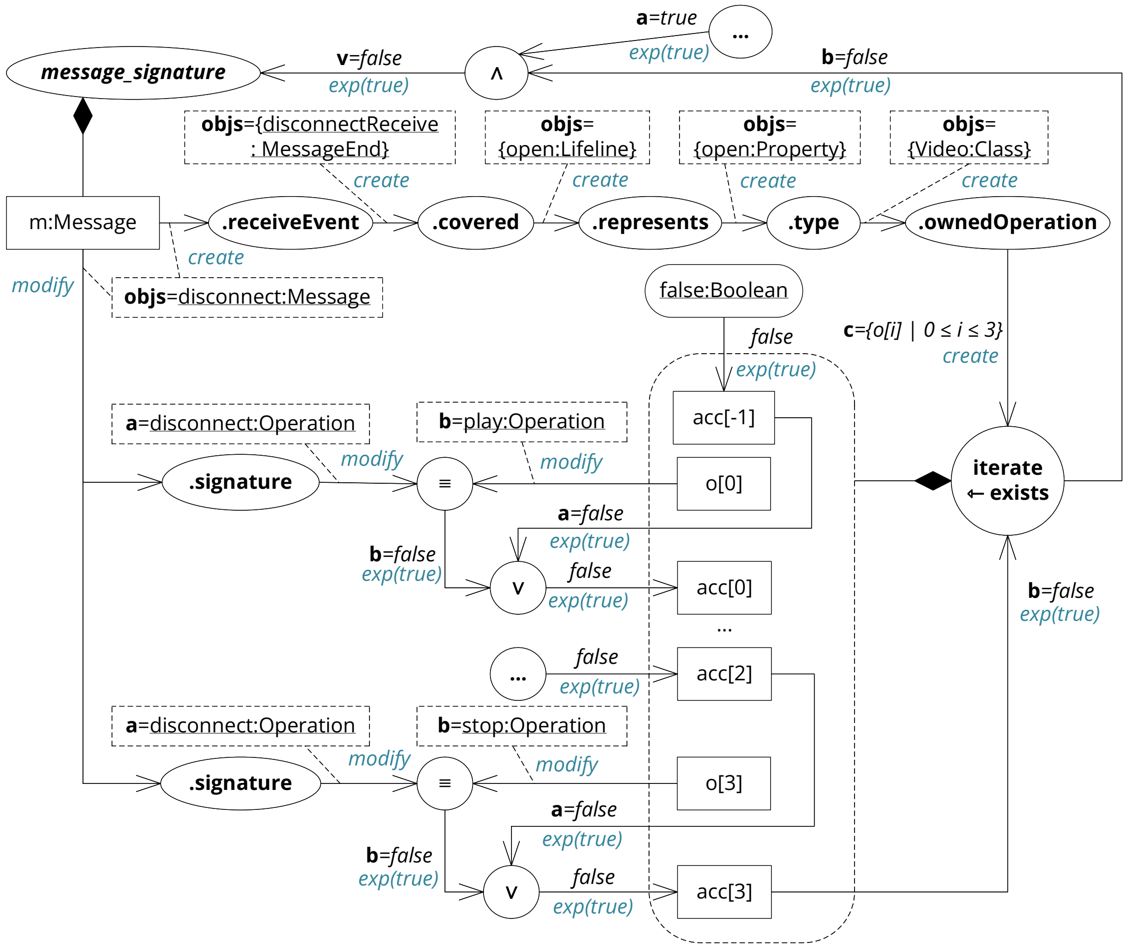


Figure 5.10. Validation trace of the second part of the consistency rule *message_signature(m:Message)* in Definition (5.1b, 5.1c) on the ASG of Version 4 in Figure 2.4 of the VoD-System. The model's ASG, including the validation scope with respect to Version 4 is illustrated in Figure 5.1.

call e recorded in the validation trace. Similar to the approach of Reder et al. [149], the repair operation computes the abstract repairs with respect to the input parameter bindings and the returned and expected value of the function call e.

In addition, a repair operation repairOp(e:CallTrace, ρ:RepairHint) is invoked with a *repair hint* ρ, which indicates the difference between the actual returned and expected result of the function call e during the validation. The repair hint *exp*(v) specifies an expected value, e.g., the expected Boolean value *exp*(*true*) if a logical operator actually returns *false*. The repair hints *increase* and *decrease* are related to numeric evaluations, e.g., the size of a set of model elements computed during the validation. Similar to their corresponding change action, the repair hints *create*, *delete*, and *modify* indicate modifications of the model's ASG.

Figure 5.11 shows the meta-model for validation traces, which is basically a graph consisting of nodes (TraceNode) and edges (TraceEdge) representing function calls, variable assignments, and parameter bindings, respectively. A function of a constraint language is declared by a subclass of the meta-class Function including its owned parameters, variables, and its repair operation repairOp(e:CallTrace, ρ:RepairHint).

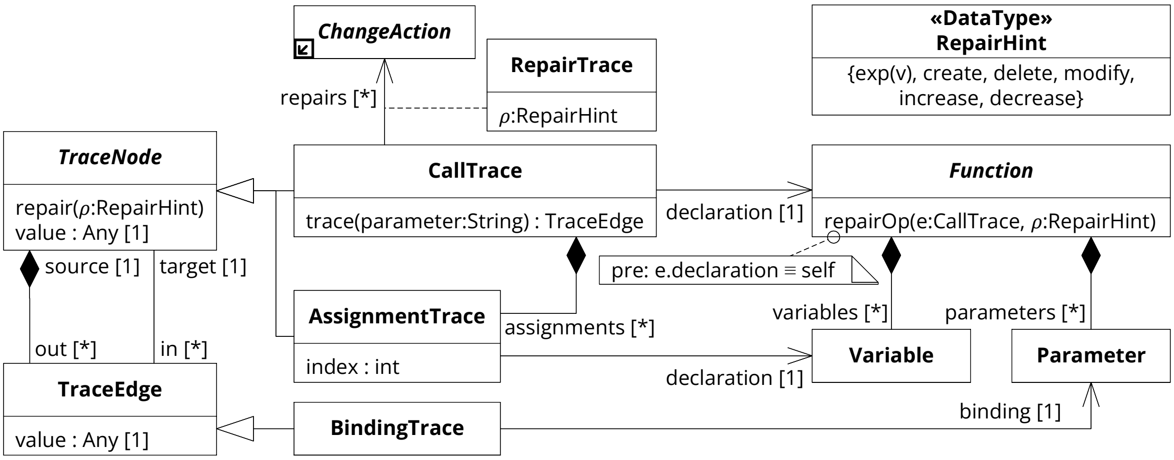


Figure 5.11. Meta-model for recording and repairing validation traces of a consistency rule. Abstract repairs are specified by changes based on the ↙ meta-model in Figure 3.8.

```
1   CallTrace::repair(ρ:RepairHint):
2     if ρ ∉ self.repairs then                   ⟩ Check if result is in the cache.
3       self.repairs[ρ] = ∅                      ⟩ Initialize repair hint cache.
4       self.declaration.repairOp(self, ρ)       ⟩ Invoke the repair operation of the recorded function call.
5     endif
    ------------------------------------------------------------------------------------------
6   AssignmentTrace::repair(ρ:RepairHint):
7     |self.in| > 0 ⇒ self.in[0].source.repair(ρ)    ⟩ Follow the writing trace of the variable assignment.
    ------------------------------------------------------------------------------------------
8   Function::repairOp(e:CallTrace, ρ:RepairHint):  ⟩ To be overwritten by concrete function implementations.
9     foreach trace ∈ e.in do                    ⟩ If no specific repair hints are specified, ...
10      trace.source.repair(modify)              ⟩ then repair all input parameters with ρ = modify.
11    endforeach
    ------------------------------------------------------------------------------------------
12  CallTrace::trace(parameter:String): TraceEdge
13    declared:Parameter = p ∈ self.declaration.parameters | p.name ≡ parameter   ⟩ Get parameter by name.
14    trace:TraceEdge = t ∈ self.in | declared ≡ t.binding     ⟩ Get incoming edge by parameter.
15    return trace  ⟩ Trace to node (call or assignment) binding the input of the given parameter.
```

Listing 5.2. Operation implementations of the meta-classes defined in the validation trace meta-model in Figure 5.11. Regarding the pseudocode notation, the variable self refers to the object on which the operation is invoked. Moreover, the set-builder notation is used to select and return a single unique value of a set.

Technically, to generate all abstract repairs R of validation trace, we invoke the repair operations following the inverse data flow in the validation trace. In the illustration of the validation trace in Figure 5.10, this means we follow the incoming edges of the function calls to generate the abstract repairs. Initially, we invoke the generation operation NodeTrace::repair(ρ:RepairHint) for abstract repairs on the root function call of the validation trace. As shown in Listing 5.2 (Line 4), the generation operation CallTrace::repair(ρ:RepairHint) of a function call invokes the repair operation of its declaring function (self.declaration).

As illustrated by the CallTrace::repairs reference in Figure 5.11, the abstract repairs are stored for each function call with respect to the provided repair hint during the generation process. The overall set of abstract repairs *R* are finally collected from all function calls in a validation trace. During the generation process, the CallTrace::repair(ρ:RepairHint) function initializes an empty set for each provided repair hint (Line 3). As an optimization, all subsequent invocations of the generation function with an already provided repair hint can be ignored (Line 2), i.e., such repair hints would lead to the same abstract repairs.

In general, if no specific repair operation is specified for a constraint language function, the generic implementation of the repair operation Function::repairOp(e:CallTrace, ρ:RepairHint) is invoked. As defined in Listing 5.2 (Line 9), in such a case, the repair generation is simply delegated to the function calls computing the input parameters of that function. The input parameters are represented as incoming edges of the function calls in the validation trace. In this context, an input parameter of a function call may be bound to a variable assignment. To generate the corresponding abstract repairs, the AssignmentTrace::repair(ρ:Repair Hint) trace repair operation (Line 6) recursively delegates the abstract repair generation to the function call computing the assignment (self.in[0]). Notably, for the sake of brevity of the notation, we use a recursive algorithm for processing the validation trace. In practice, this may be implemented by an iterative algorithm that follows the edges of the trace accordingly.

Basically, using only the generic repair operation, we invoke all repair operations in a validation trace with the *modify* repair hint. In this context, the *modify* repair hint can be understood as the most general repair hint. However, using only the generic repair operation, no abstract repairs are generated.

In order to define specific repair operations, we use the shorthand notations shown in Definition (5.3), Definition (5.4), and Definition (5.5). In this context, let e be a function call e(p) of the validation trace. The output of a function call e is stored in the TraceNode::value attribute of that validation trace node. As defined in Definition (5.3), the notation $e^{=}$ yields the computed value of e. In addition, the operation CallTrace::trace(parameter:String): TraceEdge, described in Figure 5.11 and Listing 5.2 (Line 12), determines the incoming edge of the validation trace for a given parameter name, i.e., the expression e.trace('p') returns the edge representing the parameter named p of the function call e. In this context, as defined in Definition (5.5), let $e_p^{=}$ be the value bound to the parameter named p for the function call e. Furthermore, as defined in Definition (5.4), let e_p be the source node (TraceNode) of the edge representing the parameter binding in the validation trace, i.e., the function call or variable assignment bound to the input parameter p.

$$e^{=} ::= \mathsf{e.value} \tag{5.3}$$

$$e_p^{=} ::= \mathsf{e.trace('p').value} \tag{5.4}$$

$$e_p ::= \mathsf{e.trace('p').source} \tag{5.5}$$

In the following sections, we introduce the basic functions of a constraint language and their repair operations. This approach builds upon the work introduced by Reder et al. [149] and Nentwich et al. [134]. Reder et al. directly define repair operations for Boolean operators such as conjunction, disjunction, implication, and equivalence. Here we utilize if-then-else expressions to derive Boolean operators. Moreover, Reder et al. introduce existential and universal quantifiers to make statements about sets of model elements. In contrast, in the following Section 5.1.5, we derive quantifiers from a general iterate function (as defined in OCL [139]). Additionally, Section 5.1.4 introduces support for set-based navigation for references in models.

An extension of the approach introduced by Marchezan et al. [120] include functions such as isEmpty, includes, and includesAll to check the existence of elements in sets. Such functions can be generalized by a conditional counting function (see Section 5.1.5). Similarly, Marchezan et al. introduce a greater/smaller-then-or-equal function, such as a size function for sets.

Notably, Reder et al. and Marchezan et al. consider the modification of references as atomic changes, while our approach considers such changes as a combination of creations and deletions of references. In addition, the concept for generating abstract repairs is extended to support operation invocations, derived references, and reflective access to models. Finally, basic repair operations for set theory and arithmetic operations are introduced. Based on the defined repair operations, in Section 5.1.6, we will discuss in detail the generation of abstract repairs for the *message_signature(m:Message)* validation trace recorded for our exemplary VoD-System in Figure 5.10.

5.1.4 Basic Repair Operations

For the specification of repair operations, Table 5.1 to Table 5.6 show the constraint language functions and their concrete syntax notation in the first column. The parameters as they are named in the validation trace are underlined in this notation. The second column describes the canRepair() test that defines the repair hints that can be processed by the corresponding repair function. In particular, we assume that all repair operations can handle the *modify* repair hint. For example, a function call e with a Boolean return value would interpret a *modify* repair hint as the inversion of it actually returned value ($\neg e^{=}$). Moreover, the canRepair() test typically compares the repair hint *exp*(v) with the actually returned value, i.e., if the function needs to be repaired at all. The last column specifies the actual repair operation repairOp() of each constraint language function. Notably, the templates of the canRepair() and the repairOp() are outlined in the header of each specification table.

Validation (c:EObject, T:EClass, v:Boolean, return:Boolean)

In our example in Figure 5.10 we start the process with *message_signature*.repair(*exp*(*true*)) for the consistency rule violation of the message 6:disconnect of our VoD-System. As defined by the repair operation in Table 5.1, the first (abstract) repair *delete*($e_c^{=}$) generated for the Validate function deletes the context element. To start the generation of abstract repairs for the actual consistency rule, the initial repair hint *exp*(true) is passed to the repair operation of the root predicate e_v.repair(*exp*(true)).

Get/eGet/AllInstances (objs:EObject, type:EStructuralFeature, return:Set)

We primarily use navigating between objects and reading attribute values to access information from a model. In this context, the Get function is called with a set of objects (objs) and the reference or attribute type (see meta-metamodel in Figure 3.5) to be accessed. For all given objects $e_{obj}^{=}$, the Get function collects the references or attribute values with respect to the given type in a flat set. To always return a valid result on chained model accesses, we

Core Function	Is Repairable?	Repair Operation

⟨**Function**⟩ ::=
⟨notation with $\underline{\text{parameters}}$⟩
	canRepair(e:CallTrace,
	ρ:RepairHint): Boolean
	return ⟨...⟩
		repairOp(e:CallTrace, ρ:RepairHint):
		if self.canRepair(e, ρ) **then**
		R = e.repairs[ρ]
		⟨...⟩
		endif

Validation<T> ::= $\underline{\textit{rulename}}(\underline{c}:\underline{T}) \coloneqq \underline{v}$	$e^{=} \equiv \textit{false} \wedge$ $\rho \equiv \textit{exp}(\textit{true})$	R $\cup$ delete($e^{=}_{c}$) e_{v}.repair(*exp*(*true*))

Get ::=
$\underline{\text{objs}}.\underline{\textit{type}}$
	$\rho \equiv \textit{modify} \vee$
	$\rho \equiv \textit{create} \vee$
	$\rho \equiv \textit{delete}$
		foreach o $\in e^{=}_{\text{objs}}$ **do**
		R $\cup$ change(ρ, o, $e^{=}_{\text{type}}$)
		endforeach
		e_{objs}.repair(ρ)

AllInstances ::= allInstances($\underline{\text{type}}$)	$\rho \equiv \textit{modify} \vee$ $\rho \equiv \textit{create} \vee$ $\rho \equiv \textit{delete}$	R $\cup$ change(ρ, $e^{=}_{\text{type}}$) e_{type}.repair(*modify*)
Invoke ::= $\underline{\text{objs}}.\underline{\textit{op}}(\ldots)$	$\rho \not\equiv \textit{exp}(e^{=})$	e_{objs}.repair(ρ) e_{op}.repair(ρ)
isValueOf ::= $\underline{T}$::isValueOf($\underline{\text{value}}$)	$\rho \equiv \textit{modify} \vee$ $(\rho \equiv \textit{exp}(\text{value}) \wedge$ $\text{value} \not\equiv e^{=})$	e_{value}.repair(*modify*)
ConstantLiteral ::= ⟨*literal value*⟩	*false*	
Reflection Operations		*(see Invoke, e.g., objs.eGet())*
EObject::eGet ::= eGet($\underline{\text{type}}$)	$\rho \equiv \textit{modify} \vee$ $\rho \equiv \textit{create} \vee$ $\rho \equiv \textit{delete}$	e_{type}.repair(*modify*)
EObject::eInvoke ::= eInvoke($\underline{\text{eOp}}$, [$\underline{\text{p0}}$, ..., $\underline{\text{pn}}$]) *with* eOp ::= $\underline{\textit{op}}$*(p0, ..., pn)*	$\rho \not\equiv \textit{exp}(e^{=})$	e_{op}.repair(ρ) e_{eOp}.repair(*modify*)
EDataType::eIsValueOf ::= eIsValue($\underline{\text{value}}$)	$\rho \equiv \textit{modify} \vee$ $(\rho \equiv \textit{exp}(\text{value}) \wedge$ $\text{value} \not\equiv e^{=})$	e_{value}.repair(*modify*)

Table 5.1. Definition of basic repair operations for model access.

assume that a model element navigation or reading an attribute value at least returns the empty set (∅).

The corresponding repair operation of a constraint language function in Table 5.1 is invoked with the function call e to be repaired and a repair hint ρ. The repair operation of the Get function generates an abstract repair for each accessed reference/attribute of the elements in $e^{=}_{objs}$. The type of the abstract change corresponds to the given repair hint ρ.

Consistency rules might also depend on information from the meta-model. For example, in the consistency rule Definition (3.3), which checks the multiplicity bounds of an object's references, all reference types of the object are determined dynamically using reflective model access. However, from the perspective of repairing a model, the meta-model has to be considered a constant, not changeable, structure. Therefore, we ignore the initialization of abstract changes that propose to modify the meta-model in the change() function in Listing 5.1 (Line 3).

As the consistency rule in Definition (3.3) also demonstrates, the eGet operation of EObject allows reflective model access (see Section 3.1.3). Similar to the Get function, the repair operation of eGet generates an abstract repair for the accessed reference or attribute of the object (self). Modifying the given type will change the result of an eGet call. Therefore, the corresponding repair operation in Table 5.1 additionally invokes the repair operation of the expression bound to its type parameter.

AllInstances is another function that can be used to access elements in a model. The function collect all model elements of a specific type from a model. As defined in Table 5.1, the repair operation of *allInstances* generates (abstract) object changes that ignore the concrete context of an object and just specify its type. Moreover, if the meta-class of the type parameter is computed by an expression, it can possibly be repaired.

In general, derived references or operations that are defined in a meta-model of a modeling language can be defined by queries that use the same functions as the constraint language. Such queries can also be recorded and processed in the validation trace. Alternatively, a specific repair operation can be provided for derived references or operations. As an example, a Closure function can be implemented using set-based navigation. Such a Closure function can be recorded in the validation trace as a sequence of Get functions. For example, to collect all supertypes of a meta-class, as defined by the derived reference EClass::eAllSuperTypes in Figure 3.5.

Invoke/eInvoke (objs:EObject, eOp:EOperation)
As defined in Table 5.1, the Invoke and eInvoke functions will simply delegate the repair generation to the invoked operation, i.e., the recorded query or the specific repair operation. Similar to the Get function, the repair hint is also delegated to the repair generation of the expression computing the objects (objs) on which the operation is invoked. Additionally, the reflective invocation eInvoke may generate repairs for the dynamically derived operation (eOp). The parameters of an operation invocation are to be handled by the repair operation of the invoked operation.

IsValueOf/eIsValueOf (T:EClass, value:Any)
In addition, Table 5.1 provides some repair operations for functions that handle literal values and single model elements. To check if a literal value is in the domain of a specific data

type, the IsValueOf function or EDataType::eIsValueOf operation can be used to test a value. Basically, the repair operation for literal values or single model elements either modifies the expression that computes the value or creates a new value if an empty set is given. In contrast, the repair operation of constants do not provide any abstract repairs, i.e., the repair operation invocation is ignored by the ConstantLiteral function.

Considering only the basic repair operations defined in Table 5.1 and the generic repair operation in Listing 5.2 that always propagates the *modify* repair hint, an abstract reference/ attribute creation and deletion is generated for each reference/attribute access recorded in the validation trace, i.e., the full validation scope with respect to an inconsistency. However, as mentioned, the resulting abstract repair set R is not necessarily minimal.

5.1.5 Essential Repair Operations

The first part in Definition (5.1a) and the second part in Definition (5.1b, 5.1c) of the consistency rule *message_signature(m:Message)* are connected by a conjunction. The validation trace of the conjunction in Figure 5.10, shows that the first part of the consistency rule, comparing the name of the message with the name of the operation signature, passes the validation. Thus, the first part of the validation does not need to be repaired, i.e., no abstract repairs have to be generated for the validation scope of the first part of the consistency rule.

In contrast, let us assume a conjunction with both sides evaluated to *false* and an expected value of *true*. To change the expression *false* ∧ *false* to the expected value *true*, the left-hand and right-hand side must be repaired at the same time. As described by Reder et al. [149], a repair plan that fully resolves an inconsistency always includes abstract repairs with respect to both sides of the conjunction. In practice, this can be a problem if a constraint language does a short-circuit evaluation of Boolean operators. For example, in a conjunction a ∧ b, the value of b is unknown if a is evaluated to *false*. However, to identify a given set of changes as a repair with a positive impact on the inconsistency, as described in Section 5.1.1, it is sufficient to include the abstract repairs with respect to one side of the conjunction in R.

Basically, as described by Reder et al. [149], the repair operation of conjunction and disjunction carry on the expected value to the connected Boolean expressions, while a negation ¬a inverts the expected value *exp*(¬v) for the repair operation invocation. As specified in Definition (5.6) to Definition (5.8), the repair operation of the conjunction, disjunction, negation, and, in general, other Boolean operators can be implemented by an if-then-else expression. Notably, Definition (5.6) and Definition (5.7) implement a short-circuit evaluation of the Boolean operators.

$$\mathsf{a} \wedge \mathsf{b} \quad := \quad \textbf{if } \mathsf{a} \textbf{ then } \mathsf{b} \textbf{ else } \mathsf{false} \textbf{ endif} \tag{5.6}$$

$$\mathsf{a} \vee \mathsf{b} \quad := \quad \textbf{if } \mathsf{a} \textbf{ then } \mathsf{true} \textbf{ else } \mathsf{b} \textbf{ endif} \tag{5.7}$$

$$\neg \mathsf{a} \quad := \quad \textbf{if } \mathsf{a} \textbf{ then } \mathsf{false} \textbf{ else } \mathsf{true} \textbf{ endif} \tag{5.8}$$

If-then-else (c:Boolean, th:T, el:T, return:T)

Depending on the result of the Boolean if-condition c, the repair of the then-branch th and the else-branch el is handled analogously. The repair operation always passes the repair hint to the expression computing the returned value of a branch.

Core Function	Is Repairable?	Repair Operation

⟨**Function**⟩ ::=
⟨notation with $\underline{\text{parameters}}$⟩
	canRepair(e:CallTrace,
	ρ:RepairHint): Boolean
	return ⟨...⟩
		repairOp(e:CallTrace, ρ:RepairHint):
		if self.canRepair(e, ρ) **then**
		⟨...⟩
		endif

IfThenElse<T> ::=
if $\underline{\text{c}}$ **then** $\underline{\text{th}}$ **else** $\underline{\text{el}}$ **endif**
	$\rho \not\equiv exp(e^{=})$
	canRepairThen(ρ:RepairHint): Boolean
	return $\rho \equiv exp(v) \Rightarrow v \in dom(th)$
	canRepairElse(ρ:RepairHint): Boolean
	return $\rho \equiv exp(v) \Rightarrow v \in dom(el)$
		if $e_c^{=}$ **then**
		e_{th}.repair(ρ) ⟩ then-branch
		if self.canRepairElse(ρ) **then**
		e_c.repair($exp(\neg e_c^{=})$)
		endif
		else
		e_{el}.repair(ρ) ⟩ else-branch
		if self.canRepairThen(ρ) **then**
		e_c.repair($exp(\neg e_c^{=})$)
		endif
		endif

Iterate<T,R> ::= $\underline{\text{c}}$ → iterate(o:$\underline{\text{T}}$; acc:$\underline{\text{R}}$ = $\underline{\text{init}}$ | $\underline{\text{b}}$)	$\rho \not\equiv exp(e^{=})$	$\rho_c = modify$ e_c.repair(ρ_c) e_b.repair(ρ)

Table 5.2. Definition of repair operations for the consistency rule control flow.

To change the result of an if-then-else evaluation, we may have to switch to the alternative branch by inverting the result of the if-condition (e_c.repair($exp(\neg e_c^{=})$)). However, in some cases, the alternative branch cannot contribute to the repair. In general, the repair with respect to condition c can lead to a non-minimal abstract repair set R. Such cases can be avoided by implementing and using derived functions and repair operations that filter such repairs by the repairableThen() and repairableElse() tests. By default, for a given expected value v, the repair operation requires that the expected value is at least in the domain of the expression specifying the corresponding branch. For example, the Definition (5.6) specifies the Boolean conjunction by an if-then-else expression. In this case, the domain of the then-branch can be specified as dom(th) := Boolean, and the domain of the else-branch is given by the constant dom(el) := {false}.

To illustrate the derivation of a repair operation, let us consider again the if-then-else implementation of the conjunction in Definition (5.6). To derive the repair operation, we map the parameters of the conjunction to the corresponding parameters of the IfThenElse function as defined in Table 5.2. For example, let us assume the expression a ∧ b = *false* ∧ *false* is expected to be $\rho = exp(\text{true})$. In this case, the repair operation first delegates the repair hint to the expression of the else-branch e_{el}.repair(ρ = *false*). This repair operation invocation is actually ignored by the constant value el := *false* of the else-branch. Next, the filter canRepairThen() evaluates to *true* based on the domain of the then-branch, i.e., with the domain th := b being Boolean, and *true* ∈ Boolean, the then-branch can contribute to

Derived Functions	Is Repairable?	Repair Operation

⟨**Function**⟩ ⇽
⟨**Function** (derived)⟩ ::=
⟨notation with $\underline{\text{parameters}}$⟩
	canRepair(e:CallTrace, ρ:RepairHint): Boolean
	return ⟨...⟩
		repairOp(e:CallTrace, ρ:RepairHint):
		if self.canRepair(e, ρ) **then**
		⟨...⟩
		endif

Iterate<T,R> ⇽ **Exists<T, Boolean>** (Def. 5.9) ::= $\underline{c} \rightarrow \exists\, o{:}\underline{T} \mid \underline{b}$	$\rho \equiv$ *modify* ∨ ($\rho \equiv$ *exp*(value) ∧ value $\not\equiv e^{=}$)	**if** $\rho \equiv$ *modify* **then** $\rho = \neg e^{=}$ **endif** $\rho_c =$ **if** $\rho \equiv$ *exp*(true) **then** *create* **else** *delete* **endif** e_c.repair(ρ_c) e_b.repair(ρ)
Iterate<T,R> ⇽ **ForAll<T, Boolean>** (Def. 5.10) ::= $\underline{c} \rightarrow \forall\, o{:}\underline{T} \mid \underline{b}$	$\rho \equiv$ *modify* ∨ ($\rho \equiv$ *exp*(value) ∧ value $\not\equiv e^{=}$)	**if** $\rho \equiv$ *modify* **then** $\rho = \neg e^{=}$ **endif** $\rho_c =$ **if** $\rho \equiv$ *exp*(*true*) **then** *delete* **else** *create* **endif** e_c.repair(ρ_c) e_b.repair(ρ)
Iterate<T,R> ⇽ **Count** (Def. 5.11) ::= $\underline{c} \rightarrow$ count($o{:}\underline{T} \mid \underline{b}$)	$\rho \equiv$ *modify* ∨ $\rho \equiv$ *increase* ∨ $\rho \equiv$ *decrease* ∨ ($\rho \equiv$ *exp*(value) ∧ value $\not\equiv e^{=}$)	$\rho_c =$ **if** ($\rho \equiv$ *exp*(value) ∧ $e^{=}$ > value) ∨ $\rho \equiv$ *decrease* **then** *delete* **else if** ($\rho \equiv$ *exp*(value) ∧ $e^{=}$ < value) ∨ $\rho \equiv$ *increase* **then** *create* **else** *modify* **endif** e_c.repair(ρ_c) e_b.repair(ρ)

IfThenElse<T> ⇽
Increment<Integer> (Def. 5.12) ::=
increment($\underline{i}$:Integer, $\underline{b}$:Boolean)
	$\rho \equiv$ *modify* ∨
	$\rho \equiv$ *increase* ∨
	$\rho \equiv$ *decrease* ∨
	($\rho \equiv$ *exp*(value) ∧
	value $\not\equiv e^{=}$)
		canRepairThen(e:CallTrace, ρ:RepairHint)
		return $\rho \in$ {*increase*, *modify*} ∨
		($\rho \equiv$ *exp*(value) ∧ $e^{=}$ < value)
		canRepairElse(e, ρ)
		return $\rho \in$ {*decrease*, *modify*} ∨
		($\rho \equiv$ *exp*(value) ∧ $e^{=}$ > value)

Table 5.3. Definition of repair operations for derived functions.

the repair. Therefore, the repair operation invokes e_c.repair(*exp*($\neg e_c^{=}$)) ≡ e_c.repair(*exp*(*true*)) to generate abstract repairs that invert the if-condition c ≔ a.

Iterate (c:Set<T>, o:T, acc:R, b:R, return:R)
As shown in Table 5.2, the Iterate function applies an expression b on each object in a given set o ∈ c. The result of the expression b is stored in the so-called accumulator variable acc after each iteration step. Finally, the latest value of the accumulator variable is returned by the Iterate function. In particular, the accumulator variable must be initialized with a value, i.e., an iteration on an empty set returns the initial acc value. Optionally, we allow to annotate the type T of the elements in the given set and the type of the result R in a function call.

The repair operation delegates the given repair hint ρ to the iteration expression b. In the validation trace, the result of the iteration expression b is represented by the latest accumulator variable acc[n-1] with n = |c|. As we can see in the validation trace in Figure 5.10, the iteration steps are connected by the accumulator variable.

In addition, to change the result of an iteration, we can add or remove elements from the given set c. By default, the repair operation with respect to set c is invoked with a *modify* repair hint. However, this can lead to an over-approximation of the generated abstract changes R if only creations *or* deletions can directly contribute to the repair. Therefore, we allow functions that are derived from Iterate to specify the repair hint ρ_c that is applied to the processed collection.

As an example of derived iteration functions, Definition (5.9) and Definition (5.10) show the existential and universal quantifier as functions. Their corresponding repair operations, which specifically define the repair ρ_c, are defined in Table 5.2. In particular, the collection examined by an existential quantifier can only be repaired by adding at least one of the required elements to the collection [149]. In contrast, the universal quantifier must delete the elements from the collection that do not match [149]. In particular, an existential quantifier can be derived from a universal quantifier $(c \rightarrow \exists\, o \mid p) \equiv \neg(c \rightarrow \forall\, o \mid \neg b)$, and vice versa $(c \rightarrow \forall\, o \mid b) \equiv \neg(c \rightarrow \exists\, o \mid \neg b)$.

$$c \rightarrow \exists\, o{:}T \mid b \quad := \quad c \rightarrow \text{iterate}(o{:}T;\ \text{acc:Boolean} = \textit{false} \mid \text{acc} \lor b) \tag{5.9}$$

$$c \rightarrow \forall\, o{:}T \mid b \quad := \quad c \rightarrow \text{iterate}(o{:}T;\ \text{acc:Boolean} = \textit{true} \mid \text{acc} \land b) \tag{5.10}$$

Another useful example of a derived function is shown in Definition (5.11). The Count function evaluates a Boolean expression on all elements of a given set of objects c. The function increments the accumulator variable for each object in c for which b evaluates to *true*. As shown in Definition (5.12), a conditional increment function can be specified using an if-then-else expression.

$$c \rightarrow \text{count}(b{:}\text{Boolean}) \quad := \quad c \rightarrow \text{iterate}(o{:}T;\ \text{acc} = 0 \mid \text{increment}(\text{acc}, b)) \tag{5.11}$$

$$\text{increment}(i{:}\text{Integer}, b{:}\text{Boolean}) \quad := \quad \textbf{if}\ b\ \textbf{then}\ i + 1\ \textbf{else}\ i\ \textbf{endif} \tag{5.12}$$

The corresponding repair operations are specified in Table 5.3. For the Count function, we translate a given expected number or an *increase*/*decrease* repair hint into a corresponding *create*/*delete* repair hint for the evaluated set c. To switch between the then-branch and else-branch in terms of the Increment function, the repair operation must decide whether to modify the object evaluated in the Boolean expression b. If the then-branch is executed and the returned value is too large, the repair operation can invert the value of condition b. Conversely, if the value returned by the else-branch is too small, the result of b can be inverted to execute the then-branch.

Sets (a:Set, b:Set, return:Set) **and Size** (c:Set, return:Number)

To include or exclude elements from sets, we introduce the set union and set difference in Table 5.4. Basically, the repair operations translate the repair hints *modify*, *create*, *delete* accordingly based on the set operators applied to the given sets a and b.

Core Function	Is Repairable?	Repair Operation

$\langle$**Function**$\rangle$ ::=
$\langle$notation with $\underline{\text{parameters}}\rangle$
	canRepair(e:CallTrace,
	ρ:RepairHint): Boolean
	return $\langle$...$\rangle$
		repairOp(e:CallTrace, ρ:RepairHint):
		if self.canRepair(e, ρ) **then**
		$\langle$...$\rangle$
		endif

SetUnion ::= $\underline{a} \cup \underline{b}$	$\rho \equiv \textit{modify} \lor$ $\rho \equiv \textit{create} \lor$ $\rho \equiv \textit{delete}$	e_a.repair(ρ) e_b.repair(ρ)

SetDifference ::=
$\underline{a} \setminus \underline{b}$
	$\rho \equiv \textit{modify} \lor$
	$\rho \equiv \textit{create} \lor$
	$\rho \equiv \textit{delete}$
		if $\rho \equiv \langle\textit{modify/create/delete}\rangle$ **then**
		e_a.repair($\langle\textit{modify/create/delete}\rangle$)
		e_b.repair($\langle\textit{modify/delete/create}\rangle$)
		endif

SetSize ::= $\lvert\underline{c}\rvert$	$\rho \equiv \textit{modify} \lor$ $\rho \equiv \textit{increase} \lor$ $\rho \equiv \textit{decrease} \lor$ $(\rho \equiv \textit{exp}(\text{value}) \land$ $\text{value} \not\equiv e^{=})$	**if** $\rho \equiv \textit{modify}$ **then** e_c.repair(*modify*) **else if** $\rho \equiv \textit{increase} \lor$ $\rho \equiv \textit{exp}(s) \land s > e^{=}$ **then** e_c.repair(*create*) **else if** $\rho \equiv \textit{decrease} \lor$ $\rho \equiv \textit{exp}(s) \land s < e^{=}$ **then** e_c.repair(*delete*) **endif**

Table 5.4. Definition of repair operations for working with sets.

In addition, a size function to determine the number of contained elements is introduced in Table 5.4. The repair operation of the size function translates an exact expected number, or *increase*//*decrease* repair hints into *create*/*delete* repair hints for the given set c accordingly.

Arithmetic (a:Number, b:Number, return:Number)
Table 5.5 introduces the repair operations of the basic arithmetic operators, e.g., to sum and compare the size of different sets of references in a model. First, Table 5.5 introduces the addition and multiplication operator. Subtraction and division can be derived by negation and complementing expressions, respectively. In general, numeric computations can be repaired with the repair hints *increase*, *decrease*, *exp*(v), or *modify*. Based on the operator, such repair hints are translated accordingly to the operands of a computation. In particular, the repair hint can be refined if a constant value is involved and a specific expected value is given. For the sake of brevity, the different case distinctions for constants $\langle$a/b$\rangle$, operators $\langle +/\times \rangle$, and repair hints *increase*/*decrease* are combined in the notation in Table 5.5.

In addition, to repair a greater-than(-or-equal) function, the repair operation must modify the numbers on one or both sides of the inequality accordingly, i.e., increasing or decreasing the numbers until the formula is fulfilled. Notably, if the result is expected to be *false*, the *increase*/*decrease* repair hints are swapped.

Core Function	Is Repairable?	Repair Operation

⟨**Function**⟩ ::=
⟨notation with $\underline{\text{parameters}}$⟩
	canRepair(e:CallTrace, ρ:RepairHint): Boolean
	return ⟨...⟩
		repairOp(e:CallTrace, ρ:RepairHint):
		if self.canRepair(e, ρ) **then**
		⟨...⟩
		endif

Addition ::= $\underline{a} + \underline{b}$ **Multiplication** ::= $\underline{a} \times \underline{b}$	$\rho \equiv$ *modify* ∨ $\rho \equiv$ *increase* ∨ $\rho \equiv$ *decrease* ∨ ($\rho \equiv$ *exp*(value) ∧ value $\not\equiv e^{=}$)	ρ = **if** $\rho \equiv$ *exp*(value) **then** **if** ⟨a/b⟩ ∈ Constant **then** *exp*(value ⟨−/÷⟩ ⟨$e_a^{=}$/$e_b^{=}$⟩) ⟩ for ⟨+/×⟩ **else** $e^{=}$ > value ⇒ *decrease* $e^{=}$ < value ⇒ *increase* **endif** **endif** e_a.repair(ρ) e_b.repair(ρ)

with a $\in \mathbb{R}^+$
 	
Negation ::=
$-\underline{a} := (-1) \times \underline{a}$
 	
MultiplicativeInverse ::=
$\underline{a}^{-1} := \frac{1}{\underline{a}}$
	$\rho \equiv$ *modify* ∨
	$\rho \equiv$ *increase* ∨
	$\rho \equiv$ *decrease* ∨
	($\rho \equiv$ *exp*(value) ∧
	value $\not\equiv e^{=}$)
		if $\rho \equiv$ ⟨*increase/decrease*⟩ **then**
		e_a.repair(⟨*decrease/increase*⟩)
		else if $\rho \equiv$ *exp*(value) **then**
		⟨ e_a.repair(*exp*((−1) × value)) ⟩ ⟩ for −a
		⟨ e_a.repair(*exp*(1 ÷ value)) ⟩ ⟩ for a^{-1}
		else
		e_a.repair(*modify*)
		endif

GreaterThan ::=
$\underline{a} > \underline{b}$
 	
GreaterThanOrEqual ::=
$\underline{a} \geq \underline{b}$
	$\rho \equiv$ *modify* ∨
	($\rho \equiv$ *exp*(value) ∧
	value $\not\equiv e^{=}$)
		if $\rho \equiv$ *modify* **then** $\rho = \neg e^{=}$ **endif**
		if $\rho \equiv$ *exp*(true) **then**
		e_a.repair(*increase*)
		e_b.repair(*decrease*)
		else
		e_a.repair(*decrease*)
		e_b.repair(*increase*)
		endif

Table 5.5. Definition of repair operations for basic arithmetic operations.

Equivalence (a:Any, b:Any, return:Boolean)
Table 5.6 defines the repair operations of the equivalence (a ≡ b) for different types of comparison. In general, the corresponding repair operations handle the left-hand and right-hand side of the formula in the same way, which is implemented by their repairSide() functions.

Equivalence: In the general case, when two values are compared, e.g., the names of two model elements, one side must be made equal to the other side, or both sides must be changed to the same value. Conversely, in a negated equivalence test, at least one side must be changed to another value. Summing up, this always leads to the propagation

Core Function	Is Repairable?	Repair Operation

⟨**Function**⟩ ::=
 ⟨notation with $\underline{\text{parameters}}$⟩
	canRepair(e:CallTrace, ρ:RepairHint): Boolean
	 return ⟨...⟩
		repairOp(e:NodeTrace, ρ:RepairHint):
		 if self.canRepair(e, ρ) **then**
		 ⟨...⟩
		 endif

Equivalence ::=
 $\underline{a} \equiv \underline{b}$
	$\rho \equiv$ *modify* ∨
	 ($\rho \equiv$ *exp*(value) ∧
	 value $\not\equiv e^{=}$)
		if $\rho \equiv$ *modify* **then** $\rho = \neg e^{=}$ **endif**
		 repairSide(e_a, $e_b^{=}$, ρ)
		 repairSide(e_b, $e_a^{=}$, ρ)
		 repairSide(e:NodeTrace, value:Any, ρ:RepairHint):
		 e.repair(*modify*)

Equivalence ⇽
 ConstantEquivalence ::=
 $\underline{a} \equiv \underline{b}$
	$\rho \equiv$ *modify* ∨
	 ($\rho \equiv$ *exp*(value) ∧
	 value $\not\equiv e^{=}$)
		repairSide(e:NodeTrace, value:Any, ρ:RepairHint):
		 if $\rho \equiv$ *exp*(true) ∧ value ∈ *Constant* **then**
		 e.repair(*exp*(value))
		 else
		 e.repair(*modify*)
		 endif

Equivalence ⇽
 SetEquivalence ::=
 $\underline{a} \equiv \underline{b}$
	$\rho \equiv$ *modify* ∨
	 ($\rho \equiv$ *exp*(value) ∧
	 value $\not\equiv e^{=}$)
		repairSide(e:NodeTrace, value:Any, ρ:RepairHint):
		 if $e^{=} \equiv \emptyset$ **then**
		 e.repair(*create*) ⟩ $\rho \equiv$ *exp*(*true/false*)
		 else if value $\equiv \emptyset$ **then** ⟩ $e^{=} \not\equiv \emptyset$
		 e.repair(*delete*) ⟩ $\rho \equiv$ *exp*(*true*)
		 else
		 e.repair(*modify*) ⟩ $\rho \equiv$ *exp*(*true/false*)
		 endif

Equivalence ⇽
 NumberEquivalence ::=
 $\underline{a} \equiv \underline{b}$
	$\rho \equiv$ *modify* ∨
	 ($\rho \equiv$ *exp*(value) ∧
	 value $\not\equiv e^{=}$)
		repairSide(e:NodeTrace, value:Any, ρ:RepairHint):
		 if $\rho \equiv$ *exp*(true) **then**
		 $e^{=}$ > value ⇒ e.repair(*decrease*)
		 $e^{=}$ < value ⇒ e.repair(*increase*)
		 else
		 e.repair(*modify*)
		 endif

Table 5.6. Definition of repair operations for the comparison of values, sets, constants, and numbers.

of a *modify* repair hint for both sides of an equivalence test. Notably, the left-hand and right-hand side expressions (e) of a Boolean equivalence test translate the *modify* repair hint into an expected value, i.e., their inverted result (*exp*($\neg e^{=}$)).

ConstantEquivalence: An equivalence test with any constant value (including Boolean and Number literal constants) is recorded as ConstantEquivalence function call in the validation trace. In a comparison involving a constant value, only the computed expression can be changed by a repair.

SetEquivalence: Expecting two sets to be equal or unequal, elements can be added or removed on both sets accordingly.

NumberEquivalence: For numerical equality, a number can be increased or decreased to match the compared number. In contrast, for inequality, we may increase or decrease any of the compared numbers, i.e., in this case, the repair operation is invoked with the *modify* repair hint.

5.1.6 VoD-System Scope Analysis

Let us consider again our modeling scenario of the VoD-System in Figure 2.4. After moving the operation disconnect() from class Video to Server, the consistency rule *message_signature(m:Message)* validation fails for the corresponding message 6:disconnect. To analyze the inconsistency with respect to the model's ASG in Figure 5.1, we record the validation trace as illustrated in Figure 5.10.

Starting from the root function call of the *message_signature(m:Message)* validation, we generate the first abstract repair. As the context element of the validation is message 6:disconnect, we can simply delete the context to resolve the inconsistency. Notably, a corresponding concrete repair plan would involve several additional changes, e.g., removing the source and target ends of the messages, including all the references connecting these objects to the model's ASG:

$\alpha_{\langle \textit{model version} \rangle \langle \textit{abstract change} \rangle}$: ObjectChange{ ⟨*action*⟩, ⟨*context*⟩ }

$\alpha_{4.1}$: ObjectChange{ *delete*, 6:disconnect:Message }

The first function call to be repaired is the conjunction of the first part in Definition (5.1a) and the second part in Definition (5.1b, 5.1c). This expression is expected to be *true* in order to pass the validation, i.e., the left- and right-hand-side of the conjunction must also be *true*. Technically, the repair operation of the conjunction is given by the if-then-else function defined in Table 5.2. As the first part, evaluated by an equivalence test (see Table 5.6) of the message's name and operation's name, passes the validation ($e^{=}_{a} \equiv true$), the invocation of the repair operation is ignored by the canRepair() check. The right-hand-side of the conjunction, i.e., the second part of the consistency rule evaluates to *false*. Therefore, its repair operation is invoked with an *exp*(*true*) repair hint.

The second part in Definition (5.1b, 5.1c) of the consistency rule evaluates an existential quantifier, which checks if the operation that is invoked by a message exists in the class definition of the receiving object. As already discussed for the recording of the validation trace, the existential quantifier (see Definition (5.9) and Table 5.3) can be evaluated by the Iterate function. Basically, the result of each evaluated operation in the given set is connected by a disjunction. Then each iteration step is connected by the accumulator variable (acc) of the Iterate function.

Initially, the navigation sequence m.receiveEvent.covered.represents.type.ownedOperation (part Definition (5.1b)), which determines the set operations, is considered by the repair operation of the Iterate function defined in Table 5.2. In this case, the existential quantifier in Table 5.3, as a derived iterate function, provides a specific repair hint ρ_c = *create* (with *exp*(*true*)) for the processed set.

The navigation expression is evaluated by the Get function (see Table 5.1). The corresponding repair operation generates an abstract repair for all model elements accessed during the navigation. Therefore, the *create* repair hint is passed in reverse direction of the original navigation through the validation trace. Notably, the creation of a single-valued reference (with an upper bound of 1) in the meta-model, also requires the deletion of the former reference. As described in Section 5.1.1, we do not include those dependent changes in the abstract repairs *R*, i.e., in such a case, a deletion alone does not positively affect the validation result of existential quantifier. This finally results in the following abstract changes:

$\alpha_{\langle \mathit{model\ version}\rangle\langle \mathit{abstract\ change}\rangle}$: AbstractReferenceChange{ ⟨*action*⟩, ⟨*context*⟩, ⟨*type*⟩ }

$\alpha_{4.2}$: AbstractReferenceChange{ *create*, Video:Class, *ownedOperation* }

$\alpha_{4.3}$: AbstractReferenceChange{ *create*, open:Property, *type* }

$\alpha_{4.4}$: AbstractReferenceChange{ *create*, open:Lifeline, *represents* }

$\alpha_{4.5}$: AbstractReferenceChange{ *create*, disconnectReceive:MessageEnd, *covered* }

$\alpha_{4.6}$: AbstractReferenceChange{ *create*, disconnect:Message, *receiveEvent* }

The repair generation of the existential quantifier starts at the latest accumulator variable *acc[3]* in Figure 5.10. Here both input values of the disjunction are *false*. The abstract repair set *R* is generated in reverse order of the iteration as we follow the trace between the disjunctions and the accumulator variables. Within each iteration, one operation of the class Video is compared to the signature of the message 6:disconnect, i.e., the class Video is compared to the operation disconnect() in the class Server. Finally, the repair operation ends at the initialization constant *false* of the existential quantifier.

The disjunction used in the iteration in Figure 5.10 is derived from the if-then-else function as specified in Definition (5.7). In this case, the repair operation of both sides of the conjunction are invoked with the repair hint ρ = *exp*(*true*), i.e., the accumulator of the previous iteration step or the equivalence test is expected to be *true*.

In the general equivalence test in the last part of the consistency rule in Definition (5.1c), both sides of the formula can be modified to establish equality (see Table 5.6). The navigation m.signature on the left-hand-side results in the operation disconnect() for the equivalence test. The repair operation of the equivalence function invokes the repair operation of the Get repair operation with a *modify* repair hint. Therefore, an abstract modification repair is generated for the *m.signature* navigation, i.e., an abstract creation and deletion with respect to the signature reference of the operation disconnect(). As the left-hand-side of the equivalence is identical in all iteration steps Figure 5.10, the same abstract repair are generated for all iteration steps in the validation trace:

$\alpha_{\langle \mathit{model\ version}\rangle\langle \mathit{abstract\ change}\rangle}$: AbstractReferenceChange{ ⟨*action*⟩, ⟨*context*⟩, ⟨*type*⟩ }

$\alpha_{4.7}$: AbstractReferenceChange{ *delete*, disconnect:Operation, *signature* }

$\alpha_{4.8}$: AbstractReferenceChange{ *create*, disconnect:Operation, *signature* }

The repair operation of the right-hand side of the equivalence test is invoked with a *modify* repair hint in Figure 5.10. However, the variable assignment o[i] is not further processed.

As discussed above, the abstract repair generation of the iterated set is already handled, i.e., the abstract repairs $\alpha_{4.2}$ to $\alpha_{4.6}$ generated by the repair operation of the Iterate function.

Finally, the complete set of abstract repairs R can be derived by collecting all abstract repairs generated for the function calls in the validation trace. Thereby, duplicated abstracted repairs will be eliminated.

5.2 Origin Analysis

The abstract repairs provide insight into an inconsistency in the current model version. However, to understand the cause that introduced an inconsistency, we need to examine the editing history of a model. To understand the history of an inconsistency, we first have to identify the latest version which is not affected by that inconsistency. Starting from this version, we are further interested in the historical changes which led to the current model version since a subset of these changes must have caused the inconsistency. As depicted by Step 2 in the overview of the approach in Figure 5.7 and Figure 5.8b, we refer to this process step as origin analysis.

5.2.1 Inconsistency Tracing

To find the latest model version which is not affected by the inconsistency under consideration, we need to be able to identify an inconsistency in different versions. In the sequel, let $[v_1, ..., v_n]$ be a version history, i.e., a sequence of model versions $v_1, ..., v_n$. An inconsistency that occurs in model version v_i is denoted by a tuple $\eta_i = (CR, o_i)$ where CR is a consistency rule which is violated on context element $o_i \in v_i$. In order to identify the inconsistency in the predecessor version v_{i-1}, we first have to identify the corresponding context element o_{i-1} in this version. If version v_{i-1} does not contain this context element, then the inconsistency does not exist in version v_{i-1}. Otherwise, a validation of the consistency rule CR is performed on the context element o_{i-1}. If the validation fails, the inconsistency exists in version v_{i-1}, and we repeat this predecessor check until we find a historic version V_A that is not affected by the inconsistency. If the inconsistency already exists in the initial model version v_1, we conceptually consider the empty model ϵ (see Section 3.1.3) to be the latest consistent version V_A.

As introduced in Section 3.4.1, a rather simple solution to identify corresponding model elements o_i and o_{i-1} is to use a model matcher [93]. Typically, a matcher delivers a set of correspondences, where each correspondence is a pair of model elements which are considered to be the same in versions v_i and v_{i-1}. A more sophisticated solution is to use the approach presented in Reference [193], which analyzes the correspondences of model elements over time in order to further optimize the identification of model elements, e.g., through the handling of breaks and gaps in the historical evolution of a model element. The selection of the most appropriate approach and model matcher depends on the characteristics of the models and the organization of model histories. Guidelines for this selection are out of the scope of this thesis but may be found, e.g., in References [80, 93, 193]. For the sake of efficiency, all the tracing information is saved and updated incrementally on demand, e.g., as meta-data in the version control system or by using a separate database as suggested in Reference [193].

In our modeling scenario introduced in Section 2.1.2, starting from the violation of consistency rule *message_signature(m:Message)* on context element 6:disconnect in Version 4 (see Figure 2.4b), we can easily identify this context element in the previous Version 3 (see Figure 2.3c) since the sequence diagram has not been changed at all in the inconsistency-inducing *edit step (4.1)*. No validation error of the same rule is found for that context element when validating against the class diagram in Version 3, thus this version is the latest version in the history that is not affected by the inconsistency.

5.2.2 Differencing

As a first step towards detecting the changes that may have caused the inconsistency, we calculate a structural difference (see Section 3.4.2) between the current inconsistent version V_B and the last consistent model version V_A. Corresponding model elements in V_A and V_B can be determined by exploiting the traces between model elements computed in the previous step (see Section 5.2.1).

An excerpt of the difference between the model Version 3 in Figure 2.3 and Version 4 in Figure 2.4 of our running example, including the changes of the inconsistency-inducing edit step is illustrated in Figure 5.1. Figure 5.1 shows the elementary changes in the difference graph $G_{\Delta(V_A,V_B)}$ that result from moving the operation disconnect() from class Video to Server. In the model difference $\Delta(V_A, V_B)$, the moving of the operation disconnect() is described by the following historical changes (see Section 3.3):

$c_{\langle model\ version\rangle.\langle edit\ step\rangle\langle change\rangle}$: ReferenceChange{ ⟨*action*⟩, ⟨*context*⟩, ⟨*type*⟩, ⟨*target*⟩ }

$c_{4.1a}$: ReferenceChange{ *delete*, Video:Class, *ownedOperation*, disconnect:Operation }

$c_{4.1b}$: ReferenceChange{ *delete*, disconnect:Operation, *class*, Video:Class }

$c_{4.1c}$: ReferenceChange{ *create*, Server:Class, *ownedOperation*, disconnect:Operation }

$c_{4.1d}$: ReferenceChange{ *create*, disconnect:Operation, *class*, Server:Class }

5.3 Impact Analysis

Our repair approach is based on the principle of complementing incomplete edit steps. Given a set of CPEOs, there are, in general, many possible partial executions of them in a difference. However, a detected partial execution of a CPEO and the resulting complementing changes are not necessarily inconsistency-inducing and -resolving, respectively. For example, the complementing changes can take place in a completely different part of the model or will be an optional extension of the model elements that do not directly affect the inconsistency. Technically, a partially executed CPEO can be related to a different inconsistency than the one currently being repaired. A similar problem occurs if a CPEO contains smaller yet consistent edit steps. Such edit steps will be recognized as partial executions regardless of the inconsistency. Such complements are not effective with respect to the task of repairing a specific inconsistency in a model. In this section, we will introduce a technique to discard such complements in the early phases of the repair algorithm, which also enhances its performance and scalability.

The *impact analysis*, illustrated in Step 3 in Figure 5.7 and Figure 5.8c, basically tests whether a change in a model has a positive or negative impact on a given inconsistency. In essence, a consistency rule either requires or forbids certain elements in the ASG of a model. Thus, a change performed on an ASG has a *positive* (or *negative*) *impact* on the inconsistency if it (i) creates (or deletes) a node or edge which is required by the respective consistency rule, (ii) deletes (or creates) a node or edge which is forbidden by the rule, or (iii) changes an attribute value which is forbidden (or required) by the rule. For a given inconsistency $\eta = (CR, e)$ and a change c, the predicates $positive(\eta, c)$ and $negative(\eta, c)$ state that the impact of c on η is positive and negative, respectively. Conversely, $\neg positive(\eta, c)$ (or $\neg negative(\eta, c)$) states that c does not have a positive (or negative) impact on η. The latter does not imply that a change has a negative (or positive) impact on the given inconsistency, it may also be neutral with respect to that inconsistency.

A change with a positive impact does not necessarily fully resolve an inconsistency, i.e., a positive impact indicates an improving change that may require additional changes (see Section 5.1). Likewise, an inconsistency does not necessarily have to be caused by a single change with a negative impact, i.e., an inconsistency can occur through a combination of changes. Therefore, our repair approach uses CPEOs to determine repairs for complex edit steps. In fact, the partial execution of a CPEO is considered as a repair if (i) the recognized sub-rule has a negative impact, and (ii) the complement rule has a positive impact on the inconsistency under consideration. In particular, the relation between the CPEO and a consistency rule that detects an inconsistency does not have to be explicitly known, i.e., the relation is determined by the impact analysis. A CPEO represents a complex edit step that is assumed to be consistency-preserving regarding a specific consistency rule. Therefore, complementing a partially executed CPEO typically implies that the inconsistency is resolved. As we only check for the positive impact, a complementation might require additional changes to resolve a specific inconsistency. In general, to check if a repair proposal actually resolves an inconsistency, an inefficient state space exploration would be required that simulates all possible applications of the complement rule.

As an example, let us consider the second part of the consistency rule *message_signature(m:Message)* described in Definition (5.1b, 5.1c). It states that the lifelines receiving a message must represent an instance of a class that contains the operation being specified as the signature of the message. Amongst others, two changes having a potential negative (or positive) impact are the removal (or addition) of an operation of a class. However, the inconsistency is only introduced (or resolved) if the removed (or added) operation is specified as the signature of the message.

The impact analysis can be implemented based on the set of abstract repairs R derived by the scope analysis in Section 5.1. Let $R(\eta)$ be the complete set of abstract repairs determined for a given inconsistency η. Technically, an abstract repair $\alpha \in R(\eta)$ is expressed by an abstract reference/attribute changes or an object change (action) (see Section 3.3). For a given abstract repair α the predicate $positive(\eta, \alpha)$ holds if $\alpha \in R(\eta)$. In particular, a concrete change c is contained in a set of abstract repairs $c \in R(\eta)$ if c is a valid instantiation of at least one abstract repair $\alpha \in R(\eta)$.

Definition (5.13) describes the instantiation check for the concrete change c and the abstract repair α that is represented by an abstract reference/attribute change. A concrete change c is a valid instantiation of α if c and α are of the same kind, and if the context bound by α is bound to the same model element by c. Parameters that are yet unbound in α may be bound to any element or value by c.

$$\begin{aligned}&\textit{isInitialization}(\alpha\textit{:AbstractReferenceChange, c:ReferenceChange}) \;:=\\&\textit{isInitialization}(\alpha\textit{:AbstractAttributeChange, c:AttributeChange}) \;:=\\&\quad \alpha.\mathsf{action} \equiv c.\mathsf{action}\\&\quad \alpha.\mathsf{context} \equiv c.\mathsf{context}\\&\quad \alpha.\mathsf{type} \equiv c.\mathsf{type}\end{aligned} \tag{5.13}$$

In addition, an abstract repair specifying the deletion of the context element is contained in $R(\eta)$. In this case, a concrete change c must exactly match the abstract repair α that is expressed by an concrete object change.

$$\begin{aligned}&\textit{isInitialization}(\alpha\textit{:ObjectChange, c:ObjectChange}) \;:=\\&\quad \alpha.\mathsf{action} \equiv c.\mathsf{action}\\&\quad \alpha.\mathsf{context} \equiv c.\mathsf{context}\end{aligned} \tag{5.14}$$

Definition (5.15) shows the instantiation check for the abstract repairs as they are generated by the repair operation of the AllInstances function (see Table 5.1). Here the context element of the concrete change must be assignable (see Definition (3.4)) to the context type specified by the abstract repair that is represented as an object change action.

$$\begin{aligned}&\textit{isInitialization}(\alpha\textit{:ObjectChangeAction, c:ObjectChange}) \;:=\\&\quad \alpha.\mathsf{action} \equiv c.\mathsf{action}\\&\quad \mathsf{isAssignableTo}(c.\mathsf{context.eClass}, \alpha.\mathsf{contextType})\end{aligned} \tag{5.15}$$

Such an implementation of testing abstract changes with respect to positive impacts can be re-used for implementing a negative impact test. The idea is to test whether the inverse change of c, say c^{-1}, has a positive impact on η. To invert a change, its *create* action is replaced with a *delete* action and vice versa. More formally, the following rules apply.

$$\textit{negative}(\eta, c) \iff \textit{positive}(\eta, c^{-1}) \text{ and } \neg\textit{negative}(\eta, c) \iff \neg\textit{positive}(\eta, c^{-1}) \tag{5.16}$$

5.3.1 Change Impact Analysis

As illustrated by Step 3 in Figure 5.8c, the *change impact analysis* takes the inconsistency under consideration as well as the model difference $\Delta(V_A, V_B)$ calculated in Step 2 as input. The goal is to identify the historical changes which may have caused the inconsistency η, referred to as inconsistency-inducing changes and denoted by $\Delta_\eta(V_A, V_B)$ with $\Delta_\eta(V_A, V_B) \subseteq \Delta(V_A, V_B)$. For each historical change $c \in \Delta(V_A, V_B)$, we test whether c has a negative impact on the inconsistency η.

$$\Delta_\eta(V_A, V_B) = \{c \mid c \in \Delta(V_A, V_B) \wedge \textit{negative}(\eta, c)\} \tag{5.17}$$

To give an illustration, consider the difference graph $G_{\Delta(V_A,V_B)}$ shown in Figure 5.1. The inconsistency-inducing edit step, i.e., moving the operation disconnect(), leads to four historical changes on the model's ASG, namely the deletion ($c_{4.1a}$, $c_{4.1b}$) and re-creation ($c_{4.1c}$, $c_{4.1d}$) of the edges of type ownedOperation and class between the disconnect():Operation node and the old owning Video:Class and new owning Server:Class nodes. The opposite containment edges are shown as a single undirected edge in the in Figure 5.1.

Among the four historical changes, the deletion $c_{4.1a}$ of the edge of type ownedOperation, between the class Server and the operation disconnect(), has a negative impact on the validation of the consistency rule *message_signature(m:Message)*. Intuitively, the message's receiver in the sequence diagram does not provide a suitable message signature anymore. For computing the negative impact, we compare the model difference $\Delta(V_A, V_B)$ with the abstract repairs R (see Section 5.1.6). In this case, we find the inverted historical change ${c_{4.1a}}^{-1}$ can be initialized from the abstract change $\alpha_{4.2}$ that suggests creating a new operation in the class Server. Notably, the *inverted* change ${c_{4.1a}}^{-1}$ specifies the creation of the operation disconnect() in class Video.

The fact that the disconnect() operation is moved to another class and not completely deleted from the model has no impact on the inconsistency, i.e., the two edge creations $c_{4.1c}$ and $c_{4.1d}$ of that relocation have a neutral impact on the inconsistency. Likewise, the path expression in the existence check of the consistency rule (see Definition (5.1b)) only incorporates edges of type ownedOperation. Thus, the deletion $c_{4.1b}$ of its opposite edge of type class is considered to have a neutral impact on the inconsistency, too. Finally, the set of inconsistency-inducing changes can be specified as a singleton set:

$\Delta_{\eta}(V_A, V_B) = \{$ $c_{4.1a}$: ReferenceChange{ *delete*, Video:Class, *ownedOperation*, disconnect:Operation } }

5.3.2 Action Impact Analysis

In contrast to the change impact analysis, the *action impact analysis*, shown as part of Step 3 in Figure 5.8c, works on the set of predefined CPEOs. Each CPEO is defined by an edit rule ER that specifies a set of change actions (see Section 3.5). For each ER, the goal is to determine which of the change actions defined by ER have a *potential positive* (or *negative*) *impact* on the given inconsistency η. The impact of a change action ca is *potential* in the sense that the conditions defined for checking $positive(\eta, ca)$ and $negative(\eta, ca)$ are necessary but not sufficient, i.e., the actual impact depends on the concrete arguments supplied to a change action. The result of the analysis is a set of annotated CPEOs comprising that information. We will later utilize these annotations in Step 4 (see Section 5.4). Repair proposals are only generated for CPEOs comprising at least one change action having a potential positive (and negative) impact on the given inconsistency.

Similar to the change impact analysis, the action impact analysis is based on the positive impact test $positive(\eta, \alpha)$ as introduced for abstract repairs. To determine the impact of an edit rule ER, this test is adapted to change actions by ignoring the context parameter bindings of the abstract repair. For a given change action ca and an inconsistency η, $positive(\eta, ca)$ holds if there exists an abstract repair $\alpha \in R(\eta)$ such that α can be instantiated from ca. Analogous to the negative impact, the potential negative impact can be derived from the potential positive impact.

$$negative(\eta, ca) \iff positive(\eta, ca^{-1}) \text{ and } \neg negative(\eta, ca) \iff \neg positive(\eta, ca^{-1}) \qquad (5.18)$$

In contrast to the instantiation checks in Definition (5.13) to Definition (5.15), here we check that the abstract repair α can potentially be initialized from the given change action *ca*. In terms of an abstract reference/attribute change α, this means that the context element must be assignable to the specified context type of the change action *ca*. Notably, the binding of the context element for *ca* can be further restricted by the edit rule, i.e., if the context node is also created, the types must match exactly. However, this case is handled by the sub-rule recognition described in Chapter 4 that also checks for the structural compatibility of changes.

$$
\begin{aligned}
&\mathit{isInitialization(ca{:}AbstractReferenceChange,\ \alpha{:}ReferenceChange)} \;:= \\
&\mathit{isInitialization(ca{:}AbstractAttributeChange,\ \alpha{:}AttributeChange)} \;:= \\
&\quad \mathit{ca}.\mathrm{action} \equiv \alpha.\mathrm{action} \land \\
&\quad \mathrm{isAssignableTo}(\alpha.\mathrm{context.eClass},\ \mathit{ca}.\mathrm{contextType}) \land \\
&\quad \mathit{ca}.\mathrm{type} \equiv \alpha.\mathrm{type}
\end{aligned} \tag{5.19}
$$

The above-mentioned case of object creation is handled by the object change initialization check in Definition (5.20). Notably, this may only be relevant for possible user-defined repair operations. The scope analysis in Section 5.1 only generates a context element removal as abstract repair α.

$$
\begin{aligned}
&\mathit{isInitialization(ca{:}ObjectChangeAction,\ \alpha{:}ObjectChange)} \;:= \\
&\quad \mathit{ca}.\mathrm{action} \equiv \alpha.\mathrm{action} \land \\
&\quad \mathit{ca}.\mathrm{action} \equiv \mathit{create} \Rightarrow \alpha.\mathrm{context.eClass} \equiv \mathit{ca}.\mathrm{contextType} \land \\
&\quad \mathit{ca}.\mathrm{action} \equiv \mathit{delete} \Rightarrow \mathrm{isAssignableTo}(\alpha.\mathrm{context.eClass},\ \mathit{ca}.\mathrm{contextType})
\end{aligned} \tag{5.20}
$$

For the abstract object changes *ca* and α, we have to check if any possible binding of *ca* is a valid initialization of α. An object deletion *ca* in an edit rule can be initialized with any assignable type. In contrast, object creations *ca* are initialized with the exact type of the change action. For abstract repairs, on the other hand, an abstract object changes α can always be initialized with any assignable type (see Definition (3.7)).

$$
\begin{aligned}
&\mathit{isInitialization(ca{:}ObjectChangeAction,\ \alpha{:}ObjectChangeAction)} \;:= \\
&\quad \mathit{ca}.\mathrm{action} \equiv \alpha.\mathrm{action} \land \\
&\quad \mathit{ca}.\mathrm{action} \equiv \mathit{create} \Rightarrow \mathrm{isAssignableTo}(\mathit{ca}.\mathrm{contextType},\ \alpha.\mathrm{contextType}) \land \\
&\quad \mathit{ca}.\mathrm{action} \equiv \mathit{delete} \Rightarrow \mathrm{isAssignableTo}(\mathit{ca}.\mathrm{contextType},\ \alpha.\mathrm{contextType}) \\
&\qquad\qquad\qquad\qquad\quad \lor \mathrm{isAssignableTo}(\alpha.\mathrm{contextType},\ \mathit{ca}.\mathrm{contextType})
\end{aligned} \tag{5.21}
$$

As an example, consider the consistency rule *message_signature(m:Message)* shown in Definition (5.1) and the CPEO *moveOperationAndChangeMessageTarget* shown in Figure 5.3. As illustrated by the edit rule graph of *moveOperationAndChangeMessageTarget*, to move an operation between classes and consistently change the receiving lifeline of a corresponding message, the following elementary change actions have to be performed:

$$
\begin{aligned}
&\mathit{ca}_{\langle corresponding\ to\ changes\rangle} : \ \mathrm{ReferenceChangeAction}\{\ \langle \mathit{action}\rangle, \langle \mathit{contextType}\rangle, \langle \mathit{type}\rangle, \langle \mathit{targetType}\rangle\ \} \\
&\mathit{ca}_{4.1a} : \ \mathrm{ReferenceChangeAction}\{\ \mathit{delete}, \mathrm{Class}, \mathit{ownedOperation}, \mathrm{Operation}\ \} \qquad \rangle\ \mathit{negative} \\
&\mathit{ca}_{4.1b} : \ \mathrm{ReferenceChangeAction}\{\ \mathit{delete}, \mathrm{Operation}, \mathit{class}, \mathrm{Class}\ \} \\
&\mathit{ca}_{4.1c} : \ \mathrm{ReferenceChangeAction}\{\ \mathit{create}, \mathrm{Class}, \mathit{ownedOperation}, \mathrm{Operation}\ \} \qquad \rangle\ \mathit{positive}
\end{aligned}
$$

$ca_{4.1d}$: ReferenceChangeAction{ *create*, Operation, *class*, Class }

$ca_{5.1a}$: ReferenceChangeAction{ *delete*, MessageEnd, *covered*, Lifeline } ⟩ *negative*

$ca_{5.1b}$: ReferenceChangeAction{ *delete*, Lifeline, *coveredBy*, MessageEnd }

$ca_{5.1c}$: ReferenceChangeAction{ *create*, MessageEnd, *covered*, Lifeline } ⟩ *positive*

$ca_{5.1d}$: ReferenceChangeAction{ *create*, Lifeline, *coveredBy*, MessageEnd }

The change action $ca_{5.1c}$ of type covered, which creates the connection of the message end with the receiver's lifeline may have a positive impact on the inconsistency. Intuitively, this action can change the receiving lifeline of message 6:disconnect to another lifeline that potentially resolves the inconsistency, i.e., a lifeline representing a class containing a suitable operation signature. Technically, we compare the change actions of the edit rule *moveOperationAndChangeMessageTarget* to the set of abstract repairs $R(\eta)$, as computed in Section 5.1.6. In this context, the abstract repair $\alpha_{4.5}$ represents a valid instantiation of the change action $ca_{5.1c}$ by binding the context to the model element disconnectReceive:MessageEnd (see Figure 5.1).

In addition, the change action $ca_{4.1c}$ creating the reference ownedOperation, during the moving of the operation, has a potential positive impact. The abstract change $\alpha_{4.2} \in R(\eta)$ initializes the change action $ca_{4.1c}$ by binding the context to the element Video:Class (see Figure 5.1). However, this potential positive impact is not relevant in our particular repair scenario. In general, this could be useful in a scenario in which we change the receiver of the message first and complement that edit step by moving the operation. This is due to the fact that the *potential* impact analysis on the edit rule does not consider the actual historical changes yet.

Likewise, the $negative(\eta, ca)$ impact is checked and annotated for the change actions ca of the edit rule ER. In the CPEO *moveOperationAndChangeMessageTarget* shown in Figure 5.3 the change action $ca_{4.1a}$ removing the operation from its containing class is annotated as potential inconsistency-inducing change. This case of a potential negative impact is analogous to the actual negative impact detected in the model difference in Section 5.3.1 that caused the inconsistency by moving the operation disconnect() from class Video to the Server class. In terms of the potential impact analysis, the inverted abstract change ${\alpha_{4.2}}^{-1}$ initializes the change action $ca_{4.1a}$ with the context element Video:Class (see Figure 5.1).

Contrary to the positive impact, the change action $ca_{5.1a}$ in the edit rule, which removes a message from a receiving lifeline, has a potential negative impact. This change action can be initialized as the inverted abstract change ${\alpha_{4.5}}^{-1}$ by binding the context element to disconnectReceive:MessageEnd (see Figure 5.1). Likewise, this potential negative impact is not relevant in our particular repair scenario but could be useful in the scenario mentioned above.

5.4 Repair Generation

The general idea of considering model repair as a complementation of partially executed CPEOs is illustrated in Figure 5.12. As a refined version of Figure 2.8, the Figure 5.12 visualizes a difference as a sequence of *concrete changes*. Similarly, a CPEO is visualized as a sequence of *change actions.* A subset of the change actions of the CPEO, together with a subset of concrete historical changes of the model difference, represents the partial execution of a CPEO. In order to generate a repair for the inconsistent model V_B, the remaining change actions of the CPEO need to be bound to concrete elements and values of the model's ASG. Chapter 4 describes the detection of partially executed CPEOs in model difference. In the following Section 5.4.1 to Section 5.4.3, we adapt this technique to construct case-specific repair operations.

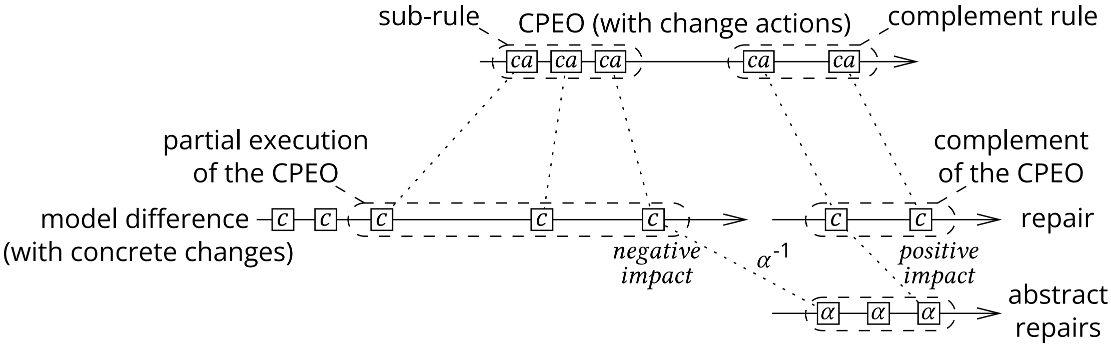


Figure 5.12. Changes in a difference and in a CPEO involved in the generation of repair proposals through complementing partially executed CPEOs.

In the context of history-based model repair, a principal question is how models can become inconsistent during an editing process. We exclude reasons such as system crashes, editing of the internal structure, etc., which "physically" damage a model. Our approach cannot repair such damages; this is beyond the scope of our approach and must be addressed by recovery techniques, notably by backup copies. We assume that models are only changed by tools that maintain a minimum degree of consistency. If one starts an editing process with an empty model, which is consistent, and later arrives at an inconsistent model, then in general, two kinds of modifications are possible to restore the model consistency. The sequence of edit steps that leads to an inconsistent model can be continued to reach a new consistent state of the model, or several edit steps can be undone to reach an older consistent state of the model. That leads to the following general repair scenarios:

Complementing edit step: The editing process can be meaningfully continued without undoing previous edit steps. This case is primarily addressed by our repair approach.

Rollback edit step: The editing process can, in principle, be continued to repair the model without undoing previous edit steps, but this does not make sense from a developer's point of view, and some or all of the former edit steps are undone at the discretion of the designer.

Our repair tool supports the developer in this case by showing the edit steps that caused the inconsistency and allowing them to make an informed decision to undo those modifications. This case-specific rollback operation is constructed by inverting the changes of the detected partially executed CPEO.

Replacing edit step: A repair will undo some previous modifications that do not directly repair the inconsistency, and after that, some new repairing changes will be applied on the model. This can be seen as equivalent to, but more efficient than, first completely undoing an incorrect edit step (a) and then redoing a similar but correct edit step.

Our approach can support these kinds of repairs for correcting attribute values of existing model fragments. For example, if a model fragment was created correctly with respect to its structure but some attribute values are specified incorrectly. Based on a corresponding CPEO, the repair tool can suggest a complement rule that overwrites those values with corrected values.

As a basis for generating history-based repair recommendations, we use the approach introduced in the previous Chapter 4 for detecting and complementing partial executions of edit rules in a model difference. Given an inconsistency and a model difference comprising changes that potentially caused the inconsistency, we have to find related partial applications of CPEOs in this difference that can be complemented to resolve the inconsistency. For a CPEO specified by an edit rule ER, this means that we want to find a sub-rule $ER_{sub} \subset ER$ whose effect is observable in the model difference $\Delta(V_A, V_B)$. At the same time, the sub-rule must be related to the inconsistency-inducing changes, i.e., we are only interested in sub-rules that might have caused the inconsistency. Next, we derive the corresponding complement rule $\overline{ER} = ER \setminus ER_{sub}$ and search for possible applications of that rule with the potential to resolve the inconsistency under consideration. Technically, the algorithm discussed in Chapter 4 allows to refine the context for detecting and complementing partially executed edit rules. Therefore, the historical impact of sub-rules and the complementary impact of complement rules, introduced in Section 4.6.1 and Section 4.6.2, are configured according to the inconsistency to be repaired.

5.4.1 Sub-Rule Recognition

As discussed in Chapter 4, while there may be many sub-rules of an edit rule and many occurrences of these sub-rules in a given difference, we are only interested in those meeting the conditions for syntactically correct sub-rules. For the preservation of elementary ASG consistency (see Section 3.3.3), all atomic change sets and their dependencies are identified in the CPEO *moveOperationAndChangeMessageTarget* in Figure 5.3. The edges of type ownedOperation and class used in our CPEO of Figure 5.3, which move the operation in the AST, must be treated as atomic fragments. Moreover, the pairs of edges of type covered and coveredBy are opposite edges that form an atomic change set.

As illustrated by Step 4 in Figure 5.9a, the *repair generation* is initialized with the model difference computed in Step 2 and a CPEO to be processed. Step 4 shows the processing of a single CPEO. Technically, all CPEOs can be processed in parallel, and the results are finally collected for the repair proposal ranking depicted in Step 5 of Figure 5.9b.

A sub-rule ER_{sub} of an edit rule ER must contain at least one change action that has a potential negative impact with respect to inconsistency η, as defined in the analysis in Section 5.3.2. This information can be obtained from the CPEO annotations calculated by the action impact analysis described in Section 5.3.2. In terms of the detection of partially executed edit rules in Section 4.6.1 of Chapter 4, at least one change action with a potential historical impact ER_μ is required; otherwise, the edit rule is ignored. Therefore, the potential historical impact ER_μ simply corresponds to the change actions annotated with a potential negative impact.

$$ER_\mu = \{ca \in ER \mid negative(\eta, ca)\} \tag{5.22}$$

This condition is fulfilled for the CPEO *moveOperationAndChangeMessageTarget* in Figure 5.3. As determined in Section 5.3.2, the change actions $ca_{4.1a}$ and $ca_{5.1a}$ deleting the edge of type ownedOperation and covered are annotated as potential negative impact.

At least one change action of ER_{sub} with a potential negative impact must finally be mapped to a historical change in the set of inconsistency-inducing changes. In other words, ER_{sub} may indeed have caused the inconsistency. The negative impact is defined by the inconsistency-inducing changes $\Delta_\eta(V_A, V_B)$ calculated by the change impact analysis described in Section 5.3.1. Therefore, the detection of partially executed edit rules in Section 4.6.1 can be restricted by specifying an actual historical impact $\Delta_\mu(V_A, V_B)$ using the inconsistency-inducing changes $\Delta_\eta(V_A, V_B)$.

$$\Delta_\mu(V_A, V_B) = \Delta_\eta(V_A, V_B) \tag{5.23}$$

For our running example, a sub-rule of the CPEO shown in Figure 5.3 that just moves the operation in the class diagram can be recognized in the model difference $\Delta(V_A, V_B)$ illustrated in Figure 5.1. The change action $ca_{4.1a}$ that deletes the edge of type ownedOperation can be mapped to a concrete historical change $c_{4.1a} \in \Delta_\eta(V_A, V_B)$, i.e., a change in the difference $\Delta(V_A, V_B)$ which, as an outcome of the change impact analysis (see Section 5.3.1), is known to have a negative impact on the inconsistency η under consideration. The following listing shows the mapping of change actions to changes in the model difference, including the inverted abstract change identified in the impact analysis:

$ca_{4.1a}$: ReferenceChangeAction{ *delete*, Class, *ownedOperation*, Operation } $\rangle$ *negative*
 $\mapsto$ $\alpha_{4.2}{}^{-1}$: AbstractReferenceChange{ *delete*, Video:Class, *ownedOperation* }
  $\mapsto$ $c_{4.1a}$: ReferenceChange{ *delete*, Video:Class, *ownedOperation*, disconnect:Operation }

$ca_{4.1b}$: ReferenceChangeAction{ *delete*, Operation, *class*, Class }
 $\mapsto$ $c_{4.1b}$: ReferenceChange{ *delete*, disconnect:Operation, *class*, Video:Class }

$ca_{4.1c}$: ReferenceChangeAction{ *create*, Class, *ownedOperation*, Operation }
 $\mapsto$ $c_{4.1c}$: ReferenceChange{ *create*, Server:Class, *ownedOperation*, disconnect:Operation }

$ca_{4.1d}$: ReferenceChangeAction{ *create*, Operation, *class*, Class }
 $\mapsto$ $c_{4.1d}$: ReferenceChange{ *create*, disconnect:Operation, *class*, Server:Class }

5.4.2 Complement Rule Matching

As described in Section 4.1, based on a recognized sub-rule, a corresponding complement rule $\overline{ER} = ER \setminus ER_{sub}$ can be derived. Conversely to the potential negative impact, at least one of the remaining change actions in the complement rule $\overline{ER}$ must have a potential positive impact on that inconsistency η. In terms of the partial edit rule detection in Section 4.6.2 of Chapter 4, at least one change action with a potential complementary impact ER_α is required; otherwise, the edit rule is ignored. Therefore, the potential complementary impact ER_α is defined by the change actions of the CPEO annotated with a potential positive impact.

$$ER_\alpha = \{ca \in ER \mid positive(\eta, ca)\} \tag{5.24}$$

Regarding our running example, given the sub-rule of CPEO *moveOperationAndChangeMessageTarget* in Figure 5.3 that moves an operation between two classes, the corresponding complement rule changes the target end of a message to another lifeline. Here, the complement rule does not comprise the change actions of types *ownedOperation* and *class*, which are already executed by the sub-rule. In this case, as determined by the action impact analysis in Section 5.3.2, the change action $ca_{5.1c}$ creating the edge of type covered indicates a complementary impact.

To ensure a positive impact of the complement rule $\overline{ER}$ application with respect to the inconsistency η, at least one change action with a potential positive impact must represent an initialization of an abstract change in $R(\eta)$. In other words, applying $\overline{ER}$ may improve or resolve the inconsistency under consideration. Likewise, the detection of partially executed edit rules in Section 4.6.2 can be configured with an actual complementary impact Δ_α. As illustrated by Step 4 in Figure 5.7, the complementary impact Δ_α is defined by the abstract repairs $R(\eta)$ computed in Step 1.

$$\Delta_\alpha = R(\eta) \tag{5.25}$$

Regarding the derived complement rule of our running example, the creation $ca_{5.1c}$ of the edge of type covered with a potential positive impact on the inconsistency must be initialized. In our current version of the VoD-System in Figure 5.1, we can find two valid application of the complement rule. One application changes the receiver of the message to the mirror:Server lifeline and the other one changes the receiver to the main:Server lifeline. Both complement rule applications have an actual complementary impact as they initialize the abstract change $\alpha_{4.5}$ with the receiving end of the message 6:disconnect. The following listing shows the initialized (abstract) changes for each change action of the complement rule as it is applied to model Version 4 resulting in model Version 5 illustrated in Figure 5.1 and Figure 5.2, respectively:

$ca_{5.1a}$: ReferenceChangeAction{ *delete*, MessageEnd, *covered*, Lifeline }
  ↦ $c_{5.1a}$: ReferenceChange{ *delete*, disconnectReceive:MessageEnd, *covered*, open:Lifeline }

$ca_{5.1b}$: ReferenceChangeAction{ *delete*, Lifeline, *coveredBy*, MessageEnd }
  ↦ $c_{5.1b}$: ReferenceChange{ *delete*, open:Lifeline, *coveredBy*, disconnectReceive:MessageEnd }

$ca_{5.1c}$: ReferenceChangeAction{ *create*, MessageEnd, *covered*, Lifeline } ⟩ *positive*
  ↦ $\alpha_{4.5}$: AbstractReferenceChange{ *create*, disconnectReceive:Lifeline, *covered* }
    ↦ $c_{5.1c}$: ReferenceChange{ *create*, disconnectReceive:MessageEnd, *covered*, mirror:Lifeline }

$ca_{5.1d}$: ReferenceChangeAction{ *create*, Lifeline, *coveredBy*, MessageEnd }
  ↦ $c_{5.1d}$: ReferenceChange{ *create*, mirror:Lifeline, *coveredBy*, disconnectReceive:MessageEnd }

5.4.3 Repair Construction

A repair suggested by our approach is a complement rule computed by the approach introduced in Chapter 4. Technically, a complement rule is defined by an in-place model transformation rule supplied with concrete parameters. The final task to come up with a concrete repair proposal is to turn all possible applications of a complement rule into a list of selectable parameter bindings. Therefore, each parameter is associated with a domain that collects the model elements of all complement rule applications. The developer can select the appropriate model elements if multiple alternatives are available. Only those parameters that lead to countless alternatives, e.g., names of new model elements, have to be specified by the developer.

In our running example, the parameters *moveOperation*, *toClass*, and *changeMessage* of the complement rule of the CPEO *moveOperationAndChangeMessageTarget* in Figure 5.3 are bound to model elements disconnect(), Server, and 6:disconnect in the model version V_B, respectively. Finally, the developer must select a specific lifeline for the parameter *toLifeline*. In Version 5 of the VoD-System in Figure 2.5b the developer can choose between the main:Server and the mirror:Server lifeline.

Technically, the selection of each parameter restricts the domains of other, not yet selected, parameters. Such restrictions can be computed incrementally by a CSP solver in Chapter 4. The domains of the complement rule are used as initial domains for the application context of the complement rule. Each time a parameter is selected, the CSP solver computes the remaining possible application context matchings. Finally, the application context matchings are propagated back to the parameter domains of the repair proposal.

5.5 Repair Ranking

Re(pair)Vision proposes repairs as pairs of a sub-rule and a complement rule. The repair proposals are ranked based on multiple criteria of both of these rules. The inverted changes of the sub-rule correspond to a rollback and the complement rule to the complementing repair, i.e., we rank and propose two kinds of repair at the same time. While each pair of a complement and a rollback is distinct in the list of repair proposals (by construction), it is likely that a rollback will be recurring. However, we can conclude from our experimental results (see Chapter 8) that a complementation is more likely in practice, preserving the efforts of former changes. In order to rank the calculated repair proposals, we consider the number of changes from the complement rule $|\overline{R}|$ as well as the historical changes of the sub-rule $|R_{sub}|$. The repairs with the greatest ratio $|R_{sub}|$ / $|\overline{R}|$ of historic and complementing changes, referred to as *change ratio* in the sequel, will be preferred. A large overlap of a CPEO with the history can be seen as an indication of an uncompleted edit step. On the other hand, a small complement rule leads to a repaired model that is close to the original model, referred to as the "least-change principle" in Reference [115].

If two repair proposals share the same change ratio, we prefer the repair with the larger sub-rule, i.e., a large sub-rule is more likely an actually uncompleted edit step than a smaller one. In order to consider the level of concreteness of the repair alternatives, the repair with the smallest number of unbound parameters is subsequently preferred. In general, modifications that just add new elements to a model are commonly less critical than modifications that delete model elements, e.g., in terms of meta-models and already existing model instances which would need migrations after deleting type definitions. Therefore, in the next ranking step, the repair with the larger number of creation changes minus the number of deletion changes is preferred.

In the unlikely case that all of the aforementioned ranking criteria do not break ties, we keep the remaining partial order on the ranked list of repair proposals. In the respective GUI component of our interactive repair tool (see Chapter 6), such proposals are presented in groups, accounting for the fact that they are ranked on the same position.

In our repair scenario in Figure 5.1, the developer selects applies the repair proposal that changes the receiver of message 6:disconnect from lifeline open:Video to mirror:Server. The result of applying this complementation is shown in Figure 5.2.

5.6 Undo Generation

In some cases, developers may find no reasonable complementation of their editing process and cannot repair the model without undoing previous edit steps. Thus, some or all of the former edit steps are undone at the discretion of the developer. Our repair tool supports this case by showing the edit steps that caused the inconsistency and by enabling developers to make the informed decision to undo those modifications.

For every partially applied CPEO that may have caused the inconsistency under consideration, a case-specific undo operation that inverts the changes of the recognized sub-rule can be constructed instantly upon request. Such a rollback is constructed as described in Section 4.9. As mentioned in Section 4.9, a rollback can cause side effects that also undo dependent changes, of which the repair tool should make the developer aware.

Our repair tool basically shows all recognized edit steps alongside with their complementing edit steps derived from the CPEO. The undo operation is calculated instantly by inverting the changes of a recognized sub-rule (see Figure 5.9c). The developer can either choose to apply a complement rule or undo a recognized sub-rule. Our approach provides no recommendation on whether to prefer undoing or complementing a partially applied CPEO; this is left at the discretion of the developers. Applying an undo operation causes the rollback of the inconsistency-inducing changes. For example, the undo operation in the scenario of Figure 5.1 would be to move the operation disconnect() from the class Server back to the class Video. This rollback would also resolve the inconsistency with respect to the message 6:disconnect.

5.7 Iterative Repair Process

Following the requirements analyzed in Section 1.2, the understandability of proposed repairs is of utmost importance for a repair recommendation system, while partial solutions may be accepted. On purpose, a repair recommended by our approach is not guaranteed to be free of negative side effects since this can lead to complex repair proposals and a possible state space explosion. The repair process described in Figure 5.7 breaks up such complex repair scenarios by focusing on a single inconsistency at a time. After performing a model repair, ReVision automatically triggers a re-validation of the resulting model, i.e., starting a new iteration beginning with Step 1 in Figure 5.7. If the repair had a negative side effect, then all new inconsistencies have to be repaired subsequently.

As shown in Figure 5.1 of the VoD-System, this is the case for the sender of the message 6:disconnect, since the sending class Server can not reference an object of type User through an owned property. This inconsistency η is detected by the consistency rule *message_property(m:Message)* specified in Definition (5.2).

For the sake of brevity, we will only discuss the abstract repairs $\alpha \in R(\eta)$ that are relevant to our repair scenario. As shown by the validation scope in Figure 5.2, the validation of the consistency rule *message_property(m:Message)* checks the sending and receiving ends of the 6:disconnect message by the existential quantifiers. Therefore, creating different message ends may have a positive impact on the validation. Analyzing the negative impact of the difference graph in Figure 5.2, we find the deletion of the covered edge of $c_{5.1a} \in \Delta(V_A, V_B)$:

$c_{\langle model\ version\rangle.\langle edit\ step\rangle\langle change\rangle}$: ReferenceChange{ ⟨*action*⟩, ⟨*context*⟩, ⟨*type*⟩, ⟨*target*⟩ }

$c_{5.1a}$: ReferenceChange{ *delete*, disconnectReceive:MessageEnd, *covered*, open:Lifeline }

$c_{5.1b}$: ReferenceChange{ *delete*, open:Lifeline, *coveredBy*, disconnectReceive:MessageEnd }

$c_{5.1c}$: ReferenceChange{ *create*, disconnectReceive:MessageEnd, *covered*, mirror:Lifeline }

$c_{5.1d}$: ReferenceChange{ *create*, mirror:Lifeline, *coveredBy*, disconnectReceive:MessageEnd }

In the next repair iteration, as described in the second corrective *edit step (6.1)* in Figure 5.2, the developer changes the sender of the message 6:disconnect to the open:Video lifeline. This time, the previous repair described by the first corrective *edit step (5.1)*, which changed the receiver of the message 6:disconnect, is recognized as an incomplete, inconsistency-inducing edit step. This repair proposal can be computed based on the CPEO *moveMessage* shown in Figure 5.5. The sub-rule is formed by the change actions $ca_{5.1a}$, $ca_{5.1b}$, $ca_{5.1c}$, and $ca_{5.1d}$ that change the receiving message end. This sub-rule is mapped to the changes of *edit step (5.1)* in the difference graph in Figure 5.2:

$ca_{5.1a}$: ReferenceChangeAction{ *delete*, MessageEnd, *covered*, Lifeline } ⟩ *negative*
$\mapsto \alpha_{5.2}{}^{-1}$: AbstractReferenceChange{ *delete*, disconnectReceive:MessageEnd, *covered* }
$\mapsto c_{5.1a}$: ReferenceChange{ *delete*, disconnectReceive:MessageEnd, *covered*, open:Lifeline }

$ca_{5.1b}$: ReferenceChangeAction{ *delete*, Lifeline, *coveredBy*, MessageEnd }
$\mapsto c_{5.1b}$: ReferenceChange{ *delete*, open:Lifeline, *coveredBy*, disconnectReceive:MessageEnd }

$ca_{5.1c}$: ReferenceChangeAction{ *create*, MessageEnd, *covered*, Lifeline }
$\mapsto c_{5.1c}$: ReferenceChange{ *create*, disconnectReceive:MessageEnd, *covered*, mirror:Lifeline }

$ca_{5.1d}$: ReferenceChangeAction{ *create*, Lifeline, *coveredBy*, MessageEnd }
$\mapsto c_{5.1d}$: ReferenceChange{ *create*, mirror:Lifeline, *coveredBy*, disconnectReceive:MessageEnd }

Based on this sub-rule, the corresponding complement rule is derived from the edit rule in Figure 5.5. As shown in the final Version 6 of the VoD-System in Figure 2.6, the complement will change the sender of the message 6:disconnect from lifeline Alice:User to open:Video. This complement rule application has a positive impact with respect to the inconsistency, i.e., the action $ca_{6.1c}$ changing the sender of the message initializes an abstract repair with respect to the inconsistency. Applying this repair proposal complements *edit step (5.1)* and restores the consistency of the VoD-System. This results in the following changes with respect to model Version 5 initialized by the complement rule:

$ca_{6.1a}$: ReferenceChangeAction{ *delete*, MessageEnd, *covered*, Lifeline }
$\mapsto c_{6.1a}$: ReferenceChange{ *delete*, disconnectSend:MessageEnd, *covered*, Alice:Lifeline }

$ca_{6.1b}$: ReferenceChangeAction{ *delete*, Lifeline, *coveredBy*, MessageEnd }
$\mapsto c_{6.1b}$: ReferenceChange{ *delete*, Alice:Lifeline, *coveredBy*, disconnectSend:MessageEnd }

$ca_{6.1c}$: ReferenceChangeAction{ *create*, MessageEnd, *covered*, Lifeline } $\rangle$ *positive*
$\mapsto \alpha_{5.4}$: AbstractReferenceChange{ *create*, disconnectSend:Lifeline, *covered* }
$\mapsto c_{6.1c}$: ReferenceChange{ *create*, disconnectSend:MessageEnd, *covered*, open:Lifeline }

$ca_{6.1d}$: ReferenceChangeAction{ *create*, Lifeline, *coveredBy*, MessageEnd }
$\mapsto c_{6.1d}$: ReferenceChange{ *create*, open:Lifeline, *coveredBy*, disconnectSend:MessageEnd }

6

Presentation of History-based Recommendations

This chapter presents the user interface of ReVision developed as a proof of concept for history-based model repair recommendation. In particular, the lack of tool support is identified as one of the major challenges faced by MDE practitioners [131]. ReVision offers a view that enhances the model editor, allowing developers to inspect a ranked list of history-based repair proposals for a selected inconsistency. Each repair proposal specifies a detected partially executed edit operation and its possible completions with respect to the inconsistency. In addition, for each of these repair proposals, a case-specific undo operation can be created upon request. The chapter also discusses the technological background and implementation insights of the tool.

We presented our interactive tool ReVision that supports developers who have to repair inconsistent models in MDE environments. Our tool, called ReVision, is implemented on top of the widely used Eclipse modeling technology stack. It is publicly available at Reference [8].

As discussed in Chapter 5, we assume that inconsistencies are introduced by former editing processes that are incomplete in the sense that additional changes are necessary to achieve a new consistent state. Assuming that the evolution of a model is managed in a version control system, the tool locates incomplete editing processes in the version history of a model. Our tool enables developers to complete such an incomplete editing process and to catch up on the missing changes. At the same time, our approach prevents undoing former edit steps. Alternatively, we enable developers to make the informed decision to simply undo these changes. As defined in Section 5.5, the generated repair proposals are ranked using properties of the history-based repair proposals. The tool supports the developer in iteratively repairing each single violation of a consistency rule.

In order to prevent inconsistencies, a restrictive editing environment could force developers to use only CPEOs for editing a model. However, this would make editing very clumsy, similar to syntax-directed editors, which are not very popular in practice [87]. Moreover, developers typically follow a task-oriented model editing process, e.g., focusing on a dedicated view, as in our example. Based on such a conceptual focus of one developer, it might also happen that inconsistencies have to be migrated by another developer who did not introduce them. As a consequence, inconsistencies have to be tolerated temporarily but must be resolved eventually.

The main idea of our approach is to consider CPEOs as ideal edit operations and to recommend the “gap” between ideal edits and the edits that have caused an inconsistency as model repairs. In particular, the tool works offline, i.e., no monitoring of the local workspace modifications is needed. In the following sections, we give a brief summary of the underlying technologies and design decisions (see Section 6.1) as well as the user interface of the tool (see Section 6.2).

6.1 Basic Technologies and Design Decisions

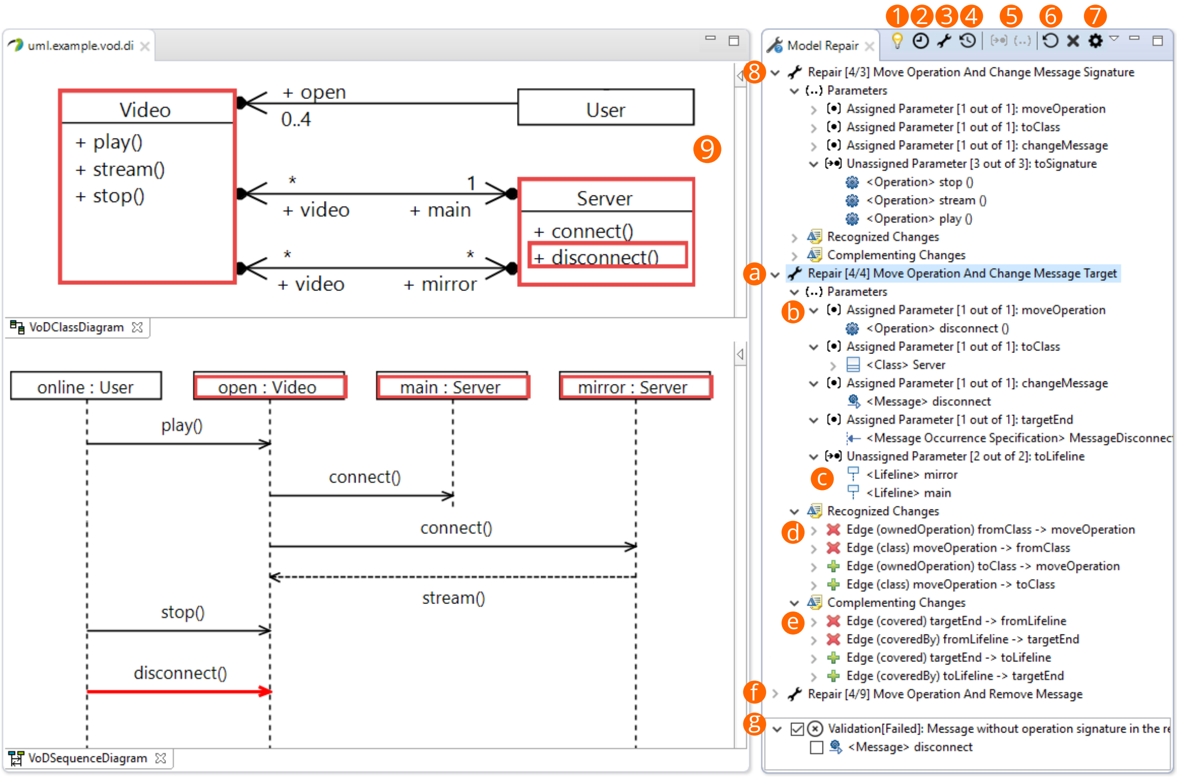
We implemented our approach to model repair as a plug-in for the Eclipse Modeling Framework (EMF) [175]. The plug-in is available from the project website of ReVision [8].

Our implementation relies on the EMF core components and on further EMF-based technologies. For specifying CPEOs (see Section 3.5), we utilize the model transformation tool Henshin [13] which is based on graph transformation concepts as used in our approach. For the calculation of model differences (see Section 3.4), we build on the model differencing framework SiLift [1] which implements the approach presented in Reference [82]. In order to determine corresponding model elements in successive model versions during inconsistency tracing in a version history (see Section 5.2), ReVision provides an extension point that can be configured with different model matching algorithms according to the modeling domain and development process (see ⑦ Figure 6.1). The scope analysis (see Section 5.1) is implemented by utilizing the concept of ASG-based repair actions as implemented in the tool Model/Analyzer [148]. To be used in our tooling environment, we re-implemented the basic concepts of the Model/Analyzer algorithm [149] for EMF-based models.

6.2 Repair Tool

To give an impression of how to use our repair tool, particularly how the generated repair proposals are presented to the developer, Figure 6.1 shows the interactive GUI components during the first corrective *edit step (5.1)* of our running example as described in Section 2.1 and shown in Section 2.4. ReVision is placed in an additional view next to the model editor. The validation of the currently opened model detects the inconsistency *message_signature(m:Message)* (see ① in Figure 6.1). Selecting this inconsistency ⓖ invokes the repair calculation. As illustrated in Figure 6.1, the repairs are presented as a list in which every entry represents the partial execution of a CPEO. In this example, change ratios induce a totally ordered ranking of the repair proposals. Every repair proposal contains three sets of information: ⓑ the parameters of the CPEO, ⓓ the change actions of the recognized sub-rule, including the inconsistency-inducing changes, and ⓔ the change actions of the complement rule that are proposed as a repair. As shown in the class and sequence diagram view in Figure 6.1, by selecting a repair proposal, parameters, or change sets in the repair view, the involved model elements are highlighted ⑨ in the diagram.

For the first corrective *edit step (5.1)*, ReVision proposes three possible repair alternatives. All of them are based on the same recognized change set, which includes the



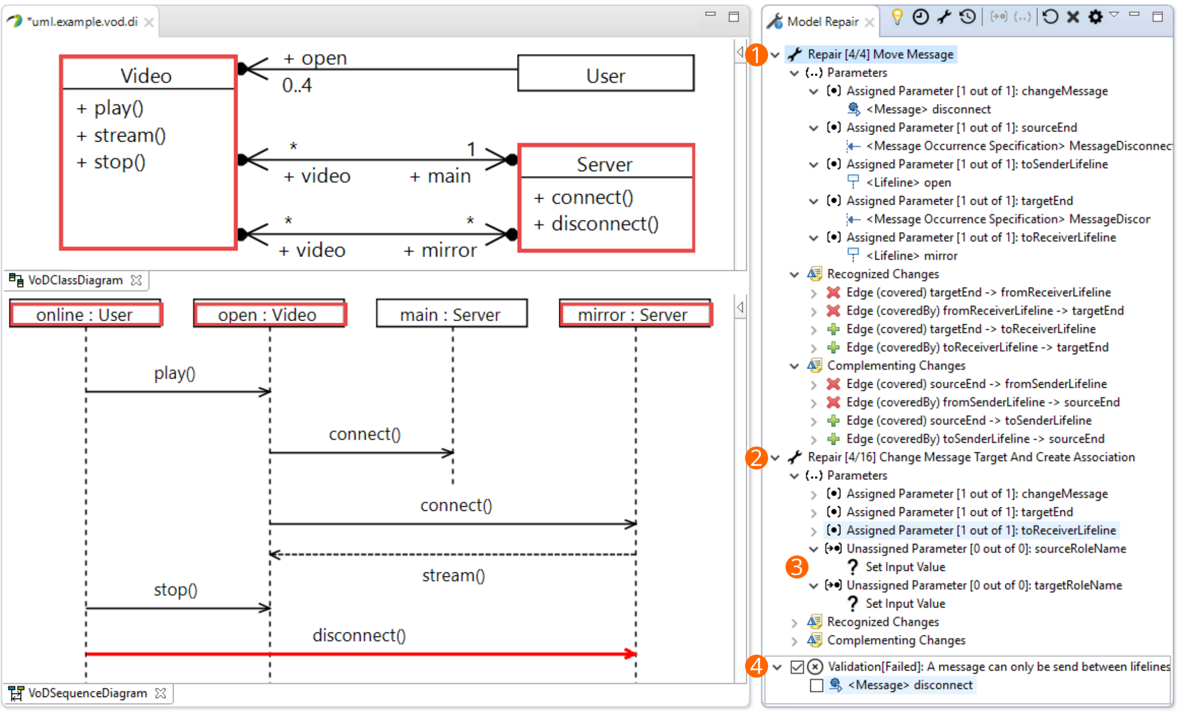
```
Repair [4/3] Move Operation And Change Message Signature
  (..) Parameters
    (•) Assigned Parameter [1 out of 1]: moveOperation
    (•) Assigned Parameter [1 out of 1]: toClass
    (•) Assigned Parameter [1 out of 1]: changeMessage
    (→•) Unassigned Parameter [3 out of 3]: toSignature
        <Operation> stop ()
        <Operation> stream ()
        <Operation> play ()
  Recognized Changes
  Complementing Changes
Repair [4/4] Move Operation And Change Message Target
  (..) Parameters
    (•) Assigned Parameter [1 out of 1]: moveOperation
        <Operation> disconnect ()
    (•) Assigned Parameter [1 out of 1]: toClass
        <Class> Server
    (•) Assigned Parameter [1 out of 1]: changeMessage
        <Message> disconnect
    (•) Assigned Parameter [1 out of 1]: targetEnd
        <Message Occurrence Specification> MessageDisconnec
    (→•) Unassigned Parameter [2 out of 2]: toLifeline
        <Lifeline> mirror
        <Lifeline> main
  Recognized Changes
    ✗ Edge (ownedOperation) fromClass -> moveOperation
    ✗ Edge (class) moveOperation -> fromClass
    + Edge (ownedOperation) toClass -> moveOperation
    + Edge (class) moveOperation -> toClass
  Complementing Changes
    ✗ Edge (covered) targetEnd -> fromLifeline
    ✗ Edge (coveredBy) fromLifeline -> targetEnd
    + Edge (covered) targetEnd -> toLifeline
    + Edge (coveredBy) toLifeline -> targetEnd
Repair [4/9] Move Operation And Remove Message

☑ ⊗ Validation[Failed]: Message without operation signature in the re
    ☐ <Message> disconnect
```

Figure 6.1. ReVision: First corrective edit step *edit step (5.1)* of the running example.

inconsistency-inducing changes moving the operation disconnect() between the classes Video and Server. To understand the inconsistency-inducing changes, the developer can also open and compare the last consistent version of the model with respect to the selected inconsistency (see ② in Figure 6.1).

The first repair alternative ⑧ proposes to change the signature of the message to one of the operations contained by class Video. The second repair ⓐ corresponds to the desired *edit step (5.1)*, as described in our running example of Section 2.1, which changes the target end of the message 6:disconnect. This repair can be parameterized with the new target lifeline that must be an instance of class Server (see ⓑ in Figure 6.1). Parameters are automatically assigned to a repair operation as long as their assignment is unique. Otherwise, a specific parameter value must be assigned by the developer. ReVision found two concrete options for the parameter toLifeline representing the new target lifeline. As in our running example, the developer can choose between the lifelines main:Server and mirror:Server. The third and last repair ⓕ alternative resolves the inconsistency by completely removing the message 6:disconnect from the sequence diagram.

As in our running example, we choose the second repair alternative *moveOperationAndChangeMessageTarget* ⓐ and select ⑤ the lifeline mirror:Server ⓒ for the parameter toLifeline. After applying ⓐ the repair, which changes the target of the message 6:disconnect to the lifeline mirror:Server, the inconsistency is resolved. In particular, all applied repairs are stored on an editing stack so that every repair step can be undone ⑥, allowing the developer to explore different repair alternatives.



```
Repair [4/4] Move Message
  {..} Parameters
    Assigned Parameter [1 out of 1]: changeMessage
      <Message> disconnect
    Assigned Parameter [1 out of 1]: sourceEnd
      <Message Occurrence Specification> MessageDisconnec
    Assigned Parameter [1 out of 1]: toSenderLifeline
      <Lifeline> open
    Assigned Parameter [1 out of 1]: targetEnd
      <Message Occurrence Specification> MessageDiscor
    Assigned Parameter [1 out of 1]: toReceiverLifeline
      <Lifeline> mirror
  Recognized Changes
    Edge (covered) targetEnd -> fromReceiverLifeline
    Edge (coveredBy) fromReceiverLifeline -> targetEnd
    Edge (covered) targetEnd -> toReceiverLifeline
    Edge (coveredBy) toReceiverLifeline -> targetEnd
  Complementing Changes
    Edge (covered) sourceEnd -> fromSenderLifeline
    Edge (coveredBy) fromSenderLifeline -> sourceEnd
    Edge (covered) sourceEnd -> toSenderLifeline
    Edge (coveredBy) toSenderLifeline -> sourceEnd
Repair [4/16] Change Message Target And Create Association
  {..} Parameters
    Assigned Parameter [1 out of 1]: changeMessage
    Assigned Parameter [1 out of 1]: targetEnd
    Assigned Parameter [1 out of 1]: toReceiverLifeline
    Unassigned Parameter [0 out of 0]: sourceRoleName
      ? Set Input Value
    Unassigned Parameter [0 out of 0]: targetRoleName
      ? Set Input Value
  Recognized Changes
  Complementing Changes
☑ Validation[Failed]: A message can only be send between lifelines
  □ <Message> disconnect
```

Figure 6.2. ReVision: Second corrective edit step *edit step (6.1)* of the running example.

As introduced in Section 5.6, instead of applying the repair with its complementing changes, the developer can also choose to undo the changes that induce the inconsistency (see ④ in Figure 6.1). Instead of applying the second repair alternative in Figure 6.1, the developer could select and undo any of the recognized change sets ⓓ of the proposed repairs. The rollback repairs the VoD-System by moving the operation disconnect() back from the class Server to the class Video.

After applying the repair, ReVision again validates the VoD-System model and recognizes a new inconsistency *message_property(m:Message)* (see ④ Figure 6.2). This inconsistency indicates the missing association between the classes User and Server of the sending and receiving instances of message 6:disconnect. As in our running example, Figure 6.2 illustrates the second corrective *edit step (6.1)* described in Section 2.1.

In this scenario, the developer can choose between two repair alternatives for this inconsistency. These repairs are recognized with the previous corrective edit step as inconsistency-inducing changes, i.e., the changing of the receiving lifeline of message 6:disconnect.

The first repair alternative ① in Figure 6.2 proposes to move the source end of the message 6:disconnect to the already existing lifeline open:Video, which represents an object of the property open [0..4] defined by class Video. This repair leads to the same corrective *edit step (6.1)* as described in Section 2.1. The second repair ② suggests creating the required association between the class User and the class Server. In this repair proposal, the developer must insert ③ the role names of the new association by a parameter argument.

As all repairs are based on the same recognized historical changes, only the number of complementing changes affects the repair ranking. The creation of a new association requires 16 change actions with respect to the model's ASG. The higher-ranked alternative, which moves the source end of the message, requires just 4 change actions.

Applying the second repair proposal changes the sender of the message 6:disconnect from the lifeline online:User to the lifeline open:Video. As in our running example in Figure 2.6, this results in resolving the inconsistencies in the model Version 6.

7

Generation of Consistency-preserving Edit Operations

Our repair tool ReVision uses edit rules as the main configuration input for adapting to a specific modeling language. However, defining a comprehensive catalog of complex edit rules manually can be a time-consuming task. To support developers in configuring the repair tool, this chapter presents an example-based generation approach for CPEOs based on graph transformation concepts. The CPEOs are assembled based on a given set of minimal modeling examples, each describing a valid model fragment in terms of a consistency rule. This approach allows developers to specify CPEOs using their preferred model editor without directly formulating edit rules using the model's abstract syntax.

To adapt our repair tool to a given modeling language, we have to specify a set of CPEOs, preserving the consistency constraints defined by this language. As already mentioned in Section 3.5.1, the goal is to provide a set of CPEOs that avoid typical inconsistencies occurring when models are edited in standard editors of the given modeling language. The most interesting kinds of CPEOs for our approach are creations, deletions, relocations, and transformations of complex model fragments. Such CPEOs typically comprise a set of change actions that must be applied together to preserve a model's consistency.

To that end, we employ a systematic process to synthesize a set of CPEOs. Specifically, we follow an example-driven approach inspired by techniques from other fields, such as model transformation by-example [79], programming by-example [109], or query-by-example [202]. The idea is to manually specify sets of minimal yet valid example model fragments, technically ASG patterns. Then, the graph patterns will be automatically composed into the following kind of CPEOs:

(1) *Pattern-Creating and -Deleting CPEOs:* The basic building blocks of a modeling language are CPEOs for creating and deleting model fragments. Such CPEOs are generated for each given graph pattern.

(2) *Pattern-Relocating CPEOs:* A CPEO which synchronizes the relocation of a graph pattern in a consistency-preserving manner and which is, therefore, a typical example that leads to inconsistencies when applied partially.

(3) *Pattern-Transforming CPEOs:* Such an edit rule transforms a match of one graph pattern into another graph pattern. To limit the number of pattern-transforming CPEOs, only the smallest transformations for each pair of ASG patterns that share at least a minimum number of overlapping nodes are generated.

7.1 Specification of Consistency Patterns

For each consistency rule CR in the overall set of consistency rules defined for the given modeling language, a domain expert specifies a set P_{CR} of ASG patterns. Each pattern $p \in P_{CR}$ represents a model fragment that shows how the rule CR can be fulfilled in a minimal context, and it includes some basic information that is later exploited for the generation of CPEOs. Conceptually, an ASG pattern $p \in P_{CR}$ can be thought of as being created in three steps:

1. We start with a minimal (non-empty) ASG pattern p'' such that the consistency rule CR evaluates to *true* when validated on p'' and none of the graph elements in p'' can be removed without violating CR.

2. Next, the elements in p'' are classified into *static* and *dynamic* parts of the pattern. Dynamic parts indicate model elements that may be *modified* by the generated CPEOs. Static parts indicate model elements that are not modified by a CPEO but which must *exist* in a model to not violate CR. Moreover, the pattern may be extended by further ASG elements that must *not exist* in a model such that CR remains fulfilled. Syntactically, the static nodes, edges, and attributes of the ASG pattern are annotated with *exist* and *not*, respectively. We refer to the pattern resulting from this step as p'. In general, we may define different static and dynamic parts for the same ASG pattern p'', leading to variants of p'.

3. Finally, p' is extended to become the final ASG pattern $p \in P_{CR}$ by adding missing *container* nodes which are needed to create (or delete) the dynamic elements of the pattern. Like static parts, container nodes must exist in a model such that a generated CPEO is applicable, but they are not modified by the CPEO.

An example of an ASG pattern is shown in Figure 7.1. The lower part shows how the consistency rule *message_signature(m:Message)* in Definition (5.1) can be fulfilled in a minimal context. The nodes called message, targetEnd, and operation, as well as their incident edges constitute the dynamic parts of the pattern, while the nodes called lifeline, receiverType and class, as well as the incident edges of type represents and type indicate static parts that must exist in a model to not violate *message_signature(m:Message)*. A variant may be obtained, e.g., by adding the node operation to the static part of the pattern. The node called container shown on top represents a container node that must exist such that the nodes message and targetEnd, which belong to the dynamic part of the pattern, can be created (or deleted) by a generated CPEO.

Edges that are incident to the nodes message, targetEnd, and operation belong to the dynamic part of the ASG pattern shown in Figure 7.1. The example shows that, in general, the dynamic part of an ASG pattern is not a graph but just a graph fragment (i.e., when removing the static or container elements, it would comprise dangling edges). Those nodes in an ASG pattern that complement such a graph fragment to form a graph will be referred to as *boundary* nodes in the sequel. In our example, the nodes called container, lifeline, and class are boundary nodes. More generally, container nodes as well as those static nodes which are adjacent to dynamic ones form the boundary nodes of an ASG pattern.

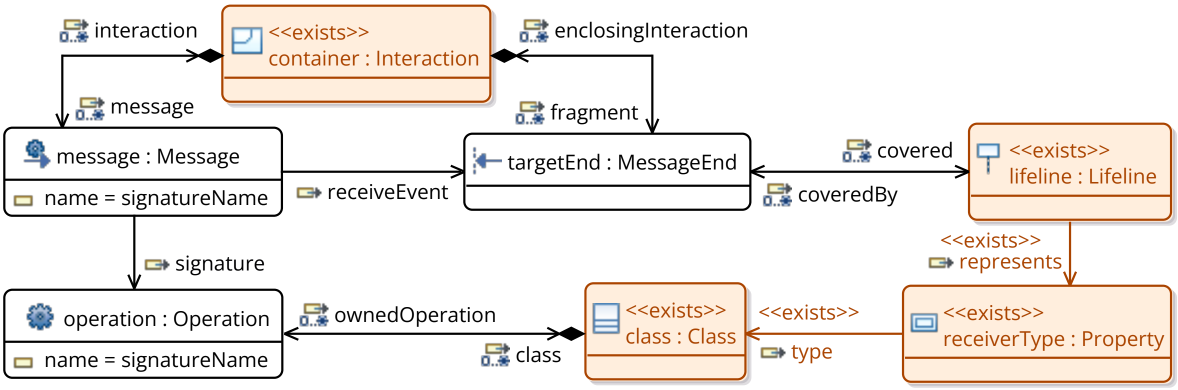


Figure 7.1. ASG pattern representing a model fragment; a message calls an operation on a lifeline. The pattern is an example of how to not violate the consistency rule *message_signature(m:Message)* in a minimal context.

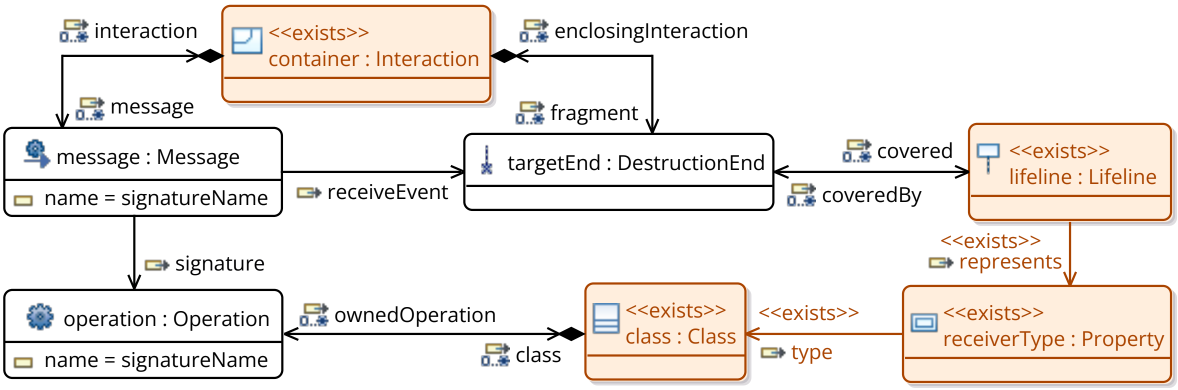


Figure 7.2. ASG pattern representing a model fragment; a destruction message is sent to a lifeline. The pattern is an example of how to not violate the consistency rule *message_signature(m:Message)* in a minimal context.

Another example ASG pattern is shown in Figure 7.2. Compared to the pattern shown in Figure 7.1, the type of the node targetEnd is replaced by the specialized sub-type DestructionEnd, a special message end used in UML sequence diagrams to denote the end of an object's lifeline (see meta-model in Figure 3.3). This is possible since the concrete type of the message end is not relevant for the consistency rule *message_signature(m:Message)*.

ReVision supports the creation of ASG patterns by providing a visual editor whose ASG notation is used in the examples shown in Figure 7.1 and Figure 7.2. A generic model-to-pattern transformation allows the tool configurator to define and validate example model fragments in the native editing environment of the given modeling language. Only refining steps, like the specification of static and dynamic parts, must be performed using the visual ASG editor.

7.2 Generation of CPEOs

The basic idea of our example-driven generation of a CPEO is to take two ASG patterns $p_a, p_b \in P_{CR}$ and a partial mapping $ab : p_a \nrightarrow p_b$ as inputs, which are then processed to a transformation rule $r(x_1, \dots, x_n) : L \rightarrow R$ in a stepwise manner (see Figure 7.3). The selection of p_a, p_b and $ab : p_a \nrightarrow p_b$, performed in an outer process, determines the kind of a generated CPEO (pattern-creating and -deleting, pattern-relocating, or pattern-transforming), which will be explained later in this section. Before that, we first describe the stepwise construction of a single CPEO, illustrated in Figure 7.3, which is the same for all kinds of CPEOs:

1. *Boundary Extension:* Unmapped dynamic edges in p_a and p_b that are incident to a boundary node shall be created and deleted by a generated CPEO, respectively. This is only possible if the respective boundary node is to be preserved by the CPEO. Thus, if such a boundary node is not included in the mapping $ab : p_a \nrightarrow p_b$, we perform a so-called boundary extension. An unmapped boundary node in p_a is created in p_b (and vice versa), and this pair of boundary nodes is added to the mapping ab. We refer to the extended mapping resulting from this step as $ab' : p'_a \nrightarrow p'_b$. The dynamic and boundary parts of p'_a and p'_b form the left- and right-hand side L and R, respectively, of the generated CPEO.

2. *Integrate Conditions:* After the boundary extension, non-boundary static elements in p'_a and p'_b may remain unmapped. These unmapped static graph elements are not part of the generated CPEO's left- and right-hand side. Instead, unmapped static graph elements in p'_a and p'_b can be considered as structural pre- and postconditions of the generated CPEO. In this step, such unmapped static graph elements are transformed to PAC and NAC graph constraints (see Section 3.5.2), i.e., graph fragments in the unified rule graph of a CPEO which are either required or forbidden by a successful rule application. Non-boundary graph elements in an ASG pattern which are annotated as static elements that must exist are transformed to PAC graph constraints while elements that are annotated as static elements that must not exist are transformed to NAC graph constraints.

3. *Derive Parameters:* Finally, we need to derive the parameters for the CPEO. Therefore, we create an input parameter for each context node of the rule, i.e., for each preserved node containing attribute modifications or having incident edges that are to be created or deleted. In addition, an input parameter is created for all nodes that are to be deleted. Furthermore, all attributes are scanned for assigned variables that have to be mapped to an input parameter. Some parameter assignments of a CPEO may be derived by the assignment of another parameter, which is automatically handled by our repair tool as shown in Chapter 6.

In general, some consistency rules may share identical ASG patterns, leading to equivalent CPEOs generated from these patterns. Since we do not store the relation between

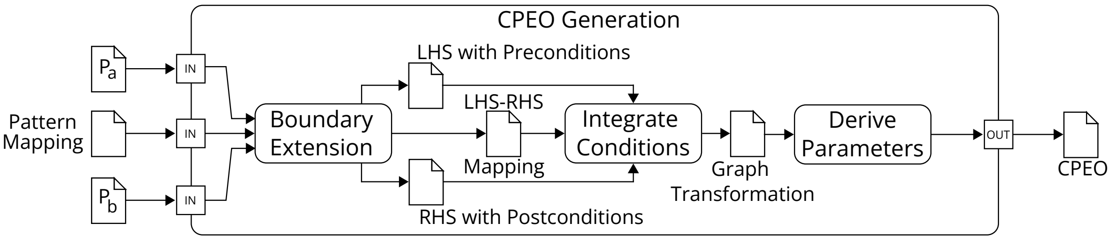


Figure 7.3. Overview of the stepwise derivation of a CPEO from a pair of example ASG patterns and a partial mapping between them.

CPEOs and consistency rules, such duplicates can be discarded. We check for duplicates before adding a generated CPEO to the overall tool configuration.

7.2.1 Generation of Pattern-Creating and -Deleting CPEOs

Given a set P_{CR} of ASG patterns, for each $p \in P_{CR}$, we generate a CPEO for creating (deleting) the model fragment defined by the dynamic part of p. The CPEO generation illustrated in Figure 7.3 starts by using p, the empty pattern p_ϵ and an empty mapping as input. Pattern-creating CPEOs are generated by choosing $p_a = p_\epsilon$ and $p_b = p$, while pattern-deleting CPEOs are generated by $p_a = p$ and $p_b = p_\epsilon$.

Boundary nodes of the pattern p lead to boundary extensions in p_ϵ such that these nodes are preserved by the generated CPEO. Non-boundary yet static elements in p lead to application conditions in the generated CPEO. All other graph elements constitute the dynamic part of the ASG pattern p. They are not included in the mapping ab and will be created (or deleted) by the generated CPEO.

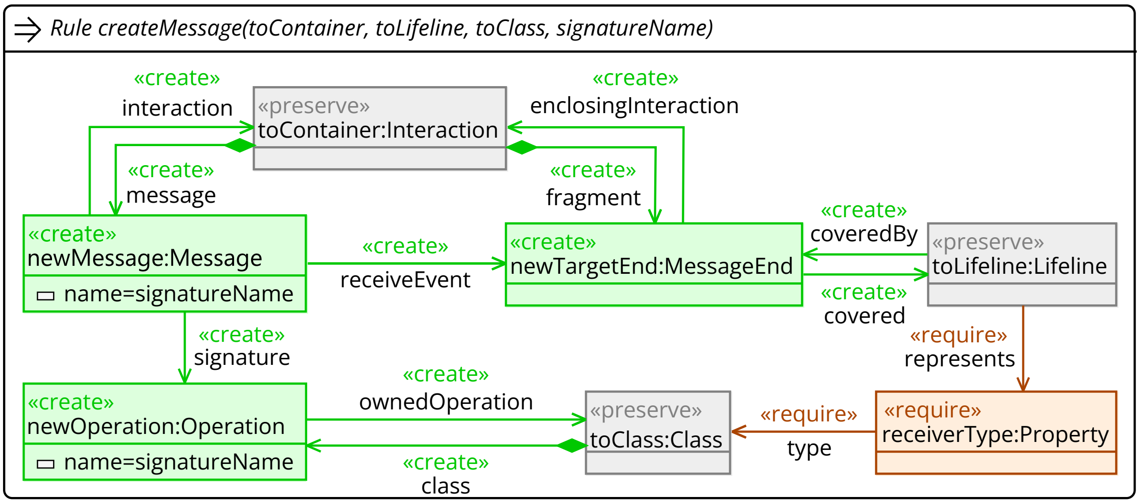


Figure 7.4. CPEO creating a message including its target end and operation signature.

Figure 7.4 shows the pattern-creating CPEO derived from the ASG pattern shown in Figure 7.1 and the empty pattern. The latter is extended by the boundary nodes container, lifeline, and class, which are to be preserved by the generated CPEO. The node receiverType, including its incident edges, is interpreted as a positive application condition. The remaining dynamic nodes message, targetEnd, and operation, as well as their incident edges, are to be created.

The integrated rule graph of the pattern-deleting CPEO derived from the same ASG pattern is isomorphic to the one shown in Figure 7.4 up to action annotations (i.e., deletions instead of creations).

7.2.2 Generation of Pattern-Relocating CPEOs

Given a set P_{CR} of ASG patterns, for each $p \in P_{CR}$, pattern-relocating CPEOs are generated by using $p_a = p_b = p$ and different variants of a partial mapping $ab : p \nrightarrow p$ as input for the CPEO generation.

A single such variant is constructed as follows. Starting from the mapping $ab_{full} : p \nrightarrow p$ that forms a graph isomorphism on the subgraph of p, which is induced by the union of dynamic and boundary nodes, we construct a reduced mapping $ab_{red} \subseteq ab_{full}$ by excluding a pair of boundary nodes. The generated CPEO then relocates the adjacent dynamic nodes by deleting and re-creating all edges being incident to the two unmapped boundary nodes excluded from ab_{full}. By a “relocation”, we mean that a dynamic node is either moved to another container (by the deletion and re-creation of a containment edge), or re-connected to a different neighbor (by the deletion and re-creation of one or several non-containment edges). The idea behind this construction is to obtain a CPEO that synchronizes the relocation of dynamic nodes of an ASG pattern in a consistency-preserving manner and which is therefore a typical example that leads to inconsistencies when applied partially.

All variants of reduced mappings are created by excluding all pairwise combinations of distinct boundary nodes of the ASG pattern p. That is, let n be the number of boundary nodes in p, then we obtain a total of $\frac{n(n-1)}{2}$ reduced mappings, each of which serves as input for the generation of a CPEO. If not all combinations are useful for a specific modeling environment, a tool configurator may refine this default behavior by explicitly defining the types of edges which may be deleted and re-created through relocations.

The example ASG pattern shown in Figure 7.1 comprises three boundary nodes, namely the nodes (1) container, (2) lifeline, and (3) class. When being excluded from ab_{full}, this leads to relocations of (1) the message and its target end to a different interaction, (2) the target end of the message to a different lifeline, and (3) the operation to a different class. In sum, we obtain three variants of a reduced mapping ab_{red}.

In fact, the combination of (2) and (3) generates a CPEO which is equivalent to *moveOperationAndChangeMessageTarget* of our running example shown in Figure 5.5. The reduced pattern mapping comprises all dynamic nodes message, targetEnd and operation, i.e., the fragment to be (partially) relocated, and the boundary node container. The boundary nodes lifeline and class are not mapped. They serve as context for the relocation and the application condition, which results from the static node receiverType and its incident edges. For the sake of simplicity, we omitted the container node of type Interaction in Figure 5.5 since it is not needed for the relocating change actions of this CPEO.

7.2.3 Generation of Pattern-Transforming CPEOs

For a given set $P_{CR} = \{p_1, ..., p_n\}$ of ASG patterns belonging to a consistency rule CR, in principle, several CPEOs could be derived for each pair (p_i, p_j) with $i \neq j$, transforming p_i into p_j. Each CPEO transforms p_i into p_j in a consistency-preserving way, they differ from each other in the number of element creations and deletions.

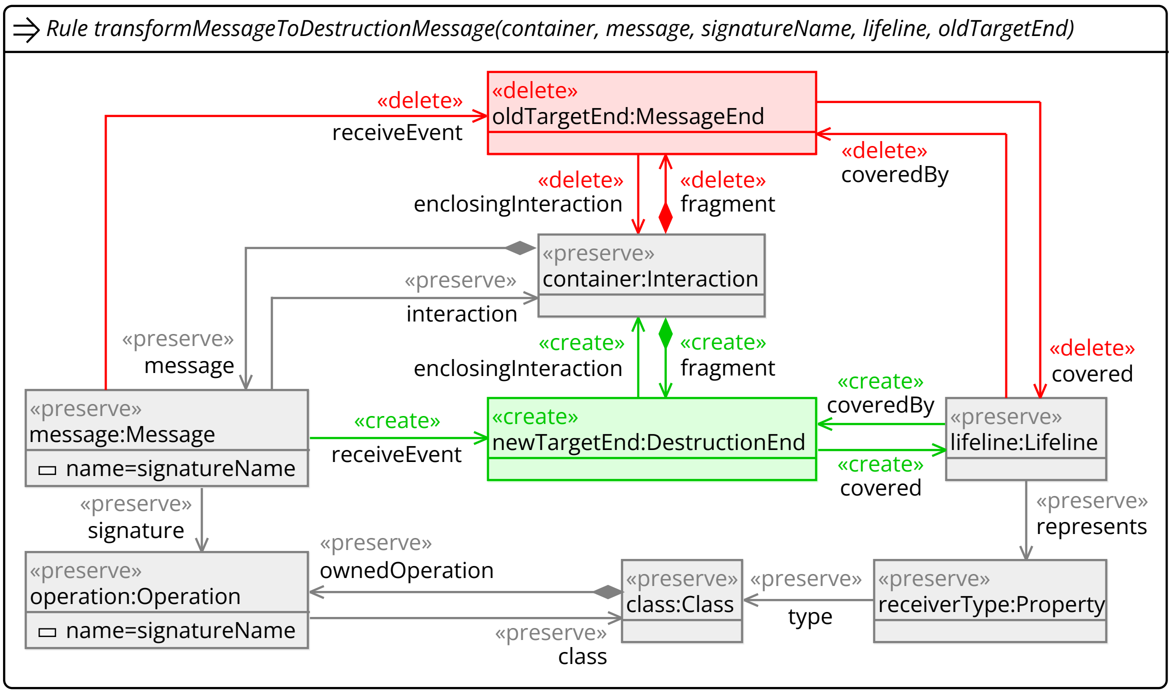

Rule transformMessageToDestructionMessage(container, message, signatureName, lifeline, oldTargetEnd)
«delete» oldTargetEnd:MessageEnd
«delete» receiveEvent
«delete» coveredBy
«delete» enclosingInteraction
«delete» fragment
«preserve» container:Interaction
«preserve» interaction
«preserve» message
«create» enclosingInteraction
«create» fragment
«delete» covered
«create» coveredBy
«preserve» message:Message
name=signatureName
«create» receiveEvent
«create» newTargetEnd:DestructionEnd
«preserve» lifeline:Lifeline
«create» covered
«preserve» signature
«preserve» represents
«preserve» operation:Operation
name=signatureName
«preserve» ownedOperation
«preserve» class:Class
«preserve» type
«preserve» receiverType:Property
«preserve» class

Figure 7.5. CPEO transforming a regular message into an object's lifeline destruction message.

In order to limit the number of pattern-transforming CPEOs, we generate the smallest transformations for each pair (p_i, p_j) in terms of the number of change actions of the resulting CPEO. To that end, the partial mapping $ab : p_i \nrightarrow p_j$ is derived from the maximum common subgraph of p_i and p_j [165]. Maximality is defined in the number of nodes, and all graph elements in the maximum common subgraph must have the same type (drawn from the meta-model) and annotation (dynamic or static). Unmapped graph elements will be created or deleted by the generated CPEO.

A matching of the pattern in Figure 7.1 will map all nodes to the pattern in Figure 7.2 except from the nodes of type MessageEnd and DestructionEnd. A node in the ASG cannot change its type, so the target end has to be replaced. The resulting CPEO is shown in Figure 7.5, which takes care of correctly replacing the node and connecting the message with the former lifeline. Optionally, the preserved elements in Figure 7.1 that do not involve any changes can be transformed to PAC graph constraints using the refactoring operation *Extract Pre-condition* as defined by Taentzer et al. [179].

8
Evaluation

This chapter discusses the evaluation of our approach to model repair. The approach is evaluated using a selected set of real-world models obtained from popular open-source Eclipse modeling projects. In particular, the experimental results confirm that most of the inconsistencies can be resolved by complementing incomplete edits. The number of repair operations generated by the approach is typically low, and the relevant proposal is ranked at the topmost position of the short list of repairs in virtually all cases. All evaluation results presented in this chapter have been published and peer-reviewed in Reference [6].

The broad discussion of the evaluation of software repair tools in Reference [132] shows that different repair scenarios lead to different, if not contradictory, evaluation criteria and evaluation goals. In the scenario of fully automated repair, the correctness and completeness of an approach are primary concerns. In contrast to this, interactive repair recommendation systems propose tentative patches to developers; thus the *understandability* of the proposed short list of patches is of primary importance [132]. Our approach clearly falls into the latter category of interactive repair.

One main feature of our approach is to describe the effect of repair proposals in terms of user-level edit operations. In this way, a developer gets an overview of the proposed repairs very quickly. A second unique feature of our approach is to give a precise description of the origin of an inconsistency. Both features are, first and foremost, qualitative advantages over existing approaches. To evaluate the helpfulness of our approach quantitatively, our evaluation is driven by the following research questions:

- **RQ1 (Coverage):** *How many inconsistencies can be resolved by our approach?* This aims at assessing the limitations of our approach of resolving inconsistencies by completing partial edit steps.

- **RQ2 (Relevance):** *If an inconsistency can be resolved by our approach, do we generate a repair alternative whose effect can be observed in the history of a model?* This aims at assessing whether proposed repair operations are capable of meeting a developer's intention of how to resolve an inconsistency.

- **RQ3 (Efficiency):** *How many repair alternatives must be inspected by developers until they find a relevant one?* This aims at assessing how quickly developers may pick and apply a relevant repair from an ordered set of generated repair alternatives.
- **RQ4 (Performance):** *How long does it take to generate the repair alternatives for a given inconsistency?* This aims at assessing the usefulness of our approach as a development support tool which should generate repairs in acceptable latency times in the order of seconds.

Following general guidelines of how to evaluate quality aspects of recommendation systems [169], we examine research questions RQ1 to RQ4 in an offline experiment using experimental datasets obtained from real-world modeling projects. Section 8.1 summarizes the subject modeling projects and justifies their selection. Section 8.2 outlines how we configured our repair tool ReVision for the selected subject models. We introduce our research methodology in Section 8.3 and present the experimental results. Threats to validity are discussed in Section 8.4.

8.1 Subject Selection

The selection of suitable subject projects for our empirical study is driven by the following requirements.

Model representation: From a technical point of view, we need to work with EMF models since ReVision is implemented on top of the Eclipse modeling technology stack.

Modeling language: A modeling language is only suitable for our evaluation if it has consistency constraints which can be specified in an OCL-like manner.

Active consistency management: We need to find real-world models for which violations of the given consistency rules, as well as resolutions of these inconsistencies can be observed in the historical evolution of a model. This requirement aims at having an oracle at hand which tells us whether the effect of a generated repair proposal can be observed in the history of a model.

We have chosen Ecore as modeling language due to its widespread usage within the Eclipse modeling project as a de-facto standard for data- and meta-modeling. Although no OCL specifications of the Ecore consistency constraints are readily available, their formalization in an OCL-like manner is straightforward. We did so for 20 Ecore consistency constraints comprised in Table 8.2 ($Supported = yes$), which we consider to be relevant to our approach. We focused on structural constraints and ignored simple checks such as well-formedness rules of certain string expressions of attribute values ($Supported = no$).

To obtain historical evolutions of real-world models, we have determined a set of Ecore models developed in popular open-source Eclipse modeling projects being hosted in the Eclipse Git repository [42]. As illustrated in Figure 8.1, we have explored the active development branches of 51 Ecore modeling projects, shown in column *Project Name* in Table 8.1. For each of those 51 projects, we extracted the version histories of all Ecore models, summarizing to 148 model histories (cf. column *All Models* in Table 8.1) spanning a time period

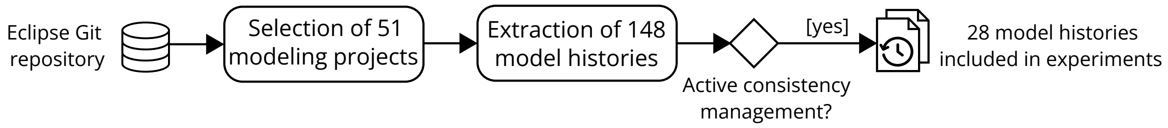

Eclipse Git repository
Selection of 51 modeling projects
Extraction of 148 model histories
Active consistency management?
[yes]
28 model histories included in experiments

Figure 8.1. Overview of the subject selection process.

of several years (2004 - 2018) for the most long-living modeling projects. Next, we filtered those 148 model histories as follows. First, we disregarded those model histories for which a validation check passes on every version of the history, i.e., the history does not expose any inconsistencies at all. Second, we also disregard those model histories which only contain inconsistencies that were never solved during the lifespan of a model history. The remaining 28 model histories (cf. column *Inc. Models* in Table 8.1) do expose inconsistencies in some historical versions, while the model history as a whole can be safely assumed to be subject of an active consistency management since all inconsistencies have been resolved eventually.

Table 8.1 further characterizes the size of the extracted models. The average numbers of model elements, calculated over all considered model histories of a project, are shown in column *Avg. Elements*. To characterize the size of the extracted model histories, column *Source Revisions* shows the number of revisions of the examined model histories per modeling project. Since an Ecore model may have cross-references to other Ecore models, we also considered these interrelated models. Interrelated models have their own development history and may co-evolve in parallel. Once an interrelated model is changed in the time interval between two revisions of the considered model history, a new intermediate version, counted in column *Co-ev. Revisions*, with the updated co-evolving model version is created. As a result, we extracted a total number of 1396 versions (incl. co-evolving versions) for the 28 considered model histories, yielding 1368 pairs of successive model versions referred to as evolution steps in the sequel. The dataset of all model histories is available on the project website of ReVision [8].

8.2 Experimental Tool Configuration

Set of CPEOs for the Ecore modeling language. In order to configure our tool, we constructed CPEOs for the 20 consistency rules shown in Table 8.1 using the approach described in Chapter 7. We manually specified a total of 89 ASG patterns, i.e., an average number of 4.5 ASG patterns per consistency rule. All Ecore ASG patterns, as well as the generated CPEOs can be found on the project website of ReVision [8], classified by the consistency rules they have been derived from. In general, some consistency rules may lead to equivalent ASG patterns. After a duplicate check, our configuration comprises 74 unique ASG patterns serving as input for the generation of 311 CPEOs. Moreover, the application conditions of the Ecore CPEOs are interpreted as invariants with respect to detection of partial CPEO executions in Section 4.5.

For each unique ASG pattern, we obtain a corresponding pattern-creating and -deleting CPEO, summarizing to a total of 148 CPEOs. As described in Section 7.2.2, the generation of pattern-relocating CPEOs allows us to define the relevant relocations that should be considered. We selected relocations that represent typical drag and drop operations of the Ecore diagram editor, including relocations of packages and classifiers (i.e., classes, interfaces, data types and enumerations), structural features (i.e., attributes, references and enumeration literals), and inheritance relationships. This configuration leads to a total number of 23 pattern-relocating CPEOs. Finally, our set of CPEOs comprises a total number of 140 pattern-transforming CPEOs.

Comparison of Ecore models. Furthermore, ReVision must be configured with a model matcher, which is used for the comparison of Ecore models. The main elements in an Ecore model have qualified names that are also used to store references between model elements on a technical level. To determine corresponding model elements in successive versions of Ecore models, we employ the signature-based matching algorithm of the SiDiff model comparison framework [80], using qualified names of model elements as signatures.

8.3 Methodology and Experimental Results

Our methodology to answer research questions RQ1 through RQ4 is outlined in Figure 8.2. The results of our empirical study are summarized in Table 8.1 for RQ1 and RQ2, and visualized in Figure 8.3 and Figure 8.4 for RQ3 and RQ4, respectively. Details will be presented in the remainder of this section.

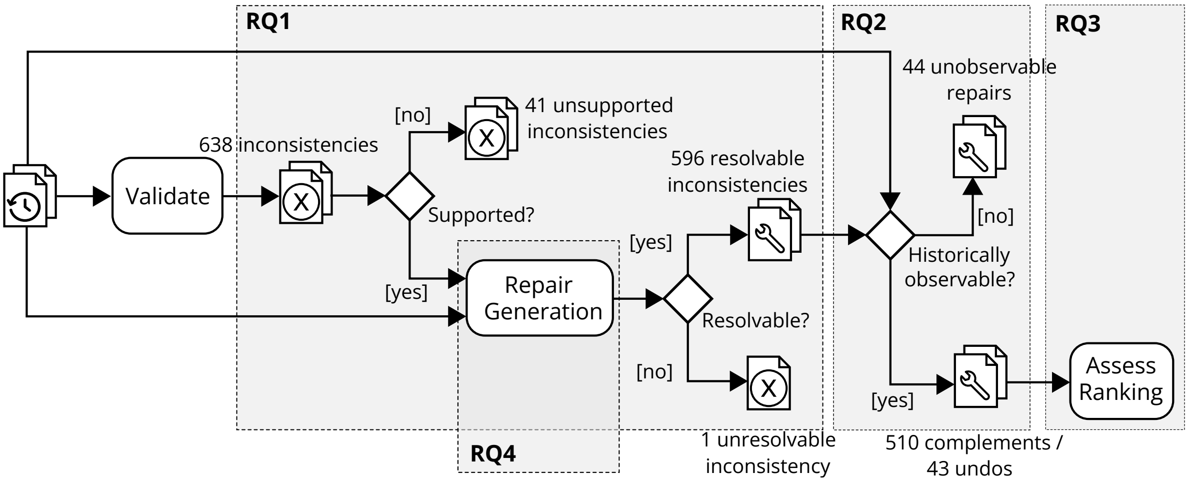


Figure 8.2. Overview of our methodology to answer research questions RQ1 through RQ4.

8.3.1 RQ.1 (Coverage)

To address RQ1, we calculate the number of inconsistencies that are resolvable by generated model repairs of our approach. An inconsistency $\gamma_i = (CR, e_i)$ introduced in model version v_i is resolvable if there is a generated repair that can be applied to v_i, yielding a changed model v_i' in which γ_i does not occur. The latter is the case if (i) there is no inconsistency $\gamma_i' = (CR, e_i')$ where $e_i' \in v_i'$ is the corresponding element of e_i, or (ii) e_i' is removed in v_i', i.e., no corresponding element of e_i can be found in v_i'.

Subject						**RQ1**				**RQ2**		
Project	Models		Elements	Revisions		Inconsistencies				Observable		
Name	All	Inc.	Avg.	Source	Co-ev.	Total	RegEx	Supp.	Resolvable	Complement	Undo	Not Obs.
(acceleo)	5	1	120	24	15	14	14	0	0	0	0	0
atl	5	2	623	32	3	32	14	18	18	11	7	0
birt	7	1	381	17	97	1	0	1	1	0	1	0
bpmn2	4	1	3297	23	11	1	0	1	1	1	0	0
buckminster	4	1	340	3	7	10	2	8	8	0	8	0
cbi	7	3	597	188	15	7	0	7	7	6	0	1
e4	1	1	286	115	7	1	0	1	1	1	0	0
eavp	1	1	188	3	1	4	0	4	4	4	0	0
edapt	2	1	294	7	8	1	0	1	1	0	1	0
emf	13	3	237	82	25	5	1	4	4	4	0	0
gmf-tooling	7	3	295	110	23	32	0	32	32	28	2	2
ocl	12	2	711	20	11	2	1	1	1	0	1	0
papyrus	14	1	89	7	2	10	9	1	1	0	1	0
qvt-oml	4	1	561	35	24	1	0	1	0	0	0	1
qvtd	29	1	82	42	214	2	0	2	2	0	1	1
sirius	7	1	3835	62	1	4	0	4	4	4	0	0
stem	17	1	289	17	22	1	0	1	1	0	1	0
uml2	9	3	3775	100	23	510	0	510	510	451	20	39
33 Others	133	0	n.a.	n.a.	n.a.	0	0	0	n.a.	n.a.	n.a.	n.a.
Summary	148	28	889	887	509	638	41	597	596	510	43	44

Table 8.1. The evaluated projects and results of the resolvable and observable inconsistencies for RQ1 and RQ2.

The considered model histories contain a total number of 638 historically resolved inconsistencies, shown in column *Total Inconsistencies* of Table 8.1, which serve as the basis for the evaluation of our generated repairs. The results in column *Resolvable Inconsistencies* show that our tool ReVision can propose at least one repair for 596 of all 638 inconsistencies (see column *Total Inconsistencies*) in our evaluation dataset.

A first explanation for this limitation are certain kinds of constraints that are not supported by our approach. Specifically, our repair approach considers inconsistent attribute values as atomic elements and does not support grammatical analyses of attribute values and related repairs. They are out of the scope of our approach, which focuses on structural inconsistencies. The upper part of Table 8.2 shows all 20 supported constraints alongside with the number of their violations in the different projects. The lower part of Table 8.2 shows the unsupported “is not well-formed” constraints. In our experiment, 41 inconsistent String values were found by 5 different regular expression constraints (see also column *RegEx Inconsistencies* in Table 8.1). We also found 3 inconsistencies that belong to 3 different domain-specific regular expression constraints that extend the default set of Ecore constraints for a specific type of Ecore models (in project *papyrus*). Since we just looked at the default Ecore constraints during the configuration phase, we obviously could not repair such inconsistencies.

Looking at the violations of supported constraints only, as summarized in column *Supp. Inconsistencies* of Table 8.1, ReVision proposes at least one repair for 596 out of 597 inconsistencies. There was only one inconsistency (in project *qvt-oml*) for which no repair was

Consistency constraint descriptions (with placeholders {i} for specific inconsistencies)	**Violations**	**Projects**	**Supported**
The generic type associated with the {0} classifier should have {1} type argument(s) to match the number of type parameter(s) of the classifier	469	gmf-tooling, uml2, eavp, cbi, emf, buckminster	yes
There may not be an operation {0} with the same signature as an accessor method for feature {1}	50	uml2	yes
A class that is an interface must also be abstract	29	gmf-tooling	yes
The default value literal {0} must be a valid literal of the attribute's type	14	gmf-tooling, sirius, edapt, atl, bpmn2, cbi, e4	yes
A containment reference of a type with a container feature {0} that requires instances to be contained elsewhere cannot be populated	10	uml2, atl, emf	yes
There may not be two features named {0}	7	qvtd, papyrus, birt, atl	yes
A container reference must have an upper bound of 1 not {0}	5	atl	yes
The attribute {0} is not transient, so it must have a data type that is serializable	3	cbi, buckminster	yes
The opposite of a transient reference must be transient if it is proxy resolving	2	uml2, stem	yes
The required feature {0} of {1} must be set	2	buckminster	yes
The typed element must have a type	2	buckminster	yes
The generic type associated with the {0} classifier must not have {1} argument(s) when the classifier has {2} type parameter(s)	1	qvt-oml	yes
The opposite of the opposite may not be a reference different from this one	1	gmf-tooling	yes
There may not be two classifiers named {0}	1	buckminster	yes
There may not be two operations {0} and {1} with the same signature	1	ocl	yes
A containment or bidirectional reference must be unique if its upper bound is different from 1	0		yes
The features {0} and {1} cannot both be IDs	0		yes
The opposite of a containment reference must not be a containment reference	0		yes
The opposite may not be its own opposite	0		yes
There may not be two parameters named {0}	0		yes
The source URI {0} is not well-formed	16	papyrus, acceleo	no
The namespace URI {0} is not well-formed	6	atl	no
The name {0} is not well-formed	7	ocl, acceleo, buckminster	no
The namespace prefix {0} is not well-formed	7	atl, emf	no
The instance type name {0} is not well-formed	2	atl	no
additional domain-specific constraints (3)	3	papyrus	no

Table 8.2. Overview of the supported and not supported inconsistencies by consistency constraints.

found, although the violated constraint is supported in principle. This inconsistency occurred as a side effect of a technical defect, namely a dangling reference to a non-existing model element. Since we ignore the defect parts of a model during inconsistency analysis, the inconsistency-inducing edit step could not be recognized. As a consequence, no repair proposal has been generated in the subsequent repair generation step.

ReVision covered 93.4% (596 of 638) of the inconsistencies of our evaluation dataset. Virtually all inconsistencies for which no repair has been generated, namely 97.6% (40 of 41), are not supported by our approach since they are caused by incorrectly formed String values. Only one inconsistency which is supported in principle has been missed. The reason for this was a technical defect in the underlying model.

8.3.2 RQ.2 (Relevance)

We use the history of a model as an oracle to answer the question whether the effect of a generated repair proposal can be observed in the changes made by developers. Given an inconsistency γ_x introduced in version v_x historically and resolved in a later version v_y, we check if there is a generated repair alternative that, when applied to v_x, leads to an effect that is observable in the evolution step from v_{y-1} to v_y. We call a repair alternative that fulfills this criterion a *historically observable resolution.* For each generated repair for an inconsistency γ_x, the oracle compares the ASG modifications of the model difference between version v_{y-1} and v_y with the change actions of the repair operation. For each change action, there must be a concrete historical change in $\Delta(v_{y-1}, v_y)$ of the same kind applied to the same model element.

ReVision has found the historically observable resolution for 553 of 597 supported inconsistencies. We can further differentiate between those resolutions that are complements of a recognized partial execution of a CPEO and those that undo recognized edit steps. In sum, we found 510 historically observable complements and 43 historically observable undo repair operations, referred to as *Observable Complement* and *Observable Undo* in Table 8.1. Only 44 inconsistencies, referred to as *Not Observable* in Table 8.1, could not be resolved by our approach in the same way as observed in the model history.

We manually investigated the inconsistencies that could not be repaired by a historically observable resolution and found two reasons for this: (1) The CPEO that would have been needed to recognize and complement the inconsistency-inducing edit step was missing in the tool configuration. Our tool configuration lacked 8 CPEOs that were needed for 38 inconsistencies in our evaluation dataset. (2) Since we always try to recognize the maximal possible sub-rule of a CPEO (see Section 4.3.3), some recognized partial CPEO executions covering inconsistency-inducing changes were actually too large. For example, if a CPEO adds a model element to a list of elements, a generated repair may propose to modify an existing element of this list instead of creating a new one. This can happen if the evolution step in which the inconsistency is introduced contains a creation of such an element in that location. We found such a case where an EAnnotation is created that stores additional information in the Ecore model. An EAnnotation is just a pair of a key and a value and ReVision currently cannot guess whether to overwrite a recently created annotation or to create a new one. In sum, we found 5 occurrences of this kind of ambiguity which eventually led to repair proposals that could not be observed historically. As already discussed for RQ1, one inconsistency that is supported in principle could not be resolved at all due to a defect in the model.

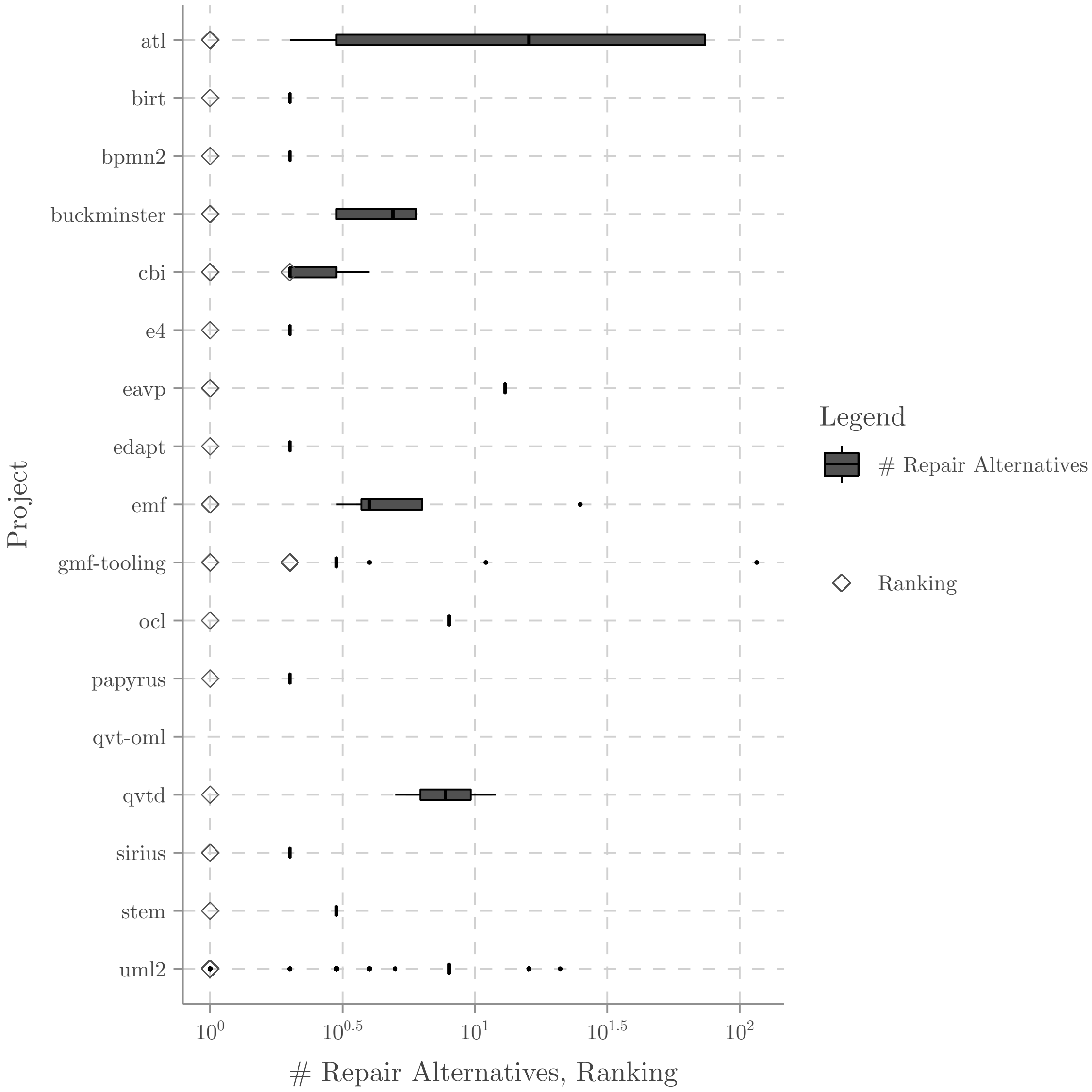


Figure 8.3. Ranges of the number of repair alternatives and the ranking of the historically observable repairs.

For 92.6% of the supported inconsistencies of our evaluation dataset, ReVision generated a repair proposal which is relevant in the sense that it could be observed in the model's editing history and thus, arguably, is capable of meeting a developer's intention. 92.2% of the historically observable repairs are completions of partial executions of CPEOs, which confirms our hypothesis that a majority of inconsistencies are the result of incomplete edits.

8.3.3 RQ.3 (Efficiency)

To address RQ3, we record the number of repair alternatives that are generated for each inconsistency and the ranking of the historically observable repairs. The latter is inspired by the so-called *EXAM measure*, which is a generally acknowledged quality measure for evaluating statistical fault localization techniques [144, 196–198] and which is defined as the number of program statements that have to be examined until the first statement containing the bug to be localized is reached. To find out how many repair alternatives must be inspected by developers until they find a relevant one, we consider those inconsistencies from which we know that our approach generates a repair operation being historically observable in a later evolution step (see RQ2). Given such an inconsistency, let $[\rho_1, \dots \rho_k]$ be the sequence of repair alternatives generated and ranked by our approach. Let ρ_i with $(1 \leq i \leq k)$ be the repair operation whose effect is historically observable in a later version of the model history. A smaller i conforms to a better prioritization since we assume that developers will consider those repair alternatives having a higher priority first.

In Figure 8.3, we show the ranges of the number of repair alternatives and the prioritization of the historically observable repairs for each modeling project. The historically observable repair is in 525 of the supported inconsistencies on the top position ($i = 1$) of the ranking and in 28 cases on the second position ($i = 2$), even in those cases with a high number of repair alternatives. Moreover, most of the time, our approach was able to limit the number of generated repair alternatives to a manageable quantity. Actually, for 92.5% of the cases, the number of repair alternatives is 10 or fewer, which is a reasonable number of options to be inspected by a developer.

In addition to selecting a suitable repair, a developer is also responsible for providing all arguments of a repair operation if not all its parameters can be bound automatically. The use of graph transformation rules makes it possible to automatically determine most of the concrete parameter values. Typically, the possible matches of a complement rule are restricted by the pre-match induced by the sub-rule which covers the inconsistency-inducing changes. However, the size of the parameter domain of a complement rule can vary significantly. This can be observed, for example, for some repair proposals for the UML meta-model history, where a parameter can be initialized with any classifier of the meta-model, i.e., the parameter domain is only limited by its type but not by the structure of the CPEO. Moreover, value parameters have to be determined by the developer in any event. On average, in addition to the selection of a suitable repair, a developer had to manually select 0.07 unbound parameters for all observable repairs of our evaluation dataset.

Repair proposals generated by ReVision are highly efficient in the sense that they do not overwhelm developers with huge numbers of irrelevant repair alternatives. For 92.5% of the supported inconsistencies, the number of repair alternatives is 10 or fewer, and the historically observable repair was ranked on the top position in 94.9% of the cases. This strongly correlates with the finding that 92.2% of the historically observable repairs are complements of partial executions of CPEOs. This confirms our hypothesis that it is a highly reasonable choice to resolve inconsistencies by complementing partial executions of CPEOs that induce an inconsistency.

8.3.4 RQ.4 (Performance)

To assess the performance and scalability of ReVision, we measured the runtime for the 3 major phases of the calculation process, namely: 1) *Analysis*: The history-based analysis comprising our origin and impact analysis (see Section 5.2 and Section 5.3), 2) *Recognition*: The recognition of partially executed CPEOs (see Chapter 4), and 3) *Complement*: The generation of repair proposals, which is dominated by the construction of complement rule applications (see Section 5.4.3). For each of the three phases, the ranges of runtimes per project are shown in Figure 8.4.

The runtime of the *analysis* phase tends to increase mainly with two parameters, namely (i) the size of the model and (ii) the number of historical revisions that, starting from the inconsistent model, have to be analyzed until we find the evolution step in which the inconsistency has been introduced. For example, the analysis runtimes for the UML meta-model history are clustered around three different values. The first time interval is around 300ms and has been measured for some smaller models with about 3393 elements. The second and third time intervals refer to later and, thus, larger versions of the UML meta-model with about 11594 elements. The runtimes around 2s have been measured for inconsistencies introduced in the predecessor version of an inconsistent model, i.e., only a very small subset of the history needs to be analyzed. In contrast, the times around 7s refer to situations that span up to 20 versions of the UML meta-model history that needed to be analyzed in order to find an inconsistency-inducing change. Note that, in order to better quantify the second parameter, we do not cache any results of an inconsistency analysis between different runs.

The runtime of the *recognition* phase mainly depends on the number of historical changes in the difference between the latest consistent model version and the inconsistent version that shall be repaired. Moreover, a minor negative influence on the runtime can also be observed for the number of change actions of a CPEO. The more change actions a CPEO has, the more changes are to be considered per CPEO. Both phenomena can be explained by the algorithm that solves the underlying partial CSP problem (see Chapter 4). In this encoding, the change actions of the CPEO are treated as variables of the CSP problem, and their domains are defined by the changes comprised by the model difference between the consistent and inconsistent model versions. Finally, another parameter affecting the overall runtime of this phase is the complexity of the consistency rule that is violated. The more complex the consistency rule, the more CPEOs might have an impact on the inconsistency and therefore need to be considered. Nevertheless, in 96.1% of the supported inconsistencies, the runtime of the partial CPEO recognition phase is less than 1s.

The runtime of the *complement* phase mainly depends on the number of possible matches that can be found for the left-hand sides of CPEOs. We combine all matches to build the elements and values for the parameter domains of a complement rule. Figure 8.4 shows that for all evaluated projects the runtime of this phase is less than 1s.

Compared to the above phases, the time needed for the inconsistency detection (step 1 in the overview pictures presented in Figure 5.7 and Figure 5.8) can be neglected. In our experiment, which uses the EMF built-in validation facilities for detecting inconsistencies, the runtimes for the detection were in the order of milliseconds. These observations are confirmed by larger and more systematic experiments on the runtime performance of OCL validation. For example, Reder et al. [150] report on an experiment where the validation of 20 consistency rules takes about a millisecond on UML models comprising 100k elements, much less when being optimized to run in an incremental fashion.

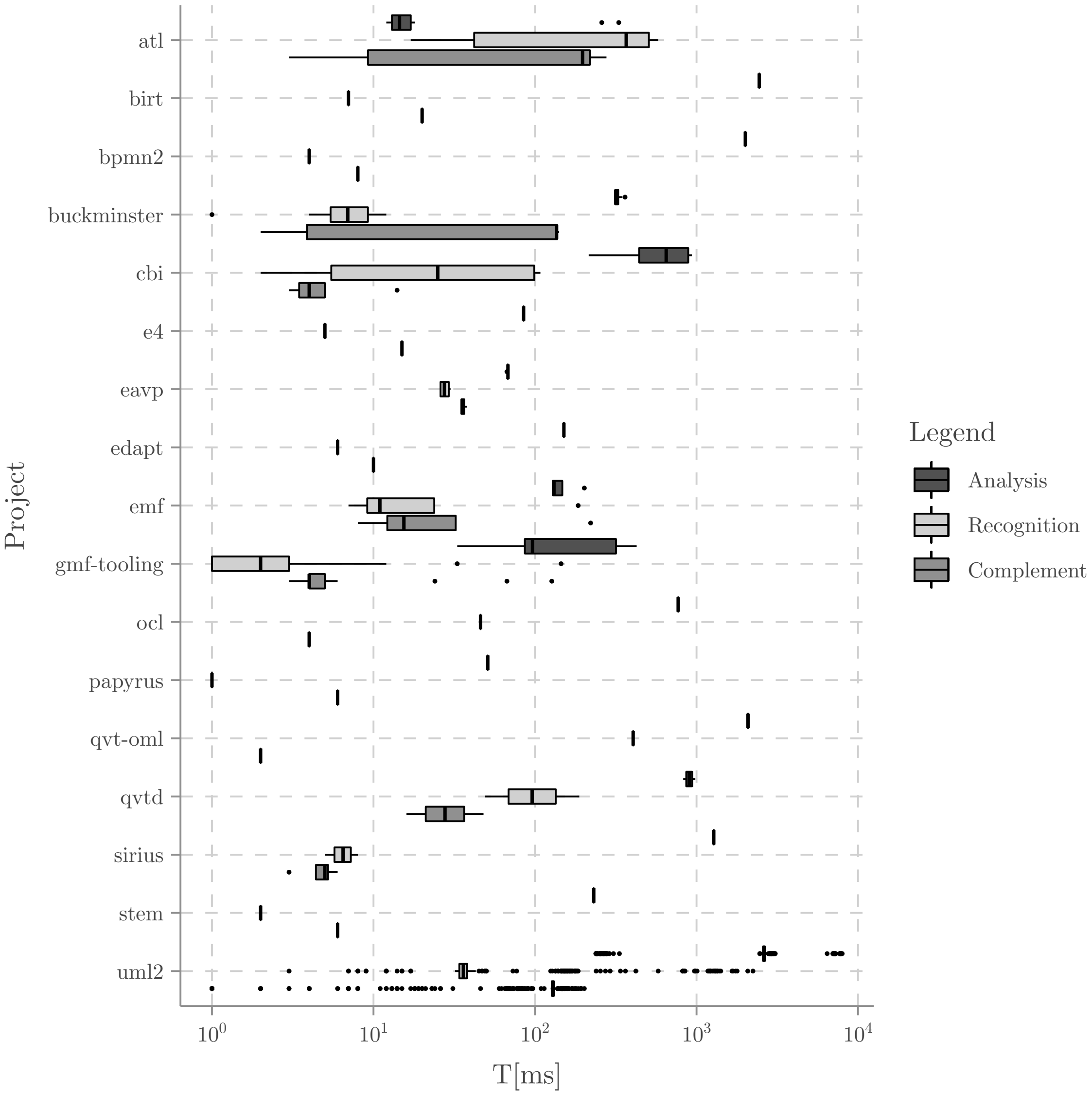


Figure 8.4. Runtime of the phases: Analysis, Recognition and Complementing.

In the majority of inconsistencies of our evaluation dataset, the overall runtimes to calculate repair proposals for a given inconsistency was in the order of seconds. Negative effects on the runtime could be observed if an inconsistency existed for a long period of time, or if a complement rule has many possible applications due to the combination of different parameter bindings. The results suggest that it is feasible to use ReVision in an interactive repair process in which inconsistencies are resolved step by step, as it is assumed by our approach.

8.4 Threats to Validity

8.4.1 Internal Validity

Our evaluation is based on an automated analysis of model histories. Several auxiliary tools have been developed for this purpose, a general threat to the internal validity of our evaluation results and thus pertains to their accuracy. First, model elements in a version history are identified by a stepwise calculation of corresponding elements between consecutive revisions. If an element temporarily does not occur in a version history, i.e., if it has been deleted in version v_i and re-created in a later version v_{i+x} with $x >= 2$, the chain of correspondences is broken, and the repair of the involved inconsistency is wrongly identified. However, as shown by an empirical study on model evolution [192], such gaps occur rather seldom in less than 2% of all correspondence chains. Second, for the calculation of corresponding elements in successive model versions, we use qualified names of model elements as signatures in our matching algorithm. This can influence the accuracy of tracing inconsistencies along version histories if (i) names are changed, but the re-named model elements remain the same or (ii) two different model elements accidentally have the same names. However, as shown in Reference [192], there is strong evidence that both errors do not have a significant influence on the correctness of matching results, particularly for class-diagram-like models which show a high stability of model element names.

The application conditions of the CPEOs are interpreted as invariants during the detection of partially applied CPEOs (see Section 4.5), i.e., the application conditions are checked stricter than necessary. This method slightly under-approximates the possibly recognizable sub-rules of a CPEO in a model difference. However, this conservative approach ensures that only valid complementations based on an edit rule's application conditions are computed. In particular, we could not observe any missing repair proposals with respect to RQ1 and RQ2, i.e., if proposals have been missed, they seem to be irrelevant with respect to the observed repairs in the model's history.

Last, our oracle slightly over-approximates the classification of historically observable repairs, i.e., repair alternatives which presumably match the intention of a developer. While we can be sure that all the change actions of a repair which is classified as historically observable occur as changes in the evolution step in which the inconsistency has been resolved, we cannot determine whether the proposed repair is complete in the sense of the intention of the developer. In other words, we observe the minimal repair but there might be some additional changes to form the actual edit step intended by the developer. Nevertheless, the observed repair is at least a subset of such an intended repair edit step, and any additional changes can be considered as out of the scope of the repair problem.

8.4.2 Construct Validity

Arguably, it is not our generic functionality that is evaluated in our experimental setup, but a configuration of ReVision which has been written by ourselves. We are familiar with the Ecore meta-model and the attached constraints. A toolchain configurator of an MDE environment, which shall use our technique for other modeling languages, might have a different background and thus come up with a less sophisticated set of CPEOs. This is a general issue pertaining to all rule-based approaches to automated model repair (see Chapter 9), which can hardly be evaluated in an academic setting, but which needs to be addressed in an empirical field study involving MDE practitioners. Nevertheless, we have lowered the formalization effort and probability of missing CPEOs by a tool-supported systematic process. Instead of formulating all consistent transformations for a modeling language, a tool configurator just has to prepare the correct model fragments regarding the consistency rules defined by a modeling language.

8.4.3 Conclusion Validity

Although our experimental results suggest that ReVision has the potential to serve as an interactive repair tool which may be effectively used in practice, we did not directly measure the effort needed by developers to turn an inconsistent model into a fully consistent one. This would need to consider the overall iterative process shown in Figure 5.7, measuring the time needed by developers to understand the generated repair proposals and assessing the impact of potential negative side effects caused by the repairs selected; all of which is not possible in an offline experiment as conducted by us. However, the general strategy of (interactively) handling one inconsistency at a time is not new but has also been suggested by others [149, 201]. More importantly, the majority of the inconsistencies in our experiments could be resolved by generated repair proposals whose effect is historically observable. We consider this as sufficient evidence that these repair proposals are understandable and plausible. Nonetheless, we acknowledge that empirical user studies and larger field studies are required to ultimately test the acceptance of ReVision in the future.

8.4.4 External Validity

A final important question is whether our results are generalizable. So far, we have only studied one particular modeling language and its consistency constraints. We choose Ecore as an evaluation target because of its widespread adoption in open-source projects. It is far more difficult to find a comprehensive and publicly available evaluation set for languages such as the UML [62]. However, from our UML case studies [5] we are confident that our repair approach is especially useful for any kind of multi-view modeling approach. In Ecore we could observe the same effect when inconsistencies arise in one model but were actually introduced by changes in a co-evolving model. In these cases, ReVision was able to detect the inconsistency-inducing changes in the co-evolving model and to generate a historically observable inconsistency resolution. The biggest challenge when adapting our approach to other modeling languages is the engineering of a set of CPEOs. The required efforts may be higher in case of larger meta-models, assuming that the number of consistency rules, and thus the number of CPEOs that need to be specified, correlates with the size of the meta-model.

9

Related Work

This chapter examines related publications to the presented approach of model repair. Related approaches are categorized into fully automated, language-specific, and adaptable model repair techniques. While the presented repair approach is fundamentally different from fully automated and language-specific approaches, it falls into the category of adaptable model repair. Regarding Chapter 7, the generation of edit rules is compared to related techniques.

This chapter addresses related work to the presented model repair recommendation approach. As discussed for the state of the art in Section 2.2, a feature-based classification system for automated model repair techniques has been proposed in Reference [115]. The classification system is driven by technical features, the engine underlying the automated repair generation procedure being the most distinguishing feature. Existing techniques are mainly categorized into *syntactic*, *rule-based* and *search-based approaches*. This scheme would classify our technique as a hybrid syntactic rule-based approach.

Regarding model repair recommendations, two other criteria are more suitable for positioning our approach. The first criterion is the supported *repair scenario*. Unlike fully automated approaches, our approach is considered a recommendation system, as discussed in Reference [132]. The second criterion is whether an approach to model repair is, like ours, *adaptable* in the sense that it can be applied to diverse modeling languages and kinds of inconsistencies. To that end, we will first review approaches that are hard to compare to ours since they address a different repair scenario (Section 9.1). Next, Section 9.2, we discuss how our approach differs from approaches that are designed to work with a specific modeling language or rely on specific assumptions, which are difficult to generalize. In Section 9.3, we will have a closer look at approaches that can be adapted to different modeling languages. Similar to the approach proposed in this thesis, such approaches consider model repair as an interactive yet semi-automated process in which a model repair tool plays the role of a recommendation system. Finally, in Section 9.4, concerning the *edit rule generation* approach introduced in Chapter 7, we have a look at related approaches and discuss whether such techniques can be helpful for generating CPEOs for our model repair scenario.

9.1 Fully Automated Model Repair

Most approaches to model repair proposed in the literature aim at a fully automated repair process, typically resolving all inconsistencies in a single step. This includes all search-based approaches which take the inconsistent model as input and search for a consistent model using off-the-shelf solvers [49, 90, 114, 186] or other domain-specific search heuristics [64, 147]. Exhaustive search strategies suffer from the well-known state space explosion problem, and turning a search-based approach into a recommendation system by collecting and presenting *all* solution candidates is simply infeasible [115].

Barriga et al. [17–19, 72] propose to tackle this problem using a reinforcement learning approach. The idea is to consider models as agents whose environment interactions are encoded as edit operations which, in turn, are rewarded or penalized based on their positive or negative impact with respect to consistency and based on potential feedback from developers. The hypothesis is that such a repair tool will learn the efficient yet relevant repairs for common inconsistencies over time. While these goals are similar to ours, their approach does not exploit domain knowledge when generating repair proposals, and there is yet no empirical evidence confirming its central hypothesis.

Finally, many rule-based approaches, including triple graph grammars (TGGs) [11, 54, 94, 166], bidirectional and incremental model transformation techniques [68], and other approaches which primarily focus on the synchronization of multi-view models [58, 92] fall into the category of fully automated model repair. Similar to search-based approaches, they typically aim at providing certain completeness and correctness guarantees, e.g., to always generate a repair turning an inconsistent model into a fully consistent one. As already mentioned, such guarantees are of minor importance or even *counterproductive* when using a repair tool as a recommendation system at design time, as it is the case in our scenario of model repair. Moreover, synchronization approaches are inherently restricted to handle multi-view consistencies, while consistency rules formulated in an OCL-like manner, as supported by our approach, may be used to specify both consistency rules over multiple views and over complex model fragments within a single model.

9.2 Language-Specific Repair

Many automated model repair techniques support only one specific type of model, e.g., UML models [45, 129, 184, 185], architectural models [38], or business process models [51]. These techniques cannot be transferred to other domain-specific modeling languages as it is the case for our adaptable technique.

Moreover, repair techniques that target a specific (family of) modeling languages often assume a very special notion of consistency. Fahland et al. [51], for instance, consider model repair as the task of aligning a given business process model with the logs of a monitored system Kautz and Rumpe [77] consider semantic model inconsistencies being introduced by erroneous model refinements. Given an inconsistent model which is supposed to be a refinement of an original model, i.e., its semantics is supposed to be subsumed by the semantics of the original model, the repair technique computes a minimal sequence of syntactic changes which repairs the inconsistent model towards refining the original one. Although the approach is not language-specific in general, it requires that the modeling language has a formal semantics. Frequently, however, there is no formal semantics and the meaning of a model is mainly defined by transformations to platform-specific models or code.

In the context of editing the source code of object-oriented programs, Jiang et al. [74] propose an approach to generate auto-completions upon so-called multi-entity edits. In response to adding a field (or method) to a class, their technique recommends missing changes such that all accesses to the added file (or method) are performed in a consistent manner. Although sharing with our approach the principle idea of complementing edits, the approach is tightly bound to fragments of object-oriented programs.

Finally, automated program repair has gained considerable attention in the software engineering research community. Since the advent of GenProg [190] in 2009 – which uses a genetic algorithm to evolve and search for source code patches that cause the buggy program to pass all test cases of a given test suite – various repair techniques have been proposed in the literature, and surveys can be found in, e.g., [106,132]. In the early years, similar to GenProg, most of the proposed techniques followed a syntax-based or generate-and-validate approach. They generate huge sets of candidate patches and validate the quality of these candidate patches against an existing test suite. Semantic program repair approaches synthesize patches based on a specific description of the correct behavior. Examples are corrected path conditions extracted from symbolic execution in SemFix [135] and Angelix [127], or a known correct reference implementation in SemGraft [126]. The approach proposed by Le et al. [105] is the only one known to us which exploits historical development data. It uses the histories of (large sets of) arbitrary software projects in order to mine recurring bug fix patterns, which are then used to control the generation of repair proposals for a given bug. Unlike their approach, we use the history of the inconsistent development artifact under consideration to analyze the cause of an inconsistency which, in turn, is used to generate repair proposals. In contrast to all approaches to program repair, we do not need executable program specifications and test cases in order to generate and validate candidate patches, and we aim at model repair on a syntactical level instead of fixing a program's behavior.

9.3 Adaptable Model Repair Recommendation

Only a minor fraction of the proposed approaches to model repair are, like ours, adaptable to various modeling languages and consider model repair as a recommendation system. They share with our approach the general process of typically handling a single consistency violation at a time.

This category includes the generic syntactic approaches producing abstract repair actions presented in References [134,148,149]. Their basic principles and disadvantages have already been discussed in Section 5.1.3 and Section 2.2. We draw from the line of research followed by Reder et al. (i) by adopting an iterative process in which each inconsistency is addressed in a single step, which helps developers in focusing on one inconsistency at a time, and (ii) by using validation techniques which let us determine the model elements on which the validation fails [151], which drastically reduces the search space for viable repairs. However, in addition to this spatial dimension of inconsistency analysis, we argue that there is also a temporal cause of an inconsistency that needs to be taken into account to optimally guide developers in resolving an inconsistency.

As discussed in Section 5.1.3, Marchezan at al. [120] extends the approach of Reder et al. [149] for computing abstract repairs. In particular, the approach introduces a so-called ownership-based filter for repair proposals based on the assumption that model elements are owned by a specific developer. Such a filter can be interesting to improve and reduce the ranking of repair techniques in general. However, the approach relies on very specific information that may not be available in a general modeling environment.

As an add-on functionality to the syntactic repair technique of Reder et al. [149,151], the work presented by Kretschmer et al. [95] specifically addresses those repair actions affecting attribute values of model elements. The goal is to help developers in turning abstract repair actions, which only mention the attribute which needs to be changed, into concrete repair actions, which also include the concrete value to be assigned to the attribute. Relying on what is known as the "plastic surgery hypothesis" [122] in the field of automated program repair, the idea is to synthesize suitable attribute values from other values of attributes that already exist in the model. In fact, this approach can be complementary to any repair generation technique that, like ours, primarily works on the structural level of a model's syntactic representation.

Kretschmer et al. [96] propose an approach to change propagation that, similar to ours, focuses on incomplete model changes. In relation to our approach, their technique does not utilize the version history of a model to identify the cause of an inconsistency. Instead, it assumes that the specific developer change that should be propagated is known. The technique of Kretschmer et al. [96] systematically tries alternative modifications for resolving inconsistencies. As discussed in Section 9.1, such state space exploration approaches can lead to huge numbers of repair alternatives. Moreover, instead of repairing one inconsistency at a time, the technique also addresses possible negative side effects in the same repair step. In comparison, the approach relies on elementary changes on a model's ASG rather than edit operations to suggest a ranking of repairs to the developer.

An approach that also deals with change propagation is presented by Marchezan et al. [119]. Similarly to Kretschmer et al. [96], the technique explores the state space of a model to find repairs and deal with possible negative side effects. The main focus of the approach is to detect conflicts of repairs with respect to changes of other developers during collaborative editing, i.e., it allows filtering repairs that would undo changes of other developers. Similar to our approach, Marchezan et al. [119] extracts these changes from a model's version history. While our repair approach prevents undoing former changes that introduced an inconsistency, their approach avoids repairs that undo recent changes of other developers. In general, such a filter can complement a repair technique in a scenario where the intent is to avoid editing conflicts rather than resolving those conflicts in a merging process (see Reference [81]) at a later time.

In general, there are a few rule-based techniques sharing some similarities with our approach [2, 28, 133, 199]. Similar to a set of CPEOs serving as configuration input of our technique, they rely on a set of predefined rules which are used to adapt the technique to a given modeling language and which are applied when an inconsistency is detected.

Nassar et al. [133] present an algorithm for repairing some kinds of inconsistencies in EMF models by trimming and completing an inconsistent model. Although being designed to work fully automatically, tool support is provided such that certain decisions of the algorithm can be interactively influenced by developers. However, only elementary constraints in EMF models [27] and multiplicity constraints defined by a meta-model are supported, while our approach can handle arbitrary OCL-like constraints.

The approach presented by Blanc et al. [28] shares with ours the idea of exploiting the changes causing an inconsistency to better control the generation of possible repairs. However, the configuration effort is much higher than for our approach. The specification of inconsistency detection rules is an integral part of their technique for inconsistency resolution, while we use consistency rules specified in an OCL-like manner which are readily available for a given modeling language. In order to use readily available OCL constraints as a basis, Xiong et al. [199] propose a language called Beanbag as an OCL add-on to describe fixing procedures along with consistency rules. A fixing procedure shall specify how to propagate inconsistency-inducing changes in order to restore consistency. In contrast to our approach, however, they do not perform a history-based origin and impact analysis at runtime, which makes it hard and often impossible to anticipate a suitable fixing procedure at design time. Moreover, we offer a semi-automated process for specifying and generating CPEOs, whereas Blanc et al. [28] and Xiong et al. [199] rely on a purely manual specification of rules.

Taentzer et al. [2] consider the construction of complement rules, which we adopt in our approach (see Section 5.4.3), from a theoretical point of view. However, the possible complement rules are already determined at design time. Therefore, the approach assumes that a set of edit operations for a given modeling language is partitioned into CPEOs and non-CPEOs, the latter being true sub-rules of the former. Partial edits are suggested to be complemented whenever possible without analyzing the cause of an inconsistency as in our approach. Thus, important requirements of a recommendation system, notably the filtering of all possible complements to the most relevant repair alternatives, are not addressed.

Finally, Schneider et al. [162–164] present an approach to graph repair in which they formalize consistency by graph conditions which are exploited to derive repair proposals using constraint solving techniques. To date, however, the work is a purely theoretical one. It would be interesting to transfer their concept into the MDE context and to implement a prototypical tool for the sake of a comparative, experimental evaluation.

9.4 Edit Rule Generation

In Chapter 7, we present a novel systematic process to synthesize CEPOs that meet our requirement of preventing typical inconsistencies when models are edited in standard editors. As already discussed in Section 2.2, some approaches exists that, similar to ours, derive edit rules from modeling examples. The demonstration-based approach of Brosch et al. [31] and Sun et al. [176, 177] record edit rules from example edits of a model. Similar to our approach, the correspondence-based approach of Saada et al. [158], Alshanqiti et al. [10] and Strüber et al. [3] derive edit rules from pairs of modeling examples. In contrast to these approaches, we derive multiple edit rules by combining several minimal modeling examples into different types of CPEOs that capture complex and error-prone edit steps with respect to complex modeling constraints. This systematic approach significantly lowers the number of required examples.

Kehrer et al. [79, 84, 85, 153] present an approach and supporting tool suite called SERGe to derive edit rules that preserve basic structure and typing constraints from a given meta-model. Similarly, some approaches derive graph grammars from meta-models [47, 55, 178] that consists of basic constructive edit rules. As already discussed in Section 2.2, these edit operations can hardly serve as CPEOs for our repair approach, which concentrates on advanced consistency constraints.

Ghannem et al. [57] present an approach that derives refactoring operations from modeling examples. With the same goal, Mokaddem et al. [57] developed a search-based approach that yields model refactorings. Regarding our repair approach, refactorings are indeed an interesting category of CPEOs. However, as analyzed in Section 1.2, we have to consider that incomplete edits can occur during several different development stages and modeling activities.

Tinnes et al. [182, 183] describe a novel unsupervised approach called OCKHAM, which learns domain-specific edit operations from histories in model repositories. OCKHAM employs frequent subgraph mining to discover actually applied domain-specific edit operations in a model difference graphs. Assuming a sufficiently large modeling history with respect to a specific modeling language, this approach indeed can help to extract interesting CPEOs. While the approach presented in Chapter 7 can produce an extensive basis of CPEOs without any modeling histories, e.g., while designing a new modeling language, additional CPEOs could be added by such a mining approach during a model's evolution.

Finally, model transformation by-example approaches such as References [15, 159, 187] deal with learning transformations from exemplary source and target models. However, the derived transformation rules are designed to translate between models of different modeling languages rather than formulating edit operations of a particular modeling language. Therefore, it has yet to be investigated whether these approaches can be adapted to generate CPEOs with respect to different complex constraints within the same modeling language or between modeling views.

Technically, ReVision can be configured with any edit rule that can be expressed declaratively and converted into an edit rule graph as described in Section 3.5. Our history-based repair approach determines on a case-specific basis if an incompletely applied edit rule might have caused the inconsistency under consideration and whether a complementation with respect to this rule could improve or resolve it. Most existing approaches do not guarantee that the generated edit rules preserve a model's consistency with respect to the complex constraints of a modeling language. Generally, the systematic approach introduced in Chapter 7 can serve as starting point for a comprehensive set of CPEOs while additional edit rule generation approaches could be used to complement this set.

10 Conclusions and Future Work

This chapter summarizes the idea, computation, and evaluation of history-based repair recommendations presented in this thesis. Finally, possible directions for future work are discussed, such as extending the approach to synchronization of multiple interrelated views and vertical model consistency, extracting CPEOs from the modeling history, and generalizing the approach to general-purpose programming languages.

To conclude this thesis, we will summarize the main contributions and findings in Section 10.1. Finally, Section 10.2 provides a perspective for future work of the approach.

10.1 Summary

In MDE, complex software systems are described by models that abstract from technical details and represent different views of the system. However, inconsistencies can arise during a model's evolution, and the number of possible repairs is usually huge.

As we observed in our evaluation of real-world model histories, such inconsistencies are often caused by incomplete editing processes. Thus, this thesis presents a new history-based approach to derive repair proposals from incomplete edits. Such edits are incomplete in the sense that additional changes are necessary to achieve a new consistent state of a model. One overly simple decision is to undo these changes. In most cases, the better decision is to complement such an incomplete edit step and to catch up on the missing changes. The proposed complementations are particularly useful in resolving inconsistencies introduced by editing complex model fragments or by isolated editing of dependent modeling views.

Our approach assumes that a model version history is available to detect an incomplete edit step within this history. This detection requires the complex transitions between consistent states to be specified as consistency-preserving edit operations (CPEOs). Such CPEOs can be expressed using model transformations based on graph transformation rules. As a compact notation, related changes can be represented using annotated graphs, which are referred to as difference graphs and edit rule graphs, respectively.

The biggest technical challenge is to find partial edit rule graphs in the difference graph that can be recognized as partial applications of the corresponding CPEOs. Such a partial graph matching algorithm can be implemented using a constraint satisfaction problem

(CSP) solver. The CSP solver detects partially executed edit operations by mapping change actions of an edit rule graph to historical changes of a model difference graph. Based on a recognized sub-rule, a complement rule is computed by subtracting the already applied changes of the sub-rule from the edit rule. To create concrete complementations, we determine all valid applications of the complement rule on the current model version. Finally, all valid applications are mapped to parameter bindings of the complement rule, which a developer can select. As an alternative to complementing an incomplete edit, the changes of a recognized sub-rule can be inverted into a case-specific rollback operation.

The presented history-based model repair recommendation approach is implemented in our tool called ReVision. The tool supports developers in repairing each single violation of a consistency rule iteratively. Based on a selected inconsistency, it analyzes the model history to detect the former model changes that induced the inconsistency. Utilizing the algorithm to detect partially executed edit operations in a model difference, ReVision generates complementing repair proposals extending these inconsistency-inducing changes. Therefore, such a complementing repair is a case-specific repair operation that extends a partial execution of a CPEO into a complete one. This way, implausible repair alternatives can be largely avoided, and we also prevent undoing former edit steps. Finally, the repair recommendations will be ranked using multiple properties of the repair operation. Basically, all repair plans are compared by their change ratio with respect to the elementary change actions of the complement and sub-rule. The pairs of inconsistency-inducing and complementing changes are then presented to the developer as a ranked list of repair proposals. The developer can either apply the complementation or create a case-specific undo operation to resolve the inconsistency.

In general, an inconsistency cannot be resolved by our approach if no suitable CPEO is available. To support the configuration of the repair tool, we provide a systematic yet semi-automated process to derive a set of CPEOs from the modeling language's meta-model and its attached set of consistency rules. The example-based generation assembles CPEOs based on a given set of minimal modeling examples, each describing a valid model fragment in terms of a consistency rule. This approach allows developers to specify CPEOs using their preferred model editor without directly formulating edit rules using the model's abstract syntax. To guide the generation process, annotations can be added to the derived graph fragments.

Compared to other model repair approaches, our approach delivers more concrete repair proposals. It avoids implausible proposals by analyzing the context and history of the inconsistency, and it avoids accidentally undoing former work. These qualitative achievements are confirmed by our experimental results. The proposed approach is evaluated using a selected set of real-world models obtained from popular open-source Eclipse modeling projects. The experimental results confirm that most of the inconsistencies can be resolved by complementing incomplete edits. The number of repair operations generated by the approach is typically low, and the relevant proposal is ranked at the topmost position of the short list of repairs in virtually all cases.

All evaluation results are published and peer-reviewed in Reference [6]. Moreover, the dataset and evaluation configuration is available from the project website of ReVision (see Reference [8]). For 92.6% of the supported inconsistencies of our experimental dataset, Re-

Vision generates a repair proposal that is relevant in the sense that it can be observed in the model's editing history and thus, arguably, meets a developer's intention. In fact, in 94.9% of the cases, the most relevant repair is ranked on the top position of the ranked repair proposals generated by ReVision, and 92.2% of these repair proposals are complements of partial executions of CPEOs inducing an inconsistency. Performance measurements show that repair proposals for a given inconsistency are typically generated in the order of seconds, which suggests that it is feasible to use ReVision in an iterative yet interactive repair process as it is assumed by our approach.

Although the configuration of ReVision used in our empirical study covers 93.4% of the inconsistencies of our evaluation dataset, our approach has two limitations: (i) It does not support consistency constraints which pertain to the well-formedness of attribute values of model elements, and (ii) it does not provide any formal guarantees with respect to the coverage of structural constraints. The latter primarily depends on the set of CPEOs specified for a given modeling language.

The history-based model recommendation approach allows developers to repair model inconsistencies efficiently and effectively. In this context, ReVision provides a valuable tool for MDE practitioners to resolve inconsistencies in models. Despite that, ReVision is not meant to serve as a replacement for existing approaches to model repair but rather as a complementary tool that should be used as the first one in a potentially larger chain of repair techniques.

10.2 Future Work

An interesting avenue for future work on conducting acceptance testing with real developers is to evaluate ReVision in settings where consistency rules span more than just two views, a scenario that is hardly addressed in the current literature on model repair. Combining the specifications of multiple views might also help the repair tool to compute more precise repair proposals, i.e., proposals that create conflicts between different views could be filtered. Technically, the specification of CPEOs can be extended to cover multiple views. The actual challenge here is to deal with the potential combinatorial explosion of the number of CPEOs that need to be specified at design time. In particular, if a consistency rule spans multiple views, not all of these views may be used in a concrete model. By using only the basic constructs of the Henshin transformation language, all of the combinations of multiple views would have to be covered by a separate CPEO. However, we assume that this problem of combinatorial explosion can be tackled by specifying optional parts in CPEOs. This may be achieved by using advanced concepts such as rule schemes or variability add-ons for the Henshin language.

This thesis has primarily focused on horizontal consistency concerning models at the same level of abstraction or within the same development phase. However, another important consistency dimension in MDE is vertical consistency, which involves synchronizing different levels of abstraction. In MDE, models are typically transformed from an abstract platform-independent model (PIM) to a more concrete platform-specific model (PSM). Existing incremental synchronization approaches automatically propagate the modifications from the abstract to the more concrete model without fully regenerating the model. Similarly, ReVision could propagate such changes to the PSM based on the applied modifications on the PIM. To enable automated synchronization without user interactions in ReVision, stricter assumptions have to be made about CPEOs, consistency rules, and the history-based analysis to guarantee a correct automatic synchronization.

Conversely, a repair tool can also support the developer in backpropagating the manual changes from a PSM to its PIM. As the model evolves, this ensures the integrity of the model in further development cycles. This synchronization direction is mainly done manually in practice or requires specifying bidirectional synchronization rules. Our repair recommendation approach could support the developer in synchronizing frequently occurring modifications without fully specifying both transformation directions.

In addition, future work can focus on extending the proposed approach beyond modeling languages and Domain-Specific Modeling Languages (DSMLs) to involve general-purpose programming languages and other textual Domain-Specific Languages (DSLs). Such a generalization of the approach includes the identification of possible inconsistencies and corresponding CPEOs with respect to these languages. One particularly interesting starting point to investigate consistency-preserving edits are applications of refactorings in such languages. In this context, our history-based repair technique could address the migration of breaking changes in Application Programming Interfaces (APIs), which often lead to inconsistencies in client libraries accessing these APIs. Technically, this requires bridging the conceptual gap between the abstract syntax of general-purpose languages and the abstract syntax defined by meta-models. Moreover, this problem is not only relevant to our repair approach but also a general challenge in transferring and utilizing modeling-related approaches to general-purpose programming languages.

Despite the widespread adoption of auto-completion techniques in programming languages, they are still rarely available in the practice of MDE. Such tools play a crucial role in helping developers to understand a language's syntax and the functionality of large code bases and libraries. Context-sensitive auto-completion tools can significantly enhance development productivity, particularly when dealing with repetitive language patterns. The technique for complementation of partially executed edits, as described in Chapter 4, can be utilized to compute history-based auto-completions for models in MDE. However, potential completions of an incomplete edit step can become ambiguous when the specific editing goal, e.g., repairing a particular inconsistency, is unclear. This ambiguity can result in a large number of possible completions, making it challenging for a developer to select the appropriate one. One interesting approach could involve combining similar completion proposals. If multiple CPEOs share common parts, the developer could first select from a ranking of combined proposals, narrowing down the appropriate one. Then, in a second interactive step, a concrete variant of the proposal can be selected.

In our current approach, CPEOs are generated from modeling fragments derived from modeling examples. An interesting future direction is to reverse this process by converting CPEOs into modeling fragments and modeling examples. Technically, this allows to backpropagate manual revision of the CPEOs (or modeling fragments) to the modeling examples, allowing an additional way to refine the set of CPEOs. More interestingly, this allows us to integrate other CPEO generation techniques and existing sets of CPEOs into the generation process. Assuming the CPEO is described by a precondition and postcondition graph, we can basically derive two new modeling fragments. By combining these two new fragments with the already existing ones, we can generate multiple CPEOs with respect to the different combinations of fragments and rule categories supported by our generation tool.

Another interesting approach for extending the CPEO generation based on modeling fragments could be to extract such complex fragments from the modeling history. There are two particularly interesting sources to explore: complex consistent edit steps within a single revision and complex intermediate inconsistent edit steps across multiple model versions. The latter source has served as an oracle in our evaluation. Additionally, we can analyze both consistent and inconsistent editing steps to learn project-specific rankings of parameter values, e.g., model elements that have to be selected for a repair proposal by a developer.

Our experimental findings support our hypothesis that the majority of inconsistencies result from incomplete edits. However, we recognize that conducting empirical user studies and more extensive field studies will be essential to definitively assess the acceptance of ReVision in the future. Furthermore, it would be valuable to extend the evaluation to various languages and compare it with related approaches to model repair. Presently, the absence of a standard benchmark makes it challenging to empirically compare these different repair approaches. Future advancements in model repair can only be achieved if the diverse approaches and repair tools can be systematically compared through experimental results obtained using common benchmarks.

References

[8] ReVision. https://repairvision.github.io/, 2023.

[9] M. Alanen, I. Porres, et al. *A relation between context-free grammars and meta object facility metamodels.* Citeseer, 2004.

[10] A. M. Alshanqiti, R. Heckel, and T. Khan. Learning minimal and maximal rules from observations of graph transformations. *Electronic Communications of the EASST*, 58, 2013.

[11] C. Amelunxen, E. Legros, A. Schürr, and I. Stürmer. Checking and enforcement of modeling guidelines with graph transformations. *Applications of graph transformations with industrial relevance*, pages 313–328, 2008.

[12] A. Anjorin, S. Rose, F. Deckwerth, and A. Schürr. Efficient model synchronization with view triple graph grammars. In *European Conference on Modelling Foundations and Applications*, pages 1–17. Springer, 2014.

[13] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: advanced concepts and tools for in-place emf model transformations. *Model Driven Engineering Languages and Systems*, pages 121–135, 2010.

[14] T. Arendt and G. Taentzer. A tool environment for quality assurance based on the eclipse modeling framework. *Automated Software Engineering*, 20(2):141–184, 2013.

[15] I. Baki and H. Sahraoui. Multi-step learning and adaptive search for learning complex model transformations from examples. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(3):1–37, 2016.

[16] R. Balzer. Tolerating inconsistency. In *Proceedings of the 13th International Conference on Software Engineering*, pages 158–165. IEEE Computer Society Press, 1991.

[17] A. Barriga, R. Heldal, A. Rutle, and L. Iovino. Parmorel: a framework for customizable model repair. *Software and Systems Modeling*, 21(5):1739–1762, 2022.

[18] A. Barriga, A. Rutle, and R. Heldal. Automatic model repair using reinforcement learning. In *MoDELS (Workshops)*, pages 781–786, 2018.

[19] A. Barriga, A. Rutle, and R. Heldal. Applying reinforcement learning to personalize automatic model repairing. *The Journal of Object Technology*, 2019.

[20] F. L. Bauer, L. Bolliet, H. Helms, P. Naur, and B. Randell. Nato software engineering conference. In *Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany. Edited by P. Naur and B. Randell*, 1968.

[21] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87, 2000.

[22] G. Bergmann, I. Ráth, G. Varró, and D. Varró. Change-driven model transformations. *Software & Systems Modeling*, 11(3):431–461, 2012.

[23] L. Bettini. *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing Ltd, 2016.

[24] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. Graphical definition of in-place transformations in the eclipse modeling framework. In *International Conference on Model Driven Engineering Languages and Systems*, pages 425–439. Springer, 2006.

[25] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. Emf model refactoring based on graph transformation concepts. *Electronic Communications of the EASST*, 3, 2007.

[26] E. Biermann, C. Ermel, and G. Taentzer. Lifting parallel graph transformation concepts to model transformation based on the eclipse modeling framework. *Electronic Communications of the EASST*, 26, 2010.

[27] E. Biermann, C. Ermel, and G. Taentzer. Formal foundation of consistent emf model transformations by algebraic graph transformation. *Software and Systems Modeling*, 11(2):227–250, 2012.

[28] X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In *Proceedings of the 30th international conference on Software engineering*, pages 511–520. ACM, 2008.

[29] S. Bobek, M. Baran, K. Kluza, and G. J. Nalepa. Application of bayesian networks to recommendations in business process modeling. In *AIBP@ AI* IA*, pages 41–50, 2013.

[30] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice.* Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[31] P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger. An example is worth a thousand words: Composite operation modeling by-example. In *Model Driven Engineering Languages and Systems: 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings 12*, pages 271–285. Springer, 2009.

[32] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan. Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, 13(1):3–30, 2001.

[33] T. Clark, P. Sammut, and J. Willans. Applied metamodelling: a foundation for language driven development., 2008.

[34] B. Combemale, J. Deantoni, B. Baudry, R. B. France, J.-M. Jézéquel, and J. Gray. Globalizing modeling languages. *Computer*, 47(6):68–71, 2014.

[35] A. Cunha, N. Macedo, and T. Guimaraes. Target oriented relational model finding. In *International Conference on Fundamental Approaches to Software Engineering*, pages 17–31. Springer, 2014.

[36] M. A. A. Da Silva, A. Mougenot, X. Blanc, and R. Bendraou. Towards automated inconsistency handling in design models. In *International Conference on Advanced Information Systems Engineering*, pages 348–362. Springer, 2010.

[37] B. Dagenais and M. P. Robillard. Semdiff: Analysis and recommendation support for api evolution. In *2009 IEEE 31st International Conference on Software Engineering*, pages 599–602. IEEE, 2009.

[38] H. K. Dam, A. Reder, and A. Egyed. Inconsistency resolution in merging versions of architectural models. In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pages 153–162. IEEE, 2014.

[39] H. K. Dam and M. Winikoff. Supporting change propagation in uml models. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[40] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *Theoretical Computer Science*, 376(3):139–163, 2007.

[41] S. Easterbrook and B. Nuseibeh. Using viewpoints for inconsistency management. *Software Engineering Journal*, 11(1):31–43, 1996.

[42] Eclipse Foundation. Eclipse Git repositories. `https://git.eclipse.org/c/`, 2019.

[43] A. Egyed. Instant consistency checking for the uml. In *Proceedings of the 28th international conference on Software engineering*, pages 381–390. ACM, 2006.

[44] A. Egyed. Automatically detecting and tracking inconsistencies in software design models. *IEEE Transactions on Software Engineering*, 37(2):188–204, 2011.

[45] A. Egyed, E. Letier, and A. Finkelstein. Generating and evaluating choices for fixing inconsistencies in uml design models. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 99–108. IEEE Computer Society, 2008.

[46] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[47] K. Ehrig, J. M. Küster, and G. Taentzer. Generating instance models from meta models. *Software & Systems Modeling*, 8(4):479–500, 2009.

[48] R. Eramo, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio. A model-driven approach to automate the propagation of changes among architecture description languages. *Software & Systems Modeling*, 11(1):29–53, 2012.

[49] R. Eramo, A. Pierantonio, J. R. Romero, and A. Vallecillo. Change management in multi-viewpoint system using ASP. In *Enterprise Distributed Object Computing Conference Workshops, 2008 12th*, pages 433–440. IEEE, 2008.

[50] R. Eshuis and R. Wieringa. A real-time execution semantics for uml activity diagrams. In *International Conference on Fundamental Approaches to Software Engineering*, pages 76–90. Springer, 2001.

[51] D. Fahland and W. M. van der Aalst. Model repair–aligning process models to reality. *Information Systems*, 47:220–243, 2015.

[52] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.

[53] U. Frank. Multilevel modeling. *Business & Information Systems Engineering*, 6(6):319–337, 2014.

[54] L. Fritsche, J. Kosiol, A. Schürr, and G. Taentzer. Efficient model synchronization by automatically constructed repair processes. In *Fundamental Approaches to Software Engineering: 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings 22*, pages 116–133. Springer, 2019.

[55] L. Fürst, M. Mernik, and V. Mahnič. Converting metamodels to graph grammars: doing without advanced graph grammar features. *Software & Systems Modeling*, 14(3):1297–1317, 2015.

[56] S. Getir, M. Rindt, and T. Kehrer. A generic framework for analyzing model co-evolution. In *ME@MoDELS*, pages 12–21, 2014.

[57] A. Ghannem, M. Kessentini, M. S. Hamdi, and G. El Boussaidi. Model refactoring by example: A multi-objective search based software engineering approach. *Journal of Software: Evolution and Process*, 30(4):e1916, 2018.

[58] H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *Software & Systems Modeling*, 8:21–43, 2009.

[59] R. L. Glass. Maintenance: Less is not more. *IEEE software*, 15(4):67–68, 1998.

[60] M. Goedicke, T. Meyer, and G. Taentzer. Viewpoint-oriented software development by distributed graph transformation: Towards a basis for living with inconsistencies. In *Requirements Engineering, 1999. Proceedings. IEEE International Symposium on*, pages 92–99. IEEE, 1999.

[61] J. Grundy, J. Hosking, and W. B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engineering*, 24(11):960–981, 1998.

[62] R. Hebig, T. H. Quang, M. R. Chaudron, G. Robles, and M. A. Fernandez. The quest for open source projects that use uml: mining github. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 173–183. ACM, 2016.

[63] R. Heckel and G. Taentzer. *Graph Transformation for Software Engineers.* Springer, 2020.

[64] A. Hegedüs, A. Horváth, I. Ráth, M. C. Branco, and D. Varró. Quick fix generation for dsmls. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 17–24. IEEE, 2011.

[65] F. Hermann, H. Ehrig, C. Ermel, and F. Orejas. Concurrent model synchronization with conflict resolution based on triple graph grammars. In *International Conference on Fundamental Approaches to Software Engineering*, pages 178–193. Springer, 2012.

[66] F. Hermann, H. Ehrig, F. Orejas, K. Czarnecki, Z. Diskin, Y. Xiong, S. Gottmann, and T. Engel. Model synchronization based on triple graph grammars: correctness, completeness and invertibility. *Software & Systems Modeling*, 14(1):241–269, 2015.

[67] T. Hettel, M. Lawley, and K. Raymond. Towards model round-trip engineering: An abductive approach. In *International Conference on Theory and Practice of Model Transformations*, pages 100–115. Springer, 2009.

[68] S. Hidaka, M. Tisi, J. Cabot, and Z. Hu. Feature-based classification of bidirectional transformation approaches. *Software & Systems Modeling*, 15(3):907–928, 2016.

[69] T. Hornung, A. Koschmider, and A. Oberweis. Rule-based autocompletion of business process models. In *CAiSE Forum*, volume 247, pages 222–232, 2007.

[70] W. S. Humphrey. The software engineering process: definition and scope. In *Proceedings of the 4th international software process workshop on Representing and enacting the software process*, pages 82–83, 1988.

[71] Z. Huzar, L. Kuzniarz, G. Reggio, and J. L. Sourrouille. Consistency problems in uml-based software development. In *International Conference on the Unified Modeling Language*, pages 1–12. Springer, 2004.

[72] L. Iovino, A. Barriga Rodriguez, A. Rutle, and R. Heldal. Model repair with quality-based reinforcement learning. *The Journal of Object Technology*, 2020.

[73] ISO/IEC/IEEE. Iso/iec/ieee international standard - systems and software engineering–vocabulary. *ISO/IEC/IEEE 24765:2017(E)*, pages 1–541, 2017.

[74] Z. Jiang, Y. Wang, H. Zhong, and N. Meng. Automatic method change suggestion to complement multi-entity edits. *Journal of Systems and Software*, 159:110441, 2020.

[75] A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[76] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, and M. Wimmer. Model transformation by-example: a survey of the first wave. In *Conceptual modelling and its theoretical foundations*, pages 197–215. Springer, 2012.

[77] O. Kautz and B. Rumpe. On computing instructions to repair failed model refinements. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 289–299. ACM, 2018.

[78] T. Kehrer. *Calculation and propagation of model changes based on user-level edit operations: a foundation for version and variant management in model-driven engineering.* PhD thesis, University of Siegen, 2015.

[79] T. Kehrer, A. Alshanqiti, and R. Heckel. Automatic inference of rule-based specifications of complex in-place model transformations. In *International Conference on Theory and Practice of Model Transformations*, pages 92–107. Springer, 2017.

[80] T. Kehrer, U. Kelter, P. Pietsch, and M. Schmidt. Adaptability of model comparison tools. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 306–309. ACM, 2012.

[81] T. Kehrer, U. Kelter, and D. Reuling. Workspace updates of visual models. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 827–830, 2014.

[82] T. Kehrer, U. Kelter, and G. Taentzer. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 163–172. IEEE Computer Society, 2011.

[83] T. Kehrer, U. Kelter, and G. Taentzer. Consistency-preserving edit scripts in model versioning. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 191–201. IEEE, 2013.

[84] T. Kehrer, M. Rindt, P. Pietsch, and U. Kelter. Generating edit operations for profiled uml models. In *ME@ MoDELS*, pages 30–39. Citeseer, 2013.

[85] T. Kehrer, G. Taentzer, M. Rindt, and U. Kelter. Automatically deriving the specification of model editing operations from meta-models. In *ICMT*, 2016.

[86] U. Kelter, J. Wehren, and J. Niere. A generic difference algorithm for uml models. *Software Engineering 2005*, 2005.

[87] A. A. Khwaja and J. E. Urban. Syntax-directed editing environments: Issues and features. In *Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing: states of the art and practice*, pages 230–237, 1993.

[88] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

[89] D. Kim, Y. Tao, S. Kim, and A. Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering*, 39(11):1597–1610, 2013.

[90] M. Kleiner, M. D. Del Fabro, and P. Albert. Model search: Formalizing and automating constraint solving in mde platforms. In *European Conference on Modelling Foundations and Applications*, pages 173–188. Springer, 2010.

[91] K. Kluza, M. Baran, S. Bobek, and G. J. Nalepa. Overview of recommendation techniques in business process modeling. In *Proceedings of 9th Workshop on Knowledge Engineering and Software Engineering (KESE9)*, pages 46–57. Citeseer, 2013.

[92] D. Kolovos, R. Paige, and F. Polack. Detecting and repairing inconsistencies across heterogeneous models. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 356–364. IEEE, 2008.

[93] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *Comparison and Versioning of Software Models, 2009. CVSM'09. ICSE Workshop on*, pages 1–6. IEEE, 2009.

[94] J. Kosiol. *Formal Foundations for Information-Preserving Model Synchronization Processes Based on Triple Graph Grammars.* PhD thesis, Philipps-Universität Marburg, 2022.

[95] R. Kretschmer, D. E. Khelladi, A. Demuth, R. E. Lopez-Herrejon, and A. Egyed. From abstract to concrete repairs of model inconsistencies: an automated approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 456–465. IEEE, 2017.

[96] R. Kretschmer, D. E. Khelladi, R. E. Lopez-Herrejon, and A. Egyed. Consistent change propagation within models. *Software and Systems Modeling*, 20:539–555, 2021.

[97] E. B. Krissinel and K. Henrick. Common subgraph isomorphism detection by backtracking search. *Software: Practice and Experience*, 34(6):591–607, 2004.

[98] T. Kuschke and P. Mäder. Pattern-based auto-completion of uml modeling activities. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 551–556, 2014.

[99] T. Kuschke, P. Mäder, and P. Rempel. Recommending auto-completions for software modeling activities. In *Model-Driven Engineering Languages and Systems: 16th International Conference, MODELS 2013, Miami, FL, USA, September 29–October 4, 2013. Proceedings 16*, pages 170–186. Springer, 2013.

[100] J. M. Küster and K. Ryndina. Improving inconsistency resolution with side-effect evaluation and costs. In *International Conference on Model Driven Engineering Languages and Systems*, pages 136–150. Springer, 2007.

[101] G. M. Lahijany, M. Ohrndorf, J. Zenkert, M. Fathi, and U. Kelte. Identibug: Model-driven visualization of bug reports by extracting class diagram excerpts. In *2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 3317–3323. IEEE, 2021.

[102] C. Larman and V. R. Basili. Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56, 2003.

[103] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of uml statechart diagrams. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 331–347. Springer, 1999.

[104] M. Lauder, A. Anjorin, G. Varró, and A. Schürr. Efficient model synchronization with precedence triple graph grammars. In *International Conference on Graph Transformation*, pages 401–415. Springer, 2012.

[105] X. B. D. Le, D. Lo, and C. Le Goues. History driven program repair. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 213–224. IEEE, 2016.

[106] C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[107] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[108] M. M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124. Springer, 1996.

[109] H. Lieberman. *Your wish is my command: Programming by example.* Morgan Kaufmann, 2001.

[110] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, 1978.

[111] W. Liu, S. Easterbrook, and J. Mylopoulos. Rule-based detection of inconsistency in uml models. In *Workshop on Consistency Problems in UML-Based Software Development*, volume 5, 2002.

[112] F. J. Lucas, F. Molina, and A. Toval. A systematic review of uml model consistency management. *Information and Software technology*, 51(12):1631–1645, 2009.

[113] N. Macedo and A. Cunha. Least-change bidirectional model transformation with qvt-r and atl. *Software & Systems Modeling*, 15(3):783–810, 2016.

[114] N. Macedo, T. Guimaraes, and A. Cunha. Model repair and transformation with echo. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 694–697. IEEE Press, 2013.

[115] N. Macedo, T. Jorge, and A. Cunha. A feature-based classification of model repair approaches. *IEEE Transactions on Software Engineering*, 2016.

[116] N. M. Macedo, A. Cunha, and H. P. Pacheco. Towards a framework for multidirectional model transformations. *CEUR Workshop Proceedings*, 1133, 2014.

[117] S. Mafazi, W. Mayer, and M. Stumptner. Conflict resolution for on-the-fly change propagation in business processes. In *Proceedings of the Tenth Asia-Pacific Conference on Conceptual Modelling-Volume 154*, pages 39–48, 2014.

[118] M. S. Mahoney. Finding a history for software engineering. *IEEE Annals of the History of Computing*, 26(1):8–19, 2004.

[119] L. Marchezan, W. K. Assuncao, R. Kretschmer, and A. Egyed. Change-oriented repair propagation. In *Proceedings of the International Conference on Software and System Processes and International Conference on Global Software Engineering*, pages 82–92, 2022.

[120] L. Marchezan, R. Kretschmer, W. K. Assunção, A. Reder, and A. Egyed. Generating repairs for inconsistent models. *Software and Systems Modeling*, pages 1–33, 2022.

[121] L. Marchezan, G. K. Michelon, W. K. Assunção, and A. Egyed. Do developers benefit from recommendations when repairing inconsistent design models? a controlled experiment. In *The International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 06 2023. (Pre-print).

[122] M. Martinez, W. Weimer, and M. Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 492–495. ACM, 2014.

[123] S. Mazanek, S. Maier, and M. Minas. Auto-completion for diagram editors based on graph grammars. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 242–245. IEEE, 2008.

[124] S. Mazanek and M. Minas. Business process models as a showcase for syntax-based assistance in diagram editors. In *MoDELS*, pages 322–336. Springer, 2009.

[125] J. J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12(1):23–34, 1982.

[126] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 129–139, 2018.

[127] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701, 2016.

[128] T. Mens. On the use of graph transformations for model refactoring. *Generative and Transformational Techniques in Software Engineering: International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*, pages 219–257, 2006.

[129] T. Mens, R. Van Der Straeten, and M. D'Hondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In *International Conference on Model Driven Engineering Languages and Systems*, pages 200–214. Springer, 2006.

[130] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 13–22. IEEE, 2005.

[131] P. Mohagheghi, W. Gilani, A. Stefanescu, and M. A. Fernandez. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. *Empirical software engineering*, 18(1):89–116, 2013.

[132] M. Monperrus. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 234–242. ACM, 2014.

[133] N. Nassar, H. Radke, and T. Arendt. Rule-based repair of emf models: An automated interactive approach. In *International Conference on Theory and Practice of Model Transformations*, pages 171–181. Springer, 2017.

[134] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 455–464. IEEE, 2003.

[135] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[136] B. Nuseibeh, S. Easterbrook, and A. Russo. Leveraging inconsistency in software development. *Computer*, 33(4):24–29, 2000.

[137] Object Management Group et al. Model driven architecture (mda)–mda guide version 1.0. 1, 2003.

[138] Object Management Group et al. Model driven architecture (mda)–mda guide revision 2.0, 2014.

[139] Object Management Group (OMG). Object Constraint Language, Version 2.4. OMG Document Number formal/14-02-03 (`https://www.omg.org/spec/OCL/2.4/`), 2014.

[140] O. M. G. (OMG). Meta object facility (version 2.5.1), 2016.

[141] O. M. G. (OMG). Unified modeling language (version 2.5.1), 2017.

[142] F. Orejas and E. Pino. Correctness of incremental model synchronization with triple graph grammars. In *International Conference on Theory and Practice of Model Transformations*, pages 74–90. Springer, 2014.

[143] R. F. Paige, J. S. Ostroff, and P. J. Brooke. Principles for modeling language design. *Information and Software Technology*, 42(10):665–675, 2000.

[144] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 609–620. IEEE, 2017.

[145] C. Pietsch, T. Kehrer, U. Kelter, D. Reuling, and M. Ohrndorf. Sipl–a delta-based modeling framework for software product line engineering. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 852–857. IEEE, 2015.

[146] J. P. Puissant, T. Mens, and R. Van Der Straeten. Resolving model inconsistencies with automated planning. In *LWI@ ASE*, pages 8–14, 2010.

[147] J. P. Puissant, R. Van Der Straeten, and T. Mens. Resolving model inconsistencies using automated regression planning. *Software & Systems Modeling*, 14(1):461–481, 2015.

[148] A. Reder and A. Egyed. Model/analyzer: a tool for detecting, visualizing and fixing design errors in uml. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 347–348. ACM, 2010.

[149] A. Reder and A. Egyed. Computing repair trees for resolving inconsistencies in design models. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 220–229. IEEE, 2012.

[150] A. Reder and A. Egyed. Incremental consistency checking for complex design rules and larger model changes. In *International Conference on Model Driven Engineering Languages and Systems*, pages 202–218. Springer, 2012.

[151] A. Reder and A. Egyed. Determining the cause of a design model inconsistency. *IEEE Transactions on Software Engineering*, 39(11):1531–1548, 2013.

[152] D. Reuling, C. Pietsch, U. Kelter, and M. Ohrndorf. Flexiple: A tool for flexible binding times in annotated model-based spls. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B*, pages 9–12, 2019.

[153] M. Rindt, T. Kehrer, and U. Kelter. Automatic generation of consistency-preserving edit operations for mde tools. In *MoDELS (Demos)*, 2014.

[154] M. Robillard, R. Walker, and T. Zimmermann. Recommendation systems for software engineering. *IEEE software*, 27(4):80–86, 2009.

[155] G. Rozenberg. *Handbook of graph grammars and computing by graph transformation*, volume 1. World scientific, 1997.

[156] D. Rubel, J. Wren, and E. Clayberg. *The Eclipse Graphical Editing Framework (GEF)*. Addison-Wesley Professional, 2011.

[157] M. Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *International Workshop on Theory and Application of Graph Transformations*, pages 238–251. Springer, 1998.

[158] H. Saada, X. Dolques, M. Huchard, C. Nebut, and H. Sahraoui. Generation of operational transformation rules from examples of model transformations. In *International Conference on Model Driven Engineering Languages and Systems*, pages 546–561. Springer, 2012.

[159] H. Saada, M. Huchard, M. Liquière, and C. Nebut. Learning model transformation patterns using graph generalization. In *CLA*, pages 11–22, 2014.

[160] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–27, 2006.

[161] M. Schmidt, S. Wenzel, T. Kehrer, and U. Kelter. History-based merging of models. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 13–18. IEEE Computer Society, 2009.

[162] S. Schneider and L. Lambers. Evaluation diversity for graph conditions. *Journal of Logical and Algebraic Methods in Programming*, 133:100862, 2023.

[163] S. Schneider, L. Lambers, and F. Orejas. A logic-based incremental approach to graph repair. In *International Conference on Fundamental Approaches to Software Engineering*, pages 151–167. Springer, 2019.

[164] S. Schneider, L. Lambers, and F. Orejas. A logic-based incremental approach to graph repair featuring delta preservation. *International Journal on Software Tools for Technology Transfer*, 23(3):369–410, 2021.

[165] A. Schultheiß, A. Boll, and T. Kehrer. Comparison of graph-based model transformation rules. *J. Object Technol.*, 19(2):3–1, 2020.

[166] A. Schürr. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1995.

[167] C. Seidl, I. Schaefer, and U. Aßmann. Deltaecore-a model-based delta language generation framework. *Modellierung 2014*, 2014.

[168] S. Sen, B. Baudry, and H. Vangheluwe. Towards domain-specific model editors with automatic model completion. *Simulation*, 86(2):109–126, 2010.

[169] G. Shani and A. Gunawardana. Evaluating recommendation systems. *Recommender systems handbook*, pages 257–297, 2011.

[170] R. Soley et al. Model driven architecture. *OMG white paper*, 308(308):5, 2000.

[171] H. Song, G. Huang, F. Chauvel, W. Zhang, Y. Sun, W. Shao, and H. Mei. Instant and incremental qvt transformation for runtime models. In *International Conference on Model Driven Engineering Languages and Systems*, pages 273–288. Springer, 2011.

[172] G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. In *Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals*, pages 329–380. World Scientific, 2001.

[173] M. Staron. Adopting model driven software development in industry–a case study at two companies. In *International Conference on Model Driven Engineering Languages and Systems*, pages 57–72. Springer, 2006.

[174] F. Steimann and B. Ulke. Generic model assist. In *Model-Driven Engineering Languages and Systems: 16th International Conference, MODELS 2013, Miami, FL, USA, September 29–October 4, 2013. Proceedings 16*, pages 18–34. Springer, 2013.

[175] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework.* Pearson Education, 2008.

[176] Y. Sun. Model transformation by demonstration. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 831–832, 2009.

[177] Y. Sun, J. Gray, and J. White. Mt-scribe: an end-user approach to automate software model evolution. In *Proceedings of the 33rd international conference on software engineering*, pages 980–982, 2011.

[178] G. Taentzer. Instance generation from type graphs with arbitrary multiplicities. *Electronic Communications of the EASST*, 47, 2012.

[179] G. Taentzer, T. Arendt, C. Ermel, and R. Heckel. Towards refactoring of rule-based, in-place model transformation systems. In *Proceedings of the First Workshop on the Analysis of Model Transformations*, pages 41–46, 2012.

[180] G. Taentzer, A. Crema, R. Schmutzler, and C. Ermel. Generating domain-specific model editors with complex editing commands. In *Applications of Graph Transformations with Industrial Relevance: Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers 3*, pages 98–103. Springer, 2008.

[181] M. Tichy, C. Krause, and G. Liebel. Detecting performance bad smells for henshin model transformations. *Amt@ models*, 1077, 2013.

[182] C. Tinnes, T. Kehrer, M. Joblin, U. Hohenstein, A. Biesdorf, and S. Apel. Learning domain-specific edit operations from model repositories with frequent subgraph mining. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 930–942. IEEE, 2021.

[183] C. Tinnes, T. Kehrer, M. Joblin, U. Hohenstein, A. Biesdorf, and S. Apel. Mining domain-specific edit operations from model repositories with applications to semantic lifting of model differences and change profiling. *Automated Software Engineering*, 30(2):17, 2023.

[184] R. Van Der Straeten and M. D'Hondt. Model refactorings through rule-based inconsistency resolution. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1210–1217. ACM, 2006.

[185] R. Van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between uml models. In *International Conference on the Unified Modeling Language*, pages 326–340. Springer, 2003.

[186] R. Van Der Straeten, J. P. Puissant, and T. Mens. Assessing the kodkod model finder for resolving model inconsistencies. In *European Conference on Modelling Foundations and Applications*, pages 69–84. Springer, 2011.

[187] D. Varró. Model transformation by example. In *Model Driven Engineering Languages and Systems: 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006. Proceedings 9*, pages 410–424. Springer, 2006.

[188] R. Wagner. A plug-in for flexible and incremental consistency management. In *Workshop Consistency Problems in UML-based Software Development, San Francisco, USA, Oct. 2003*, pages 78–85, 2003.

[189] J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise modeling with UML*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[190] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE Computer Society, 2009.

[191] J. Welsh, B. Broom, and D. Kiong. A design rationale for a language-based editor. *Software: Practice and Experience*, 21(9):923–948, 1991.

[192] S. Wenzel. *Unique identification of elements in evolving models: towards fine-grained traceability in model-driven engineering*. PhD thesis, University of Siegen, 2010.

[193] S. Wenzel. Unique identification of elements in evolving software models. *Software & Systems Modeling*, 13(2):679–711, 2014.

[194] M. Wimmer and G. Kramler. Bridging grammarware and modelware. In *International Conference on Model Driven Engineering Languages and Systems*, pages 159–168. Springer, 2005.

[195] N. Wirth. A brief history of software engineering. *IEEE Annals of the History of Computing*, 30(3):32–39, 2008.

[196] E. Wong, T. Wei, Y. Qi, and L. Zhao. A crosstab-based statistical method for effective fault localization. In *2008 1st international conference on software testing, verification, and validation*, pages 42–51. IEEE, 2008.

[197] W. E. Wong and V. Debroy. A survey of software fault localization. *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45*, 9, 2009.

[198] W. E. Wong, V. Debroy, and D. Xu. Towards better fault localization: A crosstab-based statistical approach. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(3):378–396, 2011.

[199] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei. Supporting automatic model inconsistency fixing. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 315–324. ACM, 2009.

[200] Y. Xiong, H. Song, Z. Hu, and M. Takeichi. Synchronizing concurrent model updates based on bidirectional transformation. *Software & Systems Modeling*, 12(1):89–104, 2013.

[201] A. Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[202] M. M. Zloof. Query-by-example: the invocation and definition of tables and forms. In *Proceedings of the 1st International Conference on Very Large Data Bases*, pages 1–24. ACM, 1975.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes über die Universität vom 5. September 1996 und Artikel 69 des Universitätsstatuts vom 7. Juni 2011 zum Entzug des Doktortitels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die Doktorarbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Bern, 14.07.2023