Behandlung von Inkonsistenzen in der verteilten Bearbeitung materialisierter Sichten im Kontext der modellbasierten Softwareentwicklung

Masterarbeit

Manuel Ohrndorf, B.Sc.

Universität Siegen

Siegen, 29. April 2017

Erstgutachter: **Prof. Dr. Udo Kelter**
Universität Siegen
Fakultät IV, Department Elektrotechnik und Informatik
Institut für Praktische und Technische Informatik
Lehrstuhl Softwaretechnik und Datenbanksysteme

Zweitgutachter: **Prof. Dr. Timo Kehrer**
Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik
Lehrstuhl für Modellgetriebene Softwareentwicklung

Detecting and Resolving Inconsistencies in a Distributed Multi-View Software Modeling Environment

Master Thesis

Manuel Ohrndorf, B.Sc.

Universität Siegen

Siegen, April 29, 2017

First reviewer: **Prof. Dr. Udo Kelter**
Universität Siegen
Fakultät IV, Department Elektrotechnik und Informatik
Institut für Praktische und Technische Informatik
Lehrstuhl Softwaretechnik und Datenbanksysteme

Second reviewer: **Prof. Dr. Timo Kehrer**
Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik
Lehrstuhl für Modellgetriebene Softwareentwicklung

Kurzfassung: In der Softwareentwicklung werden Modelle zur Analyse und zum Entwurf des zu implementierenden Systems eingesetzt. Die modellbasierte Softwareentwicklung macht Modelle zu zentralen Entwicklungsartefakten, welche die formale Grundlage für die Erzeugung der Implementierung bilden. Auf diese Weise wird zunächst von der technischen Zielplattform abstrahiert, was die Wiederverwendbarkeit des entwickelten Systems vereinfacht. Das erfordert allerdings die Festlegung einer formalen Syntax und Semantik für die Modellierungssprache. Außerdem können komplexe Systeme aus verschiedenen Sichten modelliert werden (z. B. Datenstrukturen und Benutzerinteraktion). Bei der verteilten Bearbeitung großer Modelle durch ein Entwicklerteam kann es darüber hinaus hilfreich sein, einen Modellausschnitt als temporäre materialisierte Sicht zu erzeugen, um die Komplexität des Modells zu reduzieren. Wenn ein Entwickler allerdings nicht alle Teile eines Modells überblickt, kann es leicht zu Inkonsistenzen zwischen verschiedenen Sichten kommen. Die Sicherstellung der Korrektheit eines Modells ist eine zentrale Aufgabe im Entwicklungsprozess, bei der die Entwickler durch entsprechende Werkzeuge unterstützt werden sollten. Diese Arbeit präsentiert einen Ansatz zur Reparatur von Inkonsistenzen in Modellen bzw. Modellsichten. Das hier vorgestellte Reparaturwerkzeug ermittelt Reparaturen auf Basis benutzerdefinierter Editierregeln. Insbesondere werden die durchgeführten Modifikationen betrachtet, um die Ursache einer Inkonsistenz besser zu verstehen.

Abstract: In software development, models are used for system analysis and design. In model-driven software development, models are primary artifacts and the formal basis for the system implementation. A model is an abstraction layer which conceals the technical target platform. This increases the re-usability of the developed components. However, this requires the definition of a formal syntax and semantics for the modeling language. Models are usually expressed from various viewpoints, e.g. the data structure and a user interaction. A multi-view modeling environment can be used to build complex system models. A viewpoint can also be constructed by slicing a specific part of the model. This can be helpful in distributing and processing large models across a development team. However, a developer can inadvertently introduce inconsistencies between different views. Ensuring the correctness of a model is a key task of the development process. Therefor, the developers need fully-fledged tool support to deal with inconsistencies. This work presents an approach to the repair of inconsistencies in a distributed multi-view modeling environment. The introduced tool determines repairs based on user-defined edit rules. To understand the cause of an inconsistency, the repair algorithm considers the modifications which have introduced the inconsistency.

Inhaltsverzeichnis

1 Motivation

In der Softwareentwicklung ist ein Modell eine Abstraktion des zu implementierenden Systems. Die Bildung von Modellen ist eine zentrale Aufgabe im Entwurfsprozess einer Software. Die Umsetzung eines komplexen Systems erfolgt üblicherweise über den Entwurf eines Modells, das die Rahmenbedingungen einer Software festlegt. Durch das Modell werden Anforderungen an die Software kommuniziert und letztendlich formalisiert. Während erste Entwürfe häufig noch als einfache Skizzen mit wenigen formalen Anforderungen erstellt werden, wird eine saubere und formale Dokumentation des Systems mit fortschreitenden Entwicklungsphasen immer wichtiger. In der Praxis kann es allerdings leicht passieren, dass Entwurf und Implementierung eines Systems, mit zunehmender Lebensdauer des Projekts, auseinander driften. Aus dieser Erkenntnis und dem Bestreben, Teile dieses Prozesses zu automatisieren, entstand die modellbasierte Softwareentwicklung (*model-driven software development*) [MM03].

Konzepte der modellbasierten Softwareentwicklung finden mittlerweile Anwendung in vielen verschiedenen Domänen (z. B. im Fahrzeugbau, Bankwesen, Druckersystemen, Web-Anwendungen usw.) [WHR14]. Modelle dienen hier nicht nur als Dokumentation des Systems, sie bilden darüber hinaus die formale Grundlage für die Erzeugung der Implementierung. Die Modelle müssen daher mit derselben Sorgfalt behandelt werden, wie der implementierte Quellcode. Das erfordert zwar zunächst zusätzlichen Aufwand von den Entwicklern, da die Modelle mit formalen Methoden erstellt und gepflegt werden müssen, dies kann sich aber in den nachfolgenden Entwicklungsphasen und bei der Weiterentwicklung des Systems wieder auszahlen. Das Konzept der modellbasierten Softwareentwicklung soll die Basis für qualitativ hochwertige und langlebige Softwareprojekte schaffen. Insbesondere bei großen Softwarefamilien und heterogenen Zielplattformen helfen plattformunabhängige Modelle, die Wiederverwendbarkeit der entwickelten Funktionalitäten zu steigern [PTN+07, GPR07].

In der modellbasierten Softwareentwicklung werden hohe Anforderungen an die Qualität eines Modells gestellt. Die Modelle werden hier zu zentralen Entwicklungsartefakten, die eine formal definierte Syntax und Semantik besitzen. Die Entwickler benötigen daher entsprechende Werkzeugunterstützung, um komplexe Systeme zu modellieren, um Mo-

delle im Team zu entwickeln und um Fehler in Modellen aufzudecken und zu behandeln. Um komplexe Systeme zu modellieren, werden Modelle häufig in verschiedene Sichten bzw. Perspektiven unterteilt. Eine Sicht kann beispielsweise die Datenstruktur und eine andere eine mögliche Benutzerinteraktion des Systems beschreiben. Wenn derselbe Aspekt eines Systems aus zwei Perspektiven beschrieben wird, dann kommt es meist zu inhaltlichen Überlappungen. Beispielsweise kann eine Sicht Operationen definieren, die von einer anderen Sicht aufgerufen werden. Daher muss, neben der syntaktischen Korrektheit einer Sicht, die Konsistenz überlappender Sichten sichergestellt werden. Hieraus ergeben sich entsprechend komplexe Editiervorschriften, welche ein Entwickler bei der Modellierung beachten muss. Die Erkennung und Behandlung von Inkonsistenzen, die durch eine Verletzung dieser Vorschriften entstehen, ist eine zentrale Aufgabe in der modellbasierten Softwareentwicklung. Die Entwickler sollten daher bei der Modellierung mit entsprechenden Werkzeugen unterstützt werden.

Große Softwareprojekte können Modelle mit tausenden von Modellelementen enthalten [Egy07]. Die einzelnen Modelle können untereinander wiederum Netzwerke von Modellen bilden. Für die Umsetzung einer bestimmten Aufgabe benötigt ein Entwickler allerdings häufig nur einen Ausschnitt des Modells. Das gesamte Modell auf einmal zu betrachten kann sich sowohl nachteilig auf die effektive Verarbeitung als auch auf die Verständlichkeit des Modells auswirken. Eine Technik, mit der die Komplexität eines Modells temporär reduziert werden kann, ist das sogenannte Slicing (*model slicing*) [ACH+13], bei dem ein Teil des Modells ausgeschnitten wird. Alle erforderlichen Modifikationen werden nun auf dem Slice durchgeführt. Das ermöglicht es dem Entwickler, sich auf einen Ausschnitt der Modellierung zu konzentrieren, ohne immer das gesamte Systemmodell im Blick haben zu müssen.

Die isolierte Bearbeitung eines Modellausschnitts kann allerdings auch zu Inkonsistenzen in den restlichen Teilen des Modells führen. Ein Slice kann in diesem Zusammenhang als materialisierte Sicht auf ein Systemmodell verstanden werden. Der Modellausschnitt wird als persistente Version gespeichert und wird dann durch den Entwickler zu einer neuen Version modifiziert. Entstehen Inkonsistenzen zwischen dem Slice bzw. der materialisierten Sicht und überlappenden Teilen des Modells, so können diese analog zu Inkonsistenzen in den vordefinierten Sichten einer Modellierungssprache behandelt werden. Die Inkonsistenzen entstehen allerdings erst, wenn die auf der materialisierten Sicht durchgeführten Modifikationen wieder in das Ursprungsmodell reintegriert werden. Zu diesem Zeitpunkt sollte der Entwickler auf die entstandenen Probleme hingewiesen werden und entsprechende Werkzeugunterstützung erhalten, um die Inkonsistenzen zu beheben.

Aufbau dieser Arbeit: Das Ziel dieser Arbeit ist die Entwicklung eines Verfahrens zur Reparatur von Inkonsistenzen in Modellen. Zu Beginn werden wir in *Kapitel 2* die hierfür relevanten Grundlagen besprechen und eine entsprechende Terminologie vereinbaren. Darauf folgt die Vorstellung des Konzepts. In *Kapitel 3 und 4* wird ein Algorithmus zur Erkennung unvollständig durchgeführter Editierschritte auf Basis von benutzerdefinierten Editierregeln hergeleitet. Dieser Algorithmus bildet die Basis für die Berechnung der Reparaturen. Hierfür wird der Algorithmus in *Kapitel 5* mit einer syntaktischen Analyse der Inkonsistenzen kombiniert. Somit können unvollständige Editierschritte bestimmt werden, welche ggf. mit der Entstehung der Inkonsistenz zusammenhängen. In einem nachfolgenden Berechnungsschritt werden dann mögliche Ergänzungen dieser Editierschritte bestimmt, um herauszufinden, ob so eine Reparatur der Inkonsistenzen erreicht werden kann. Dem Entwickler wird daraufhin eine Liste möglicher Reparaturen vorgeschlagen, welche mit Hilfe des entwickelten Werkzeugs inspiziert und angewandt werden können.

Neben syntaktischen Fehlern innerhalb eines Modells können durch das Verfahren insbesondere Inkonsistenzen zwischen verschiedenen Modellsichten repariert werden. In *Kapitel 6* wird das Reparaturverfahren im Kontext der Reintegration von Modellausschnitten eingesetzt. Wie bereits festgestellt wurde, ist ein Slice eine materialisierte Sicht des Ursprungsmodells. Inkonsistenzen zwischen dem Ursprungsmodell und dem Slice können analog zu Inkonsistenzen zwischen zwei Sichten behandelt werden. Hierfür wird das Reparaturverfahren in einen Slicing- und Reintegrationsprozess eingebunden. Treten bei der Reintegration eines modifizierten Modellausschnitts Inkonsistenzen auf, so wird ausgehend von den durchgeführten Modifikationen nach möglichen Reparaturen gesucht. Der Entwickler wird dann bei der Reintegration durch ein interaktives Werkzeug unterstützt, um die Modifikationen zu überprüfen und ggf. anzupassen. In dieser Hinsicht präsentiert diese Arbeit einen Ansatz zur Reparatur, Synchronisation und Reintegration von Modellsichten unter Beachtung der durchgeführten Modifikationen.

In *Kapitel 7* wird die Benutzerschnittstelle und Evaluierung des implementierten Reparaturwerkzeugs vorgestellt. Die Arbeit schließt in *Kapitel 8* mit einer Zusammenfassung der gewonnenen Erkenntnisse und einem Ausblick auf zukünftige Arbeiten.

2 Grundlagen

Abbildung 2.1 zeigt den Entwurf eines Video-on-Demand-Systems (VoD) [Egy06, KK99]. Die hier erstellten Modelle sind als Diagramme der Unified Modeling Language (UML) [OMG15] dargestellt. Die UML beinhaltet eine Reihe standardisierter Notationen, um ein System aus verschiedenen Sichten zu modellieren. Die Diagramme lassen sich in Struktur- und Verhaltensdiagramme unterteilen [Kec09]. Ein Strukturdiagramm zeigt die statische Sicht eins Systems, wie beispielsweise Komponenten oder Datenstrukturen. Verhaltensdiagramme modellieren dynamische Prozesse oder Abläufe, die innerhalb des Systems auftreten können. Das VoD-System in Abbildung 2.1 wird durch ein Klassen-, Sequenz- und Zustandsdiagramm modelliert. Das Klassendiagramm zeigt die statische Datenstruktur und durchführbare Operationen. Ein Benutzer (`User`) kann maximal `4` Videos gleichzeitig von einem Server abrufen. Ein Video lässt sich über die angegebenen Operationen starten und stoppen. Die Operationen lösen auch die entsprechenden Zustandsübergänge (Transitionen) im Zustandsdiagramm der Video-Klasse aus. Das Sequenzdiagramm zeigt, wie die Interaktionen zwischen dem Benutzer und den beteiligten Komponenten abläuft. Der Benutzer starten zunächst das Video. Daraufhin wird, über die angegebene Adresse, eine Verbindung zum Server aufgebaut. Zu einem späteren Zeitpunkt beendet der Benutzer das Video und trennt die Verbindung anschließend wieder.

2.1 Konsistenz von Modellen

Nach dem Erstentwurf werden einige Verbesserungen an der Modellierung vorgenommen (Abbildung 2.2). Zunächst wird die Operation `disconnect()` aus der Klasse `Video` in die Klasse `Server` verschoben. Anschließend wird die Transition `stop` als Zustandsübergang zum Endzustand festgelegt. Die Transition wird konzeptionell durch die Transition `pause` ersetzt. In einem letzten Schritt wird dann noch der Zustand `offline` entfernt. Wie sich bereits intuitiv erkennen lässt, hängen die einzelnen Diagramme inhaltlich zusammen. Eine Veränderung im Klassendiagramm zieht beispielsweise auch Änderungen im Sequenzdiagramm nach sich. Werden diese Änderungen nicht vollzogen, so sprechen

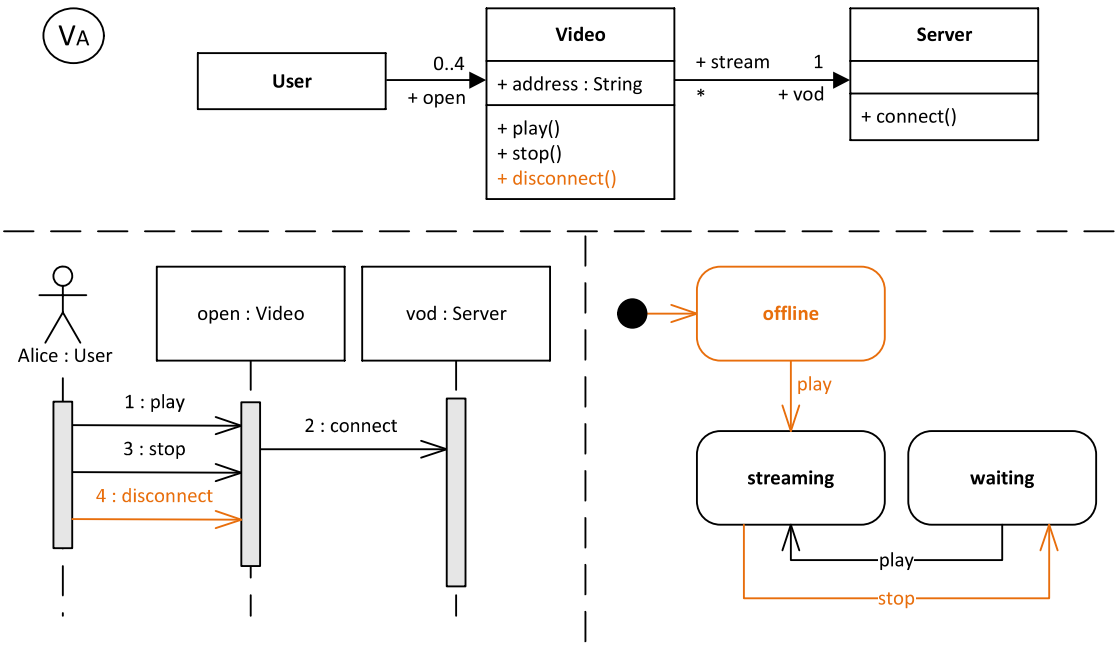


Abbildung 2.1: Entwurf eines VoD-Systems - Version A

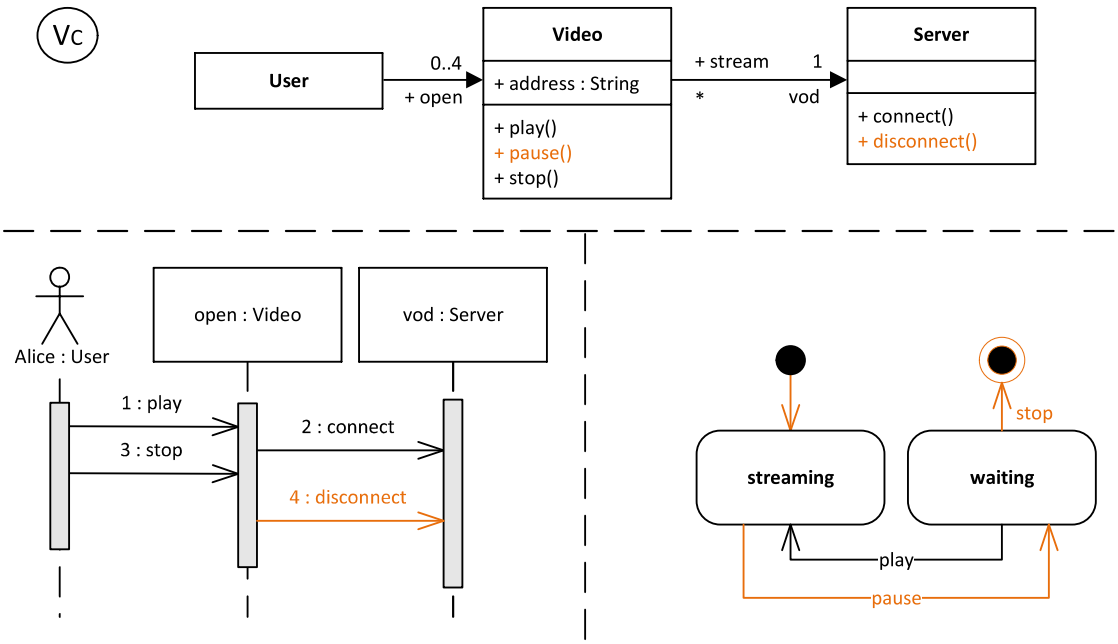


Abbildung 2.2: Entwurf eines VoD-Systems - Version C

wir von einer **Inkonsistenz**, die zwischen den Diagrammen entstanden ist. Die verschiedenen Sichten treffen unterschiedliche Aussagen über das selbe System [SZ01].

Die Änderungen betreffen zunächst die Nachricht `4:disconnect`, die im Sequenzdiagramm zwischen dem Benutzer und dem Video gesendet wird. Das Senden der Nachricht modelliert hier den Aufruf der gleichnamigen Funktion im Ziel-Objekt. Da die Operation `disconnect()` verschoben wurde, muss zunächst das Ziel der Nachricht auf die Lebenslinie des Server-Objekts gesetzt werden. Außerdem muss der Sender einer Nachricht den Empfänger kennen, d.h. es muss eine Assoziation zwischen den Klassen existieren, welche Sender und Empfänger modellieren. Laut Klassendiagramm kommen hierfür nur Objekte der Klasse Video infrage. Daher wird das Senden der Nachricht `4:disconnect` der Lebenslinie des Video-Objekts zugeordnet. Inhaltlich liest sich das Sequenzdiagramm nun so, dass ein stoppen des Videos dazu führt, dass die Verbindung zum Server getrennt wird.

Um das Video anzuhalten, ohne die Verbindung zum Server zu trennen, wurde im Zustandsdiagramm die Transition `stop` durch `pause` ersetzt. Die Anpassung des Modells ist allerdings noch unvollständig. Für die Transition muss noch ein entsprechender Trigger festgelegt werden. Da das Zustandsdiagramm das Verhalten der Video-Klasse beschreibt, wird hier eine entsprechende Operation `pause()` eingefügt und der Trigger der Transition auf die Operation gesetzt. Nachdem alle Modifikationen vervollständigt und auf die Sichten propagiert wurden, befindet sich die Modellierung des VoD-Systems wieder in einem konsistenten Zustand.

Das Beispiel zeigt, wie Inkonsistenzen zwischen verschiedenen Sichten eines Modells auftreten können. Die Modelle befinden sich hier auf der selben konzeptionellen Abstraktionsebene. In nachfolgenden Entwicklungsphasen werden die Modelle auf eine bestimmte technische Zielplattform übertragen. Ein Modell wird dazu mit den entsprechenden technischen Details angereichert oder sogar in eine andere plattformspezifische Sprache transformiert [MM03]. Engels et al. [EKHG01] unterteilt die Konsistenz der Modelle auf Basis der dadurch entstehenden Abstraktionsebenen. Die Konsistenz von Modellen auf der selben Abstraktionsebene wird als **horizontale** bzw. **Intra-Modell-Konsistenz** (*intra-model consistency*) bezeichnet [BLK+16]. Wie im Beispiel müssen hier Teilmodelle oder Sichten der Systemmodellierung konsistent gehalten werden. Die Konsistenz zwischen Modellen, die auf unterschiedlichen Abstraktionsebenen liegen, wird hingegen als **vertikale** bzw. **Inter-Modell-Konsistenz** (*inter-model consistency*) bezeichnet [BLK+16]. Eine Inkonsistenz kann hier z. B. entstehen, wenn die Namensänderung einer Klasse nicht vom plattformunabhängigen auf das entsprechende plattformspezifische Modell übertragen wird.

Sowohl für Intra- als auch für Inter-Modell-Konsistenz unterscheidet Engels et al. [EKHG01] zusätzlich zwischen syntaktischer und semantischer Konsistenz. Die **syntaktische Konsistenz** eine Modells wird durch sein Metamodell und zusätzlich Konsistenzregeln festgelegt. Das **Metamodell** beschreibt die grundlegenden Strukturen, aus denen eine Modellinstanz aufgebaut werden kann. Komplexere Konsistenzbedingungen werden durch zusätzliche **Konsistenzregeln** mit Hilfe einer **Constraintsprache** formuliert. Im Rahmen der UML wurde hierfür die Object Constraint Language (OCL) [OMG14] definiert. Die Überprüfung eines Modells durch Konsistenzregeln wird als **Validierung** bezeichnet. Jede gescheiterte Validierung einer Konsistenzregel wird dem Entwickler als Inkonsistenz berichtet. Eine strengere Form der Konsistenz ist die **semantische Konsistenz**, d.h. ein semantisch konsistentes Modell ist auch syntaktisch Konsistent. Die automatische Überprüfung semantischer Konsistenz ist schwierig, da es sich hierbei meist um inhaltliche oder auf das Verhalten bezogene Bedingungen handelt, die formell bzw. statisch nicht beschrieben werden können.

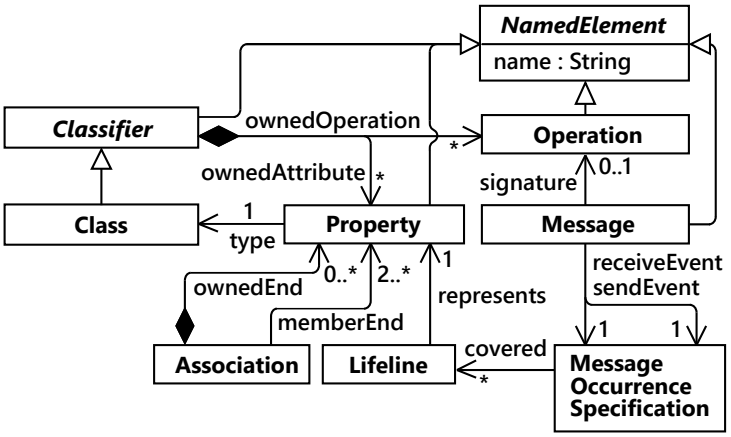


Abbildung 2.3: Ausschnitt des UML-Metamodells [TOLR17]

Abbildung 2.3 zeigt ein vereinfachtes Metamodell für das in Abbildung 2.1 vorgestellte Klassen- und Sequenzdiagramm. Neben den Metaklassen, Attributen und Referenzen definiert das Metamodell auch die Kompositionsstruktur für den abstrakten Syntaxbaum (*abstract syntax tree, AST*) eines Modells. Der Syntaxbaum wird auch als **abstrakte Syntax** des Modells bezeichnet. Berechnungen und automatische Verarbeitungen finden i.d.R. auf der abstrakten Syntax der Modellierungssprache statt. Auf Basis der abstrakten Syntax bzw. des Metamodells lassen sich auch die syntaktischen Verbindungen zwischen den einzelnen Diagrammen erkennen. Beispielsweise kann eine Nachricht im Sequenzdiagramm explizit eine Operation als Signatur (`signature`) referenzieren.

Zum Entwurf eines Modells verwendet der Entwickler i.d.R. die **konkrete Syntax**, in Form von Diagrammen oder textuellen Darstellungen.

Im Kontext dieser Arbeit wird nur die syntaktische Konsistenz auf der horizontalen Abstraktionsebene (Intra-Modell-Konsistenz) betrachtet. Ein Modell wird als Konsistent bezeichnet, wenn alle festgelegten Konsistenzregeln eingehalten werden. Die Konsistenzregeln werden auf Basis der abstrakten Syntax formuliert. Als Constraintsprache wird eine auf Prädikatenlogik erster Stufe basierende Notation verwendet. Jede Konsistenzregel besitzt ein **Kontextelement**, welches den Startpunkt der Validierung darstellt.

Konsistenzregel 2.1 beschreibt die Validierung einer Nachricht (`Message`) eines Sequenzdiagramms. Die Konsistenzregel legt fest, dass der Empfänger einer Nachricht eine gleichnamige Operation besitzen muss. Zunächst wird überprüft, ob die Nachricht eine Operation als Signatur festlegt. Diese Operation muss den selben Namen tragen, wie die Nachricht. Im zweiten Teil der Konsistenzregel werden alle Lebenslinien (`Lifeline`) betrachtet, welche die Nachricht empfangen. Jede Lebenslinie repräsentiert ein Objekt bzw. ein Assoziationsende (`Property`), das einen bestimmten Typ besitzt. Der Typ wird durch eine Klasse definiert, welche wiederum die als Signatur angegeben Operation besitzen muss. Im Beispiel in Abbildung 2.2 wird diese Konsistenzregel zwischenzeitlich verletzt, nachdem die Operation `disconnect()` im Klassendiagramm verschoben wurde. Erst durch die Korrektur des Empfängers der Nachricht `4:disconnect`, wird auch der zweite Teil der Konsistenzregel wieder erfüllt.

Konsistenzregel 2.1

$$
\begin{aligned}
&\textit{Message } m : \\
&\quad (\exists\ s \in m.signature : s.name = m.name) \\
&\qquad\qquad \wedge \\
&\quad (\forall\ l \in m.receiveEvent.covered : \\
&\quad\ \exists\ o \in l.represents.type.ownedOperation : \\
&\qquad o = m.signature)
\end{aligned}
$$

2.2 Reparaturwerkzeuge

Die Identifikation, Diagnose, Dokumentation und Reparatur von Inkonsistenzen gehören zu den grundlegenden Tätigkeiten in der Softwaremodellierung. Die Tätigkeiten im Rahmen der Behandlung von Inkonsistenzen werden als **Konsistenzverwaltung** (*consistency management*) bezeichnet [LMT09]. Ausgehend von einem konsistenten Modell wird eine Inkonsistenz durch eine Menge von Änderungen ausgelöst, die nicht wieder zu

einem konsistenten Zustand des Modells führen. Eine solche Veränderung wird im Folgenden als **inkonsistenter Editierschritt** bezeichnet. Eine darauf folgende Modifikation, welche die Konsistenz des Modells wieder herstellt oder die Inkonsistenz zumindest teilweise ausbessert, wird als (vollständige) **Reparatur** bezeichnet. Die Reparatur einer Inkonsistenz kann in den meisten Fällen nicht eindeutig bestimmt werden. Unter Umständen gibt es sogar unendlich viele Möglichkeiten eine Inkonsistenz zu beheben. Zwei Klassen eines Klassendiagramms dürfen beispielsweise nicht denselben Namen tragen. Es gibt aber praktisch unendlich viele Möglichkeiten eine Klasse umzubenennen. Alternativ könnte man auch eine der beiden Klassen löschen oder in einen anderen Namensraum verschieben.

Nachdem eine Inkonsistenz erkannt wurde, muss der Entwickler deren Ursprung diagnostizieren und eine geeignete Reparatur entwickeln. Entsprechende Werkzeuge können den Entwickler hierbei unterstützen. Ein **Reparaturwerkzeug** generiert eine Menge von Reparaturvorschlägen zur Behandlung einer bzw. mehrerer Inkonsistenzen. Die meisten Reparaturwerkzeuge erstellen dann ein Ranking der Reparaturvorschläge, aus denen der Entwickler eine passende Reparatur auswählen kann. Um eine zu große Anzahl von Reparaturen zu vermeiden, werden ggf. einzelne Änderungen der Reparaturen nicht initialisiert. Eine Reparatur kann beispielsweise festlegen, dass eine Klasse umbenannt werden muss, der konkrete Name der Klasse wird allerdings nicht vorgegeben. Eine nicht initialisierte Reparatur wird als **abstrakte Reparatur** bezeichnet (Definition 2.1). Nach der Initialisierung wird diese dann zu einer **konkreten Reparatur**. Ein **Reparaturplan** besteht aus mehreren reparierenden Änderungen und kann sowohl konkrete als auch abstrakte Reparaturen beinhalten.

Definition 2.1
Eine **abstrakte Reparatur** (*repair action*, [RE12a]) ist eine Tripel $\tau = \langle \lambda, \omega, \gamma \rangle$, welches das zu reparierende Modellelement ω und dessen Eigenschaft γ (Attribut oder Referenz) angibt. Eine abstrakte Reparatur legt nur die Art der Änderung λ fest, welche konkrete Änderung durchzuführen ist wird zunächst noch offen gelassen. Unter der angegebenen Eigenschaft γ des Modellelements ω kann entweder eine Einfügung, Löschung oder eine Modifikation $\lambda \in \{create, delete, modify\}$ stattfinden[a] [RE12a].

[a] Im Originaltext $\lambda \in \{add, delete, modify\}$

Werden mehrere Inkonsistenzen durch eine Reparatur behoben, wird dies als **positiver Seiteneffekt** bezeichnet. Es können auch **negative Seiteneffekte** auftreten, wenn eine Reparatur neue Inkonsistenzen auslöst. Negative Seiteneffekte lassen sich allerdings

nicht immer vermeiden. In solchen Fällen muss die Reparatur als ein inkrementeller Prozess betrachtet werden, in dem Inkonsistenzen schrittweise behoben werden müssen [MVDSD06, MVDS06].

Macedo et al. [MJC16] klassifizieren die existierenden Reparaturalgorithmen im Wesentlichen in syntax-, such- und regelbasierte Ansätze. Ein **syntaxbasiertes** Verfahren generiert Reparaturen durch eine Analyse der Konsistenzregeln und der fehlgeschlagenen Validierungen. Nentwich et al. [NEF03] beschreiben ein Verfahren, bei dem zu jeder Konsistenzregel eine Menge von abstrakten Reparaturen generiert wird. Die Reparaturen werden statisch generiert und ggf. vorgeschlagen, wenn die Validierung der entsprechenden Konsistenzregel fehlschlägt. Reder und Egyed [RE12a] beschreiben eine Weiterentwicklung dieses Verfahren, bei dem zusätzlich die konkrete Auswertung der Konsistenzregel auf dem Modell berücksichtigt wird. Dadurch lassen sich unnötige Reparaturen ausschließen. Die abstrakten Reparaturen geben dann alle Ansatzpunkte für konkrete Reparaturen innerhalb des Modells vor.

Ein **suchbasierter** Reparaturalgorithmus startet mit dem inkonsistenten Zustand eines Modells und versucht eine Folge von Zustandsübergängen zu finden, die zu einem konsistenten Zustand führt. Diese Technik wird daher auch als **Zustandsraumsuche** (*state-space exploration, state-space planning*) bezeichnet. Einige Reparaturalgorithmen führen die Zustandsübergänge mit benutzerdefinierten Editierregeln aus [MGC13, HHR+11]. Bei anderen Ansätzen werden generische Modifikationen auf Basis der abstrakten Syntax des Modells durchgeführt [PVDSM15]. Ein grundsätzliches Problem dieser Ansätze ist die effiziente Durchsuchung des Zustandsraums. Der Zustandsraum ist meist zu groß, um diesen vollständig zu betrachten. Daher müssen entsprechende Heuristiken entwickelt werden, um die konsistenten Zustande schneller zu finden.

Bei **regelbasierten** Ansätzen wird das Reparaturwerkzeug mit einer Reihe von Lösungsstrategien für häufig auftretende Inkonsistenzen vorkonfiguriert [VDSD06]. Tritt eine Inkonsistenz auf, dann kann der Entwickler eine der Lösungsstrategien als Reparatur anwenden. Dieser Ansatz erlaubt es spezielle Reparaturen vorzugeben. Allerdings wird dadurch sehr viel Komplexität auf die Konfiguration des Werkzeugs verlagert.

Das in dieser Arbeit beschriebene Reparaturwerkzeug leitet die Reparaturen aus benutzerdefinierten Editierregeln ab. Es werden daher keine direkten Lösungsstrategien für Inkonsistenzen vorgegeben. Die Editierregeln legen fest, wie ein Modell korrekt Bearbeitet werden kann. Taentzer et al. [TOLR17] beschrieben einen ähnlichen Ansatz, der auf konsistenzerhaltenden Editierregeln basiert. Eine Inkonsistenz tritt auf, wenn ein durchgeführter Editierschritt durch keine der konsistenzerhaltenden Editierregeln beschrieben werden kann. Für die Berechnung der Reparaturen sind zusätzliche Edi-

tierregeln anzugeben, welche die inkonsistenten Editierschritte beschreiben. Kann die inkonsistente Ausführung einer Editierregel durch eine konsistenzerhaltende Editierregel ergänzt werden, dann wird diese Ergänzung als Reparatur ausgegeben. Im Rahmen dieser Arbeit werden hingegen Konsistenzregeln verwendet, um Inkonsistenzen zu erkennen. Außerdem müssen keine zusätzlichen Editierregeln angegeben werden, welche die inkonsistenten Editierschritte beschreiben.

2.3 Editierregeln und Graphtransformationen

Ein Metamodell inklusive der zugehörigen Konsistenzregeln beschreibt alle syntaktisch korrekten Modellinstanzen. In diesem Abschnitt werden wir definieren, wie ein korrekter Übergang zwischen zwei Versionen einer Modellinstanzen beschrieben werden kann. Eine abgeschlossene Änderung, beispielsweise das Einfügen und Benennen eines neuen Modellelements im Diagrammeditor, wird als **Editierschritt** bezeichnet. Ein sinnvoller bzw. syntaktisch korrekter Editierschritt wird durch eine **Editierregel** definiert.

Komplexe Editierregeln, wie sie beispielsweise Diagrammeditoren benötigen, werden allerdings meist nur in sogenannte Kommandos (*command*) codiert ([SBMP08]). Ein Kommando ist im Wesentlichen eine Blackbox, bei der nur das Resultat beobachten werden kann, z. B. um das Kommando rückgängig (*undo*) zu machen. Es ist allerdings nicht direkt zu erkennen, wie sich das Kommando unter verschieden Randbedingungen verhält. Um die korrekten Editierschritte bzgl. eines Metamodells zu verstehen, wird ein deklaratives Konzept zur Definition von Editierregeln benötigt.

Graphtransformationsregeln bieten eine formale Grundlage, um Editierregeln für eine Modellierungssprache zu formulieren. Eine **Graphtransformationsregel** (bzw. ein Graphersetzungssystem) besteht formal aus zwei Mustergraphen und einem Klebegraphen, der diese beiden verbindet. Ein **Mustergraph** ist typisiert und besteht aus attributierten Knoten und gerichteten Kanten. Eine Graphtransformationsregel wird auf den sogenannten **Arbeitsgraphen** angewendet. Der Arbeitsgraph enthält in diesem Zusammenhang das zu editierende Modell. Die Anwendung einer Graphtransformationsregel erfolgt, indem zunächst eine Abbildung des linken Mustergraphen (*left-hand side, LHS*) auf den Arbeitsgraphen gesucht wird. Für eine korrekte **Abbildung** (*match*) müssen den Knoten des Mustergraphen passende Modellelemente zugeordnet werden, d.h. der angegebene Typ muss übereinstimmen und vorgegebene Kanten bzw. Referenzen und Attributwerte müssen vorhanden sein. Die Abbildung wird dann durch den rechten Mustergraphen (*right-hand side, RHS*), unter Beachtung des Klebegraphen, ersetzt. Alle Modellelemente bzw. Referenzen, die auf Knoten bzw. Kanten $P = LHS \cap RHS$

abgebildet wurden, welche auf beiden Seiten der Regel vorkommen, bleiben unverändert. Alle verbleibenden Knoten bzw. Kanten $D = LHS \setminus RHS$ der linken Seite führen zum Löschen der zugeordneten Modellelemente und Referenzen. Alle Knoten und Kanten $C = RHS \setminus LHS$, die nur auf der rechten Seite des Mustergraphen vorkommen, müssen neu erzeugt werden. Abschließend müssen noch alle Attributwerte $S = \{Attribut = Wert, \ldots\}$ gesetzt werden, die links- und rechtsseitig abweichend angegeben sind.

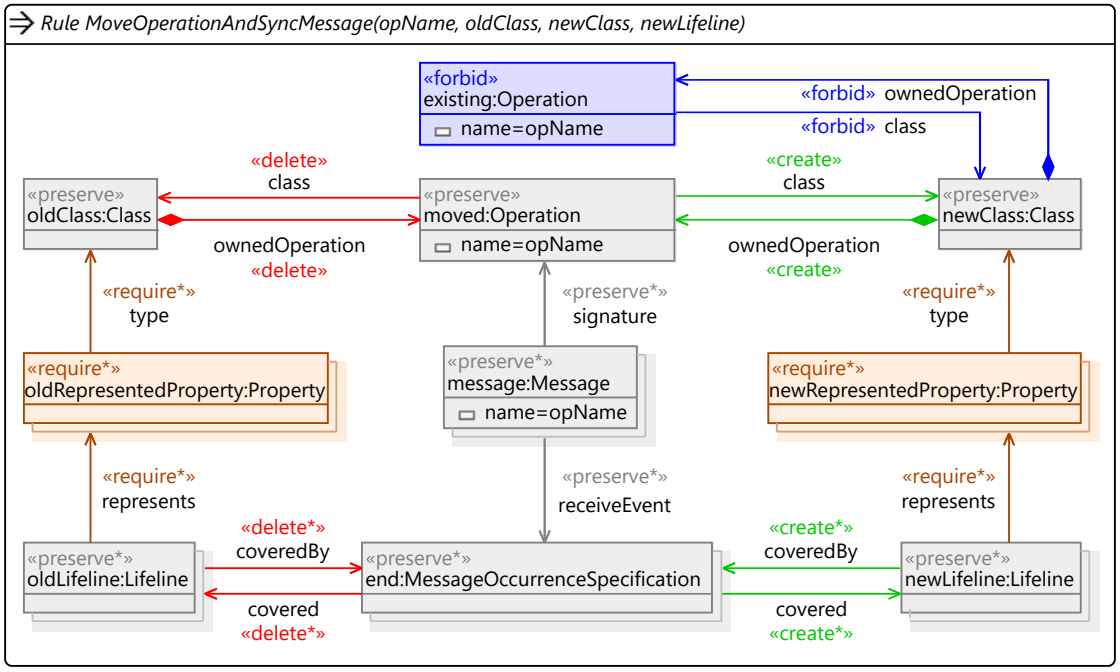


Abbildung 2.4: Editierregel - Verschieben einer Operation

Abbildung 2.4 zeigt eine Editierregel, die ein Klassendiagramm mit einem Sequenzdiagramm synchronisiert. Nachdem eine Operation in die Klasse *newClass* verschoben wurde, werden die auf der Operation basierenden Nachrichten angepasst. Falls die Klasse *newClass* eine Lebenslinie im Sequenzdiagramm besitzt, kann diese als Empfänger der Nachrichten verwendet werden. Die Editierregel entspricht dem Verschieben der Operation `disconnect()` und dem Anpassen des Empfängers der Nachricht `disconnect` in Abbildung 2.2.

Das Diagramm in Abbildung 2.4 zeigt die Vereinigung der linken und rechten Seite der Graphtransformationsregel [ABJ+10]. Diese Notation erlaubt eine intuitive Darstellung der durchzuführenden Transformation. Aus der Graphtransformationsregel ergeben sich bestimmte **Aktionen** (*action*) für alle Kante und Knoten, die als Stereotypen an den entsprechenden Elementen notiert werden. Die integrierte Darstellung der Mustergra-

phen wird daher im Folgenden als **Aktionsgraph** bezeichnet. Alle Knoten und Kanten ($P = LHS \cap RHS$), die mit der Aktion «preserve» markiert wurden, geben vor, wie die Änderungen zusammenhängen und werden daher als **Kontextgraph** bezeichnet. Alle zu löschenden ($D = LHS \setminus RHS$) bzw. zu erzeugenden ($C = RHS \setminus LHS$) Knoten und Kanten werden mit der Aktion «delete» bzw. «create» gekennzeichnet. Für die Editierregeln wird außerdem gefordert, dass der Kontextgraph und alle Aktionen einen zusammenhängenden Graphen bilden.

Einer Graphtransformationsregel können zusätzliche Mustergraphen als positive (*positive application condition, PAC*) und negative (*negative application condition, NAC*) **Anwendungsbedingungen** ($AC = PAC \cup NAC$) hinzugefügt werden. Für jede positive Anwendungsbedingung («require») muss eine Abbildung in den Arbeitsgraphen existieren. Wird hingegen eine Abbildung für eine negative Anwendungsbedingung gefunden («forbid»), dann ist eine Anwendung der Regel nicht möglich. Außerdem lassen sich die einzelnen Anwendungsbedingungen über logische Formeln (*and, or* usw.) miteinander verknüpfen. Eine Anwendungsbedingung kann beispielsweise den Kontext für eine Aktion einschränken.

Einige der Knoten in Abbildung 2.4 sind durch einen angedeuteten Stapel dargestellt. Diese Knoten sind Teil einer sogenannten Multiregel, d.h. dieser Teil der Regel wird ggf. mehrfach ausgeführt. Zunächst wird nach einer Abbildung der angrenzenden Kernregel gesucht, in welche die Multiregel eingebettet wurde. Danach werden alle Abbildungen der Multiregel gesucht, die sich in die Abbildung der Kernregel einbetten lassen. Die Transformation der Multiregel wird schließlich für alle gefunden Abbildungen ausgeführt. Dieses Konzept wird in der Graphtransformation als **Amalgamierung** der Transformationsregel bezeichnet[BEE+10]. In der Editierregel in Abbildung 2.4 werden dadurch alle Nachrichten (Multiregel), die sich auf die verschobene Operation beziehen (Kernregel), entsprechend synchronisiert.

Graphtransformationsregeln bieten einen formalen Ansatz, um komplexe Editierregeln einer Modellierungssprache zu beschreiben. Ein Aktionsgraph beschreibt deklarativ, wie einzelne Aktionen einer Editierregel zusammenhängen. Vereinfacht kann eine Editierregel auch als Menge der enthaltenen Aktionen betrachtet werden.

Definition 2.2
Eine **Editierregel** δ_x beschreibt einen komplexen bzw. syntaktisch korrekten Zustandsübergang zwischen zwei Modellinstanzen. Die Editierregel $\delta_x = \{\alpha_0, \ldots, \alpha_n\}$ gibt das Muster für die durchzuführenden Änderungen vor. Eine **Aktion** $\alpha_i = (\lambda, \kappa)$

des Typs $\lambda \in \{create, delete\}$ bezieht sich auf einen Knoten, eine Kante oder ein Attribut[a] κ des Aktionsgraphen. Die Menge der Aktionen beschreibt den Effekt der Editierregel. Eine parametrisierte Editierregel $\delta_x(P_1, \ldots, P_m)$ bestimmt den Kontext bzw. die Werte der Aktionen auf Basis der Eingabeparameter P_i. Die Eingabeparameter werden als partielle Vorgabe für die Abbildung der Graphtransformationsregel gesetzt.

[a] Aktuell werden nur Änderungen von Attributwerten in Kontextknoten betrachtet. Die Initialisierung eines Attributs wird durch einen Editierschritt nicht wiedergegeben (Definition 2.3).

2.4 Modelldifferenzen

Modelle werden im Laufe eines Entwicklungsprozesses immer wieder angepasst und weiterentwickelt. Um die Modifikationen zu verwalten und nachzuvollziehen, werden entsprechende Versionsverwaltungswerkzeuge benötigt. Einen persistenten Zustand eines Modells bezeichnen wir als **Modellversion**. Eine zeitlich aufeinanderfolgende Kette von Modellversionen bildet dann eine **Modellhistorie**. Die Historie wird i.d.R. durch ein Repository verwaltet. Neben der Speicherung von Modellversionen, besteht eine zentrale Aufgabe von Versionsverwaltungswerkzeugen darin, verschiedene Versionen miteinander zu vergleichen. Die Gemeinsamkeiten und Unterschiede zwischen zwei Modellversionen werden durch eine **Differenz** beschrieben. Eine Differenz wird beispielsweise benötigt, um in Konflikt stehende Modifikationen mehrerer Entwickler interaktiv zu mischen.

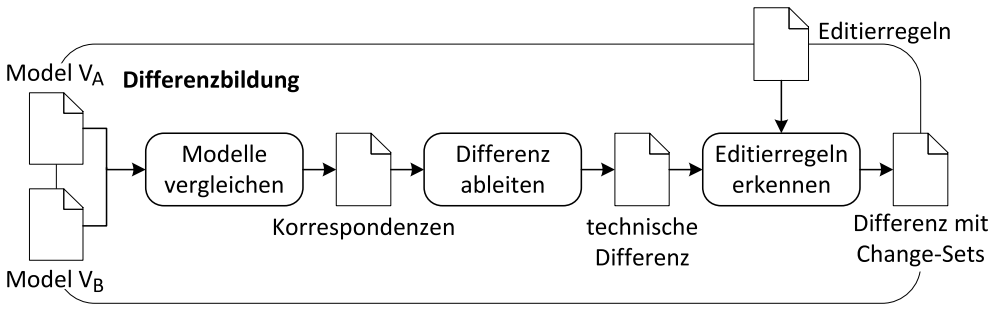


Abbildung 2.5: Datenfluss - Differenzbildung

Abbildung 2.5 zeigt die Differenzbildung zwischen zwei Modellversionen V_A und V_B. Im ersten Schritt wird nach den gemeinsamen Elementen beider Modelle gesucht. Der Entwickler kann hierfür zwischen verschieden Matchingtechniken wählen (z.B. ID-basiert, signaturbasiert oder ähnlichkeitsbasiert ([KKPS12])). Die **Korrespondenzen**, welche zwischen den Modellelementen beider Versionen ermittelt wurden, bilden die Basis für die Ableitung der **technischen Differenz** $D_{A \to B}$. Alle Elemente und Referenzen, die

nur in Modell V_A vorkommen, werden als gelöscht betrachtet. Elemente und Referenzen, die hingegen nur in Modell V_B vorkommen, werden als hinzugefügt angesehen. Außerdem müssen alle Attribute der korrespondierenden Elemente verglichen werden, um herauszufinden, ob diese modifiziert wurden. Die so berechneten **technischen Änderungen** werden auf Basis der abstrakten Syntax notiert. Abbildung 2.6 zeigt das Datenmodell zur Darstellung der technischen Differenz. Jede Modifikation des abstrakten Syntaxbaums eines Modells lässt sich durch 5 technische Änderungen beschreiben: *AddObject, RemoveObject, AddReference, RemoveReference, AttributeValueChange.*

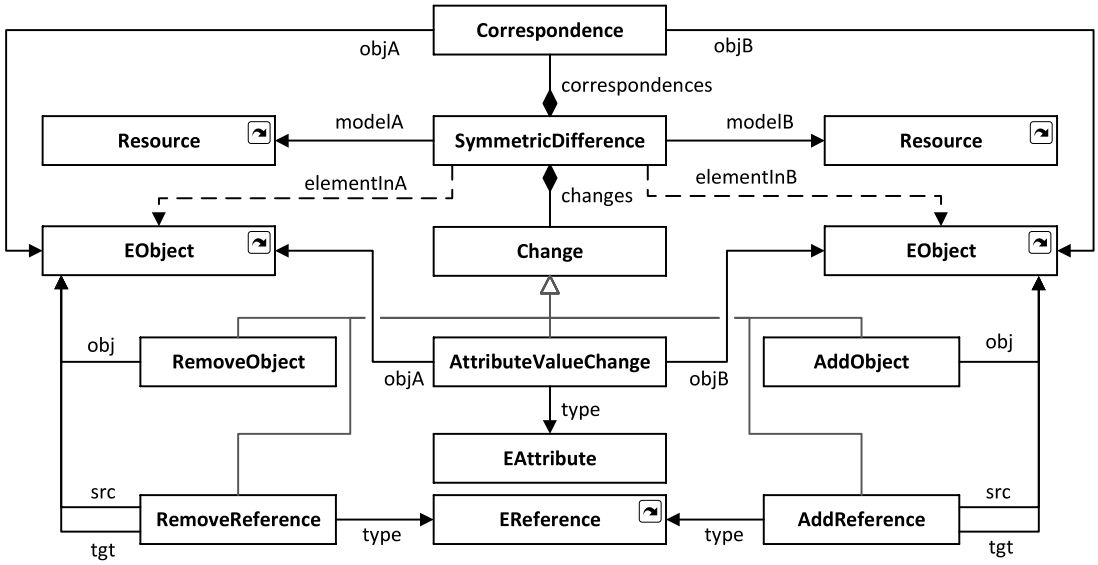


Abbildung 2.6: Datenmodell der technischen Differenz

Definition 2.3
Eine **technische Differenz** $D_{A\to B} = \{c_0, \ldots, c_m\}$ enthält alle **technischen Änderungen** $typeOf(c_i) \in \{AddObject,\ RemoveObject,\ AddReference,\ RemoveReference,\ AttributeValueChange\}$, die zwischen den Modellversionen V_A und V_B durchgeführt wurden. Ein **Editierschritt** $\Delta_x \subseteq D_{A\to B}$ enthält eine Teilmenge der technischen Änderungen einer Differenz und beschreibt damit einen Zustandsübergang zwischen zwei Modellzuständen $\Delta_x = V_{A'} \xrightarrow{x} V_{A''}$.

Die Änderungen der technischen Differenz sind aus der Sicht des Entwicklers sehr kleinteilig und daher schwer nachzuvollziehen. Eine Möglichkeit eine Differenz übersichtlicher darzustellen, ist die Gruppierung einzelner Änderungen zu sogenannten **ChangeSets**. Dazu muss zunächst festgelegt werden, welche Änderungen als (semantisch) zu-

sammenhängend betrachtet werden sollen. Kehrer et al. [KKT11, Keh15] beschreiben ein Verfahren, bei dem Änderungen so gruppiert werden, dass jedes Change-Set einer benutzerdefinierten Editierregel entspricht. Das Ziel des Verfahrens besteht darin, eine nicht überlappende Gruppierung der technischen Änderungen zu ermitteln, so dass jedes Change-Set dem Editierschritt Δ_x einer Editierregel δ_x entspricht. Hierfür wird die technische Differenz nach entsprechend zusammenhängenden Änderungen durchsucht.

Die Editierregeln werden als Graphtransformationsregeln beschrieben. Der Effekt einer Graphtransformationsregel lässt sich anhand der Aktionen des Aktionsgraphen ablesen. Jede Aktion einer Graphtransformations- bzw. Editierregel erzeugt bei Anwendung eine technische Änderung in der Differenz. Jede Editierregel hinterlässt somit ein Änderungsmuster in der Differenz. Dieses Änderungsmuster kann aus der Editierregel abgeleitet und innerhalb der Differenz gesucht werden. Ein **Änderungsmuster** beschreibt im Wesentlichen die technischen Änderungen und den Kontext, der die Änderungen verbindet. Dadurch erstreckt sich das Graphmuster sowohl über die beiden Modellversionen V_A und V_B als auch über die Änderungen und Korrespondenzen der Differenz. Abbildung 2.7 zeigt das Änderungsmuster zum Verschieben einer Operation (ohne die Synchronisation der Nachricht), wie es in der Editierregel in Abbildung 2.4 enthalten ist. Das Änderungsmuster wurde hier in die sogenannte **Erkennungsregel** eingebettet. Für jede vollständige Abbildung des Änderungsmusters auf die Differenz, wird ein neues Change-Set erzeugt, das alle gefundenen technischen Änderungen enthält. Das Change-Set entspricht somit dem durch die Editierregel erzeugten Editierschritt.

Die Menge der nicht überlappenden Change-Sets lässt sich allerdings nicht immer eindeutig bestimmen. Um die technischen Änderungen eindeutig einem Change-Set bzw. Editierschritt zuzuordnen, müssen ggf. Change-Sets verworfen werden. Ein sinnvolles Optimierungskriterium besteht z. B. darin, die Anzahl der Change-Sets möglichst gering zu halten und damit möglichst große und aussagekräftige Change-Sets zu erzeugen.

Eine eindeutige Erkennung der Editierschritte ist insbesondere dann relevant, wenn ein **Editierskript** erstellt werden soll. Ein Editierskript ist ebenfalls eine Differenz, welche die Modifikationen zwischen den zwei Modellversionen V_A und V_B durch Editierregeln beschreibt. Das in [KKT13, Keh15] beschriebene Verfahren bestimmt zunächst die Belegung der Eingabeparameter für jeden erkannten Editierschritt. Eine Editierregel mit initialisierten Parametern werden wir im Folgenden als **Editieroperation** $\vec{\delta_x}(p_1, \ldots, p_n)$ bezeichnen. Zusätzlich werden noch die sequentiellen Abhängigkeiten zwischen den Editieroperationen bestimmt. Erzeugt beispielsweise eine Editieroperation $\vec{\delta_x}()$ ein Modellelement, das von einer anderen Editieroperation $\vec{\delta_y}()$ verwendet wird, dann ist $\vec{\delta_y}()$ von $\vec{\delta_x}()$ abhängig, d.h. die Ausführung von $\vec{\delta_x}()$ muss vor der Ausführung von $\vec{\delta_y}()$ erfolgen.

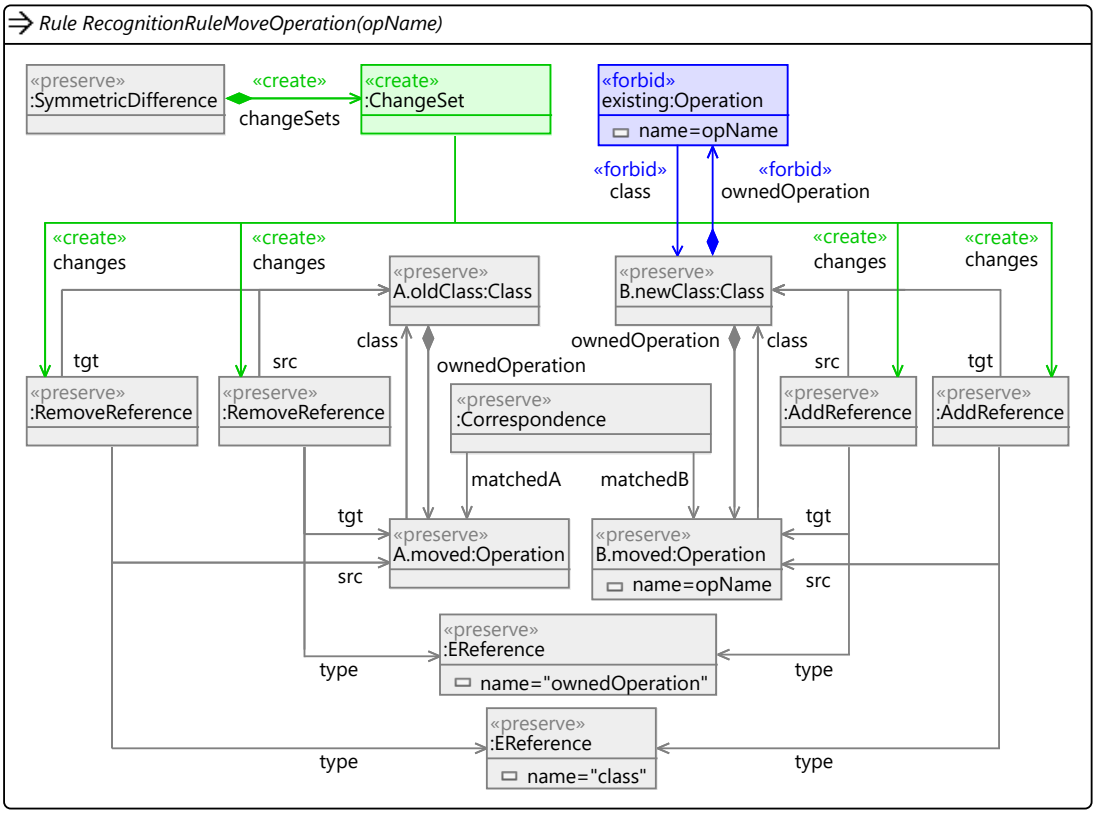


Abbildung 2.7: Erkennungsregel - Verschieben einer Operation

Die Editieroperationen des Editierskripts bilden somit eine partielle Ordnung, in der die Operationen ausgeführt werden müssen.

Definition 2.4
Ein **Editierskript** $\vec{D}_{A \to B} = \{\vec{\delta}_{x.0}(), \ldots, \vec{\delta}_{x.m}()\}$ besteht aus einer partiell geordneten Menge von **Editieroperationen** $\vec{\delta}_{x.i}(p_1, \ldots, p_n)$. Jede Editieroperation wird durch eine Editierregel $\delta_x(P_1, \ldots, P_n)$ definiert.

Die technische Differenz (mit Change-Sets) eignet sich hauptsächlich für Aufgaben bei denen zwei aufeinander folgende Modellversionen verarbeitet werden sollen. Eine typische Aufgabe ist beispielsweise das Inspizieren und Einchecken neuer Versionen in ein Repository. Ein Editierskript ist insbesondere dann nützlich, wenn die durchgeführten Änderungen als Patch extrahiert werden sollen. Der Patch kann dann die Änderungen auf ein anderes Modell, beispielsweise aus einem anderen Entwicklungszweig, übertragen. Die Darstellung der Differenz auf Basis von benutzerdefinierten Editierregeln erleichtert in jedem Fall die Lesbarkeit der Änderungen für den Entwickler.

3 Dekomposition von Editierregeln

Ein wichtiges Werkzeug zur Bearbeitung technischer Dokumente ist die automatische Erkennung und Ergänzung von unvollständigen Editierschritten. Werkzeuge zur Autovervollständigung von textuellen Programmiersprachen helfen sowohl unerfahrenen als auch fortgeschrittenen Entwicklern Arbeitsabläufe zu beschleunigen [FGL12]. Wie Kuschke et al. [KMR13, KM14] feststellen, wurde allerdings die Entwicklung von Werkzeugen, welche einen Modellierungsprozess durch Autovervollständigungen unterstützen, bisher weitestgehend vernachlässigt. Für Programmiersprachen lassen sich die möglichen Autovervollständigungen meist aus den sequentiell vorhergehenden Anweisungen ableiten. Bei Modellierungssprachen besteht eine wesentliche Herausforderung darin, dass Editierschritte häufig zu komplexen strukturellen Mustern führen, wodurch die Abschätzung nachfolgender Editierschritte, aufgrund der zahlreichen Alternativen, erschwert wird.

Graphtransformationsregeln bieten eine formale Grundlage, um sowohl einfache als auch komplexe Editierregeln einer Modellierungssprache zu beschreiben. Wie bereits in Abschnitt 2.3 und 2.4 beschrieben wurde, definiert eine Graphtransformationsregel bzw. Editierregel die Durchführung und Randbedingungen eines Editierschritts. Eine Autovervollständigung lässt sich dann durch die Aufteilung eines Editierschritts $V_A \xrightarrow{g} V_C$ in zwei aufeinanderfolgende Schritte $V_A \xrightarrow{t} V_B \xrightarrow{k} V_C$ beschreiben. Bei der Berechnung einer Autovervollständigung wurde der erste Schritt $\Delta_t = V_A \xrightarrow{t} V_B$ durch den Entwickler ausgeführt, woraufhin der vollständige Editierschritt $\Delta_g = V_A \xrightarrow{g} V_C$ automatisch erkannt und daraus der ergänzende Schritt $\Delta_k = V_B \xrightarrow{k} V_C$ abgeleitet wird.

Die Editierregel $\delta_g = \{\alpha_0, \ldots, \alpha_n\}$ beschreibt den Editierschritt Δ_g, wobei α_0 bis α_n die durchzuführenden Aktionen festlegen (Definition 2.2). Die Editierregeln der Teilschritte Δ_t und Δ_k lassen sich aus der **Gesamtregel** δ_g ableiten. Die bereits ausgeführte unvollständige **Teilregel** $\delta_t \subset \delta_g$ enthält eine Teilmenge der Aktionen aus δ_g. Die ergänzende Editierregel wird als **Komplementregel** $\overline{\delta_k}$, der Teilregel δ_t, bezeichnet. Wobei $\overline{\delta_k} = \delta_g \setminus \delta_t = \{\alpha_0, \ldots, \alpha_n\} \setminus \{\alpha_0, \ldots, \alpha_p\} = \{\alpha_0, \ldots, \alpha_q\}$ mit $n > q > 0$ gilt, d.h. die Komplementregel ergibt sich, indem die Aktionen der Teilregel aus der Gesamtregel entfernt werden. Daher gilt $\delta_t \setminus \overline{\delta_k} = \emptyset$, d.h. die Aktionen der Teil- und Komplementregel sind disjunkt. Daraus folgt, dass zu jeder Teilregel genau eine Komplementregel exis-

tiert. Eine Teil- bzw. Komplementregel besitzt mindesten eine aber insgesamt weniger Aktionen als die zugehörige Gesamtregel. Im Allgemeinen könnten für eine Gesamtregel mit $a = |\delta_g|$ Aktionen $2^a - 2$ Teilregeln existieren, d.h alle Kombinationen von Aktionen. Aufgrund der Struktur und Randbedingungen von Editierregeln lassen sich allerdings einige dieser Teil- bzw. Komplementregeln ausschließen. Außerdem werden für Autovervollständigungen nur die Teilregeln benötigt, welche sich maximal mit den technischen Änderungen $\Delta_t = V_A \xrightarrow{t} V_B = \{c_0, \ldots, c_p\}$ überschneiden, die bereits durch den Entwickler ausgeführt wurden. Ansonsten würden bereits durchgeführte Änderungen durch die Komplementregel wiederholt. Wir betrachten in den folgenden Abschnitten zunächst die Ableitung aller korrekten Teil- bzw. Komplementregeln, unabhängig von den konkret ausgeführten Änderungen des Entwicklers.

3.1 Ableiten von Teilregeln

Eine Editierregel $\delta_g = \{\alpha_0, \ldots, \alpha_n\}$ kann zunächst vereinfacht als eine Menge betrachtet werden, die beschreibt, welche Aktionen durchzuführen sind. Allerdings bildet nicht jede Teilmenge der Aktionen eine korrekte Teilregel. Verschiedene Aktionen können voneinander abhängen. Die Editierregel in Abbildung 3.1 erzeugt eine Transition in einem Zustandsdiagramm. Außerdem wird eine Operation eines Klassendiagramms als Trigger der Transition festgelegt. Der Trigger kann aber beispielsweise erst nach der Transition erzeugt werden, da die Transition den Container für den Trigger bildet.

Die Abhängigkeiten zwischen den verschiedenen Aktionen lassen sich als **Abhängigkeitsgraph** ausdrücken. Ein Knoten entspricht (mindestens) einer Aktion der Editierregel. Eine Kante, welche ausgehend von Knoten d_2 zu Knoten d_1 verläuft, gibt eine Abhängigkeit an. In diesem Fall ist Knoten d_2 von d_1 abhängig. Die Kante kann auch gelesen werden als: „d_2 benötigt d_1“. Übertragen auf die Erzeugung von Teilregeln bedeutet dies, dass die Aktionen, welche d_1 entspricht, vor der Aktion von d_2 ausgeführt werden muss. Zur einfacheren Formulierung ist es außerdem zulässig, dass einem Knoten des Abhängigkeitsgraphen mehrere Änderungen zugeordnet werden. Dies würde einem vollständig verbundenen Teilgraphen innerhalb des Abhängigkeitsgraphen entsprechen, d.h. die Änderungen müssen immer gemeinsam ausgeführt werden. Ein Element eines Modells wird beispielsweise immer zusammen mit seiner Container- und Containment-Referenz erzeugt oder gelöscht. Solche untrennbaren Abhängigkeiten werden im Folgenden als **atomare Abhängigkeit** bezeichnet. Jede gestrichelte Box in Abbildung 3.1 entspricht einer atomaren Abhängigkeit bzw. einem Knoten des Abhängigkeitsgraphen. Die Kanten zwischen den gestrichelten Boxen geben die Abhängigkeiten an.

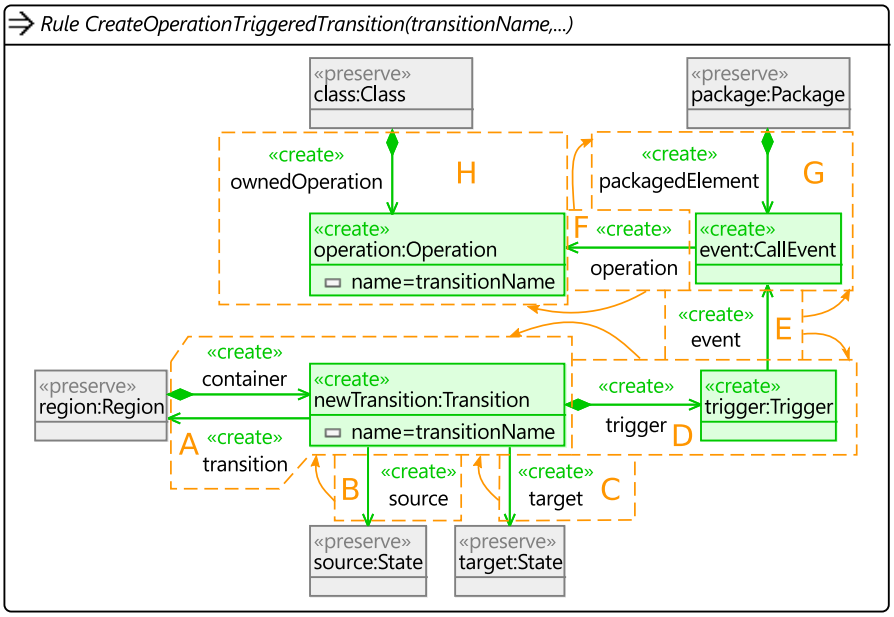


Abbildung 3.1: Editierregel - Erzeugen einer getriggerten Transition

Aus der Sicht einer einzelnen Aktion existiert immer eine (ggf. leere) Menge von direkten Vorgängern, die bereits ausgeführt sein müssen, bevor die Aktion selbst ausgeführt werden kann. Um den Abhängigkeitsgraph zu konstruieren, muss für jede Aktion die Menge der direkten Vorgänger bestimmt werden. Hierfür lassen sich eine Reihe von Regeln festlegen. Eine Aktion α_i der Editierregel $\delta_g = \{\alpha_0, \ldots, \alpha_n\}$ kann entweder einem erzeugenden/löschenden Knoten oder einer erzeugenden/löschenden Kante eines Aktionsgraphen bzw. einer Graphtransformationsregel entsprechen (Definition 2.2). Außerdem können Container-/Containment-Beziehungen (Eltern-/Kind-Beziehungen) sowie entgegengerichtete Kanten (*opposite*) in der Editierregel enthalten sein:

1. Jeder *erzeugende/löschende Knoten* bildet, zusammen mit seiner Container- und Containment-Kante eine atomare Abhängigkeit bzw. einen Knoten des Abhängigkeitsgraphen. Die Knoten A, D, G und H in Abbildung 3.1 sind Beispiele für atomare Abhängigkeiten.

2. In einem Metamodell kann eine ungerichtete Referenzen durch zwei entgegengerichtete Referenzen (*opposite*) definiert werden. Analog dazu bilden im Aktionsgraphen zwei *erzeugende/löschende entgegengerichtete Kanten* zusammen eine atomare Abhängigkeit bzw. einen Knoten des Abhängigkeitsgraphen. Die Kanten `covered` und

`coveredBy` der Editierregel in Abbildung 2.4 sind beispielsweise entgegengerichtete Kanten.

3. Der abstrakte Syntaxbaum eines Modells wird immer ausgehend von der Wurzel erstellt. Werden neuen Elemente durch eine Editierregel erzeugt, dann hängt ein *erzeugender Kindknoten* von seinem Container-Knoten ab, falls der Container selbst ein erzeugender Knoten ist. Dieser Fall tritt in Abbildung 3.1 zwischen `trigger` und `newTransition` auf, d.h. es ergibt sich eine Abhängigkeit zwischen den Knoten bzw. atomaren Abhängigkeiten D und A.

4. Um eine Referenz anzulegen, muss sowohl das Element, von dem die Referenz ausgeht, als auch das Element, auf welches die Referenz zeigt, zunächst erzeugt werden. Daher haben alle *erzeugenden Kanten* eine Abhängigkeit auf ihre Quell- und Zielknoten, falls diese Knoten erzeugende Knoten sind. Die Knoten B und C haben jeweils eine Abhängigkeit bezüglich des Quellknotens A. Die Knoten E und F haben jeweils eine Abhängigkeit bezüglich des Quell- (D und G) und Zielknotens (G und H).

5. Ein Teilbaum des abstrakten Syntaxbaums muss immer (rekursiv) von den Blättern ausgehend gelöscht werden. Jeder *löschende Container-Knoten* hat daher Abhängigkeiten, auf alle direkten (löschenden) Kindknoten. Abbildung 3.2 zeigt die Umkehroperation zur Editierregel in Abbildung 3.1. Hier tritt diese Abhängigkeit für den Knoten A zum Knoten D auf, d.h. der Trigger muss gelöscht werden, bevor die Transition gelöscht wird.

6. Eine Referenz muss immer gelöscht werden, bevor das entsprechende Element gelöscht wird. Ansonsten würden hängende Referenzen im Modell entstehen. Daher hängt jeder *löschende Knoten* von seinen inzidenten *löschenden Kanten* ab. Hieraus ergeben sich die restlichen Abhängigkeiten der Editierregel in Abbildung 3.2.

7. Die Änderung eines Attributwerts kann nur auf einem bereits existierenden Element erfolgen. Eine *Attributwertänderung* kann daher unabhängig von anderen Aktionen ausgeführt werden, d.h. es gibt in diesem Fall *keine* Abhängigkeiten.

Aus diesen Regeln lässt sich nun der Abhängigkeitsgraph für eine beliebige Editierregel konstruieren. Aus dem Abhängigkeitsgraph folgt, dass eine einzelne Abhängigkeit immer eine (ggf. leere) Menge von direkten **Vorgängern** (ausgehende Abhängigkeit) und **Nachfolgern** (eingehende Abhängigkeit) besitzt. Außerdem besitzt der Abhängig-

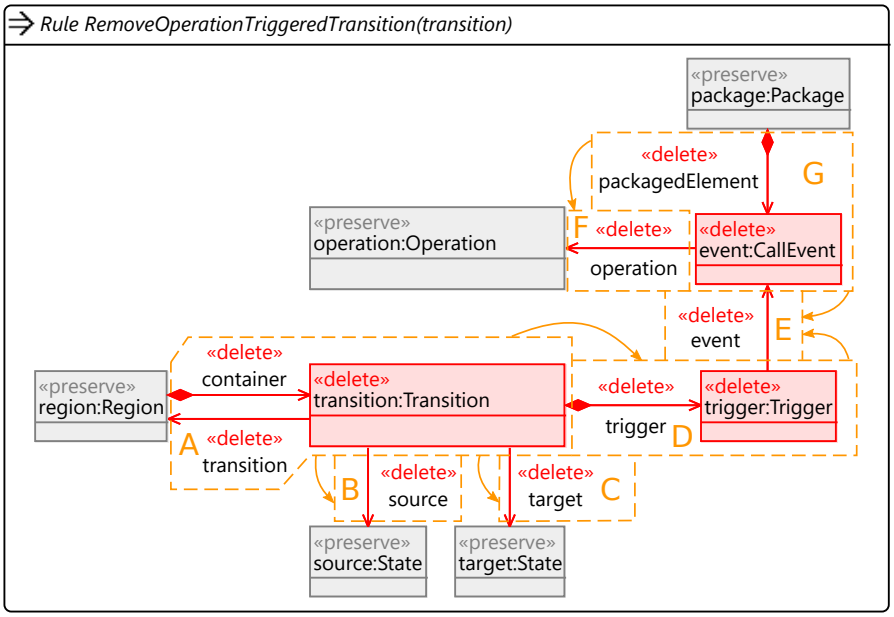


Abbildung 3.2: Editierregel - Löschen einer getriggerten Transition

keitsgraph eine Menge von **unabhängigen Knoten** I. Ein Knoten wird als unabhängig betrachtet, falls dieser keine Vorgänger besitzt. Im Sinne einer Editierregel legt die Menge I fest, welche Aktionen zuerst und unabhängig voneinander ausgeführt werden können. Betrachtet man eine Aktion aus I als ausgeführt und entfernt diese aus dem Abhängigkeitsgraphen, dann ergeben sich wiederum neue ausführbare Aktionen in I. Für die Editierregeln in Abbildung 3.1 und 3.2 ergeben sich damit die initial unabhängigen Knoten $I_{3.1} = \{A, G, H\}$ und $I_{3.2} = \{B, C, E, F\}$.

Neben der Auswahl der Aktionen stellt sich noch die Frage, welche Teile des Kontextgraphen und welche Anwendungsbedingungen aus der Gesamtregel δ_g in die Teilregel δ_t übernommen werden sollen. Im Kontext der Autovervollständigungen von Editierschritten betrachten wir zunächst nur **relaxierte zusammenhängende Teilregeln**:

- Für die Teilregel werden zunächst *keine Anwendungsbedingungen* aus der Gesamtregel übernommen. Die Überprüfung der Anwendungsbedingungen wird später durch die Komplementregel $\overline{\delta_k}$ nachgeholt. Attribute des Kontextgraphen mit variablen Werten werden ebenfalls als Anwendungsbedingung behandelt. Attribute in Kontextknoten der Gesamtregel, die *feste Attributwerte* besitzen, werden dagegen in die Teilregel übernommen.

- Außerdem wird für eine Teilregel gefordert, dass diese einen *zusammenhängend Aktionsgraph* bildet, d.h. es existiert mindestens ein Pfad zwischen allen Aktionen der Editierregel. Auf diese Weise werden bei einer Autovervollständigung nur die bereits ausgeführten Aktionen betrachtet, die lokal zusammenhängende Änderungen durchführen (Kapitel 4). Aus der Editierregel in Abbildung 3.1 darf beispielsweise nicht nur die erzeugende Kante `event` entfernt werden, um eine Teilregel zu konstruieren. Diese Einschränkung lässt sich aber grundsätzlich durch alternative Editierregeln umgehen. In diesem Fall würde eine Editierregel benötigt, bei der das `CallEvent` nicht neu erzeugt wird.

Im Allgemeinen kommt die gesamte LHS der Graphtransformationsregel für die Auswahl eines zusammenhängenden Kontextgraphen in Frage. Daraus folgt für den Aktionsgraphen, dass löschende Knoten und Kanten, die nicht in der Teilregel enthalten sind, in Kontext umgewandelt werden. Anzumerken ist hierbei, dass bei der Darstellung als Graphtransformationsregel – entgegen der begrifflichen Intuition – die RHS einer Teilregel größer sein kann, als die RHS der entsprechenden Gesamtregel. Durch die Umwandlung von löschenden Knoten und Kanten in Kontext, wird die RHS um die entsprechenden Graphanteile erweitert. Abbildung 3.3 zeigt eine Teilregel der Editierregel aus Abbildung 3.2. Die noch zu löschende `Transition` wird hier als Kontextknoten für den gelöschten `Trigger` verwendet.

3.2 Ableitung der Komplementregel

Ein Editierschritt $V_A \xrightarrow{g} V_C$ zwischen zwei Modellzuständen, der sich in zwei Teilschritte $V_A \xrightarrow{t} V_B \xrightarrow{k} V_C$ zerlegen lässt, wird durch die Editierregeln δ_g, δ_t und $\overline{\delta_k}$ beschrieben. Wir gehen zunächst davon aus, dass die Gesamtregel δ_g und die Teilregel δ_t bekannt sind. Nun soll die Komplementregel $\overline{\delta_k}$ abgeleitet werden. Die Komplementregel soll die Teile der Gesamtregel abdecken, die nicht bereits in der Teilregel enthalten sind. Dies lässt sich durch die Subtraktion der Teilregel von der Gesamtregel beschreiben. Dabei wird zunächst nur der Effekt der Teilregel subtrahiert, d.h. es werden nur enthaltene Aktionen entfernt und nicht die enthaltenen Anwendungsbedingungen.

Tabelle 3.1 zeigt die Ableitung der Komplementregel, aus den Änderungen der Gesamt- und Teilregel. Erzeugende Knoten bzw. Kanten («create»), die bereits in der Teilregel enthalten sind, werden in der Komplementregel in Kontext («preserve») umgewandelt (Zeile 1). Die Knoten bzw. Kanten können nicht ohne weiteres aus der Regel entfernt werden, da diese ggf. als Kontext für andere Änderungen benötigt werden. Ein Knoten

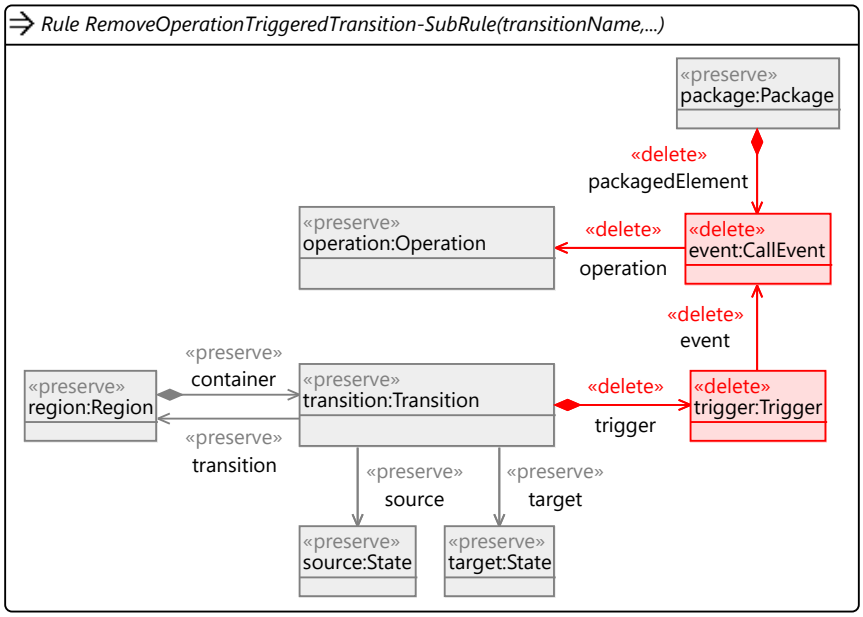


Abbildung 3.3: Teilregel - Löschen einer getriggerten Transition

könnte beispielsweise einen Container für enthaltene Kindelemente darstellen. Löschende Knoten bzw. Kanten («delete»), die bereits in der Teilregel vorkommen, müssen hingegen aus der Komplementregel entfernt werden (Zeile 3). Dies gilt auch für Änderungen von Attributwerten (Zeile 6). Die Änderung eines Attributwerts ($a = x \rightarrow y$) kann nur auf einem bereits existierenden Modellelement durchgeführt werden, d.h. diese Aktionen beziehen sich immer auf Kontextknoten der Gesamtregel. Alle weiteren Änderungen, die nicht durch die Teilregel abgedeckt sind, werden aus der Gesamtregel δ_g in die Komplementregel $\overline{\delta_k}$ übernommen (Zeile 2,4,5,7). Wie im vorherigen Abschnitt beschrieben, werden löschende Knoten bzw. Kanten ggf. in Kontext umgewandelt (Zeile 4), d.h. die

	δ_g	δ_t	$\overline{\delta_k}$
1	«create»	«create»	«preserve»
2	«create»	–	«create»
3	«delete»	«delete»	–
4	«delete»	«preserve»	«delete»
5	«delete»	–	«delete»
6	$a = x \rightarrow y$	$a = x \rightarrow y$	–
7	$a = x \rightarrow y$	–	$a = x \rightarrow y$

Tabelle 3.1: Ableitung der Komplementregel

entsprechenden Aktionen sind nicht in der Teilregel enthalten. Abbildung 3.4 zeigt die abgeleitete Komplementregel zur Teilregel in Abbildung 3.3.

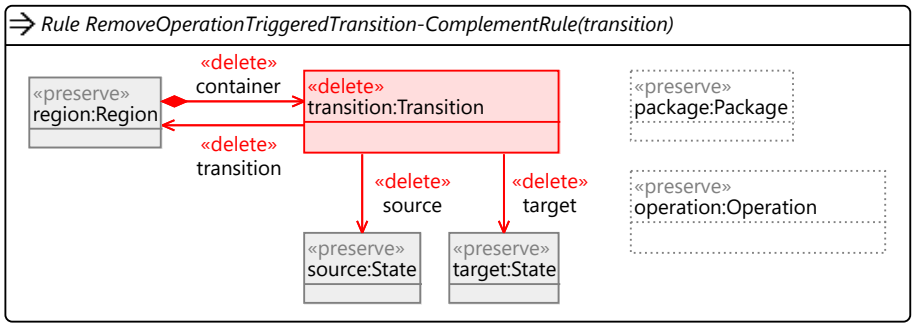


Abbildung 3.4: Komplementregel - Löschen einer getriggerten Transition

Ein grundsätzliches Problem besteht darin zu entscheiden, welche Vorbedingungen einer Gesamtregel in eine Komplementregel zu übernehmen sind. Eine Vorbedingung kann entweder ein Teil des Kontextes oder eine Anwendungsbedingung sein. Im Allgemeinen wäre zu überprüfen, ob sich der Editierschritt Δ_t der Teilregel δ_t, innerhalb einer Sequenz von Editierschritten $V_A \xrightarrow{\cdots} V_{A'} \xrightarrow{t} V_{A''} \xrightarrow{x} V_B$, durch den Editierschritt Δ_g ersetzen lässt, wobei alle Vorbedingungen der Gesamtregel δ_g zu diesem Zeitpunkt erfüllt sein müssen. Die Vorbedingungen der Komplementregel werden allerdings erst auf Modellzustand V_B überprüft. Es kann aber passieren, dass ein nachfolgender Editierschritt Δ_x die erwarteten Vorbedingungen wieder aufhebt. Die Komplementregel in Abbildung 3.4 ist beispielsweise nicht auf Modell V_B anwendbar, wenn die Operation ebenfalls gelöscht wurde, da die Operation hier als (unnötiger) Kontext gefordert wird.

Bisher wurde der vollständige Kontext aus der Gesamt- in die Komplementregel übernommen. Grundsätzlich werden dadurch bestimmte Komplementregeln ausgeschlossen, wenn Vorbedingungen auf Modell V_B nicht mehr erfüllt sind. Als Lösungsstrategie wird der Kontextgraph einer Komplementregel ggf. minimiert. Alle isolierten Kontextknoten, d.h. Knoten, die keine inzidenten Kanten (Aktionen oder Anwendungsbedingungen) oder Attribute mehr besitzen, können aus der Komplementregel entfernt werden. Diese Knoten werden nur als Vorbedingungen für die Teilregel benötigt und können für die Komplementregel vernachlässigt werden. Für die Komplementregel in Abbildung 3.4 folgt daraus, dass die Kontextknoten `operation` und `package` entfernt werden können. Damit ist die Editierregel auch anwendbar, wenn beispielsweise die Operation aus dem Modell gelöscht wurde.

Die Anwendungsbedingungen werden weiterhin vollständig aus der Gesamtregel übernommen. Eine Anwendungsbedingung wird damit wie eine Invariante behandelt, welche

durch die Editierregel impliziert wird. Zukünftig könnten stattdessen Lösungsstrategien eingesetzt werden, bei denen die Vorbedingungen abgeschwächt werden. Beispielsweise könnte man eine Vorbedingung als erfüllt ansehen, wenn diese zumindest auf eine der beiden Modellversion V_A oder V_B erfüllt ist. Damit hätte man eine gute Abschätzung dafür, dass ein nachvollziehbarer[1] Zustand zwischen V_A und V_B existiert, in dem die Vorbedingungen erfüllt sind.

[1]Eine Sequenz ohne transiente Editierschritte, z. B. mehrfaches überschreiben eines Attributwerts.

4 Erkennung partieller Editierregeln

Ein Werkzeug zur Autovervollständigung von Modellen benötigt zunächst Informationen über die zu ergänzenden Teilstrukturen. Ein entsprechendes Verfahren kann entweder zustands- oder differenzbasiert arbeiten, d.h. die Ergänzung wird entweder auf Basis einer unvollständigen Modellierung oder über die zuletzt ausgeführten Änderungen berechnet. Ein differenzbasiertes Verfahren hat den Vorteil, dass nicht nur der aktuelle Zustand des Modells betrachtet wird, sondern auch Löschungen und Verschiebungen von Elementen.

Das im Folgenden vorgestellte Verfahren erkennt unvollständig ausgeführte Editierschritte, basierend auf einer Differenz zwischen zwei Modellversionen V_A und V_B. Eine Differenz $D_{A \to B} = \{c_0, c_1, \ldots\}$ enthält alle technischen Änderungen c_i, die zwischen Modellversion V_A und V_B durchgeführt wurden (Definition 2.3). Ziel des Algorithmus ist es, alle Autovervollständigungen der Differenz $D_{A \to B}$ bezüglich einer Editierregel δ_g zu ermitteln. Dafür sind alle Teilregeln $\delta_t \subset \delta_g$ der Gesamtregel δ_g zu betrachten, die einen Editierschritt $\Delta_t = \{c_0, c_1, \ldots\}$ innerhalb der Differenz $D_{A \to B} = V_A \xrightarrow{\cdots} V_{A'} \xrightarrow{t} V_{A''} \xrightarrow{\cdots} V_B$ beschreiben. Aus der Gesamtregel δ_g und der Teilregel δ_t lässt sich dann wiederum die Komplementregel $\overline{\delta_k} = \delta_g \setminus \delta_t$ ableiten. Abschließend müssen alle möglichen ergänzenden Editierschritte $\Delta_k = V_B \xrightarrow{k} V_C$ ermittelt werden. Dafür sind alle Initialisierungen der Komplementregel $\overline{\delta_k}(P_1, \ldots, P_n)$ für die aktuelle Modellversion V_B zu berechnen. Jede initialisierte Komplementregel stellt dann eine konkrete Editieroperation $\vec{\delta_k}(p_1, \ldots, p_n)$ bzw. Autovervollständigung dar (Definition 2.4). Einzelne Parameter P_i, wie beispielsweise Namen für neu zu erzeugende Modellelemente, die sich nicht aus der Komplementregel abgeleitet lassen, sind ggf. noch durch den Entwickler zu initialisieren.

Um zu erkennen, ob eine Differenz technische Änderungen enthält, die den Aktionen einer Editierregel entsprechen, wird das in Abschnitt 2.4 beschriebene Verfahren verwendet. Zunächst wird die Editierregel δ_g in ein sogenanntes Änderungsmuster transformiert. Lässt sich das Änderungsmuster vollständig auf einen Teil der Differenz (Modell V_A, V_B, Korrespondenzen und technische Änderungen) abbilden, dann kann die Menge der erkannten Änderungen durch die Editierregel wiederholt werden. Da in diesem Fall nur ein Teil einer Editierregel in der Differenz erkannt werden soll, wird ein spezieller

Matchingalgorithmus benötigt. Der Algorithmus muss nach allen maximalen partiellen Abbildungen zwischen dem Änderungsmuster und der Differenz suchen. Das partielle Änderungsmuster bezüglich der gefundenen Abbildung muss wiederum dem Änderungsmuster einer Teilregel $\delta_t \subset \delta_g$ entsprechen.

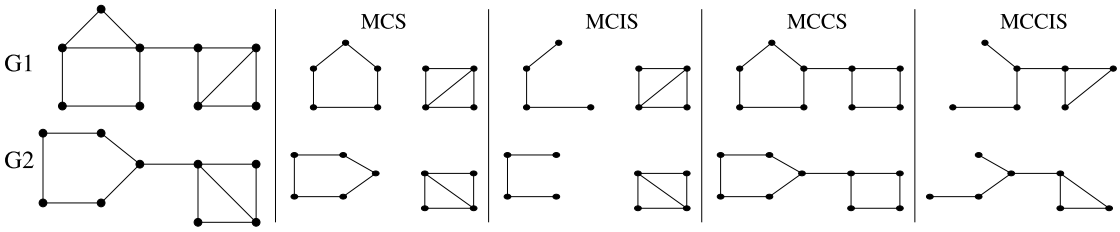


Abbildung 4.1: MCS-Probleme: MCS, MCIS, MCCS, MCCIS [VV08]

Die Berechnung entspricht im Allgemeinen einem MCS-Problem (*maximum common subgraph, MCS*). Zur Lösung eines MCS-Problems müssen alle maximalen Teilgraphen zweier Graphen $G1$ und $G2$ gefunden werden. Wie Abbildung 4.1 zeigt, kann bei der Beschreibung eines MCS-Problems außerdem zwischen zwei Eigenschaften des gemeinsamen Teilgraphen unterschieden werden. Eine mögliche Einschränkung besteht darin, dass die Teilgraphen zusammenhängenden sein müssen (*maximum common connected subgraph, MCCS*), d.h. alle Knoten der Graphen sind miteinander verbunden. Eine weitere häufig verwendete Eigenschaft ist die des induzierten Teilgraphen (*maximum common induced subgraph, MCIS*). Die Teilgraphen dürfen dabei nur durch das Entfernen von Knoten und allen inzidenten Kanten aus den beiden Graphen $G1$ und $G2$ entstehen. Grundsätzlich können auch beide Eigenschaften gleichzeitig gefordert werden (*maximum common connected subgraph, MCCIS*). Die meisten in der Literatur beschriebenen Algorithmen zur Lösung des MCS-Problems beziehen sich auf induzierte Teilgraphen [VV08].

Die erforderliche Berechnung lässt sich im Wesentlichen in zwei Teilprobleme zerlegen: Die Ableitung aller korrekten Teilregeln einer Editierregel und die Berechnung der partiellen Abbildungen des entsprechenden Änderungsmusters auf die Differenz. Die partielle Abbildung muss strukturerhaltend und typkompatibel sein, d.h. es existiert ein Graphmorphismus, der alle getypten Knoten und Kanten des Änderungsmusters zusammenpassend abbildet. Außerdem muss ein partielles Änderungsmuster einer Gesamtregel δ_g dem Änderungsmuster einer Teilregel $\delta_t \subset \delta_g$ entsprechen. Da für Teilregeln in Abschnitt 3.1 gefordert wurde, dass diese zusammenhängend sind, muss der Matchingalgorithmus für Änderungsmuster ebenfalls auf zusammenhängende Abbildungen eingeschränkt werden. Außerdem muss nicht der vollständige Kontextgraph aus der Gesamtregel übernommen werden, um die Aktionen der Teilregel zu verbinden, d.h. es können ggf. einzelne Kanten

fehlen. Die Teilregel ist damit kein induzierter Teilgraph der Gesamtregel. Daraus folgt, dass der Matchingalgorithmus in diesem Fall ein MCCS-Problem lösen müsste.

Für die Erkennung der Teilregeln ist es allerdings ausreichend, wenn mindestens eine Abbildung des Kontextgraphen existiert, so dass die Änderungen eines Editierschritts über Modell V_A bzw. V_B miteinander verbunden sind. Es wird daher nicht jede einzelne Kombination von Modellelementen benötigt, die einen passenden Kontext bilden. In Abschnitt 4.1 und 4.2 wird deshalb zunächst nur das kombinatorische Problem betrachtet, bei dem einzelne technische Änderungen bezüglich der Aktionen einer Editierregel ausgewählt werden. Der Algorithmus wird dann in Abschnitt 4.3 um eine strukturelle Analyse erweitert, so dass nur Änderungen kombiniert werden können, die dem Kontextgraph entsprechend miteinander verbunden sind.

MCS-Algorithmen unterscheiden sich außerdem darin, ob die Anzahl der Kanten oder Knoten im gemeinsamen Teilgraphen maximiert werden. Der hier beschriebene Algorithmus beschränkt sich darauf, eine Teilmenge der Knoten eines Graphmusters zu maximieren. Diese Knoten werden im Folgenden als **Variablen** der zu maximierenden Menge $V = \{v_0 \times \ldots \times v_n\}$ bezeichnet. Die Menge der Elemente c_i, mit denen eine Variable belegt werden kann, bildet den zugehörigen **Wertebereich** (*domain*) dieser Variablen $v_i[c_0, c_1, \ldots]$. Der in Abschnitt 4.1 beschriebene Algorithmus erzeugt – solange keine weiteren Einschränkungen getroffen werden – für alle Teilmengen von V das vollständige Kreuzprodukt aus den Wertebereichen aller Variablen. Eine gefundene Belegung der Variablen wird als **Lösung** (*solution*) bzw. Teillösung bezeichnet. Um die Menge der Lösungen einzuschränken, werden sowohl für die möglichen Teilmengen von V, als auch für die Kombinationen von Elementen aus den Wertebereichen, Bedingungen festgelegt. Die Bedingungen werden in diesem Kontext als **Constraints** bezeichnet. Ein Problem auf diese Weise zu formulieren, wird dann als Constraint-Satisfaction-Problem (CSP) bezeichnet ([VV08, McG82]).

> *„Consider the general problem of selecting a value for each of n variables, x_i $i = 1, \ldots n$. The value for each x_i, must be selected from a corresponding finite domain D_i. A set of values $(x_l, x_2, \ldots x_n)$ satisfying some specified conditions represents a solution to the problem. Many non-numerical problems can be expressed in this form and backtrack algorithms are widely used in solving such problems, usually in conjunction with other refinement techniques for reducing the number of combinations of values to be considered.“ [McG82]*

Die Aktion α_i einer Editierregel wird im Änderungsmuster durch einen Knoten v_i für die entsprechende technische Änderung repräsentiert. Der Knoten v_i bildet eine Variable

des Constraint-Satisfaction-Problems $V = \{v_0 \times \ldots \times v_n\}$. Eine Variable entspricht somit indirekt einer Aktion ($v_i \mathrel{\widehat{=}} \alpha_i$) der zu maximierenden Teilregel. Der Wertebereich (*domain*) für die Variable $v_i[c_0, c_1, \ldots]$ eines Änderungsmusters wird zunächst durch die Änderungen der technischen Differenz $D_{A \to B} = \{c_0, \ldots, c_m\}$ vorgegeben. Der Wertebereich wird außerdem durch den Typ der technischen Änderung (*AddObject, RemoveObject, AddReference, RemoveReference, AttributeValueChange*) eingeschränkt. Des Weiteren muss der Typ des erzeugten/gelöschten Modellelements bzw. der modifizierten Eigenschaft (Referenz, Attribut) mit dem von der Aktion vorgegebenen Typ übereinstimmen. Die Aktionskante in Abbildung 4.2 gibt sowohl den Typ des Quell- und Zielknoten (`Class, Operation`) als auch den Typ der Referenz (`ownedOperation`) vor. Der Aktionsknoten in Abbildung 4.3 gibt zumindest den Typ des zu löschenden Elements vor. Grundsätzlich wird der Wertebereich einer Variablen v_i bereits statisch, durch alle von der Aktion $\alpha_i \mathrel{\widehat{=}} v_i$ abhängigen Aktionen (Abschnitt 4.4) und deren unmittelbaren Kontext, vorgegeben.

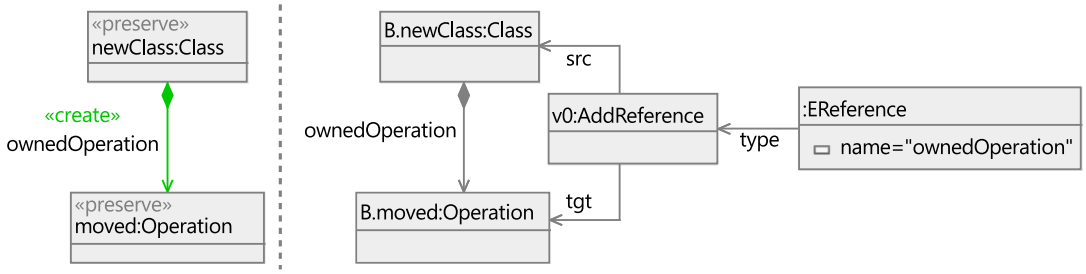


Abbildung 4.2: Änderungsmuster einer Aktionskante (Editierregel Abbildung 2.4)

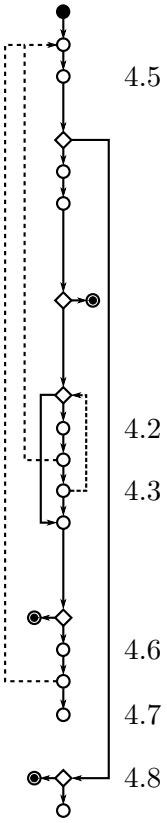
Abbildung 4.3: Änderungsmuster eines Aktionsknotens (Editierregel Abbildung 3.2)

4.1 Grundform des Matchingalgorithmus

Der Pseudocode 4.1 zeigt die Grundform des Matchingalgorithmus [KH04]. Die Funktion `expand()` erzeugt, durch rekursive Aufrufe, das Kreuzprodukt der Wertebereiche für alle möglichen Teilmengen von $V = \{v_0, \ldots, v_n\}$. Zunächst werden schrittweise einzelne Elemente für die Variablen festgelegt, bis eine maximal mögliche Lösung erreicht wird. Der Algorithmus verwendet dann die Strategie des Backtrackings, um den restlichen

Lösungsraum zu durchsuchen. Ist für die zuletzt ausgewählte Variable v_i keine weitere Belegung mehr möglich, dann wird nach allen weiteren Lösungen gesucht, in denen die Variable v_i *nicht* enthalten ist.

```
1  Set remainingVariables = {v0, ..., vn}
2  Stack removedVariables = {}
3  Stack pickedVariables = {}
4  Stack solution = {}
5
6  function expand(int unassigned) {
7      int next = (unassigned == 0) ? pickVariables() : unassigned
8
9      if (next != 0) {
10         Variable vi = pickedVariables[solution.size()]
11         Iterator domain = getDomain(vi)
12
13         // Ensure atomic dependencies:
14         if ((unassigned > 0) && domain.isEmpty()) return
15
16         // Assign all subsets which include the picked variable:
17         while (domain.hasNext()) {
18             assignVariable(vi, domain.next())
19             expand(next - 1)
20             freeVariable(vi)
21         }
22
23         // Create all subsets which exclude the picked variable:
24         if (unassigned == 0) {
25             removeVariables(next)
26             expand(unassigned)
27             addVariables(next)
28         }
29     } else if (isAMaxSolution()) {
30         storeSolution()
31     }
32 }
```



Algorithmus 4.1: Matchingalgorithmus

Initial sind alle Variablen $V = \{v_0 \times \ldots \times v_n\}$ in der Menge *remainingVariables* enthalten. Die aktuell ausgewählte Teilmenge der Variablen wird in *pickedVariables* gespeichert und ist initial leer. Die jeweils zugewiesenen Elemente der aktuellen Teillösung werden parallel auf dem Stack *solution* abgelegt. Der Parameter *unassigned* der `expand`-Funktion gibt die Anzahl der Variablen an, die für die Teilmenge *pickedVariables* ausgewählt aber noch nicht belegt wurden. Der Algorithmus wird daher initial mit `expand(0)` gestartet, da noch keine Variablen ausgewählt und belegt wurden.

Aufbauen einer Lösung: Zu Beginn des Algorithmus (Zeile 7) müssen die ersten zu belegenden Variablen ausgewählt werden. Die Funktion ⟨07⟩`pickVariables()` überprüft, ob Variablen in *remainingVariables* vorhanden sind, um die Teilmenge *pickedVariables* der bisher ausgewählten Variablen zu erweitern. Im einfachsten Fall wählt die Funktion immer genau eine Variable aus und fügt diese der Teilmenge *pickedVariables* hinzu. In diesem Zusammenhang können auch zunächst alle Fallunterscheidungen bezüglich des *unassigned* Parameters ignoriert werden. Wurde mindestens eine neue Variable eingefügt, dann wird die erste noch nicht belegte Variable v_i aus der Teilmenge *pickedVariables* selektiert (Zeile 10). Dann wird zunächst der Wertebereich der Variablen abgefragt (⟨11⟩`getDomain()`) und anschließend eine Schleifeniteration über alle enthaltenen Werte durchgeführt (Zeile 17-21).

Die Variable $v_i[c_0, c_1, \ldots]$ wird in jeder Iteration durch die Funktion ⟨4.2⟩`assignVariable()` mit einem neuen Element c_i (in unserem Fall einer technischen Änderung) belegt. Zusätzlich wird durch die $\mathbb{S}$-Funktion die **Injektivität** der Lösung sichergestellt. Damit kein Element in einer Lösung mehrfach vorkommt, wird das gerade zugewiesene Element temporär aus allen anderen Wertebereichen entfernt. Anschließend wird die Funktion ⟨19⟩`expand()` rekursiv aufgerufen, um die nächsten Variablen auszuwählen und zu belegen. Die Rekursion wird fortgesetzt, bis die Lösung nicht mehr erweitert werden kann, d.h. in *remainingVariables* sind keine passenden Variablen mehr vorhanden, die durch die Funktion ⟨07⟩`pickVariables()` ausgewählt werden können (*next = 0*).

```
1 function assignVariable(Variable vi, Element ci) {
2     solution.push(value)
3
4     // restrict domains:
5     𝕊(vi, ci)
6 }
```

Algorithmus 4.2: Festlegen der Variablen

Sobald die Lösung nicht mehr erweitert werden kann, wird die aktuelle Lösung gespeichert (Zeile 29-31). Die Funktion ⟨29⟩`isAMaxSolution()` kann – in der vereinfachten Betrachtung des Algorithmus – zunächst ignoriert werden. Die Lösung wird nun kopiert und an einer geeigneten Stelle abgelegt (⟨30⟩`storeAssignment()`). Die Kopie ist erforderlich, da die Variablen im weiteren Verlauf mit neuen Elementen überschrieben werden.

Backtracking: Nachdem die Lösung gespeichert wurde, wird der aktuelle Aufruf von `expand()` beendet, wodurch die Rekursion auf den letzten Aufruf von ⟨19⟩`expand()`

zurückgesetzt wird. Die letzte Variable v_i wird anschließend, durch die Funktion `free Variable()` in Algorithmus 4.3, wieder aus der Lösung (*solution*) entfernt. Durch die $\mathbb{S}^{-1}$-Funktion werden schließlich die Einschränkungen, zur Sicherstellung der Injektivität, wieder aufgehoben. Solange noch Werte zur Verfügung stehen, wird die Zuweisungsschleife (Zeile 17-21) fortgesetzt. Grundsätzlich kann der Wertebereich für eine Variable (⟨11⟩`getDomain()`) auch leer sein. Die Zuweisungsschleife würde in diesem Fall einfach übersprungen.

```
1 function freeVariable(Variable variable) {
2     solution.pop();
3
4     // undo restrictions of the domains:
5     S^-1(v_i, value)
6 }
```

Algorithmus 4.3: Freigeben der Variablen

Berechnung aller Teillösungen: Nachdem der Wertebereich einer Variablen v_i abgearbeitet wurde, müssen noch alle Lösungen ermittelt werden, in denen v_i nicht vorkommt. Dazu wird die Variable aus der Menge *pickedVariables* der ausgewählten Variablen entfernt und auf dem Stack *removedVariables* abgelegt (⟨25⟩`removeVariable()`). Anschließend wird – analog zur Zuweisungsschleife – die Funktion ⟨26⟩`expand()` rekursiv für die nächste Variable aufgerufen. Nachdem alle weiteren Lösungen ermittelt wurden, wird die Variable v_i wieder in die Menge *remainingVariables* der verfügbaren Variablen eingefügt (⟨27⟩`addVariables()`).

$\{0,0,0\}$	$\{0,1,\#\}$	$\{1,0,1\}$	$\{1,\#,0\}$	$\{\#,0,\#\}$	$\{\#,\#,1\}$
$\{0,0,1\}$	$\{0,\#,0\}$	$\{1,0,\#\}$	$\{1,\#,1\}$	$\{\#,1,0\}$	$\{\#,\#,\#\}$
$\{0,0,\#\}$	$\{0,\#,1\}$	$\{1,1,0\}$	$\{1,\#,\#\}$	$\{\#,1,1\}$	
$\{0,1,0\}$	$\{0,\#,\#\}$	$\{1,1,1\}$	$\{\#,0,0\}$	$\{\#,1,\#\}$	
$\{0,1,1\}$	$\{1,0,0\}$	$\{1,1,\#\}$	$\{\#,0,1\}$	$\{\#,\#,0\}$	

Tabelle 4.1: Lösungen für $V = \{v_1 \times v_2 \times v_3\}$ (spaltenweise)

Das temporäre Entfernen der Variablen hat denselben Effekt, wie den Wertebereich aller Variablen um einen Platzhalter zu erweitern. Nehmen wir als Beispiel eine Menge $V = \{v_1, v_2, v_3\}$ mit drei Variablen. Der Wertebereich aller Variablen ist $v_1 = v_2 = v_3 = [0, 1, \#]$, inklusive des Platzhalters $\#$. Insgesamt ergeben sich so $3 \times 3 \times 3 = 27$ Lösungen. Davon sind $2 \times 2 \times 2 = 8$ vollständige Belegungen und eine leere Belegung, die nur Platzhalter enthält. Tabelle 4.1 zeigt alle Lösungen, in der Reihenfolge (spaltenweise),

wie sie durch den Algorithmus erzeugt werden. Bisher wurden noch keine Einschränkungen für den Lösungsraum vorgenommen. Grundsätzlich lassen sich zwei Dimensionen des Lösungsraums einschränken. Wir werden zunächst die Begrenzung der möglichen Teilmengen von $V = \{v_0 \times \ldots \times v_n\}$ betrachten und anschließend in Abschnitt 4.3 die inkrementelle Einschränkungen der Wertebereiche $v_i[c_0, c_1, \ldots]$.

4.2 Matching von Teilregeln

Jede Aktion α_i einer Editierregel ist eindeutig einer Variablen v_i des entsprechenden Änderungsmusters zugeordnet ($\alpha_i \mathrel{\widehat{=}} v_i$), d.h. indirekt wird durch die Auswahl der Variablen eine Teilregel konstruiert. Um eine *korrekte* Teilregel zu konstruieren sind bei der Auswahl die Abhängigkeiten zwischen den Aktionen zu beachten. Abschnitt 3.1 beschreibt, welche Abhängigkeiten zwischen verschiedenen Aktionen auftreten können. Für jede Editierregel wird dann ein Abhängigkeitsgraph ermittelt, welcher die Reihenfolge wiedergibt, in der die Aktionen der Regel ausgeführt werden können.

Jeder Abhängigkeitsgraph besitzt eine Menge von **unabhängigen Knoten** I. Ein Knoten wird als unabhängig betrachtet, falls dieser keine Vorgänger besitzt. Es dürfen immer nur Knoten ausgewählt werden, die aktuell in der Menge I der unabhängigen Knoten enthalten sind. Ein ausgewählter Knoten wird danach aus dem Abhängigkeitsgraphen entfernt. Ein Knoten d_p wird zu einem unabhängigen Knoten, sobald alle Knoten aus dem Graphen entfernt wurden, von denen d_p abhängt. Die Aktualisierung der Menge I kann dabei inkrementell erfolgen. Wird ein Knoten d_i aus dem Abhängigkeitsgraphen entfernt, so werden alle Nachfolger von d_i, die keine weiteren Vorgänger mehr haben, in die Menge I der unabhängigen Knoten aufgenommen.

Der Matchingalgorithmus 4.1 erzeugt alle Teilregeln einer Editierregel mittels eines Backtracking-Algorithmus. Ein Pfad innerhalb des Backtrackings entspricht hier einer Sequenz von ausgewählten Abhängigkeitsknoten. Es wird also ein Algorithmus benötigt, der die Auswahl und das Zurücklegen (*undo*) von Knoten unterstützt. Das Zurücklegen erfolgt dabei in umgekehrter Reihenfolge wie die Auswahl der Knoten. Die Menge I der unabhängigen Knoten ist dabei ebenfalls inkrementell zu aktualisieren, d.h. ein Knoten d_i wird wieder in die Menge I aufgenommen, woraufhin alle von d_i abhängigen Knoten aus der Menge I entfernt werden müssen.

Der Algorithmus in Pseudocode 4.4 zeigt die inkrementelle Auswertung des Abhängigkeitsgraphen. Wir gehen zunächst davon aus, dass der Abhängigkeitsgraph bereits nach den oben beschriebenen Regeln konstruiert wurde. Die Funktionen `successors()` und `predecessors()` ermitteln jeweils die Vorgänger und Nachfolger eines Abhängig-

```
1  Set allDependencies = {d_1, ..., d_n};
2  Set I = independent = {d_i ∈ allDependencies | isIndependent(d_i)};
3  Stack removed = {};
4
5  function boolean isIndependent(DependencyNode d_i) {
6      Set predecessors = {predecessors(d_i) \ removed};
7      return predecessors.isEmpty();
8  }
9
10 function boolean canRemoveDependency(DependencyNode d_i) {
11     return independent.contains(d_i);
12 }
13
14 function removeDependency(DependencyNode d_i) {
15     removed.push(d_i);
16     independent.remove(d_i);
17     independent.add({d_q ∈ successors(d_i) | isIndependent(d_q)})
18 }
19
20 function undoRemoveDependency() {
21     DependencyNode d_undo = removed.pop();
22     independent.add(d_undo);
23     independent.remove(successors(d_undo));
24 }
```

Algorithmus 4.4: Auswertung des Abhängigkeitsgraphen

keitsknoten. Aus der Menge aller Abhängigkeitsknoten (*allDependencies*) werden initial alle unabhängigen Knoten (I = *independent*) berechnet (Zeile 02). Die Funktion `isIndependent()` bestimmt dazu, unter Beachtung der bereits entfernten Knoten (*removed*), ob der Knoten d_i aktuell keine Vorgänger besitzt. Ein Knoten lässt sich aus dem Abhängigkeitsgraph entfernen (`canRemoveDependency()`), sobald dieser in der Menge der unabhängigen Knoten (*independent*) vorkommt. Die Funktion `removeDependency()` entfernt den ausgewählten Abhängigkeitsknoten d_i. Um den Fortschritt auf dem Abhängigkeitsgraphen zu simulieren, wird der Knoten d_i aus der Menge der unabhängigen Knoten (*independent*) entfernt und auf dem Stack *removed* abgelegt. Danach wird die Menge um alle nun unabhängigen Knoten erweitert (Zeile 17). Dieser Vorgang kann solange fortgesetzt werden, bis keine unabhängigen Knoten mehr existieren.

Die Funktion `undoRemoveDependency()` legt die Abhängigkeitsknoten in der umgekehrten Reihenfolge wieder in den Abhängigkeitsgraphen zurück. Dazu wird der letzte Knoten d_{undo} vom Stack *removed* entfernt und wieder in die Menge der unabhängigen

```
1  function int pickVariables() {
2      Set pickable = {v_i ∈ remainingVariables | canRemoveDependency(𝔻(v_i))}
3      DependencyNode independent = 𝔻(pickAny(pickable))
4
5      for (Variable atomicVariable in 𝕍(independent)) {
6          pickedVariables.push(atomicVariable)
7          remainingVariables.remove(atomicVariable)
8      }
9      removeDependency(independent)
10     return |𝕍(independent)|
11 }
```

Algorithmus 4.5: Auswahl der Variablen

Knoten eingefügt (Zeile 21-22). Schließlich entfernt der Algorithmus noch alle Knoten, die von d_{undo} abhängen aus der Menge der unabhängigen Knoten (Zeile 23).

Die Einschränkungen des Abhängigkeitsgraphen sollen nun in den Matchingalgorithmus 4.1 übertragen werden. Hierfür wird der Matchingalgorithmus um den Algorithmus 4.4 zur Auswertung des Abhängigkeitsgraphen erweitert. Die Funktion ⟨4.5⟩`pickVariables()` wählt nun aus allen noch verbleibenden Variablen in *remainingVariables* eine Variable aus, die aktuell keine Abhängigkeit auf andere Variablen besitzt (Zeile 2-3). Die 𝔻-Funktion bildet zunächst eine Variable auf den Abhängigkeitsknoten der zugehörigen Aktion ab. Der Knoten wird dann durch die Funktion `canRemoveDependency()` in Algorithmus 4.4 auf noch vorhandene Abhängigkeiten geprüft. Aus der Menge *pickable* aller unabhängigen Variablen wird nun eine beliebige Variable ausgewählt (`pickAny()`). Ein Abhängigkeitsknoten kann mehrere Variablen (bzw. Aktionen) atomar zusammenfassen. Ist die ausgewählte Variable Teil einer atomaren Abhängigkeit (Zeile 3), dann werden zunächst durch die 𝕍-Funktion alle zusammenhängenden Variablen ermittelt (Zeile 5). Die neu ausgewählten Variablen werden nun auf dem Stack *pickedVariables* abgelegt (Zeile 6-7). Der entsprechende Abhängigkeitsknoten wird abschließend durch den Aufruf ⟨09⟩`removeDependency()` aus dem Abhängigkeitsgraphen entfernt, wodurch ggf. neue unabhängige Variablen entstehen.

Wurden durch die Funktion ⟨7⟩`pickVariables()` in Matchingalgorithmus 4.1 mehrere atomar zusammenhängende Variablen ausgewählt, dann sind diese Variablen auch gemeinsam zu belegen. Der Parameter *unassigned* der `expand`-Funktion zeigt an, wie viele Variablen ausgewählt, aber noch nicht belegt wurden. Erst wenn alle Variablen (`unassigned == 0`) belegt sind, wird die Lösung um die nächsten Variablen erweitert (Zeile 7). Falls nicht für alle zusammenhängenden Variablen eine Belegung existiert, d.h. wenn der Wertebereich einer Variablen leer ist, so wird diese Teillösung verworfen (Zeilen

```
1  function removeVariables(int count) {
2      for (i = 0; i < count; ++i) {
3          Variable v_atomic = pickedVariables.pop()
4          removedVariables.push(v_atomic)
5      }
6      undoRemoveDependency()
7  }
```

Algorithmus 4.6: Entfernen der Variablen

```
1  function addVariables(int count) {
2      for (i = 0; i < count; ++i) {
3          Variable v_atomic = removedVariables.pop()
4          remainingVariables.add(v_atomic)
5      }
6  }
```

Algorithmus 4.7: Einfügen der Variablen

14). Außerdem werden die zusammenhängenden Variablen immer gemeinsam aus dem Lösungsraum entfernt (Zeile 24-28)[2].

Nachdem alle weiteren Lösungen gefunden wurden, werden die zuletzt eingefügten Variablen durch die Funktion ⟨4.6⟩`removeVariables()` aus dem Lösungsraum entfernt (Zeile 2-5). Da die Variablen nun nicht mehr ausgewählt werden können, muss der zugehörige Abhängigkeitsknoten wieder in den Abhängigkeitsgraphen aufgenommen werden. Dies wird durch den Aufruf ⟨06⟩`undoRemoveDependency` erledigt. Dadurch werden ggf. auch potentiell auswählbare Variablen in der Menge *remainingVariables* blockiert, wenn diese von den gerade entfernten Variablen abhängen. Wodurch ggf. der Lösungsraum verkleinert wird. Nachdem alle Teillösungen ohne die entfernten Variablen betrachtet wurden (Zeile 25-26, Matchingalgorithmus 4.1), werden die Variablen wieder in die Menge *remainingVariables* eingefügt. Die Funktion ⟨4.7⟩`addVariables()` setzt dadurch die Berechnung wieder auf den Ausgangszustand zurück, woraufhin alle rekursiven Aufrufe der `expand`-Funktion schrittweise beendet werden. Der Matchingalgorithmus hat nun den gesamten Lösungsraum abgearbeitet und alle gefundenen Lösungen wurden gespeichert.

Für das Resultat des Matchingalgorithmus sind nur maximale Lösungen relevant, die nicht mehr erweitert werden können. Ansonsten würde der Algorithmus auch alle Teillösungen einer maximalen Lösung ausgeben. Das einfache Beispiel in Tabelle 4.1 enthält

[2]Die Prüfung (`unassigned == 0`) in Zeile 24 impliziert, dass neue Variablen ausgewählt wurden.

```
1  function boolean isAMaxSolution() {
2      Set pickable = {v_i ∈ removedVariables | canRemoveDependency(𝔻(v_i))}
3
4      for (Variable v_i in pickable) {
5          if (getDomain(v_i).isNotEmpty()) {
6              return false
7          }
8      }
9      return true
10 }
```

Algorithmus 4.8: Erkennung nicht maximaler Lösungen

nur Variablen mit Wertebereichen, die voneinander unabhängig sind. Daher wären hier tatsächlich nur die Ergebnisse ohne Platzhalter maximale Lösungen. Alle nicht maximalen Lösungen lassen sich filtern, indem für jede entfernte Variable (Platzhalter bzw. *removedVariables*) überprüft wird, ob noch Elemente zugewiesen werden können. Die Überprüfung, ob eine der maximalen Lösungen gefunden wurde, wird durch die Funktion ⟨29⟩ `isAMaxSolution()` durchgeführt. Die Funktion ermittelt, ob die gefundene Teillösung noch erweitert werden kann. Hierzu müssen alle Variablen überprüft werden, die bereits explizit aus der Teillösung entfernt wurden (*removedVariables*), aber ansonsten wieder aufgenommen werden könnten. Dies wird durch den Test ⟨4.4⟩`canRemoveDependency()` des Abhängigkeitsgraphen ermittelt (Zeile 2). Nun muss überprüft werden, ob die Wertebereiche der infrage kommenden Variablen leer sind (Zeile 5). Wird eine Variable gefunden, die zwar aus dem Lösungsraum entfernt wurde, allerdings die Teillösung noch erweitern kann, dann wird die aktuelle Lösung verworfen (`isAMaxSolution()` → *false*).

Die Überprüfung der nicht vollständigen Teillösungen wird erst dann benötigt, wenn die Zuweisung eines Elements an eine Variable ggf. zur Einschränkung von Wertebereichen anderer Variablen führt. Wir sprechen im Kontext des Constraint-Satisfaction-Problems von Constraints, die zwischen den Wertebereichen erfüllt sein müssen. Im Fall von Graphmustern handelt es sich im Wesentlichen um strukturelle Constraints, die sich aus den Elementen und Referenzen der Modelle ergeben. Im nächsten Abschnitt 4.3 werden wir die Einschränkung der Wertebereiche durch pfadbasierte Constraints einführen.

4.3 Matching von Änderungsmustern

Der vorgestellte Matchingalgorithmus 4.1 sucht nach allen Teilregeln einer Editierregel $\delta_g = \{\alpha_0 \times \ldots \times \alpha_n\}$, die bezüglich einer Differenz $D_{A \to B} = \{c_0, c_1, \ldots\}$ zwischen zwei Versionen V_A und V_B eines Modells ausgeführt wurden. Hierfür muss der Algorithmus

alle möglichen Zusammensetzungen der technischen Änderungen c_j mit den Aktionen α_i der Editierregel vergleichen. Die ausgewählten Aktionen der Editierregel δ_g müssen zusammen mit dem Kontextgraph der Teilregel $\delta_t \subset \delta_g$ einen verbundenen Aktionsgraphen ergeben. Hierbei ist zu beachten, dass nicht ausgewählte Knoten und Kanten mit löschenden Aktionen ggf. in Kontext der Teilregel umgewandelt werden. Knoten und Kanten mit erzeugenden Aktionen dürfen hingegen nur so ausgewählt werden, dass diese mit der bisher ausgewählten Teilregel verbunden sind (Abschnitt 3.1). Um die Verbindung zwischen den Aktionen sicherzustellen, wird in diesem Abschnitt eine strukturelle lokale Suche vorgestellt, welche den Lösungsraum des Constraint-Satisfaction-Problems, auf Basis der bereits ausgewählten technischen Änderungen, inkrementell einschränkt. Insbesondere bei größeren Differenzen wären ansonsten sehr viele mögliche Kombinationen von technischen Änderungen zu überprüfen.

Bei der Erkennung einer Editierregel auf Basis einer technischen Differenz kann der Kontext ggf. auf die beiden Modellversionen V_A und V_B verteilt sein. Wurden Teile des Kontexts gelöscht, dann kann dieser nur auf der historischen Modellversion V_A überprüft werden. Wurde der Kontext hingegen neu erzeugt, so kann nur auf der überarbeiteten Modellversion V_B überprüft werden, ob bestimmte Änderungen verbunden sind. Das Änderungsmuster einer Editierregel verbindet dazu die Kontextknoten für V_A und V_B über die Korrespondenzen der Differenz. Abbildung 4.4 zeigt das Muster, mit dem zwei Modellelemente in einer Differenz als korrespondierend festgelegt werden.

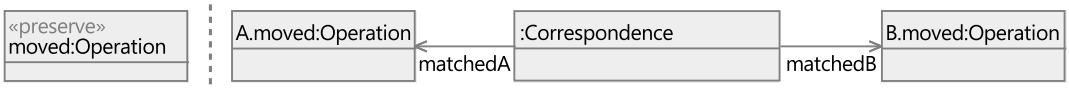


Abbildung 4.4: Korrespondenzmuster (Editierregel Abbildung 2.4)

Die einzelnen Änderungen einer Lösung, die durch den Matchingalgorithmus 4.1 gefunden wurden, müssen sich also über die Modelle V_A oder V_B verbinden lassen. Abbildung 4.5 zeigt eine Editierregel und das abgeleitete Änderungsmuster sowie den zugehörigen Abhängigkeitsgraphen in einer integrierten Darstellung. Die Editierregel aus Abbildung 2.4 verschiebt eine Operation zwischen zwei Klassen eines Klassendiagramms. Parallel werden die Empfänger aller Nachrichten der Sequenzdiagramme angepasst, welche die Operation als Signatur angeben. Eine Nachricht muss von der Lebenslinie eines Objekts empfangen werden, dessen Typ bzw. Klasse die angegebene Operation enthält. Für die Berechnung der Teilregeln werden die Anwendungsbedingungen zunächst vernachlässigt (Abschnitt 3.1). Außerdem wird die Multiregel als einfacher Teil der Kernregel interpretiert. Letzteres kann ggf. dazu führen, dass eine mögliche Autovervollständigung

mehrfach ausgegeben wird, falls die Multi-Regel mehrfach als Teilregel mit derselben Komplementregel ausgeführt wurde. Dies würde hier beispielsweise auftreten, wenn die Empfänger mehrerer Nachrichten (mit derselben Signatur) ausgetauscht werden, ohne die Operation entsprechend zu verschieben. Es handelt sich hierbei aber nur um eine Vereinfachungen der Editierregel, es werden dadurch keine Lösungen ausgeschlossen.

Der Aktionsgraph einer Editierregel δ_g besteht aus einer Menge von Knoten $\{t_0, \ldots, t_p\}$ und Kanten $\{e_0, \ldots, e_q\}$, welche mit Aktionen $\{\alpha_0, \ldots, \alpha_n\}$ annotiert sind (Abschnitt 2.3). Jeder Kontextknoten und löschende Knoten t_i des Aktionsgraphen ist ein potentieller Kontextknoten für eine Teilregel von δ_g (z.B. Teilregel in Abbildung 3.3). Diese Knoten werden im Änderungsmuster als Korrespondenzmuster repräsentiert (Abbildung 4.4). Im kompakten Änderungsmuster in Abbildung 4.5 wird jedes Korrespondenzmuster stellvertretend durch einen einzelnen Knoten t_i mit einem zweidimensionalen Wertebereich $t_i[\sigma_0, \sigma_1, \ldots][\sigma_0, \sigma_1, \ldots]$ für Modell V_A und V_B dargestellt. Knoten t_j mit erzeugenden Aktionen benötigen hingegen nur einen eindimensionalen Wertebereich $t_j[\sigma_0, \sigma_1, \ldots]$ für Modell V_B. Ein Wertebereich kann Modellelemente des entsprechenden Typs (bzw. Subtyps) aus Modellversion V_A bzw. V_B aufnehmen. Die zweidimensionalen Wertebereiche $t_i.A = [\sigma_0, \sigma_1, \ldots]$ und $t_i.B = [\sigma_0, \sigma_1, \ldots]$ sind bei jeder Veränderung miteinander zu synchronisieren. Wird ein neues Modellelement aus Modell V_A in den Wertebereich $t_i.A$ eingefügt oder entfernt, so ist das korrespondierende Modellelement aus Modell V_B in $t_i.B$ einzufügen bzw. zu entfernen, falls eine entsprechende Korrespondenz existiert. Dies gilt vice versa für Veränderungen des Wertebereichs $t_i.B$. Man kann sich dies veranschaulicht als Vereinigung der beiden Modelle V_A und V_B vorstellen.

Die Variablen v_i, welche die technischen Änderungen repräsentieren, beziehen sich zum einen auf die Wertebereiche $t_i.A$ und $t_i.B$ der Kontextknoten bzw. Aktionsknoten und zum anderen auf die Typen der entsprechenden ($\alpha_i \mathrel{\widehat{=}} v_i$) Aktionen (vgl. Änderungsmuster Abbildung 4.2 und 4.3). Zwei Aktionen α_x und α_y einer Editierregel δ_g müssen minimal über einen Pfad des Aktionsgraphen miteinander verbunden sein. Ein **Pfad** $T = \{t_0, t_1, \ldots\}$ entspricht einer Reihe von adjazenten Knoten t_i des Aktionsgraphen einer Editierregel. Ein Pfad kann daher sowohl über Kontextknoten als auch über löschende/erzeugende Knoten verlaufen. Jeweils zwei adjazente Knoten eines Pfades bezeichnen wir als **Teilpfad**. Die Richtung und Anzahl der Kanten ist für die Auswahl eines Teilpfads aus dem Aktionsgraphen nicht relevant, d.h. auch wenn mehrere Kanten zwischen zwei Knoten existieren, wird dies nur als ein möglicher Teilpfad betrachtet.

Ausgehend von einer Aktion lassen sich alle möglichen Pfade über eine Tiefensuche (*depth-first search, DFS*) auf dem Aktionsgraphen ermitteln. Die Operation $dfs(\alpha_x) = \{t_0, t_1, \ldots\}$ ermittelt iterativ alle Teilpfade $T_i = \{t_i, t_{i+1}\}$ ausgehend von der Aktion α_x.

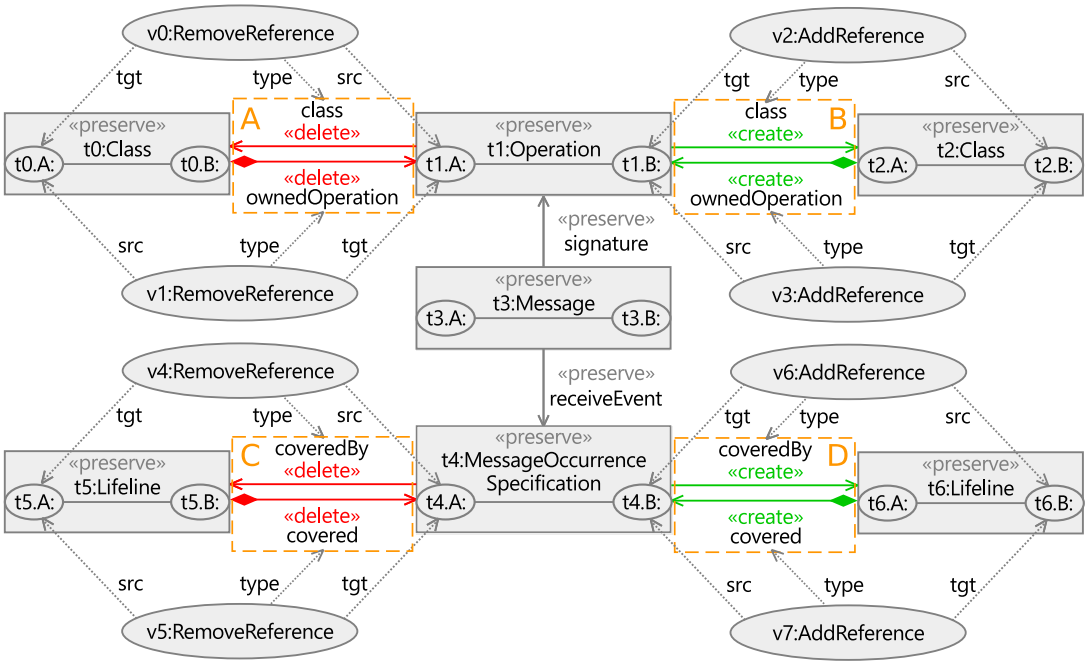


Abbildung 4.5: Visualisierung des Änderungsmusters zur Editierregel aus Abbildung 2.4

Ein maximaler Pfad innerhalb der Tiefensuche darf dabei keinen Zyklus bilden, d.h. ein Knoten darf in einem Pfad ausgehend von Startknoten t_0 nicht doppelt vorkommen. Für die Editierregel bzw. den Aktionsgraph aus Abbildung 4.5 ergibt sich beispielsweise für die Aktion $\alpha_1 = (delete, ownedOperation)$ bei einer Tiefensuche $dfs(\alpha_1)$ die Folge der Knoten $\{t_0, t_1, t_2, t_3, t_4, t_5, t_6\}$.

Der Matchingalgorithmus 4.1 belegt die Variablen v_x schrittweise mit Elementen, d.h. in jedem Schritt wird eine technische Änderung c_r passend zu einer Aktion α_x der Editierregel ausgewählt ($\alpha_x \mathrel{\widehat{=}} v_x[c_r]$). Nach jeder Belegung einer Variablen v_x werden die Wertebereiche der Knoten t_i und der anderen Variablen v_y inkrementell eingeschränkt. Für jeden Teilpfad ausgehend von der Variablen v_x sind alle möglichen Abbildungen in die Differenz $D_{A \to B}$ bzw. in Modellversion V_A und V_B zu berechnen. Dazu wird jeder Teilpfad $T_i = \{t_i, t_{i+1}\}$, der durch die Tiefensuche $dfs(\alpha_x) = \{t_0, t_1, \ldots\}$ ermittelt wird, schrittweise ausgewertet. Die beiden adjazenten Knoten t_i und t_{i+1} sind über eine Menge von Kanten $E = \{e_0, e_1, \ldots\}$ verbunden. Die Auswertung einer Kante e_i wird als **Auswertungsschritt** $t_i \xrightarrow{e_j} t_{i+1}$ eines Teilpfades bezeichnet. Das heißt, es werden immer zwei Knoten und eine Kante der Editierregel bzw. des Aktionsgraphen betrachtet. Für diesen Teilpfad wird dann das entsprechende partielle Änderungsmuster bestimmt

und auf die Differenz bzw. Modell V_A und V_B abgebildet. Daraus ergibt sich der folgende Algorithmus $\mathbb{S}(v_x, c_r)$ zur Einschränkung (*restriction*) des Lösungsraums:

1. **Zuweisung einer Variablen:** Für eine Variable v_x wird eine technische Änderung c_r aus dem entsprechenden Wertebereich ausgewählt und zugewiesen. Zur Sicherstellung der Injektivität der Lösung wird c_r aus den Wertebereichen der anderen Variablen entfernt (Abschnitt 4.1). Ein Aktionsknoten (Abbildung 4.3) legt eindeutig den Startknoten t_0 der Tiefensuche $dfs(\alpha_x)$ mit der Aktion $\alpha_x \mathrel{\widehat{=}} v_x[c_r]$ fest. Eine Aktionskante (Abbildung 4.2) legt den Quell- und Zielknoten der Kante als Startknoten t_0, t_1 fest, die Kante selbst muss aber nicht nochmal ausgewertet werden.

2. **Auswertungsschritt:** Für jeden Teilpfad $T_i = t_i \xrightarrow{E} t_{i+1}$ der Tiefensuche $dfs(\alpha_x)$ $= \{t_0, t_1, \ldots\}$ werden alle Kanten $e_j \in E$ und der Zielknoten t_{i+1} ausgewertet. Die ermittelten Modellelemente und technischen Änderungen werden in den entsprechenden Wertebereichen zunächst nur markiert. Die verschiedenen Pfade können sich dadurch ggf. überlagern:

   a) **Aktion bestimmen:** Zunächst wird ggf. die Aktion $\alpha_y \in \{delete, create\}$ der Kante e_j ausgewertet. Dazu werden alle technischen Änderungen (*RemoveReference, AddReference*) im Wertebereich $v_y[c_0, c_1, \ldots]$ markiert, die über Modellelemente im Wertebereich $t_i.A$ bzw. $t_i.B$ des Startknotens t_i erreichbar sind, d.h. die Modellelemente sind entweder Quelle (*src*) oder Ziel (*tgt*) einer gelöschten oder erzeugten Referenz. Ausgehend von den markierten technischen Änderungen, werden dann umgekehrt alle Modellelemente in den Wertebereichen $t_{i+1}.A$ bzw. $t_{i+1}.B$ des Zielknotens t_{i+1} markiert.

   Anschließend wird ggf. die Aktion $\alpha_z \in \{delete, create\}$ des Zielknotens t_{i+1} ausgewertet, d.h. es werden ebenfalls die technischen Änderungen (*AddObjekt, RemoveObjekt*) im Wertebereich v_z markiert, welche Modellelemente im Wertebereich $t_i.A$ bzw. $t_i.B$ des Zielknotens t_{i+1} als gelöscht bzw. erzeugt annotieren (*obj*).

   b) **Kontext bestimmen:** Für alle Kanten mit einer Aktion $\alpha_y \in \{delete, preserve\}$ müssen, ausgehend vom aktuellen Startknoten, alle erreichbaren Modellelemente des Zielknotens ermittelt werden. Ist der Zielknoten ein Kontextknoten ($\alpha_y = preserve$), so sind zusätzlich die Wertebereiche $t_i.A$ und $t_i.B$ über die Korrespondenzen der Differenz zu synchronisieren. Nachdem alle er-

reichbaren Modellelemente über die Kante e_j ausgelesen wurden, startet ggf. der nächste Auswertungsschritt für die Kante e_{j+1}.

3. **Einschränkung des Lösungsraums:** Nachdem alle Pfade $dfs(\alpha_x)$, ausgehend von der Variablen $v_x \mathrel{\widehat{=}} \alpha_x$, ausgewertet wurden, werden aus allen Wertebereichen die *nicht* markierten Elemente entfernt. Die entfernten Elemente werden für jeden Wertebereich ($t_i.A$, $t_i.B$ oder v_j) auf einem eigenen Stapel gespeichert. Der Lösungsraum wird somit bei jeder Zuweisung einer Variablen v_x inkrementell eingeschränkt.

4. **Freigeben einer Variable:** Wird eine Variable v_x wieder freigegeben, müssen auch die zuvor gesetzten Einschränkungen der Wertebereiche wieder aufgehoben werden. Die Aufhebung erfolgt immer in umgekehrter Reihenfolge zur Einschränkung. In einem Backtracking-Schritt des Matchingalgorithmus 4.1 werden alle Einschränkungen, die bezüglich der Variablen v_x gemacht wurden, durch den Aufruf $\mathbb{S}^{-1}(v_x, c_r)$ in der Funktion ⟨4.3⟩`freeVariable()` wieder zurückgesetzt.

Der Lösungsraum wird durch den Matchingalgorithmus 4.1 schrittweise eingeschränkt, bis alle möglichen Variablen belegt wurden. Die Einschränkung $\mathbb{S}(v_x, c_r)$ wird durch die Funktionen ⟨4.2⟩`assignVariable()` nach der Belegung einer Variablen ausgeführt[3]. Schließlich enthält der Lösungsraum nur die Elemente, die Teil einer partiellen Abbildung des Änderungsmusters auf die technische Differenz sind. Die partiellen Abbildungen verbinden die ausgewählten Änderungen über Modell V_A und V_B. Der Lösungsraum enthält somit die Vereinigung aller möglichen partiellen Abbildungen für den Kontext einer Teilregel. Dadurch lassen sich zwar die einzelnen Abbildungen nicht direkt ablesen, es kann aber bereits die Aussage getroffen werden, dass mindestens eine partielle Abbildung existiert, welche die Lösung des Constraint-Satisfaction-Problems strukturell miteinander verbindet. Eine exakte Prüfung des Kontexts wird zunächst auf die Komplementregel verlagert (Abschnitt 3.2). Ein Vorteil der pfadbasierten Analyse besteht darin, dass nur zusammenhängende Lösungen durch den Matchingalgorithmus betrachtet und erzeugt werden. Der Lösungsraum wird somit auf die lokale Struktur der bereits selektierten technischen Änderungen begrenzt. Durch die Restriktion $\mathbb{S}(v_x, c_r)$ kann auch der gesamte Wertebereich einer anderen Variable eingeschränkt werden, d.h. kein Element

[3]Optimierung: $\mathbb{S}(v_x, c_r)$ muss nur einmal für atomar abhängige Aktionen ausgeführt werden.

dieses Wertebereichs ist von der aktuellen Lösung des Constraint-Satisfaction-Problems aus erreichbar. Damit wird diese Variable aus dem Lösungsraum des CSPs entfernt.

4.4 Heuristisches Matching

Die Auswertung eines rekursiven Algorithmus lässt sich als Baum darstellten, wobei jede Ausführung der Funktion einen Knoten und jeder rekursive Aufruf eine Kante darstellt. Wird ein Funktionsaufruf ausgesetzt, dann werden auch alle nachfolgenden rekursiven Aufrufe übersprungen. Der Baum stellt den Lösungsraum des Algorithmus dar. Um den Lösungsraum einzugrenzen, muss nachvollzogen werden, welche korrekten Lösungen in welchem Teilbaum liegen. Lassen sich Teilbäume identifizieren, die keine korrekten Lösungen enthalten können, dann handelt es sich um eine exakte Optimierung. Die in den vorherigen Abschnitten beschriebenen Einschränkungen sind exakte Optimierungen, da hier keine korrekte Lösung ausgeschlossen wird. Werden korrekte Lösungen durch die Vermeidung eines Teilbaums ggf. nicht gefunden, handelt es sich hingegen um eine heuristische Optimierung. Bei einer Heuristik nimmt man ggf. den Verlust einzelner Lösungen in Kauf, um eine schnellere Berechnung zu ermöglichen.

Ein Werkzeug zur Autovervollständigung (bzw. Reparatur in Kapitel 5 und 6) von Modellen soll einen bestimmten Editierschritt automatisieren. Je länger die Berechnung dauert, desto unwahrscheinlicher wird es, dass das Ergebnis noch Relevanz für den Entwickler besitzt. Außerdem muss schließlich ein Ranking der gefundenen Lösungen erstellt werden, um dem Entwickler die Auswahl zu erleichtern. Daher wäre es akzeptabel, wenn die Lösungen, welche eine relativ schlechte Bewertung erhalten, gar nicht erst durch den Matchingalgorithmus berechnet werden. Eine Editierregel, die eine große Überschneidung mit der Differenz besitzt, ist ein gutes Indiz dafür, dass diese Regel eine wahrscheinliche Ergänzung des aktuellen Modellzustands darstellt. Außerdem führt eine große Teilregel (bezüglich der Gesamtregel) zu einer kleinen Komplementregel, was das *„Principle of Least Change“* [Mee98] unterstützt. Daher werden im Folgenden Heuristiken vorgeschlagen, die dazu führen, dass der Matchingalgorithmus 4.1 primär nach möglichst großen Lösungen sucht.

Minimale Lösungen: Um kleinere Lösungen auszublenden lässt sich ein Parameter s_{min} einführen, der die minimale Größe der zu suchenden Lösungen festlegt [KH04]. Sobald nur noch Lösungen gefunden werden können, die weniger als s_{min} Variablen besitzen, wird dieser Teil des Lösungsraums verlassen. Hierfür wird ein globaler Parameter eingeführt, der die Minimalgröße als Prozentsatz bezüglich der Anzahl der Aktionen der

Editierregeln angibt. Der Entwickler kann somit das Ergebnis des Matchingalgorithmus mit wenig Konfigurationsaufwand steuern.

Der Matchingalgorithmus 4.1 verkleinert die Lösung sobald Variablen aus dem Lösungsraum entfernt werden (⟨25⟩`removeVariables()`). Nach dem Entfernen der Variablen muss die maximal mögliche Größe der Lösung abgeschätzt werden. Liegt die abgeschätzte Größe unter der angegebenen Minimalgröße, dann wird die Suche an dieser Stelle nicht fortgesetzt (⟨26⟩`expand()`). Für eine präzisere Abschätzung wird der Algorithmus zunächst erweitert. Wird eine Variable aus dem Lösungsraum entfernt, so werden auch alle abhängigen Variablen entfernt. Die Abhängigkeiten lassen sich (statisch) aus dem Abhängigkeitsgraphen ermitteln. Die maximal mögliche Größe ergibt sich dann aus den bereits belegten Variablen (*pickedVariables*) und den noch auswählbaren Variablen (*remainingVariables*), deren Wertebereich noch mindestens ein Element enthält.

Vermeidung unvollständiger Lösungen: Zur Optimierung des Algorithmus muss möglichst frühzeitig erkannt werden, ob eine Teillösung noch zu einer korrekten Lösung ergänzt werden kann. Eine Lösung ist – im Sinne des Constraint-Satisfaction-Problems in Abschnitt 4.1 – nicht korrekt, wenn die Lösung unvollständig ist. Diese nicht maximalen Lösungen werden durch die Funktion ⟨4.8⟩`isAMaxSolution()` ausgefiltert. Die unvollständigen Lösungen entstehen dadurch, dass zu jeder maximalen Lösung auch alle Teillösungen untersucht werden müssen, um herauszufinden, ob sich diese zu einer anderen Lösung ergänzen lassen. Eine unvollständige Lösung ist also immer Teil einer bereits gefundenen maximalen Lösung. Es wird daher eine Strategie benötigt, mit der abgeschätzt werden kann, ob zu einer Teillösung noch *neue* Lösungen existieren.

Eine mögliche Strategie besteht darin, Teillösungen nur dann zu betrachten, wenn im aktuellen Lösungsraum noch Elemente vorhanden sind, die in keiner bisherigen Lösung vorgekommen sind. Dazu merkt sich der Matchingalgorithmus pro Wertebereich einer Variablen, welche Elemente bereits Teil einer Lösung waren. Nachdem der Matchingalgorithmus 4.1 die Variablen (⟨25⟩`removeVariables()`) entfernt hat, wird überprüft, ob noch nicht verwendete Elemente in den verbleibenden Variablen existieren. Wenn dieser Test positiv ausfällt wird die Rekursion ⟨26,19⟩`expand()` fortgesetzt; ansonsten wird die Teillösung verworfen.

Bei dieser Heuristik ist es für eine bessere Abschätzung ebenfalls sinnvoll, Variablen gemeinsam mit ihren abhängigen Variablen zu entfernen (⟨25⟩`removeVariables()`). Zusätzlich wird die Auswahl der Variablen so angepasst, dass immer die Variable als nächstes ausgewählt wird (⟨07⟩`pickVariables`), welche die meisten noch nicht verwendeten Elemente besitzt. Die Heuristik stellt zumindest sicher, dass jedes Element in mindestens

einer Lösung vorkommt. Daraus folgt, dass Lösungen, die sich vollständig mit mehreren anderen Lösungen überlappen, ggf. nicht gefunden werden.

5 Behandlung von Inkonsistenzen

Die Erkennung und Behandlung von Inkonsistenzen gehört zu den grundlegenden Tätigkeiten der Softwaremodellierung. Die Entwickler müssen daher über entsprechende Werkzeuge verfügen, um die Aufgaben, die im Zusammenhang mit der Konsistenzverwaltung (*consistency management*) anfallen, bewältigen zu können [LMT09]. Es existieren bereits einige Konzepte und Werkzeuge, mit denen Konsistenzregeln formuliert und Inkonsistenzen effizient erkannt werden können [NEF03, BMMM09, RE12b]. Wir werden uns daher in diesem Kapitel im Wesentlichen mit der Behandlung von Inkonsistenzen beschäftigen. Die Behandlung einer Inkonsistenz kann auf verschiedene Arten erfolgen. Nuseibeh et al. [NER00] beschreiben beispielsweise einen Prozess, bei dem Inkonsistenzen temporär toleriert werden können. Dies kann insbesondere in frühen Entwicklungsphasen nützlich sein, wenn noch nicht genügend Informationen zur Reparatur der Inkonsistenz vorliegen. Falls eine Inkonsistenz fälschlicherweise erkannt wurde, kann es auch erforderlich sein, die Konsistenzregel anzupassen oder eine Ausnahme für diese Regel festzulegen. Temporär tolerierte Inkonsistenzen müssen jedoch zu einem späteren Zeitpunkt repariert werden. Eine Inkonsistenz, die nicht sofort repariert werden kann, sollte entsprechend dokumentiert werden. Somit kann der Entwickler während einer Reparatur einfacher nachvollziehen, wie eine Inkonsistenz entstanden ist. Die Ursachen für die Entstehung von Inkonsistenzen können vielfältig sein:

> *„(i) different models may be developed in parallel by different persons; (ii) the interdependencies between models may be poorly understood; (iii) the requirements may be unclear or ambiguous at an early design stage; (iv) the models may be incomplete because some essential information may still be unknown.“ [MVDSD06]*

Für eine Inkonsistenz lässt sich sowohl ein räumlicher als auch ein zeitlicher Ursprung angeben. Der **räumliche Ursprung** einer Inkonsistenz bezeichnet im Folgenden die Elemente und Eigenschaften (Referenzen oder Attribute) innerhalb des Modells, an denen die Auswertung einer Konsistenzregel scheitert. Zur Bestimmung der betroffenen Modellelemente muss die Auswertung der Konsistenzregel analysiert werden [RE12a].

Für das in diesem Kapitel beschriebene Reparaturwerkzeug ist es außerdem erforderlich den **zeitlichen Ursprung** innerhalb einer Modellhistorie zu kennen, in der eine Inkonsistenz aufgetreten ist. Es wird zunächst die Version (V_A) des Modells benötigt, unmittelbar bevor die Inkonsistenz auftritt, um festzustellen, welche Änderungen die Inkonsistenz ausgelöst haben. Von den nachfolgenden Versionen wird nur die aktuellste Version (V_B) benötigt, welche den zu reparierenden Zustand des Modells darstellt. Eine Inkonsistenz wird dabei durch die zugehörige Konsistenzregel und das Kontextelement, über mehrere Modellversionen hinweg, identifiziert.

Die meisten der in Abschnitt 2.2 vorgestellten Verfahren ermitteln Reparaturen nur auf Basis des aktuellen Modellzustands. Dabei gehen aber wichtige Informationen für die Berechnung der Reparaturen verloren. Die hier ermittelten Reparaturen beachten insbesondere die zuvor durchgeführten Modifikationen des Modells. Indem man die Modellhistorie hinzuzieht, lässt sich sowohl die Ursache einer Inkonsistenz als auch die Intention des Entwicklers, bezüglich der durchgeführten Änderungen, besser verstehen. Eine Inkonsistenz kann z. B. dadurch entstehen, dass Ausschnitte eines Modell gelöscht oder hinzugefügt wurden. Falls diese Änderungen auch andere Teile des Modells betreffen, kann es leicht passieren, dass der Entwickler diese Teile übersieht und nicht konsistent anpasst. Inkonsistenzen können auch auf ähnliche Weise durch das Verschieben oder Umbenennen von Modellfragmenten entstehen.

Die Modellversion V_B in Abbildung 5.1 zeigt einen inkonsistenten Zwischenzustand, zu der bereits in Kapitel 2 vorgestellten Modellierung eines Video-on-Demand-Systems (VoD). Ausgehend von der Modellversion V_A in Abbildung 2.1 wurde die Operation `disconnect()` aus der Video- in die Server-Klasse verschoben. Aus Sicht der Nachricht `4:disconnect` fehlt nun die entsprechende Operationssignatur in der Video-Klasse. Ein ähnliches Problem entsteht für die neu eingefügte Transition `pause`. Die Transition erwartet ebenfalls eine entsprechende Operation in der Klasse `Video`. Wie das Beispiel 5.1 zeigt, lässt sich auf Basis des aktuellen Modellzustands nicht mehr erkennen, ob die Inkonsistenzen durch das Löschen, Hinzufügen oder Verschieben von Elementen entstanden sind.

In Entwicklungsumgebungen, welche auf eine bestimmte Programmiersprache spezialisiert sind, werden häufig handgeschriebene Reparaturroutinen eingesetzt. Dieses Vorgehen ist, insbesondere für die Entwicklung von domänenspezifischen Modellierungssprachen, aufwendig und meist unvollständig. Ein Reparaturwerkzeug sollte daher entsprechende Reparaturen automatisch bestimmen können. In der Literatur wurden bereits verschiedene Ansätze vorgeschlagen, welche Reparaturen entweder durch die Analyse von Konsistenzregeln oder durch eine Zustandsraumsuche, ausgehend von den defek-

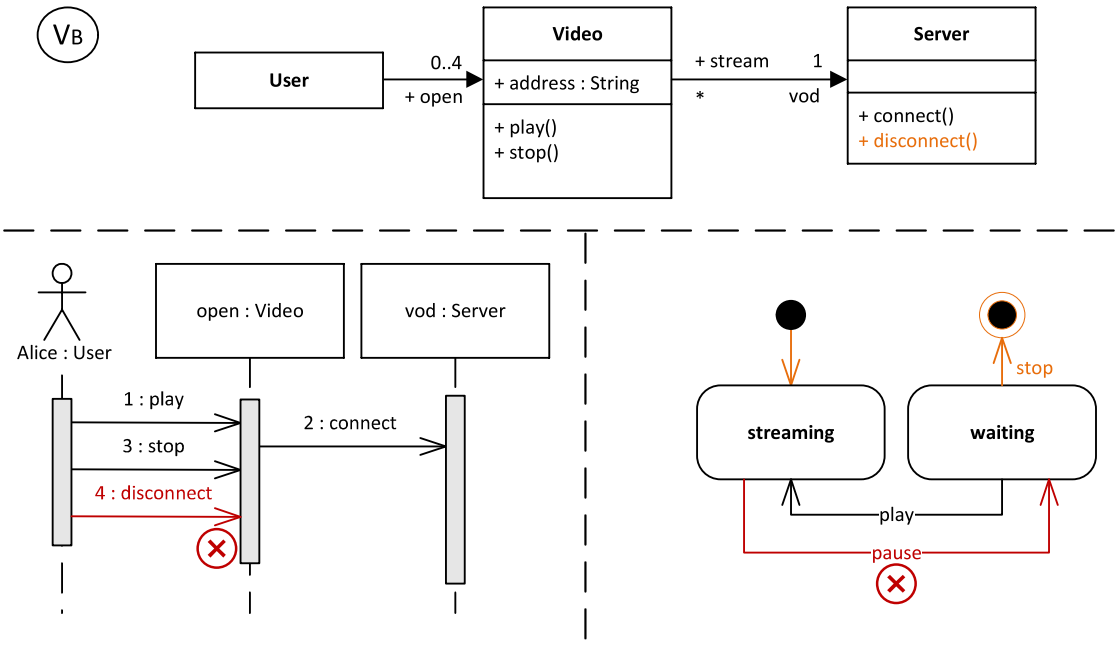


Abbildung 5.1: Entwurf eines VoD-Systems - Version B

ten Modellen, generieren (Abschnitt 2.2). Der hier beschriebene Ansatz zur Reparatur von Modellen verwendet sowohl eine Technik der syntaxbasierten Konsistenzregelanalyse als auch ein regelbasiertes Verfahren, welches Ähnlichkeiten mit einer Zustandsraumsuche besitzt. In einem ersten Schritt wird die Inkonsistenz analysiert, um Ansatzpunkte innerhalb des Modells für mögliche Reparaturen zu finden. Anschließend wird die Modellhistorie nach *unvollständig* ausgeführten Editierschritten durchsucht, die sich als mögliche Ursache der Inkonsistenz identifizieren lassen. Ähnlich wie bei einer Zustandsraumsuche, wird dann nach *vervollständigenden* Editierschritten gesucht, welche die Inkonsistenz reparieren. Der Unterschied zu anderen zustandsraumbasierten Suchverfahren besteht darin, dass nach einem konsistenten Editierschritt ausgehend von der historischen Modellversion gesucht wird. Die aktuelle Modellversion bildet hier einen erwarteten Zwischenzustand und nicht den Startzustand der Suche. Die Suche nach unvollständig ausgeführten Editierschritten erfolgt durch die partielle Erkennung des Änderungsmusters einer Editierregel in der Modellhistorie und nicht durch die Erzeugung und Untersuchung aller möglichen Folgezustände eines Modells.

Das Verfahren hat gegenüber einer reinen syntaxbasierten Analyse außerdem den Vorteil, dass die speziellen Eigenschaften eines Modelltyps durch benutzerdefinierte Editierregeln beachtet werden können. Zur Konfiguration des Reparaturwerkzeugs werden daher

alle (relevanten) domänenspezifischen Editierregeln benötigt. Die Menge aller Editierregeln wird im Folgenden als Regelbasis bezeichnet. Die Erstellung der Regelbasis erfolgt durch einen Domänenexperten, der mit den Details der Modellierungssprache vertraut ist und das Werkzeug für den Entwickler konfiguriert. Der Reparaturalgorithmus stellt keine Anforderungen an die Vollständigkeit oder das Konsistenzlevel (z. B. nur Regeln für eine Modellsicht oder sichtenübergreifende Regeln) der Regelbasis. Interessant sind insbesondere solche Regeln, die aufgrund ihrer Komplexität häufig zu Inkonsistenzen führen. Dies können z. B. Editierregeln sein, welche mehrere Sichten gleichzeitig editieren oder solche die komplexe Teilstrukturen erzeugen.

5.1 Konsistenz und Validierung der Modelle

Im Kontext der Softwaremodellierung behandelt der Begriff der Konsistenz die umfassende Korrektheit eines Modells. Hierbei kann es sich sowohl um formal definierte syntaktische Bedingungen als auch um in natürlicher Sprache formulierte semantische Anforderungen handeln [SC02]. In den nachfolgenden Betrachtungen wird Konsistenz immer auf formale und damit automatisch überprüfbare Bedingungen bezogen. Ein Modell gilt als **konsistent** (Abbildung 5.2), wenn es eine Menge von Konsistenzregeln erfüllt. Für jede Konsistenzregel wird gefordert, dass diese einen **Kontexttyp** Γ definiert. Eine Konsistenzregel wird auf alle Elemente σ des Modells angewendet, die dem angegebenen Kontexttyp entsprechen, d.h. σ ist eine Instanz des Typs Γ oder eines Subtyps von Γ. Die Auswertung einer Konsistenzregel, ausgehend von einem **Kontextelement** σ, wird als **Validierung** bezeichnet. Ein negatives Ergebnis der Validierung markiert das Auftreten einer Inkonsistenz am Kontextelement σ. Die Konsistenzregeln werden als Ausdrücke einer auf Prädikatenlogik erster Stufe (*first-order logic, FOL*) basierenden Sprache formuliert. Das vorzustellende Verfahren lässt sich somit auch auf Sprachen, wie beispielsweise die Object Constraint Language (OCL), übertragen. Die Konsistenzregel 2.1 erkennt z. B. die Inkonsistenz der Nachricht `4:disconnect` in Abbildung 5.1.

Zur Abgrenzung des hier verwendeten Konsistenzbegriffs wird zunächst ein minimales Konsistenzlevel der betrachteten Modelle definiert. Ein Modell das dieses Konsistenzlevel erfüllt liegt gemäß Abbildung 5.2 innerhalb des **inkonsistenten oder konsistenten Zustandsbereichs**. Dazu wird minimal gefordert, dass ein Modell konform zu seinem Metamodell ist. Das Metamodell definiert alle Typen (Metaklassen), Attribute und Referenzen aus denen eine Modellinstanz aufgebaut werden kann. Die Multiplizitäten des Metamodells werden zunächst abgeschwächt, d.h. alle [1..1] und $[n..m]$ mit $(m > 1)$ Multiplizitäten werden zu [0..1] und $[0..*]$ Multiplizitäten verallgemeinert. Anders formuliert

wird nur zwischen (optionalen) Feldern und Listen unterschieden. Des Weiteren müssen entgegengerichtete Referenzen (*opposites*) konsistent angelegt sein. Strukturell muss außerdem die vorgegebene Kompositionsstruktur (Eltern-Kind-Beziehungen der Elemente) eingehalten werden, wobei alle Elemente des Modells einen Baum bilden. Auf diese Weise ist sichergestellt, dass ein Modell in Form seines abstrakten Syntaxbaums dargestellt und verarbeitet werden kann.

Ein Modell wird als **defekt** betrachtet, wenn das minimale Konsistenzlevel unterschritten wird. Beispielsweise können nicht auflösbare Referenzen, unbekannte Typen (Metaklassen) oder XML-Parserfehler dazu führen, dass ein Modell (partiell) defekt wird. Die Behandlung von Defekten liegt außerhalb der hier betrachteten Reparaturmechanismen. Ein Modell kann ggf. trotzdem verarbeitet und repariert werden, indem partielle Defekte, wie beispielsweise eine nicht auflösbare Referenz, zunächst aus dem Modell entfernt werden. Modellierungstechnologien, wie z. B. das Eclipse Modeling Framework [SBMP08], bieten hierfür i.d.R. entsprechende generische Rettungsmechanismen an.

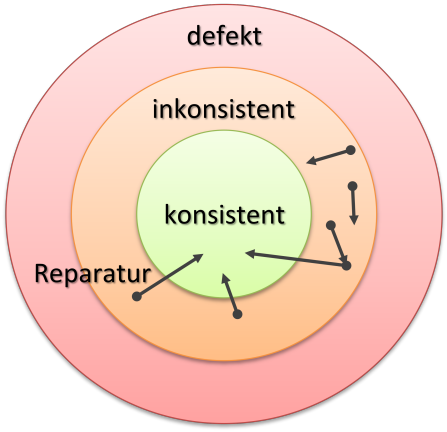


Abbildung 5.2: Konsistenzlevel eines Modells

Es lässt sich leicht bestimmen, ob ein Sprung zwischen den Konsistenzleveln stattgefunden hat. Hierzu wird jeweils eine Menge der Inkonsistenzen betrachtet. Der Entwickler könnte beispielsweise eine Teilmenge der zu reparierenden Inkonsistenzen auswählen. Ein Modell wird konsistent, wenn keine der Inkonsistenzen mehr erkannt wird. Ein Sprung zwischen den Konsistenzleveln lässt sich somit einfach durch eine erneute Validierung der entsprechenden Konsistenzregel ermitteln. Problematischer ist die Einschätzung einer Modifikation des Modells, die eine Inkonsistenz nicht vollständig behebt, sondern diese nur ausbessert. Intuitiv lässt sich eine Reparatur als eine Folge von Änderungen beschreiben, welche die Validierung einer Konsistenzregel zumindest minimal verbessert.

Demnach muss eine Reparatur eine Inkonsistenz nicht vollständig beheben, d.h. das Ergebnis der Validierung verändert sich ggf. nicht. Es muss allerdings festgestellt werden, ob eine Folge von Änderungen eine positive Auswirkung auf die Validierung besitzt. In solchen Fällen kann die Reparatur als ein inkrementeller Prozess aufgefasst werden, der sich aus mehreren aufeinander folgenden Reparaturen zusammensetzt.

5.2 Reparaturalgorithmus

Abbildung 5.3 gibt einen Überblick über die Berechnungsschritte des Reparaturwerkzeugs. Das Werkzeug muss zunächst mit den Konsistenzregeln und den domänenspezifischen Editierregeln konfiguriert werden. Zur Analyse der Konsistenz wird zunächst nur das aktuelle Modell V_B benötigt. Nach Abschluss der Validierung liegt die Menge aller in V_B enthaltenen Inkonsistenzen vor. Der Benutzer muss nun eine Teilmenge der Inkonsistenzen auswählen, die repariert werden sollen. Alle ausgewählten Inkonsistenzen werden dann einer syntaktischen Analyse unterzogen, um festzustellen, welche Modellelemente an der Inkonsistenz beteiligt sind. Im Zusammenhang mit der Konsistenzregel lassen sich dadurch alle Ansatzpunkte für mögliche Reparaturen bestimmen. Hierfür wird der Algorithmus von Reder und Egyed [RE12a] verwendet, um alle abstrakten Reparaturen (Definition 2.1) für eine Inkonsistenz zu generieren. Details zur *Validierung* und *syntaktischen Analyse* werden wir in Abschnitt 5.2.1 besprechen.

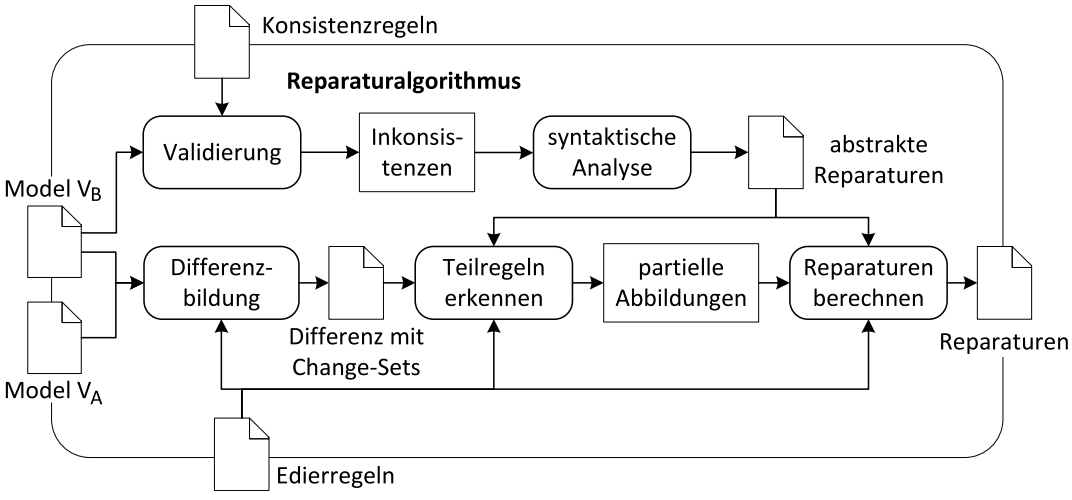


Abbildung 5.3: Datenfluss - Reparaturalgorithmus

In den nächsten Schritten wird die Historie des Modells analysiert, um unvollständige Editierschritte zu finden. Hierfür wird zusätzlich die historische Version V_A des Modells

benötigt, welche den Zustand unmittelbar vor Auftreten der Inkonsistenzen enthält. Zwischen den beiden Modellversionen V_A und V_B wird nun eine technische Differenz gebildet (Abschnitt 2.4), welche die Gemeinsamkeiten und Unterschiede der beiden Modelle beschreibt (Abbildung 5.3 – *Differenzbildung*).

Im nächsten Schritt werden die technischen Änderungen der Differenz mit allen Editierregeln der vorgegebenen Regelbasis verglichen. Mit Hilfe des in Kapitel 4 beschriebenen Matchingalgorithmus wird nach unvollständig ausgeführten Editierschritten gesucht (Abbildung 5.3 – *Teilregeln erkennen*). Hieraus ergeben sich alle partiellen Abbildungen für das Änderungsmuster einer Editierregel (Abschnitt 4.3). Die partiellen Abbildungen dienen später als Ausgangspunkt für die Berechnung der vervollständigenden Reparaturen. Damit der Matchingalgorithmus nur unvollständige Editierschritte erkennt, die im Zusammenhang mit den betrachteten Inkonsistenzen stehen, wird der Lösungsraum zusätzlich durch die syntaktische Analyse der Inkonsistenzen eingeschränkt. Die Einbindung des Matchingalgorithmus in die Reparaturberechnung wird in Abschnitt 5.2.2 behandelt.

Nachdem ermittelt wurde, welche Teilregeln unvollständige Editierschritte darstellen, müssen die entsprechenden Komplementregeln abgeleitet werden (Abbildung 5.3 – *Reparaturen berechnen*). Dafür wird die ermittelte Teilregel von der vollständigen Editierregel abgezogen (Abschnitt 3.2). Die Teilregel stellt somit den historischen bereits ausgeführten und die Komplementregel den noch auszuführenden Teil eines konsistenten Editierschritts dar. Schließlich müssen noch die Parameter bestimmt werden, mit denen die Komplementregel initialisiert werden kann. Jede mögliche Belegung der Parameter bildet dann einen konkreten Reparaturplan. Die einzelnen Reparaturen werden abschließend bewertet, um daraus ein Ranking zu erstellen. Das Ranking soll dem Entwickler dabei helfen, aus der Menge aller konkreten Reparaturen, die aus seiner Sicht korrekte herauszufinden. Die Berechnung der konkreten Reparaturpläne werden wir in Abschnitt 5.2.3 diskutieren.

5.2.1 Syntaktische Analyse der Inkonsistenzen

Der Reparaturprozess beginnt mit der Validierung des aktuellen Modellzustands V_B. Durch die Auswertung der modelltypspezifischen Konsistenzregeln wird zunächst die Menge der in V_B enthaltenen Inkonsistenzen erkannt. Als Ausgangspunkt für die Reparaturberechnung wählt der Entwickler dann eine Teilmenge der Inkonsistenzen aus, die behoben werden soll. Hierfür können ggf. noch weitere Analysen der Inkonsistenzen nützlich sein, um den Entwickler bei der Auswahl zu unterstützen. Interessant sind ins-

besondere Inkonsistenzen mit inhaltlichen Überschneidungen bei der Validierung. Jede Validierung einer Konsistenzregel besitzt, ausgehend vom Kontextelement der Auswertung, einen sogenannten **Auswertungsbereich** (*scope*) [Egy06]. Der Auswertungsbereich beschreibt die Menge der Modellelemente, die während der Validierung der Konsistenzregel evaluiert wurden. Die Auswertungsbereiche lassen sich während der Auswertung der Konsistenzregeln protokollieren. Eine Modifikation des Modells innerhalb der Auswertungsbereiche kann zur Veränderung der entsprechenden Validierungen führen. Überschneiden sich die Auswertungsbereiche mehrerer Validierungen, dann haben diese Inkonsistenzen möglicherweise einen inhaltlichen Zusammenhang oder sogar denselben Ursprung [Egy06].

Reder und Egyed [RE12a] haben ein Verfahren entwickelt, bei dem Reparaturen durch eine syntaktische Analyse der Konsistenzregeln und Inkonsistenzen generiert werden[4]. Die Reparaturen beschreiben die Modifikationen auf der Ebene des abstrakten Syntaxbaums des Modells, d.h. es werden keine komplexen domänenspezifischen Editierregeln verwendet. Zu jeder Inkonsistenz wird eine Menge von sogenannten abstrakten Reparaturen berechnet (Definition 2.1). Für die inkonsistente Nachricht `4:disconnect` in Abbildung 5.1 würde beispielsweise eine abstrakte Reparatur $\tau = \langle create, Class[Video], ownedOperation\rangle$ ausgegeben, welche eine Operation in der Klasse `Video` fordert. Die abstrakte Reparatur gibt aber nicht an, ob diese Operation neu erzeugt oder aus einer anderen Klasse verschoben werden soll. Die abstrakte Reparatur gibt sozusagen nur den Ansatz eines Editierschritts vor.

Die zu überprüfenden Konsistenzregeln sind als Ausdrücke einer auf Prädikatenlogik erster Stufe *(first-order logic, FOL)* basierenden Constraintsprache, wie beispielsweise der Object Constraint Language (OCL), anzugeben. Die Berechnung der abstrakten Reparaturen erfolgt in zwei Schritten. Zunächst wird die Konsistenzregel ausgewertet und jeder Evaluierungsschritt protokolliert. Aus diesem Protokoll lassen sich nun die abstrakten Reparaturen generieren. Jeder Evaluierungsschritt, welcher ein nicht erwartetes Ergebnis liefert, ist ein potentieller Ansatzpunkt für eine Reparatur. Jedes Modellelement, das durch den Teilausdruck ausgewertet wird, könnte verändert werden, um das Ergebnis der gesamten Validierung zu korrigieren. Zur Erzeugung der abstrakten Reparaturen wird für jeden Operator der Constraintsprache eine Generierungsfunktion definiert. Die

[4]Eine detaillierte Analyse dieses Verfahrens wurde bereits in [Ohr15] durchgeführt.

Generierungsfunktion erzeugt, auf Basis des erwarteten Ergebnisses und der evaluierten Modellelemente, alle möglichen abstrakten Reparaturen.

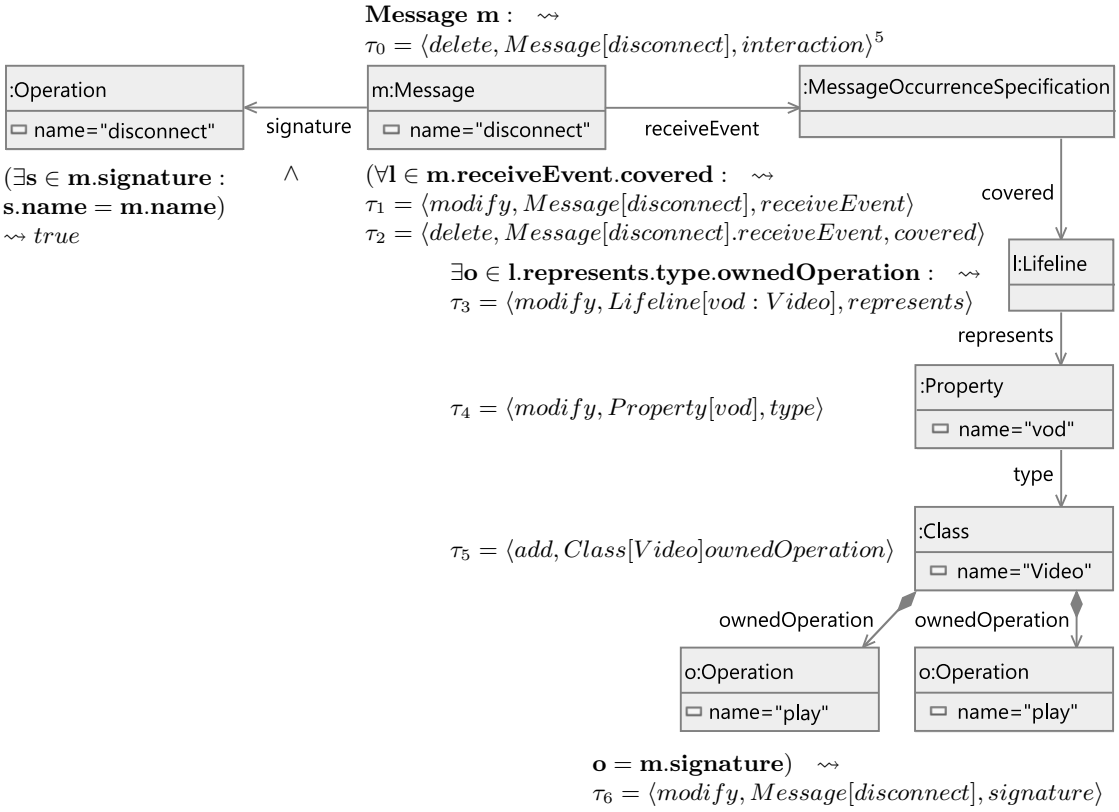


Abbildung 5.4: Berechnung der abstrakten Reparaturen

Abbildung 5.4 zeigt den Auswertungsbereich der Konsistenzregel 2.1 für die inkonsistente Nachricht `4:disconnect` in Abbildung 5.1. Für jedes Element bzw. jede Referenz wird ein Teilausdruck der Konsistenzregel ausgewertet. Für jede Auswertung lassen sich die entsprechenden abstrakten Reparaturen generieren. Der Algorithmus kann außerdem einen Reparaturbaum (*repair tree*) erzeugen, mit dem der Entwickler eine Sequenz von abstrakten Reparaturen aus den verfügbaren Alternativen zusammenstellen kann. Für die nachfolgenden Betrachtungen ist es allerdings ausreichend, einen Reparaturbaum als Menge von abstrakten Reparaturen pro Inkonsistenz aufzufassen. Abschließend lassen

[5] Entfernt das Kontextelement der Validierung.

sich noch mögliche positive und negative Seiteneffekte zwischen den Reparaturbäumen verschiedener Inkonsistenzen bestimmen.

> *„Indeed, any existing approaches for computing concrete repairs may be used to complement our repair trees with concrete values [...].“ [RE12a]*

Die abstrakten Reparaturen können als Ansatzpunkte verwendet werden, um komplexe konkrete Reparaturpläne, basierend auf domänenspezifischen Editierregeln, zu berechnen. Reder und Egyed [RE12a] zeigen, dass die berechneten Reparaturbäume *vollständig* sind, d.h. alle Modellelemente bzw. Eigenschaften, die man reparieren könnte, werden beachtet. Die Menge der abstrakten Reparaturen ist außerdem *minimal* in dem Sinne, dass keine unnötigen Reparaturen enthalten sind, d.h. jede Reparatur kann die Auswertung der entsprechenden Konsistenzregel beeinflussen. Daraus folgt, dass eine konkrete Reparatur mindestens eine abstrakte Reparatur enthalten bzw. initialisieren muss, um das Ergebnis einer Validierung positiv zu beeinflussen. Diese Schlussfolgerung wird für die nachfolgenden Berechnungen verwendet, um allgemeine Autovervollständigungen von Reparaturen zu unterscheiden.

5.2.2 Erkennung inkonsistenter Editierschritte

Zur Berechnung der komplexen Reparaturen wird zunächst die Historie des Modells nach unvollständig durchgeführten Editierschritten durchsucht. Hierfür wird die Differenz zwischen dem aktuellen inkonsistenten Modell V_B und der zuletzt konsistenten Modellversion V_A benötigt. Die Konsistenz des Modells wird hierbei immer auf eine ausgewählte Teilmenge der Inkonsistenzen in Modell V_B bezogen. Die Differenzbildung erfolgt, wie in Abschnitt 2.4 beschrieben. Nachdem die gemeinsamen Elemente beider Modellversionen bestimmt wurden, kann die technische Differenz abgeleitet werden. Die technische Differenz $D_{A \to B}$ wird als Menge der gemeinsamen Elemente und Änderungen zwischen zwei Modellversionen V_A und V_B angegeben. Eine Änderung beschreibt die Modifikation eines Attributs oder das Löschen/Hinzufügen von Objekten und Referenzen auf der Granularitätsebene des abstrakten Syntaxbaums. Gemeinsame Modellelemente in beiden Versionen werden als korrespondierend bezeichnet. Abschließend wird die Differenz nach *vollständig durchgeführten Editierregeln* durchsucht. Für jede Editierregel ergibt sich somit eine Anzahl von Change-Sets, welche die entsprechenden Änderungen für jede Regel zusammenfasst. Die Change-Sets werden im nachfolgenden Schritt für die Optimierung der Suche nach *partiell durchgeführten Editierregeln* verwendet.

Wird durch die Differenz $D_{A \to B}$ eine Inkonsistenz in Modell V_B eingefügt, so kann diese durch eine Modifikation der durchgeführten Änderungen behoben werden. Zur Korrektur der Inkonsistenz müssen entweder bereits ausgeführte Änderungen entfernt (*undo*), neue Änderungen hinzugefügt oder bestehende Änderungen ersetzt werden. Im Folgenden sollen alle möglichen Erweiterungen der Differenz ermittelt werden, welche die ausgewählten Inkonsistenzen reparieren können. Hierfür werden die benutzerdefinierten Editierregeln mit den bereits durchgeführten Änderungen verglichen. Wird ein Teil einer Editierregel in der Differenz erkannt, so muss ermittelt werden, ob ein vollständiges Ausführen der Regel zur Reparatur einer Inkonsistenz führen würde.

Die durch die syntaktische Analyse der Inkonsistenzen ermittelten abstrakten Reparaturen geben alle Ansatzpunkte für mögliche Reparaturen vor. Häufig sind allerdings komplexe Editierschritte notwendig, um eine Inkonsistenz vollständig zu beheben. Komplexe Editierschritte können durch benutzerdefinierte Editierregeln definiert werden. Wie bereits in Abschnitt 5.2.1 festgestellt wurde, muss eine komplexe Reparatur mindestens eine abstrakte Reparatur enthalten, um die Inkonsistenz zumindest auszubessern. Die Überschneidung zwischen abstrakten Reparaturen und den Änderungen der Editierregel lässt sich zunächst syntaktisch, d.h. unabhängig von der Modellinstanz, bestimmen. Dazu werden die Änderungen der Editierregel auf Basis ihres Typs mit den abstrakten Reparaturen verglichen. Hat eine Editierregel mindestens eine Überschneidung mit den abstrakten Reparaturen, dann wird diese Regel im Folgenden als **potentiell konsistenzverbessernd** bezeichnet. Im ersten Schritt (*Editierregeln filtern*) in Abbildung 5.5 werden alle Editierregeln der Regelbasis ermittelt, die potentiell konsistenzverbessernd sind.

Die inkonsistente Nachricht `4:disconnect` in Abbildung 5.1 kann beispielsweise durch die Editierregel in Abbildung 2.4 behoben werden. Die Editierregel verschiebt eine Operation und passt gleichzeitig die Empfänger der Nachrichten an, welche die Operation als Signatur besitzen. Bei einem syntaktischen Vergleich der Editierregel und der abstrakten Reparaturen wird genau eine Überschneidung festgestellt. Die Aktion $\alpha_5 = (delete, covered)$ überschneidet sich mit der abstrakten Reparatur $\tau_2 = \langle delete, Message[disconnect].receiveEvent, covered \rangle$, welche das Ende der Nachricht `4:disconnect` löscht (Abbildung 5.4).

Die Reparaturen, die sich aus einer Editierregel δ_g ergeben, können unabhängig für jede Regel der Regelbasis berechnet werden. Abbildung 5.5 zeigt zunächst die parallele Erkennung der partiell ausgeführten Editierregeln. Um die Menge der zu betrachtenden Änderungen einzugrenzen, werden zu Beginn alle vollständigen Vorkommen bezüglich der Editierregel δ_g aus der Differenz ausgefiltert. Überlappungen mit Änderungen von

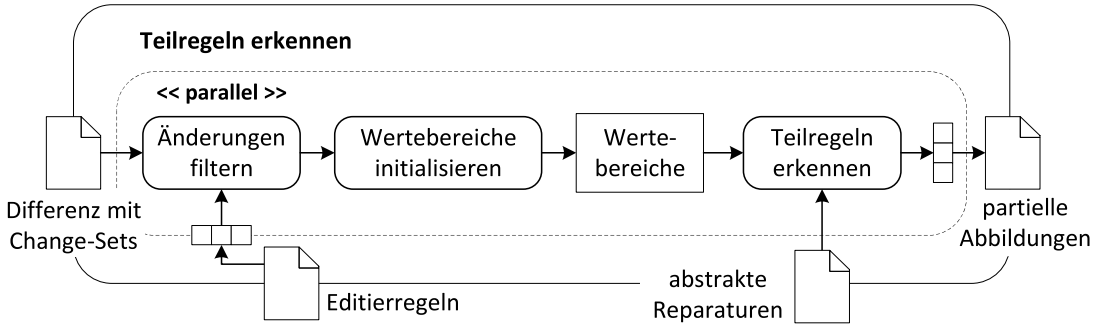


Abbildung 5.5: Datenfluss - Teilregeln erkennen

anderen vollständig erkannten Editierregeln sind grundsätzlich erlaubt. Dies kann insbesondere dann auftreten, wenn inkonsistente Editierschritte durch kleinere Regeln abgedeckt werden. Das Auftreten einer Inkonsistenz kann vom Kontext eines Modellelements abhängen. Die Transition `pause` in Abbildung 5.1 ist beispielsweise inkonsistent, da kein Trigger vorhanden ist. Eine Transition ausgehend vom Startzustand benötigt aber hingegen keinen Trigger. In diesem Fall würde eine Editierregel als Teilregel einer anderen Regel in der Regelbasis auftreten. Aus der verbleibenden Differenz und den Modellversionen V_A und V_B werden dann die Wertebereiche für den Matchingalgorithmus erzeugt (Abbildung 5.5 – *Wertebereiche initialisieren*).

Der Matchingalgorithmus zur Erkennung von Teilregeln (Abbildung 5.5 – *Teilregeln erkennen*) wurde bereits in Kapitel 4 besprochen. Im Gegensatz zur Berechnung der Autovervollständigungen, können in diesem Fall noch weitere Einschränkungen des Constraint-Satisfaction-Problems vorgenommen werden. Nach jeder Einschränkung des Lösungsraums erfolgt ein Vergleich mit den abstrakten Reparaturen. Der bereits ausgeführte Teil eines Editierschritts muss sich um mindestens eine abstrakte Reparatur erweitern lassen. Die Überschneidung der Änderungen der Editierregel und der abstrakten Reparaturen $\tau = \langle \lambda, \omega, \gamma \rangle$ ist bereits bei der Filterung der Editierregeln berechnet worden. An dieser Stelle werden nun zusätzlich die Kontextelemente ω der abstrakten Reparatur mit den noch enthaltenen Modellelementen in den entsprechenden Wertebereichen $t_i.B$ für Modellversion V_B der Kontextknoten verglichen (Abschnitt 4.3). Das Matching kann fortgesetzt werden, wenn das Modellelement ω als Kontextelement für die noch auszuführende Aktion der Editierregel gefunden wurde. Irrelevante Matchings lassen sich somit frühzeitig erkennen, wodurch der Lösungsraum des Constraint-Satisfaction-Problems verkleinert wird. Alle gefundenen Lösungen sind Hinweise auf inkonsistente Editierschritte, deren Ergänzung eine mögliche Reparatur darstellen.

Der Matchingalgorithmus sucht nach partiellen Abbildungen der Erkennungsregel auf die Differenz und die Modellversionen V_A und V_B. Diese Abbildung lässt sich wiederum der ursprünglichen Editierregel δ_g zuordnen. Jedem Knoten der Aktionsgraphen einer Editierregel können Modellelemente aus V_A und V_B zugewiesen werden. Kontextknoten, die nur für den bereits ausgeführten Teil der Editierregel benötigt werden, besitzen ggf. nur eine Abbildung auf die historische Modellversion V_A. Neu erzeugte Modellelemente in Modell V_B und bereits gelöschte Modellelemente in Modell V_A werden den entsprechenden erzeugenden bzw. löschenden Knoten der Aktionsgraphen zugeordnet. Noch zu löschende Knoten besitzen ggf. mehrere Abbildungen auf Modell V_B. Knoten, die noch zu erzeugen sind, besitzen folglich keine Abbildungen. Wir bezeichnen dies im Folgenden als **partielle Editierregelabbildung**. Alle Änderungen, die auf die Differenz abgebildet wurden, legen die bereits ausgeführte Teilregel δ_t der Editierregel δ_g fest.

5.2.3 Berechnung der Reparaturen

Aus den bisher erkannten Teilregeln und partiellen Editierregelabbildungen müssen nun alle möglichen Reparaturen berechnet werden. Hierfür muss die Teilregel durch eine Komplementregel ergänzt werden (Abbildung 5.6 – *Komplementregel ableiten*). Eine mögliche Initialisierung der Komplementregel mit passenden Parametern wird dann als Reparaturplan bezeichnet. Ob sich eine Teilregel zu einer Reparatur ergänzen lässt, muss letztlich auf Basis der abgeleiteten Komplementregel überprüft werden. Eine Komplementregel ist **potentiell konsistenzverbessernd**, wenn syntaktisch mindestens eine abstrakte Reparatur in der Regel enthalten ist (Abschnitt 5.2.2). Eine Komplementregel gilt dann als **konsistenzverbessernd**, wenn mindestens eine Änderung einen positiven Effekt auf die Validierung des Modells hat. Da die abstrakten Reparaturen alle möglichen Ansatzpunkte für Reparaturen vorgeben, lässt sich ein positiver Effekt auf die Auswertung einer Konsistenzregel feststellen, wenn mindestens eine abstrakte Reparatur durch die Komplementregel initiiert wird.

Als nächstes werden die konkreten Reparaturen ermittelt, d.h. die resultierende Komplementregel wird mit Parametern für die noch auszuführenden Änderungen belegt. Im Sinne der Graphtransformationsregel bedeutet dies eine vollständige Abbildung für die linke Seite (LHS) der Regel in der aktuellen Modellversion V_B zu finden (Abbildung 5.6 – *Komplement-Matching*). Aus der Abbildung werden dann die Parameter der Editieroperation abgeleitet. Einzelne Parameter, wie beispielsweise Namen für neu zu erzeugende Modellelemente, sind ggf. noch durch den Entwickler zu initialisieren, falls der Reparaturplan angewendet werden soll. Außerdem werden geforderte Anwendungsbedingungen

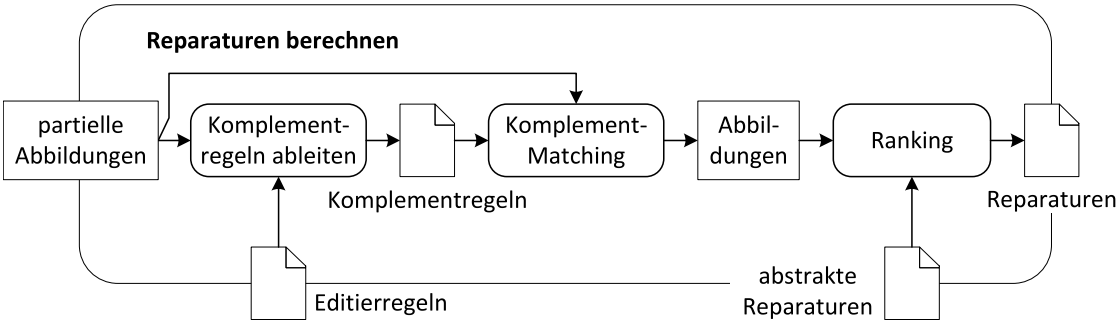


Abbildung 5.6: Datenfluss - Reparaturen berechnen

der Regel überprüft. Der Arbeitsgraph für die Graphtransformation wird bereits durch die partielle Editierregelabbildung vorgegeben. Alle gefundenen vollständigen Abbildungen, die konsistenzverbessernd sind, werden dann als konkrete Reparaturen gespeichert. Alle anderen Abbildungen werden ausgefiltert. Es kann auch der Fall eintreten, dass keine vollständige Abbildung der Komplementregel existiert, weil z. B. eine Anwendungsbedingung nicht erfüllt werden kann. Die Komplementregel entfällt damit als mögliche Reparatur.

Aus allen berechneten Reparaturen muss der Entwickler schließlich eine Reparatur auswählen. Um die Auswahl zu erleichtern, wird eine Bewertung und entsprechende Sortierung der Reparaturen durchgeführt (Abbildung 5.6 – *Ranking*). Ein typisches Bewertungskriterium ist die Minimierung der durchzuführenden Änderungen am Modell, d.h. je weniger Änderungen eine Reparatur hat, desto besser fällt die Bewertung aus. *„Principle of Least Change: The action taken by the maintainer of a constraint after a violation should change no more than is needed to restore the constraint“* [Mee98]. Intuitiv lässt sich dieses Prinzip mit dem Versuch begründen den Informationsverlust, der durch eine Reparatur entsteht, möglichst gering zu halten. Außerdem sollen nicht unnötige neue Informationen in das Modell eingefügt werden. Das hier verwendete Bewertungskriterium beachtet die bereits durchgeführten Modifikationen und hält die Anzahl der neuen Modifikationen gering. Dazu wird das Verhältnis zwischen der Anzahl der historischen und der Anzahl der komplementierenden Änderungen berechnet. Eine Reparatur wird damit positiv bewertet, wenn die Editierregel bzw. Teilregel δ_t eine große Überschneidung mit der Differenz aufweist und möglichst wenig Modifikationen durch die Komplementregel $\overline{\delta_k}$ durchgeführt werden. Die Reparaturen mit dem größten Resultat $Q = \frac{|\delta_t|}{|\overline{\delta_k}|}$ werden damit im Ranking weit oben ausgegeben. Die Bestimmung des korrekten Reparaturplans wird aber schließlich dem Entwickler überlassen.

> *„Since the repair of inconsistencies goes hand in hand with the creative process of modeling, we strongly advocate against heuristics that replace the role of the human designer. For example, a repair that favors the fewest model changes is frequently the same as an undo.“ [RE12a]*

Grundsätzlich lassen sich auch noch weitere Bewertungskriterien auf das hier vorgestellte Verfahren anwenden; wobei dann immer die konkreten Reparaturpläne, ohne historische Editierschritte, bewertet würden. Mens et al. [PVDSM15] verwendet beispielsweise eine Reihe von gewichteten Bewertungskriterien. Zum Beispiel kann das Erzeugen von Modellelementen gegenüber dem Löschen bevorzugt werden. Es kann auch sinnvoll sein, bestimmte Teile des Modells möglichst zu bewahren, so dass beispielsweise primär Verhaltensdiagramme modifiziert werden und die Strukturdiagramme erhalten bleiben. Eine weitere Möglichkeit die Historie des Modells mit in die Bewertung einzubeziehen, ist neuere Modellelemente bevorzugt vor älteren zu modifizieren.

Die Differenz zwischen der Modellversionen V_A in Abbildung 2.1 und der inkonsistenten Version V_B in Abbildung 5.1 enthält zwei inkonsistente Editierschritte. Das Verschieben der Operation `disconnect()` kann als Teilregel der Editierregel in Abbildung 2.4 erkannt werden. Daraus lässt sich die Komplementregel in Abbildung 5.7 ableiten, welche die Empfänger der entsprechenden Nachrichten anpasst. Hieraus ergibt sich dann der konkrete Reparaturplan, die Nachricht `4:disconnect` von der Lebenslinie des Objekts `vod:Server` empfangen zu lassen. Für die neu erzeugte Transition `pause` lässt sich aus der Editierregel in Abbildung 3.1 das Einfügen einer neuen Operation `pause()` in die Klasse `Video` ableiten. Abbildung 5.8 zeigt die entsprechende Komplementregel. Nach Anwenden der Reparaturen ergibt sich schließlich die Modellversion V_C in Abbildung 2.2. Zusätzlich wurde hier noch der Sender der Nachricht `4:disconnect` auf die Lebenslinie des Objekts `open:Video` gesetzt. Die beiden Inkonsistenzen wurden damit behoben.

Nachdem eine Reparatur ausgeführt wurde, erfolgt eine erneute Validierung des aktuellen Modells. Im besten Fall wurden die ausgewählten Inkonsistenzen vollständig behoben. Ansonsten wird die Berechnung der Reparaturen mit der aktuellen Modellversion erneut gestartet. Die Reparaturen müssen dann inkrementell angewendet werden. Es kann allerdings auch vorkommen, dass durch eine (vollständige) Reparatur neue Inkonsistenzen ausgelöst werden. Solche negativen Seiteneffekte müssen ebenfalls durch weitere Reparaturen behoben werden. Eine Inkonsistenz kann nur dann nicht in einem Schritt behoben werden, wenn keine Editierregel vorhanden ist, welche die benötigten Änderungen in einem Schritt ausführt.

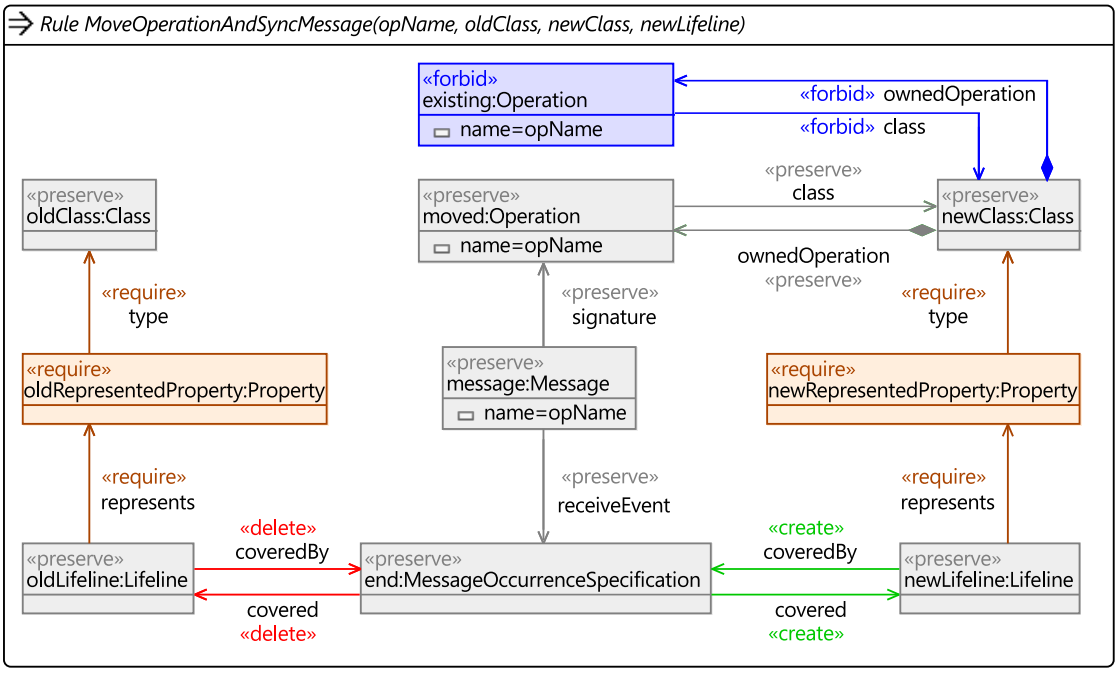


Abbildung 5.7: Komplementregel - Verschieben einer Operation

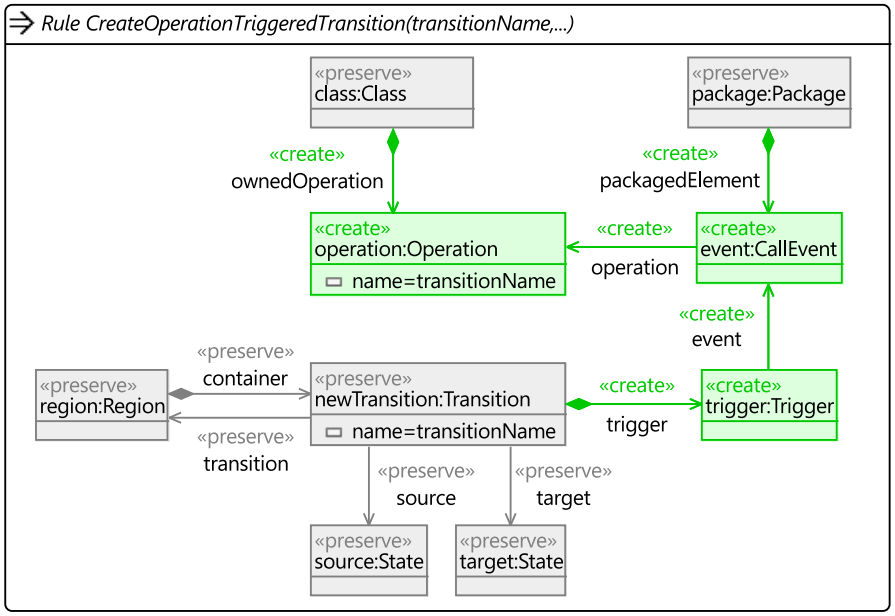


Abbildung 5.8: Komplementregel - Erzeugen einer getriggerten Transition

6 Bearbeitung materialisierter Sichten

Der Entwurf eines komplexen Systems macht es erforderlich, dass die Modellierung in verschiedene Teilmodelle und Sichten gegliedert wird. Damit sich ein Entwickler während einer Entwurfsaufgabe besser auf einen bestimmten Aspekt des Systems konzentrieren kann, sollte es möglich sein, nicht relevante Teile des Modells auszublenden. Hierfür muss der relevante Teil des Modells ausgeschnitten werden. Diese Technik wird daher auch als **Slicing** (*model slicing*) [ACH+13] bezeichnet. Der Entwickler wählt zunächst einen Teil des Modells aus, der für seine Arbeit relevant ist. In Anlehnung an das Slicing von Quellcode wird diese Auswahl als **Slicing-Kriterium** (*slicing criterion*) bezeichnet [Wei79]. Ein Slicing-Algorithmus ergänzt dieses Kriterium dann zu einem sinnvollen Teilmodell. Der so entstehende Modellausschnitt wird daher als **Slice** bezeichnet. Insbesondere bei sehr großen Netzwerken von Modellen lässt es sich so vermeiden, dass immer das gesamte Systemmodell aus einem Repository in den lokalen Arbeitsbereich des Entwicklers kopiert wird. Das Slicing des Systemmodells hat außerdem den Vorteil, dass ein kleines Teilmodell effizienter durch die verwendeten Werkzeuge verarbeitet werden kann, als das vollständige Systemmodell. Darüber hinaus lässt sich einfacher feststellen, ob parallel durchgeführte Modifikationen verschiedener Entwickler ggf. in Konflikt stehen.

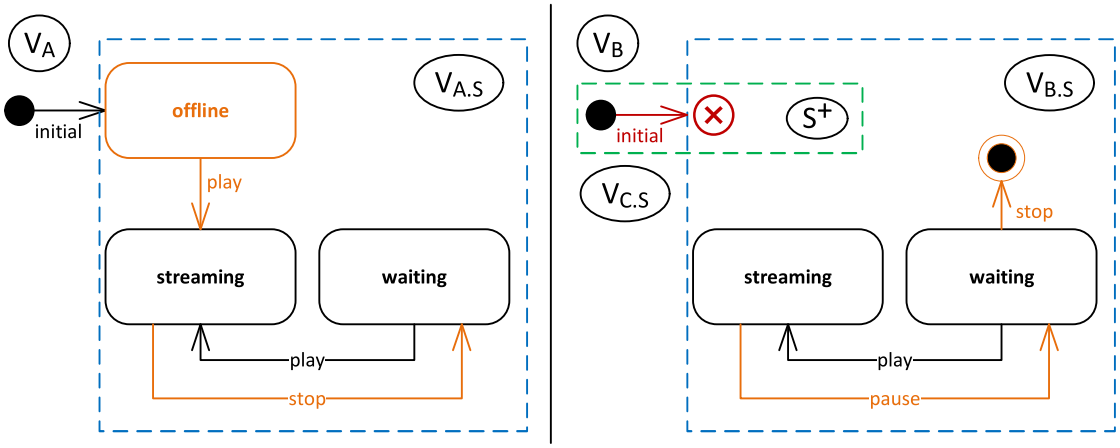


Abbildung 6.1: Entwurf eines VoD-Systems - Slice

Abbildung 6.1 zeigt ein einfaches Beispiel für einen Slice, welcher nur den Startzustand des ursprünglichen Zustandsdiagramms nicht enthält. Der Modellausschnitt $V_{A.S}$ des Systemmodells V_A wird zunächst aus dem Repository in den lokalen Arbeitsbereich des Entwicklers übertragen. Der Entwickler startet nun die isolierte Bearbeitung des Modellausschnitts. In diesem Fall wird der Zustand `offline` und die damit verbundene Transition `play` aus dem Diagramm entfernt. Außerdem wird ein neuer Endzustand und die Transition `pause` eingefügt. Dadurch entsteht die neue Version $V_{B.S}$ des Modellausschnitts. Da die Modifikationen des Modellausschnitts abgeschlossen sind, sollen die Änderungen nun wieder in das Ursprungsmodell V_A reintegriert werden. Daraus ergibt sich die neue Version des Systemmodells V_B. Die isolierte Bearbeitung des Modellausschnitts hat allerdings zu einer Inkonsistenz in der neuen Version V_B geführt. Die Transition des Startzustands besitzt nun keinen Zielzustand mehr.

Ein Werkzeug zur Reintegration von Modellausschnitten muss den Entwickler auf die neu entstandenen Inkonsistenzen hinweisen. Der Entwickler muss dann entscheiden, ob die Inkonsistenzen behandelt werden sollen. Dafür muss es allerdings möglich sein, die Inkonsistenzen im lokalen Arbeitsbereich nachzuvollziehen. Hierfür müssen alle Teile des Systemmodells, welche für die Behandlung der Inkonsistenz relevant sind, in den lokalen Arbeitsbereich des Entwicklers übertragen werden. Der Slice in Abbildung 6.1 muss minimal um den Startzustand und die „hängende Transition“ `initial` erweitert werden. Entsprechend der Modellversion in Abbildung 2.2 kann der Entwickler die Inkonsistenz beheben, indem die Transition mit dem Zustand `streaming` verbunden wird. Die so entstehende Version $V_{C.S}$ kann dann, ohne neue Inkonsistenzen auszulösen, in das ursprüngliche Systemmodell reintegriert werden, wodurch die neue konsistente Version V_B entsteht.

Ein Slice stellt eine materialisierte Sicht eines Systemmodells V_A dar. Entstehen Inkonsistenzen zwischen der materialisierten Sicht und einem Teil des Ursprungsmodells, so können diese analog zu Inkonsistenzen in den vordefinierten Sichten einer Modellierungssprache behandelt werden. Dieses Kapitel beschreibt den Entwurf eines Konzepts zur konsistenten Reintegration von Modellausschnitten. Dabei wird das in Kapitel 5 vorgestellte Reparaturwerkzeug verwendet, um Inkonsistenzen, die bei der Reintegration entstehen, zu behandeln. Abbildung 6.2 gibt einen Überblick über die verschiedenen Aktivitäten des Konzepts. Zunächst muss der Teil des Modells V_A festgelegt werden, der durch den Entwickler bearbeitet werden soll. Das Slicing-Kriterium könnte beispielsweise aus einer generierten Übersicht der Fragmente eines Systemmodells ausgewählt werden. Anschließend wird ein Modellausschnitt berechnet, der die ausgewählten Fragmente in einem sinnvollen Kontext darstellt. Ein konkreter Slicing-Algorithmus wird im Rahmen

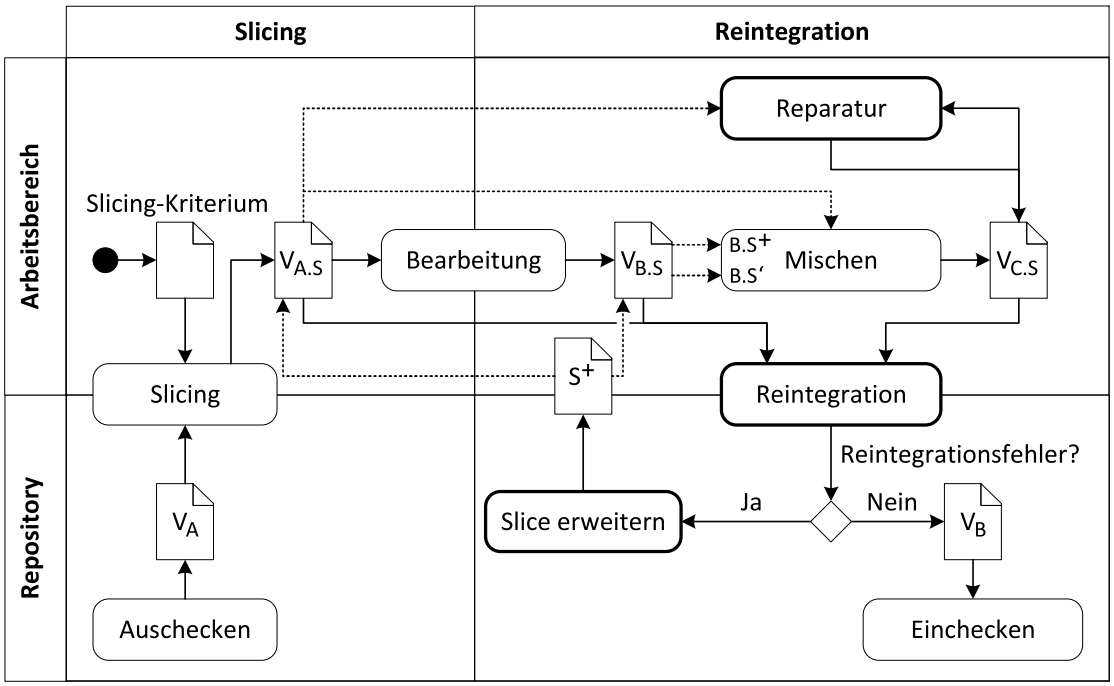


Abbildung 6.2: Slicing und Reintegration von Modellen

dieser Arbeit als gegeben angenommen. Der entstandene Slice $V_{A.S}$ wird dann in den lokalen Arbeitsbereich des Entwicklers kopiert. Nachdem die erforderlichen Modifikationen mit einem entsprechenden Modellierungswerkzeug durchgeführt wurden, entsteht schließlich die neue lokale Modellversion $V_{B.S}$.

Als nächstes erfolgt die Reintegration des Modellausschnitts $V_{B.S}$ in das Ursprungsmodell V_A des Repositorys. Hierfür wird zunächst die Differenz zwischen den Versionen $V_{A.S}$ und $V_{B.S}$ des Modellausschnitts berechnet. Danach werden die Änderungen auf das Ursprungsmodell übertragen. Wir gehen hier davon aus, dass die Zuordnung der Modellelemente zwischen Slice und Ursprungsmodell bekannt sind. Dies lässt sich beispielsweise über eindeutige Surrogatschlüssel (z. B. Universally Unique Identifier, UUID) umsetzen, die in den Modellelementen gespeichert werden. Bei der Reintegration der lokalen Änderungen können allerdings Konflikte und neue Inkonsistenzen im Ursprungsmodell entstehen. Diese Probleme werden im Folgenden als **Reintegrationsfehler** bezeichnet. Um die Reintegrationsfehler im lokalen Arbeitsbereich nachzuvollziehen, muss der Modellausschnitt ggf. erweitert werden. Hierfür müssen zunächst die Ursachen für die Reintegrationsfehler identifiziert werden. Als Ursache eines Reintegrationsfehlers lässt sich eine Menge von Modellelementen bestimmen. Die ermittelten Modellelemente wer-

den in der Menge S^+ gespeichert. Die einzelnen ausgewählten Modellelemente können ggf. noch durch den Slicer zu sinnvollen Fragmenten vergrößert werden:

- **Konflikte durch konkurrierende Bearbeitungen:** Falls das Ursprungsmodell V_A in der Zwischenzeit parallel durch einen anderen Entwickler modifiziert wurde, müssen diese Änderungen vor der Reintegration mit dem lokalen Arbeitsbereich synchronisiert werden. Dies ist insbesondere dann erforderlich, wenn die konkurrierenden Änderungen mit den lokalen Änderungen in Konflikt stehen.

  Alle modifizierten und in Konflikt stehenden Modellelemente (inklusive modifizierter Attribute und Referenzen) werden in die Menge S^+ aufgenommen, damit die Konflikte im lokalen Arbeitsbereich behandelt werden können.

- **Konflikte bezüglich des Ursprungsmodells:** Besitzt eine Referenz im Metamodell die Multiplizität [0..1] oder [1..1], dann kann es ggf. zu Konflikten zwischen Slice und Ursprungsmodell kommen. Würden Referenz mit diesen Multiplizitäten aus einem Modellausschnitt entfernt und bei der Bearbeitung durch neue Referenzen ersetzt, so würde die Reintegration die ursprünglichen Referenzen einfach überschreiben. Der Entwickler sollte aber auf solche Seiteneffekte hingewiesen werden und ggf. die Möglichkeit bekommen, darauf zu reagieren.

  Konflikte bezüglich des Ursprungsmodells können bei Bedarf auch durch den Slicer verhindert werden, indem Referenzen mit [0..1] bzw. [1..1] Multiplizitäten immer ergänzt werden. Hier gilt es, einen sinnvollen Kompromiss zwischen der Erweiterung des Modellausschnitts und möglichen Reintegrationsfehlern zu finden.

  Um den Konflikt lokal nachvollziehen zu können, müssen minimal die in Konflikt stehenden referenzierten Modellelemente des Ursprungsmodells V_A in die Erweiterung des Modellausschnitts S^+ aufgenommen werden.

- **hängende Referenzen:** Werden Modellelemente im Modellausschnitt gelöscht, so kann es bei der Reintegration zu sogenannten hängenden Referenzen kommen. Existieren im Ursprungsmodell weitere Modellelement, die das gelöschte Modellelement referenzieren, dann werden diese Referenzen als Seiteneffekt der Reintegration ebenfalls gelöscht.

  Es müssen zunächst alle Referenzen ermittelt werden, die auf zu löschende Modellelemente zeigen. Die Menge S^+ wird dann um die bereits gelöschten Modellelemente und um die referenzierenden Modellelemente erweitert.

- **neue Inkonsistenzen:** Durch die Reintegration der lokalen Änderungen kann das entstehende Modell V_B neue Inkonsistenzen enthalten, die bisher weder im lokalen Modell $V_{B.S}$ noch im Ursprungsmodell V_A aufgetreten sind. Ein Editierschritt kann aus Sicht des Modellausschnitts konsistent sein, während der gleiche Editierschritt im Ursprungsmodell zu einer Inkonsistenz führt.

  Damit die Inkonsistenzen ebenfalls im lokalen Arbeitsbereich des Entwicklers auftreten, müssen sich die entsprechenden Validierungen des Modellausschnitts identisch zu den fehlgeschlagenen Validierung des Ursprungsmodells verhalten. Daher werden die Auswertungsbereiche (Abschnitt 5.2.1) aller fehlgeschlagenen Validierungen auf Modellversion V_B protokolliert und in die Menge S^+ aufgenommen.

Eine voll automatische Reparatur der Reintegrationsfehler ist nicht immer möglich oder sinnvoll. Daher wird der aktuelle Modellausschnitt im nächsten Schritt so erweitert, dass der Entwickler die Reintegrationsfehler im lokalen Arbeitsbereich nachvollziehen und beheben kann. Dazu wurden alle Ursachen der Reintegrationsfehler ermittelt und die beteiligten Modellelemente S^+ bestimmt. Aus den ermittelten Modellelementen S^+ und dem modifizierten Slice $V_{B.S}$ ergeben sich dann zwei neue Modellversionen. Zunächst werden nur die Modellelemente in $V_{B.S}$ eingefügt, welche nicht in Konflikt mit den Änderungen des Modellausschnitts stehen. Daraus ergibt sich die erweiterte Version des Modellausschnitts $V_{B.S'}$. Für die zweite Version $V_{B.S^+}$ werden alle Modellelemente aus S^+ in den aktuellen Slice $V_{B.S}$ übernommen, auch wenn dadurch in Konflikt stehende Teile des Modellausschnitts überschrieben werden. Der Slice $V_{B.S'}$ entspricht der Sicht des Entwicklers, während der Slice $V_{B.S^+}$ die Sicht des Repositorys darstellt. Beide Modellausschnitte basieren auf dem gleichen Ursprungsmodell $V_{A.S}$, das um alle Modellelemente in S^+ erweitert wird. Die bei der Reintegration gefundenen Konflikte können somit durch ein 3-Wege-Mischen [SWK09] der beiden Slice-Versionen $V_{B.S^+}$ und $V_{B.S'}$ gelöst werden. Hierbei kann auch das implizite Löschen hängender Referenzen nochmal abfragt werden. Die Behandlung der Konflikte kann durch ein entsprechendes Werkzeug ([KKR14]) im lokalen Arbeitsbereich des Entwicklers durchgeführt werden.

Nachdem der konfliktfreie Modellausschnitt $V_{C.S}$ erstellt wurde, müssen noch neu aufgetretene Inkonsistenzen behandelt werden. Für die Einbindung des Reparaturwerkzeugs aus Kapitel 5 in den Reintegrationsprozess können Konflikte zu diesem Zeitpunkt vernachlässigt werden. Alle neu entstandenen Inkonsistenzen werden nun auf Basis der Versionen $V_{A.S}$ und $V_{C.S}$ des erweiterten Modellausschnitts repariert. Dazu sucht das Reparaturwerkzeug nach unvollständigen Editierschritten bezüglich der neu aufgetretenen Inkonsistenzen. Im einfachsten Fall können die Reparaturen aus dem aktuellen

Modellausschnitt ermittelt werden. Sind die benötigten Modellelemente nicht im Modellausschnitt enthalten, so muss der Entwickler den Slice bzw. die Slicing-Konfiguration entsprechend erweitern. Sobald alle Inkonsistenzen behoben wurden, kann ein neuer Versuch zur Reintegration des Modellausschnitts unternommen werden. Falls keine neuen Reintegrationsfehler des erweiterten Modellausschnitts $V_{C.S}$ auftreten, wird eine neue Revision V_B des Ursprungsmodells im Repository erzeugt; ansonsten muss der Reintegrationsprozess fortgesetzt werden.

Alle in Abbildung 6.2 beschriebenen Aktionen verwenden verschiedene Versionen eines Modells als Eingabe. Die Modellversionen $(V_{A.S}, V_{B.S}, V_{C.S})$ können nach den einzelnen Schritten in einem Zweig des Repositorys abgelegt werden. Die Werkzeuge (Slicing, Modellierung (Bearbeiten), Mischen, Reparatur, Reintegration), welche die einzelnen Aktionen umsetzen, können daher über das zentrale Repository kommunizieren, ohne direkt untereinander Daten auszutauschen. Das hat den Vorteil, dass die Werkzeuge in unterschiedlichen Prozessen ausgeführt werden können. So könnte beispielsweise das Repository auf einem anderen Server ausgeführt werden als die Reintegration. Der Entwickler hat außerdem die Freiheit verschiedene Werkzeuge, beispielsweise zur Modellierung und zum Mischen, einzusetzen.

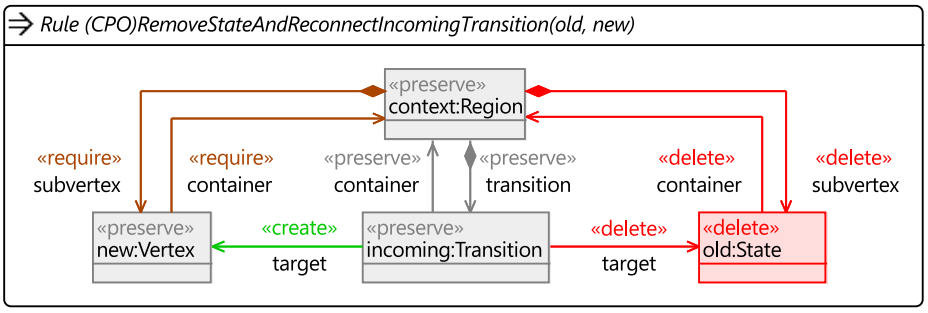


Abbildung 6.3: Editierregel - Transition neu verbinden

Konsistenzregel 6.1

Transition t : not(isEmpty(t.source)) ∧ not(isEmpty(t.target))

Mit diesem Prozess lässt sich die Reparatur des Modellausschnitts in Abbildung 6.1 automatisieren. Für die Berechnung der erforderlichen Reparatur wird zunächst die Konsistenzregel 6.1 benötigt, die keine „hängenden Transitionen“ zulässt. Des Weiteren wird die Editierregel in Abbildung 6.3 benötigt, welche das Ziel einer Transition neu verbindet. Bei der Reintegration wird der Zustand `offline` aus dem Ursprungsmodell gelöscht. Implizit wird damit auch der Zielzustand der Transition `initial` ge-

löscht. Abbildung 6.1 zeigt den Auswertungsbereich (S^+) der Konsistenzregel 6.1, welcher den Startzustand und die inkonsistente Transition einschließt. Dieser Teil wird nun in den Slice übertragen. Der Reparaturalgorithmus berechnet für die Editierregel in Abbildung 6.3 genau eine Komplementregel. Diese initialisiert die abstrakte Reparatur $\tau = \langle create, Transition[initial], target \rangle$. Als Ziel der Transition ergeben sich innerhalb des Modellausschnitts drei mögliche Zustände (`streaming`, `waiting` und der Endzustand), welche die konkreten Reparaturpläne festlegen. Jede dieser Reparaturen behebt die Inkonsistenz vollständig. Nachdem der Entwickler eine Reparatur ausgewählt hat, können die Modifikationen des Modellausschnitts in das Ursprungsmodell übertragen werden, wodurch die neue konsistente Version V_B des Systemmodells entsteht.

7 Proof of Concept

Das Reparaturwerkzeug in Abbildung 7.2 wurde im Eclipse Modeling Framework [EMF] implementiert. Zur Konfiguration des Werkzeugs wird das Ecore-Metamodell und die Konsistenzregeln der Modellierungssprache benötigt. Zur Formulierung der Konsistenzregeln wurde eine auf Prädikatenlogik erster Stufe basierende Constraintsprache erstellt. Abbildung 7.1 zeigt den mit Hilfe einer Xtext-Grammatik [Xte] umgesetzten Editor. Das entsprechende Validierungswerkzeug kann dann alle angegebenen Konsistenzregeln auf einem Modell auswerten und zu allen aufgetretenen Inkonsistenzen die entsprechenden abstrakten Reparaturen berechnen (7.2.C). Des Weiteren werden die benutzerdefinierten Editierregeln für eine Modellierungssprache benötigt (7.2.D). Die Graphtransformationsregeln bzw. Aktionsgraphen können mit Hilfe des graphischen Henshin-Editors [Hen] formuliert werden. Die Transformation der Editierregeln in Änderungsmuster übernimmt das SiLift-Projekt [SiL].

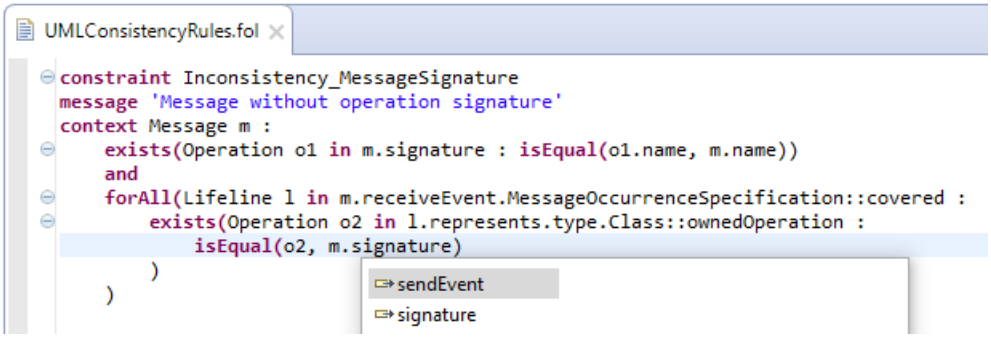


Abbildung 7.1: Editor der Constraintsprache

Das Reparaturwerkzeug wird dann mit zwei Modellversionen gestartet. Die historische konsistente Version V_A (7.2.A) und die aktuelle inkonsistente Version V_B (7.2.B). Zunächst muss die Differenz zwischen den Modellen gebildet werden. Für die Berechnung der Korrespondenzen kann zwischen verschiedenen Verfahren des SiDiff-Projekts [SiD] gewählt werden (7.2.E). Nachdem die Berechnung der Reparaturen (7.2.F) beendet wurde, werden die gefundenen Komplementregeln entsprechend des ermittelten

Rankings ausgegeben. Unterhalb der Komplementregeln werden dann die konkreten Reparaturpläne aufgelistet. Wird ein Reparaturplan ausgewählt, dann werden die an der Reparatur beteiligten Modellelemente im grafischen Editor gehighlightet. Um den Effekt einer Reparatur zu testen kann der Entwickler die ausgewählte Reparatur auf das aktuelle Modell V_B anwenden (7.2.G). Bei Bedarf lassen sich die angewendeten Reparaturen auch wieder zurücksetzen (7.2.H). Ansonsten wird das reparierte Modell erneut validiert, um festzustellen, ob noch weitere Inkonsistenzen zu beheben sind.

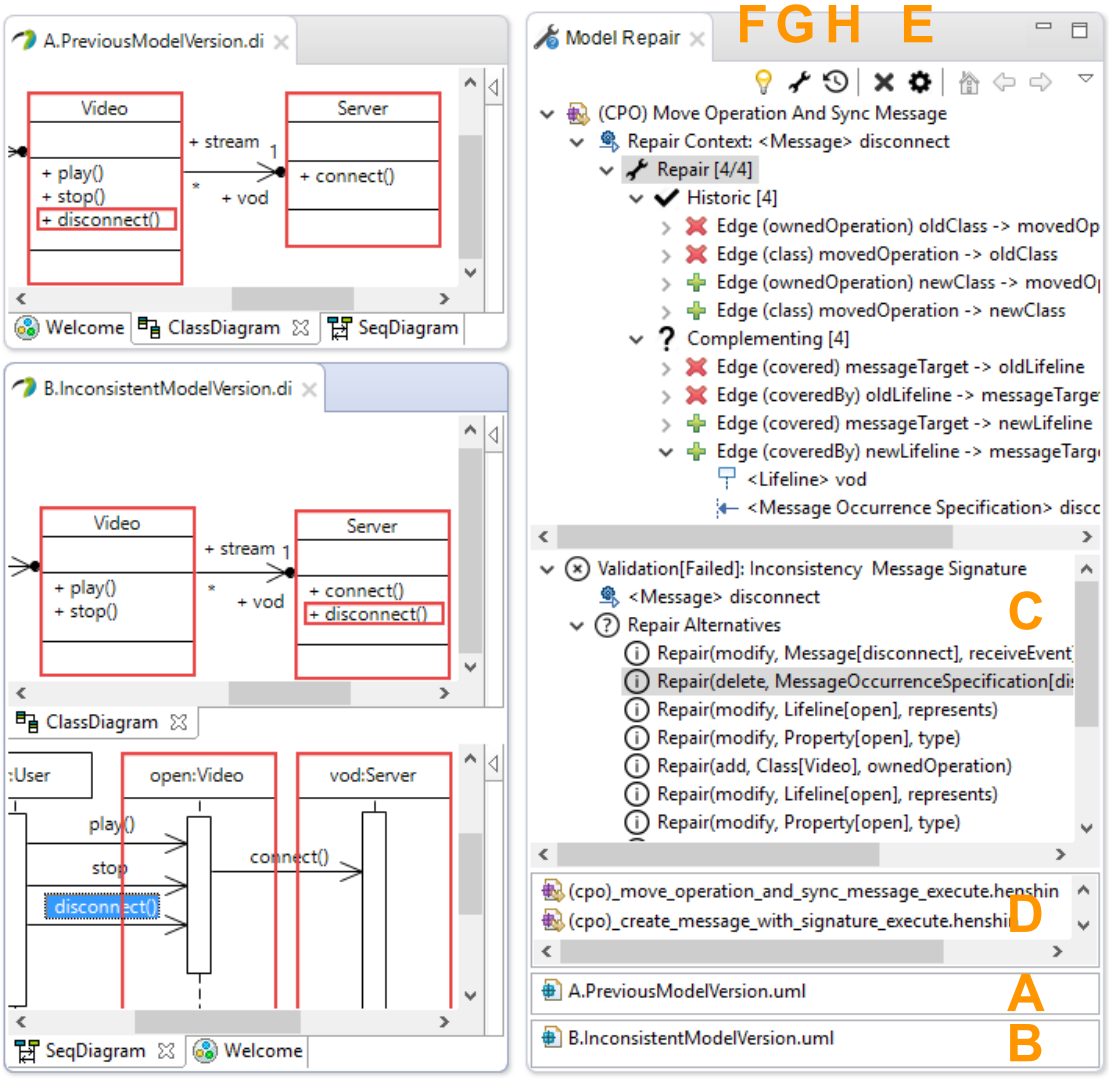


Abbildung 7.2: Benutzeroberfläche des Reparaturwerkzeugs

Für eine erste Einschätzung der Laufzeiten[6] und Skalierbarkeit wurde das Reparaturwerkzeug auf eine Historie der bCMS-Fallstudie (Barbados Crash Management System) [CCGI11] angewandt. Hierbei handelt es sich um ein UML-Modell, das Klassen-, Zustands- und Sequenzdiagramme enthält. Das Modell V_A enthält etwa 2600 Modellelemente mit 22.000 Referenzen. Ausgehend von Modellversion V_A wurden Teile der Zustands- und Sequenzdiagramme eingefügt und gelöscht. In der berechneten Differenz zwischen Modellversion V_A und V_B wurden etwa 2500 Änderungen an Modellelementen (erzeugt 290 / gelöscht 300) und Referenzen (erzeugt 1200 / gelöscht 700) durchgeführt. Die Validierung des aktuellen Modells V_B ergibt 148 Inkonsistenzen für 8 Konsistenzregeln. Die Validierung und die Generierung der abstrakten Reparaturen ($\approx 10ms$) kann unabhängig von der Berechnung der Differenz ($\approx 1500ms$) durchgeführt werden. Um Reparaturen während der Bearbeitung eines Modells vorzuschlagen, könnte das Reparaturwerkzeug zukünftig um eine inkrementelle Differenzbildung und Validierung erweitert werden.

ER	Aktionen	Änderungen (Wertebereiche)	Teilregeln / Reparaturen	Matching-algorithmus	Heuristiken (Min. 30%)
01	2	96	96 / 8	3ms	3ms
02	3	188	48 / 6	3ms	2ms
03	5	430	377 / 5	25ms	24ms
04	6	77	12 / 4	3ms	3ms
05	7	61	11 / 4	35ms	10ms
06	10	87	4 / 5	20ms	10ms
07	11	71	29 / 35	15ms	5ms
08	15	115	34 / 46	80ms	35ms
09	21	1304	4 / 4	800ms	250ms
10	22	1366	4 / 108	2300ms	900ms

Tabelle 7.1: Laufzeiten des Reparaturalgorithmus

Nachdem die Analyse der Inkonsistenzen abgeschlossen ist, startet die Berechnung der Reparaturpläne. Tabelle 7.1 zeigt die Anzahl der Änderungen (bzw. die Summe aller Wertebereiche), die pro Editierregel (ER) betrachtet werden müssen. In einem zweiten Durchlauf werden die in Abschnitt 4.4 vorgestellten Heuristiken verwendet. Als minimale Lösung werden (abgerundet) 30% der Aktionen gefordert. Bei den hier untersuchten

[6] Intel(R) Core(TM) i7-3612QM CPU, 8GB Arbeitsspeicher

Editierregeln wurden trotzdem alle erwarteten Lösungen gefunden. Insgesamt wurden 10 Editierregeln ausgewählt, welche konsistente Modifikationen der Klassen-, Zustands- und Sequenzdiagramme durchführen. Allerdings wird teilweise nur eine begrenzte Auswahl an möglichen Belegungen der Parameter betrachtet. Beispielsweise wird eine „hängende Transition“ nur mit Zuständen in derselben Region verbunden. Solche Einschränkungen lassen sich über zusätzliche Anwendungsbedingungen der Editierregel angegeben. Parameter einer Editierregel, für die zu viele Modellelemente infrage kommen, sollten in Zukunft erst auf Nachfrage des Entwicklers berechnet werden.

Für die Komplexität und Laufzeit des Matchingalgorithmus (Abschnitt 4) ist im Wesentlichen die Anzahl der Aktionen und Änderungen pro Editierregel entscheidend. Die Größe des Modells beeinflusst dagegen hauptsächlich die Berechnung der Differenz. Grundsätzlich hat auch die Größe der Regelbasis Einfluss auf die Gesamtlaufzeit der Reparaturberechnung. Die Berechnung der Reparaturen lässt sich durch eine parallelisierte Verarbeitung der einzelnen Editierregeln beschleunigen. Außerdem könnte man eine große Editierregel δ_x ggf. zunächst in kleinere Teilschritte aufteilen, beispielsweise, wenn zwei Modellsichten bearbeitet werden. Dadurch würden zwei Teilregeln $\delta_x = \delta_p \cup \delta_q$ für die beiden Sichten entstehen. Eine Reparatur würde dann zunächst nur eine der beiden Sichten behandeln. Innerhalb der größeren Editierregel δ_x könnten dann allerdings die Aktionen der kleineren Regel δ_p und δ_q als atomare Abhängigkeit (Abschnitt 3) behandelt werden, wodurch die Komplexität der Berechnung deutlich verringert würde.

8 Schlussfolgerung

Das in dieser Arbeit vorgestellte Verfahren kann unvollständig durchgeführte Editierschritte in Modellen erkennen. Falls diese Editierschritte im Zusammenhang mit einer Inkonsistenz auftreten, können dadurch mögliche Reparaturpläne ermittelt werden, welche die unvollständigen Editierschritte ergänzen. Um die speziellen Eigenschaften einer domänenspezifischen Modellierungssprache abzubilden, verwendet das Verfahren komplexe graphtransformationsbasierte Editierregeln. Diese werden durch einen Matchingalgorithmus verarbeitet, der partiell ausgeführte Teile der Editierregeln in einer Differenz zwischen der zuletzt konsistenten und der aktuellen inkonsistenten Version eines Modells erkennt. Anschließend wird für jede partiell ausgeführte Editierregel eine Komplementregel abgeleitet, welche die noch nicht ausgeführten Aktionen einer erkannten Editierregel enthält. Alle möglichen Initialisierungen der Komplementregel ergeben dann die möglichen Ergänzungen eines unvollständig durchgeführten Editierschritts. Da nur ergänzende Editierschritte benötigt werden, welche Inkonsistenzen behandeln, wird der Matchingalgorithmus um eine syntaktische Analyse der fehlgeschlagenen Validierungen erweitert. Durch eine Analyse der ausgewerteten Konsistenzregel können alle möglichen abstrakten Reparaturen als Ansatzpunkte für mögliche Reparaturpläne gefunden werden. Ergänzende Editierschritte, welche eine abstrakte Reparatur initialisieren, werden dem Entwickler dann als Reparaturplan angeboten.

Die abstrakten Reparaturen werden auf Basis der abstrakten Syntax notiert. Im Gegensatz dazu werden die hier berechneten konkreten Reparaturpläne durch benutzerdefinierten Editierregeln beschrieben. Die so vorgeschlagenen Reparaturen sind für den Entwickler daher einfacher zu verstehen, da die Editierregeln meist an Operationen auf der konkreten Syntax angelehnt sind. Außerdem muss der Entwickler die Reparaturpläne nur mit wenigen zusätzlichen Parametern initialisieren, während abstrakte Reparaturen zunächst interpretiert und schrittweise initialisiert werden müssen. Reparaturalgorithmen ([HHR+11, MGC13]), welche benutzerdefinierte Editierregeln verwenden, führen meist eine Form der Zustandsraumsuche durch, bei der Editierregeln auf ein inkonsistentes Modell angewendet werden, bis ein konsistenter Zustand erreicht wird. Das hier vorgestellte Verfahren verwendet zusätzlich das zuletzt konsistente historische Modell

als Startzustand; das inkonsistente Modell stellt hier einen erwarteten Zwischenzustand dar. Außerdem werden die Editierregeln bei der Berechnung der Reparaturen nicht ausgeführt, sondern durch ein musterbasiertes Erkennungsverfahren ermittelt.

Die ermittelten Reparaturpläne ergänzen bereits durchgeführte Editierschritte. Eine korrekte Ergänzung existiert allerdings nicht immer. Es können auch fehlerhafte Veränderungen des Modells durchgeführt werden, welche sich nur durch das Zurücknehmen (*undo*) einzelner Änderungen korrigieren lassen. Ein Beispiel hierfür wäre das Erzeugen einer eingehenden Transition auf den Startzustand eines Zustandsautomaten. Reparierende Undo-Aktionen könnte man aber bei Bedarf aus der Differenz ableiten. Dadurch wird allerdings nicht ausgeschlossen, dass einzelne Aktionen innerhalb eines Reparaturplans auftreten, die bestimmte Änderungen der Differenz (als Seiteneffekt) zurücksetzten. Dies kann beispielsweise auftreten, wenn ein Attribut erneut modifiziert wird oder wenn ein gerade erzeugtes Element gelöscht wird. Sollen solche Reparaturen verhindert werden, wäre zusätzlich sicherzustellen, dass ein Reparaturplan keine kausalen Abhängigkeiten zu den technischen Änderungen der Differenz $D_{A \to B} \setminus D_{A \to A'}$ besitzt; wobei $D_{A \to A'}$ die minimale Differenz, inklusive des zu ergänzenden Editierschritts, enthält. In diesem Fall haben die berechneten Reparaturen die Eigenschaft, die bereits durchgeführten Änderungen zu bewahren (*„change-preserving“*) [TOLR17].

Die Qualität der Reparaturvorschläge hängt im Wesentlichen von der Menge der vorkonfigurierten Editierregeln ab. Für die Regelbasis sind insbesondere komplexe Editierregeln interessant die häufig zu Inkonsistenzen führen. Die Erstellung der Regelbasis erfordert allerdings ein tiefgehendes Verständnis der Modellierungsdomäne. Einfache Editierregeln, welche die grundlegenden Konsistenzbedingungen des Metamodells abbilden, können bereits automatisch abgeleitet werden [RKK14, KTRK16]. Eine weiterführende Aufgabe besteht darin, komplexe Editierregeln möglichst automatisch zu bestimmen. Ein möglicher Ansatz besteht darin, komplexe Editierregeln aus existierenden konsistenten Modellen abzuleiten. Sofern die entsprechenden Konsistenzregeln gegeben sind, könnte man alle komplexen Editierregeln ermitteln, welche das Modell ohne zwischenzeitliche Inkonsistenzen aufbauen. Eine andere Möglichkeit besteht darin, die Inkonsistenzen zunächst manuell zu beheben und diesen Vorgang aufzuzeichnen, um dann die entsprechende konsistenzerhaltende Editierregel aus der Modellhistorie und der Aufzeichnung abzuleiten.

Des Weiteren kann die berechnete Differenz Einfluss auf die Qualität der Reparaturvorschläge haben. Im Allgemeinen lässt sich die Differenzen zwischen zwei Modellversionen nicht eindeutig bestimmen. Je nachdem, welches Verfahren verwendet wird, können beispielsweise Verschiebungen nicht immer erkannt werden, was grundsätzlich dazu füh-

ren kann, dass bestimmte Reparaturvorschläge nicht gefunden werden. Um die Bildung der Korrespondenzen beeinflussen zu können, wird ein entsprechender Matchingalgorithmus benötigt, welcher auf die domänenspezifische Modellierungssprache angepasst werden kann [KKPS12]. Um mehr Reparaturen zu erkennen, könnte es außerdem interessant sein, die Überprüfung der Anwendungsbedingungen der Editierregeln abzuschwächen. Man könnte die Anwendungsbedingungen als erfüllt ansehen, wenn sich diese aus der aktuellen Modellversion und der historischen Modellversion zusammensetzen lassen. Dies wäre eine Abschätzung dafür, dass eine Anwendungsbedingung zum Zeitpunkt der Ausführung einer partiell erkannten Editierregel erfüllt war. Alternativ könnte man auch die Anzahl der nicht erfüllten Anwendungsbedingungen einer Komplementregel zählen und die Regel dann im Ranking der Reparaturen entsprechend abwerten.

Das vorgestellte Reparaturwerkzeug eignet sich insbesondere dazu, Inkonsistenzen zwischen verschiedenen Modellsichten zu reparieren. Eine Modellsicht kann entweder explizit im Modell definiert werden (z. B. durch verschiedene Diagramme) oder wird nach Bedarf als materialisierte Sicht durch einen Slicing-Algorithmus erzeugt. In beiden Fällen kann das Reparaturwerkzeug die Entwickler mit konkreten Reparaturplänen unterstützen, welche die bereits durchgeführten Modifikationen beachten. Auf Basis des aktuellen Modellzustands lässt sich beispielsweise nicht mehr entscheiden, ob eine Inkonsistenz durch das Hinzufügen, Löschen oder Verschieben von Modellelementen entstanden ist. Die Analyse der Modellhistorie hilft dabei, die Ursache einer Inkonsistenz und die Intention hinter den in diesem Zusammenhang durchgeführten Modifikationen besser zu verstehen.

Abbildungsverzeichnis

Literaturverzeichnis

[ABJ+10] Arendt, Thorsten ; Biermann, Enrico ; Jurack, Stefan ; Krause, Christian ; Taentzer, Gabriele: Henshin: advanced concepts and tools for in-place EMF model transformations. In: *Model Driven Engineering Languages and Systems* (2010), S. 121–135

[ACH+13] Androutsopoulos, Kelly ; Clark, David ; Harman, Mark ; Krinke, Jens ; Tratt, Laurence: State-based model slicing: A survey. In: *ACM Computing Surveys (CSUR)* 45 (2013), Nr. 4, S. 53

[BEE+10] Biermann, Enrico ; Ehrig, Hartmut ; Ermel, Claudia ; Golas, Ulrike ; Taentzer, Gabriele: Parallel independence of amalgamated graph transformations applied to model transformation. In: *Graph transformations and model-driven engineering*. Springer, 2010, S. 121–140

[BLK+16] Bashir, Raja S. ; Lee, Sai P. ; Khan, Saif Ur R. ; Chang, Victor ; Farid, Shahid: UML models consistency management: Guidelines for software quality manager. In: *International Journal of Information Management* 36 (2016), Nr. 6, S. 883–899

[BMMM09] Blanc, Xavier ; Mougenot, Alix ; Mounier, Isabelle ; Mens, Tom: Incremental detection of model inconsistencies based on model operations. In: *International Conference on Advanced Information Systems Engineering* Springer, 2009, S. 32–46

[CCGI11] Capozucca, Alfredo ; Cheng, Betty ; Guelfi, Nicolas ; Istoan, Paul: OO-SPL modelling of the focused case study. In: *ReMoDD repository, at http://www. cs. colostate. edu/remodd/v1/content/bcms-case-study-models-oo-spl-approach* (2011)

[Egy06] Egyed, Alexander: Instant consistency checking for the UML. In: *Proceedings of the 28th international conference on Software engineering* ACM, 2006, S. 381–390

[Egy07] Egyed, Alexander: UML/Analyzer: A tool for the instant consistency checking of UML models. In: *Proceedings of the 29th international conference on Software Engineering* IEEE Computer Society, 2007, S. 793–796

[EKHG01] Engels, Gregor ; Küster, Jochem M. ; Heckel, Reiko ; Groenewegen, Luuk: A methodology for specifying and analyzing consistency of object-oriented behavioral models. In: *ACM SIGSOFT software engineering notes* 26 (2001), Nr. 5, S. 186–195

[EMF] *Eclipse Modeling Framework (EMF).* `http://www.eclipse.org/modeling/emf/`, . – 2017-04-27

[FGL12] Foster, Stephen R. ; Griswold, William G. ; Lerner, Sorin: WitchDoctor: IDE support for real-time auto-completion of refactorings. In: *Proceedings of the 34th International Conference on Software Engineering* IEEE Press, 2012, S. 222–232

[GPR07] Gruhn, Volker ; Pieper, Daniel ; Röttgers, Carsten: *MDA®: Effektives Software-Engineering mit UML2® und EclipseTM.* Springer-Verlag, 2007

[Hen] *Henshin.* `https://www.eclipse.org/henshin/`, . – 2017-04-27

[HHR+11] Hegedüs, Abel ; Horváth, Akos ; Ráth, István ; Branco, Moisés C. ; Varró, Dániel: Quick fix generation for DSMLs. In: *VL/HCC* Bd. 11, 2011, S. 17–24

[Kec09] Kecher, Christoph: UML 2. In: *Das umfassende Handbuch. Bonn* (2009)

[Keh15] Kehrer, Timo: *Calculation and propagation of model changes based on user-level edit operations*, Ph. D. thesis, University of Siegen, Diss., 2015

[KH04] Krissinel, Evgeny B. ; Henrick, Kim: Common subgraph isomorphism detection by backtracking search. In: *Software: Practice and Experience* 34 (2004), Nr. 6, S. 591–607

[KK99] Kim, Dohyung ; Kim, Jihong: Design and implementation of a Java-based MPEG-1 video decoder. In: *IEEE Transactions on Consumer Electronics* 45 (1999), Nr. 4, S. 1176–1182. – `http://peace.snu.ac.kr/dhkim/java/MPEG/`

[KKPS12] Kehrer, Timo ; Kelter, Udo ; Pietsch, Pit ; Schmidt, Maik: Adaptability of model comparison tools. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* ACM, 2012, S. 306–309

[KKR14] Kehrer, Timo ; Kelter, Udo ; Reuling, Dennis: Workspace updates of visual models. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering* ACM, 2014, S. 827–830

[KKT11] Kehrer, Timo ; Kelter, Udo ; Taentzer, Gabriele: 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* IEEE Computer Society, 2011, S. 163–172

[KKT13] Kehrer, Timo ; Kelter, Udo ; Taentzer, Gabriele: Consistency-preserving edit scripts in model versioning. In: *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on* IEEE, 2013, S. 191–201

[KM14] Kuschke, Tobias ; Mäder, Patrick: Pattern-based auto-completion of UML modeling activities. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering* ACM, 2014, S. 551–556

[KMR13] Kuschke, Tobias ; Mäder, Patrick ; Rempel, Patrick: Recommending auto-completions for software modeling activities. In: *International Conference on Model Driven Engineering Languages and Systems* Springer, 2013, S. 170–186

[KTRK16] Kehrer, Timo ; Taentzer, Gabriele ; Rindt, Michaela ; Kelter, Udo: Automatically deriving the specification of model editing operations from meta-models. In: *International Conference on Theory and Practice of Model Transformations* Springer, 2016, S. 173–188

[LMT09] Lucas, Francisco J. ; Molina, Fernando ; Toval, Ambrosio: A systematic review of UML model consistency management. In: *Information and Software Technology* 51 (2009), Nr. 12, S. 1631–1645

[McG82] McGregor, James J.: Backtrack search algorithms and the maximal common subgraph problem. In: *Software: Practice and Experience* 12 (1982), Nr. 1, S. 23–34

[Mee98] Meertens, Lambert: *Designing constraint maintainers for user interaction.* 1998

[MGC13] Macedo, Nuno ; Guimaraes, Tiago ; Cunha, Alcino: Model repair and transformation with Echo. In: *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on* IEEE, 2013, S. 694–697

[MJC16] Macedo, Nuno ; Jorge, Tiago ; Cunha, Alcino: A feature-based classification of model repair approaches. In: *IEEE Transactions on Software Engineering* (2016)

[MM03] Miller, Joaquin ; Mukerji, Jishnu: Model Driven Architecture (MDA) 1.0. 1 Guide. Object Management Group. In: *Inc.(June 2003)* (2003)

[MVDS06] Mens, Tom ; Van Der Straeten, Ragnhild: Incremental resolution of model inconsistencies. In: *International Workshop on Algebraic Development Techniques* Springer, 2006, S. 111–126

[MVDSD06] Mens, Tom ; Van Der Straeten, Ragnhild ; D'Hondt, Maja: Detecting and resolving model inconsistencies using transformation dependency analysis. In: *International Conference on Model Driven Engineering Languages and Systems* Springer, 2006, S. 200–214

[NEF03] Nentwich, Christian ; Emmerich, Wolfgang ; Finkelstein, Anthony: Consistency management with repair actions. In: *Software Engineering, 2003. Proceedings. 25th International Conference on* IEEE, 2003, S. 455–464

[NER00] Nuseibeh, Bashar ; Easterbrook, Steve ; Russo, Alessandra: Leveraging inconsistency in software development. In: *Computer* 33 (2000), Nr. 4, S. 24–29

[Ohr15] Ohrndorf, Manuel: *Algorithmen zur Erkennung und Reparatur von Inkonsistenzen in Software-Modellen.* Forschungsseminar - Uni Siegen, Fakultät IV, Institut für Praktische und Technische Informatik, Lehrstuhl Softwaretechnik und Datenbanksysteme, 2015

[OMG14] Object Management Group, OMG: Object Constraint Language Version 2.4. In: *Object Management Group* (2014). – http://www.omg.org/spec/OCL/2.4/

[OMG15] Object Management Group, OMG: OMG Unified Modeling Language TM (OMG UML) Version 2.5. In: *Object Management Group* (2015). – http://www.omg.org/spec/UML/2.5

[PTN+07] Pietrek, Georg ; Trompeter, Jens ; Niehues, Benedikt ; Kamann, Thorsten ; Holzer, Boris ; Kloss, Michael ; Thoms, Karsten ; Beltran, Juan Carlos F. ; Mork, Steffen: Modellgetriebene Softwareentwicklung: MDA und MDSD in der Praxis. In: *Software+ Support* (2007)

[PVDSM15] Puissant, Jorge P. ; Van Der Straeten, Ragnhild ; Mens, Tom: Resolving model inconsistencies using automated regression planning. In: *Software & Systems Modeling* 14 (2015), Nr. 1, S. 461–481

[RE12a] Reder, Alexander ; Egyed, Alexander: Computing repair trees for resolving inconsistencies in design models. In: *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on* IEEE, 2012, S. 220–229

[RE12b] Reder, Alexander ; Egyed, Alexander: Incremental consistency checking for complex design rules and larger model changes. In: *International Conference on Model Driven Engineering Languages and Systems* Springer, 2012, S. 202–218

[RKK14] Rindt, Michaela ; Kehrer, Timo ; Kelter, Udo: Automatic Generation of Consistency-Preserving Edit Operations for MDE Tools. In: *Demos@ MoDELS* Citeseer, 2014

[SBMP08] Steinberg, Dave ; Budinsky, Frank ; Merks, Ed ; Paternostro, Marcelo: *EMF: eclipse modeling framework*. Pearson Education, 2008

[SC02] Sourrouille, Jean L. ; Caplat, Guy: Constraint checking in UML modeling. In: *Proceedings of the 14th international conference on Software engineering and knowledge engineering* ACM, 2002, S. 217–224

[SiD] *SiDiff*. http://www.sidiff.org/, . – 2017-04-27

[SiL] *SiLift*. http://pi.informatik.uni-siegen.de/projects/SiLift/, . – 2017-04-27

[SWK09] Schmidt, Maik ; Wenzel, Sven ; Kelter, Udo: Transaktionsorientiertes Mischen von Modellen. In: *Software Engineering*, 2009, S. 165–170

[SZ01] Spanoudakis, George ; Zisman, Andrea: Inconsistency management in software engineering: Survey and open research issues. In: *Handbook of software engineering and knowledge engineering* 1 (2001), S. 329–380

[TOLR17] Taentzer, Gabriele ; Ohrndorf, Manuel ; Lamo, Yngve ; Rutle, Adrian: Change-Preserving Model Repair. In: *International Conference on Fundamental Approaches to Software Engineering* Springer, 2017, S. 283–299

[VDSD06] Van Der Straeten, Ragnhild ; D'Hondt, Maja: Model refactorings through rule-based inconsistency resolution. In: *Proceedings of the 2006 ACM symposium on Applied computing* ACM, 2006, S. 1210–1217

[VV08] Vismara, Philippe ; Valery, Benoît: Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms. In: *Modelling, Computation and Optimization in Information Systems and Management Sciences*. Springer, 2008, S. 358–368

[Wei79] Weiser, Mark D.: *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. Ann Arbor, MI, USA, University of Michigan, Diss., 1979. – AAI8007856

[WHR14] Whittle, Jon ; Hutchinson, John ; Rouncefield, Mark: The state of practice in model-driven engineering. In: *IEEE software* 31 (2014), Nr. 3, S. 79–85

[Xte] *Xtext*. https://eclipse.org/Xtext/, . – 2017-04-27

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig unter ausschließlicher Nutzung der angegebenen Quellen und Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche gekennzeichnet.

Diese Arbeit lag in gleicher oder ähnlicher Weise noch keiner Prüfungsbehörde vor und wurde bisher noch nicht veröffentlicht.

Siegen, 29. April 2017

Manuel Ohrndorf