Evolution von Meta-Modellen mit sprachbasierten Mustern

Matthias Jahn

Evolution von Meta-Modellen mit sprachbasierten Mustern

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von
Matthias Stefan Jahn
geboren in Sonneberg

1. Gutachter: Prof. Dr.-Ing. Stefan Jablonski
2. Gutachter: Prof. Dr. rer. pol. Ulrich Frank

Tag der Einreichung: 20. Juni 2014
Tag des Kolloquiums: 5. Dezember 2014

Für meine Familie.

Zusammenfassung

In der Softwareentwicklung der vergangenen Jahre wird häufig ein Sachverhalt durch eine speziell auf ein Anwendungsgebiet zugeschnittene Sprache beschrieben, die man domänenspezifische Modellierungssprachen (DSML) nennt. Dadurch können Ausschnitte aus einer Anwendungsdomäne präzise abgebildet werden. Um den Aufwand während der Erstellung dieser Sprache gering zu halten, kommen meist Meta-Modelle zum Einsatz, die die konkrete und abstrakte Syntax definieren. Basierend auf diesen Meta-Modellen werden Instanz-Modelle erstellt, die dann einen konkreten Teil einer Domäne abbilden. Durch den hohen Grad an Spezialisierung den DSMLs aufweisen, stellt sich speziell in frühen Phasen der Entwicklung häufig eine Evolution ein, um neue Anforderungen oder eine Änderung im Verständnis einer Domäne in der entwickelten Sprache abbilden zu können. Als Konsequenz müssen Instanz-Modelle dahingehend angepasst werden, dass sie der neuen Sprachversion genügen, um die in ihnen enthaltenen Informationen weiterhin verwenden zu können.

Die Entwicklung einer DSML ist geprägt durch den Anspruch, die Verständlichkeit und Ausdrucksstärke von Modellen voranzutreiben. Im Bereich der Meta-Modellierung wurden deshalb verschiedene Sprachmuster wie die Vererbung, Deep Instantiation, Materialisierung, Instanz-Spezialisierung, Clabjects und der Powertyp entwickelt, die dazu dienen die Qualität eines Meta-Modells zu verbessern. Nichtsdestotrotz führt die Anwendung eines Sprachmusters auf ein existierendes Meta-Modell zu einer Sprachevolution, von der viele Elemente im Modell beeinflusst werden können. Die Migration ist folglich nicht trivial. Da in aktuellen Systemen oder Ansätzen innerhalb der Forschung solch eine Sprachevolution nur wenig berücksichtigt wird, müssen diese manuell durch den Modellierer vollzogen werden. Daher ist diese Aufgabe zeitaufwändig und fehleranfällig.

Die vorliegende Dissertation widmet sich diesem Problem und liefert Methoden, um Meta-Modelle mit Sprachmustern in einfacher Weise zu ändern. Dafür wird eine Bibliothek an Evolutionsoperatoren vorgestellt, die den Umgang mit dieser Evolution ermöglicht. Genauer gesagt wird für jede Änderung einer Eigenschaft eines Modellelements, die einen ungültigen Zustand eines Modells verursachen könnte, ein Operator bereitgestellt. Dieser übernimmt die Evolution und berücksichtigt dabei auch vorhandene Sprachmuster im Modell. Weiterhin führt der Operator auch die Migration ungültiger Elemente im Modell durch, die von der Änderung betroffen sind. Als weiterer Beitrag wurden für die Sprachmuster Operatoren definiert, die sie in ein existierendes Meta-Modell einführen, existierende Sprachmuster anpassen oder sie aus dem Modell entfernen. Diese verwenden als Grundlage die elementaren Operatoren und sind somit modular aufgebaut. Als weiterer Punkt stellt diese Arbeit einen Ansatz vor, der die Sprachmuster und deren Evolution für den Benutzer transparent darstellt. Dadurch wird dem Benutzer eine Möglichkeit geboten, verschiedene Evolutionsvarianten anhand der elementaren Auswirkungen zu bewerten. Außerdem wird der Lernprozess im Umgang mit Sprachmustern, durch die Visualisierung der Auswirkungen auf betroffene Modellelemente, unterstützt.

Abstract

Nowadays, software developers tend to use domain specific modelling languages (DSML) to describe parts of the software system using terms of the domain. As a consequence, the resulting DSML is able to precisely represent the domain part in a concise way. Nevertheless, creating languages is not a trivial task. That is why several tools exist supporting efficient development of DSMLs. These tools often use meta models defining integral parts of the DSML like abstract and concrete syntax. Conforming to these meta models, instance models can be built representing the concrete parts of the domain of interest. The high degree of specialization of a DSML causes a language adaption especially in early language development phases to cope with arising domain requirements. Hence, models expressed in the old meta model version might get invalid and have to be migrated to be utilized in the system again.

The development of a DSML is driven by the aim to improve expressiveness and conciseness. Therefore, the current research in the field of meta modelling has discovered language patterns like inheritance, deep instantiation, materialization, prototyping, clabjects and (extended) powertypes which help to improve meta model quality. However, introducing such patterns into an existing meta model is coupled with language evolution influencing various model elements. Hence, discovering a migration strategy is not trivial. Owing to the fact that such kind of language evolution is nearly not considered in current systems or research approaches, this migration needs to be defined manually by the modeller. Thus, this task is time-consuming and error-prone.

This thesis focuses the above stated challenge and provides a set of evolution operators which support evolution of meta models using language patterns. For each change of a model elements property that might cause an invalid model state, an operator is defined that transforms the model. These transformations consider the model paradigm constraints presented in this thesis and the compliance of the language patterns' semantic. Furthermore, a catalogue of operators is presented enabling an semiautomatic introduction of language patterns into existing meta models or a change of them. These catalogue leverages the set of elementary modelling operators in order to reuse their functionality. Additionally, an approach is presented that outlines modellers the pattern effects to specific model elements and hence supports dealing with language patterns. With that, different evolution possibilities can be reviewed by the modeller.

Danksagung

Die Erstellung einer Dissertation ist ein Prozess, der einer gewissen Evolution unterliegt. Diese Evolution wurde in meinem Fall auch maßgeblich von anderen Personen beeinflusst, denen ich an dieser Stelle meinen Dank aussprechen möchte.

Zu allererst möchte ich meinen Doktorvater Prof. Dr.-Ing. Stefan Jablonski danken, der mir in den vergangenen vier Jahren stets zur Seite stand und mir in zahlreichen Diskussionen viele Hinweise, Ratschläge und Ideen unterbreitet hat, die es mir ermöglichten, diese Dissertation zu verfassen. Daneben möchte ich mich auch bei allen anderen Mitarbeitern des Lehrstuhls für die tolle Zeit bedanken, in der wir auch außerhalb der Arbeit viele schöne Momente geteilt haben.

Besonders möchte ich dabei meinen Kollegen Bastian Roth erwähnen, der mir in vielen Situationen mit Ideen und Kritiken zur Seite stand und damit einen maßgeblichen Anteil an der Erstellung dieser Dissertation trägt. Weiterhin möchte ich mich bei Ihm bedanken für die gemeinsame Entwicklung der Model Workbench, die als Grundlage des Prototyps dieser Arbeit dient. Zusätzlich möchte ich mich noch bei meine Kollegen Lars Ackermann, Stefan Schönig und Michael Zeising bedanken, die mir in der Zeit am Lehrstuhl ebenfalls sehr ans Herz gewachsen sind und dabei einige Male mein Leid beim Überwinden mancher Schwierigkeiten während der Erstellung dieser Dissertation ertragen mussten.

Prof. Dr. rer. pol. Ulrich Frank möchte ich Dank aussprechen für den Austausch von Anregungen und Hinweisen, sowie die Übernahme des Zweitgutachtens.

Weiterhin möchte ich mich bei meiner Familie bedanken, die der wichtigste Teil meines Lebens ist und die mich auch in schwierigsten Zeiten zum Lachen bringt. Ihr wart mir stets eine große Hilfe und meine Inspiration. Ich liebe euch!

Zu guter Letzt möchte ich auch meinen Eltern und meinem Bruder danken, denen ich nicht nur sehr viel zu verdanken habe, sondern denen auch die Aufgabe zu Teil wurde, diese Arbeit auf typografische Fehler zu untersuchen.

Inhaltsverzeichnis

1 Einleitung

Durch die zunehmende Komplexität der Anwendungsszenarien wurde die Softwareentwicklung in den letzten Jahren vor große Herausforderungen gestellt [21, 34, 39]. Neben der eigentlichen Entwicklung und Umsetzung erschwert dies auch die Wartung und Evolution eines Softwaresystems. Deshalb hat sich das Gebiet des Software Engineerings zum Ziel gesetzt, Methoden und Konzepte zur systematischen Erstellung von großen und komplexen IT-Systemen bereitzustellen. Diese Systeme werden für den jeweiligen Bereich, auch Anwendungsdomäne genannt, speziell zugeschnitten. Um die komplexen Abläufe und Strukturen einer Anwendungsdomäne schrittweise zu analysieren, werden Modelle [69, 108, 123] erstellt. Diese abstrahieren von irrelevanten Details eines konkreten Abschnittes der Domäne [107] und dienen als Repräsentant für das jeweilige Original im Kontext des Softwaresystems. Sie werden folglich nach der Erstellung dazu verwendet, das Verhalten in der Anwendungsdomäne abzubilden. Dadurch ermöglichen sie es die Modelle zu validieren und daraus Quellcode zu generieren, um ein lauffähiges System zu erhalten. Diese Vorgehensweise nennt man aufgrund des zentralen Modellierungsgedankens folglich Modellgetriebene Softwareentwicklung (MDSD) [108].

Bei der MDSD sind also Modelle die Kernelemente eines Softwaresystems. Sie müssen nicht nur vom Menschen (Entwickler oder Dritte) sondern auch vom Computer interpretiert werden können. Dazu wird eine formale Sprache (Modellierungssprache) benötigt, in der Modelle der Anwendungsdomäne beschrieben werden. Im Laufe der Jahre haben sich für bestimmte Anwendungsbereiche Sprachen entwickelt, die von verschiedenen Gruppen standardisiert wurden. Dadurch entstand eine Grundlage zur Interoperabilität von Softwaresystemen, die diese Sprachen verwenden, was entscheidend zur Verbreitung einer Modellierungssprache beitrug. Beispiele solcher Sprachen sind *Business Process Model and Notation* (BPMN) für die Beschreibung von Geschäftsmodellen [84], *Entity-Relationsship* Modelle (ER) für die Modellierung von Datenschemata [33] oder die *Unified Modeling Language* für die Modellierung objektorientierter Software [86]. Besonders die UML besitzt den Anspruch in möglichst jedem Anwendungsgebiet nutzbar zu sein. Dieser Anspruch wird durch den Erweiterungsmechanismus der UML2 über Profile realisiert. Ein Beispiel für ein solches Profil stellt das Profil zur Modellierung von Enterprise Java Beans [43] dar, die häufig im Kontext von Java EE Anwendung finden.

Trotz der beliebigen Erweiterbarkeit der UML2 setzen immer mehr Softwareentwickler für die Modellierung einer Anwendungsdomäne eine domänenspezifische Sprache (DSL) oder auch domänenspezifische Modellierungssprache (DSML) ein, die im Vergleich zur UML lediglich das Ziel besitzt, eine gewisse Klasse von Problemen zu lösen. Diese Sprachen werden also für das jeweilige Anwendungsgebiet definiert und nutzen das Vokabular der Domäne, um Strukturen und Abläufe darin zu beschreiben. Um den Aufwand bei der Definition einer DSML in Grenzen zu halten, haben sich in den letzten Jahre vielzähligen Werkzeuge wie das Eclipse Modeling Framework (EMF) [109] und Xtext [122], Microsoft DSL Tools [22] oder MPS [115] entwickelt. Neben der Definition der Sprache werden in solche Umgebungen auch Mittel geliefert, um einfach Werkzeuge zu erstellen, die es erlauben, einfach mit der Sprache umzugehen und Modelle in ihr zu erzeugen.

Für die Erstellung einer DSML bedarf es neben einem Mapping zwischen den im Folgenden beschriebenen Bestandteilen im Wesentlichen drei Definitionen ([21], S16ff.):

- Definition der *abstrakten Syntax*: Die abstrakte Syntax einer Sprache definiert die verwendeten Konstrukte der Sprache und wie diese miteinander in Verbindung gesetzt werden können.

- Definition einer *konkreten Syntax*: Die konkrete Syntax beschreibt die Notation, in der die Sprache dargestellt ist. Üblicherweise ist diese meist grafisch oder textuell.
- Definition einer *Semantik*: Die Semantik einer Sprache definiert die Bedeutung der in der abstrakten Syntax definierten Konstrukte.

Um die Sprache zu formulieren, geht der Trend dahin, neben der Beschreibung der Anwendungsdomäne mit Hilfe von Modellen, auch die abstrakte und konkrete Syntax einer DSML mit Modellen zu beschreiben [65, 66, 116], um die Techniken der Modellierung wiederzuverwenden. Da diese Modelle beschreiben, wie Modelle aufgebaut werden, nennt man sie Meta-Modelle. Durch die Definition der abstrakten und der konkreten Syntax mittels Meta-Modellen lassen sich sowohl Modelle, deren Sprachen und auch die Sprachen der Modellierungssprachen (Meta-Meta-Modelle) auf die gleiche Art und Weise verarbeiten, wodurch die Nutzung in einem entsprechenden Werkzeug wesentlich erleichtert. Folglich [65] dienen in diesem Fall die Meta-Modelle, die die DSML definieren, als Grundlage für die Erstellung von Editoren und Code Generatoren. Diese Werkzeuge können dann wiederum dazu verwendet werden, um mit ihnen Modelle konform zum Meta-Modell zu definieren und damit den mit der DSML beschriebenen Teil der Anwendungsdomäne zu modellieren. Die Begriffe Modell, Meta-Modell bzw. Meta-Meta-Modell werden im Folgenden immer mit einem Bezugspunkt verwendet, da die in dieser Arbeit betrachteten Meta-Hierarchien nicht abgeschlossen sind. Folglich wird der Begriff des Modells zunächst ebenen-unabhängig verwendet. Die anderen Begrifflichkeiten wie Meta- oder Instanz-Modell beziehen sich dann immer auf das Modell, welches die Sprache des aktuellen Modells beschreibt oder konform zu diesem ist.

1.1 Beispiel einer Modellierungssprache

In Abbildung 1-1 ist eine beispielhafte DSML zur Modellierung einfacher Prozesse dargestellt. Sie beinhaltet drei grundsätzliche Arten von Konzepten. Dabei ist eine Instanz von `Start` als Anfang, eine

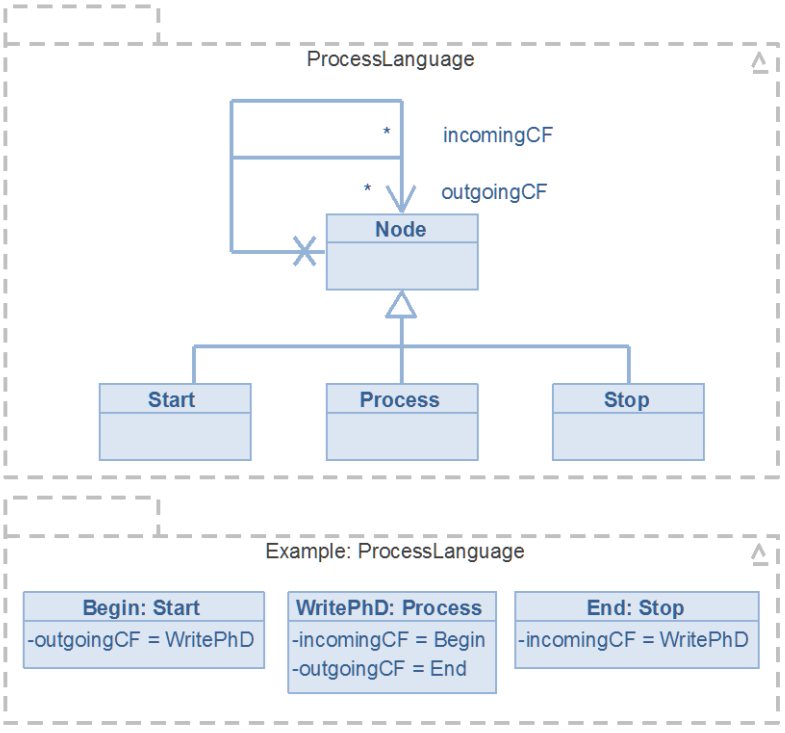

Abbildung 1-1 Einfache Prozessmodellierungssprache

Instanz von `Stop` als Ende eines Prozesses anzusehen und Instanzen von `Process` bilden Unterprozesse des modellierten Prozesses. Die gezeigte Sprache ist sehr einfach und könnte zum Beispiel noch um verschiedene Gateways (And, Or, Xor) erweitert werden. Alle beschrieben Konzepte sind Spezialisierungen von `Node`, das andere Konten als Vorgänger mittels `incomingCF` und Nachfolger mittels `outgoingCF` besitzen kann.

Neben der eigentlichen Sprache (Ebene `ProcessLanguage`) wurde in diesem Beispiel auch ein Modell (`Example`) in der Sprache erstellt. Es beschreibt den (stark vereinfachten) Ablauf beim Erstellen einer Dissertation. Zunächst beginnt der Prozess mit dem Konzept `Begin`, das eine Instanz von `Start` ist, und wird dann mit dem Teilprozess `WritePhD`, also dem Erstellen der Dissertation, fortgesetzt. Dieses Konzept ist folglich auch eine Instanz von `Process`, während der letzte Schritt (`End`) eine Instanz von `Stop` darstellt. Der Kontrollfluss von `Begin` über `WritePhD` zu `End` wird durch die Zuweisungen an den drei Konzepten für die Attribute `incomingCF` bzw. `outgoingCF` modelliert.

1.2 Problemstellung

Im Laufe der Modellierung einer Anwendungsdomäne kommt es häufig neben Änderungen in den erstellten Modellen auch zu Änderungen in der DSML selbst. Diese beruhen meist darauf, dass sich das Verständnis für die Anwendungsdomäne entwickelt oder der Abstraktionsgrad der Sprache angepasst werden muss, da dieser zunächst falsch gewählt wurde [39, 40]. Wie schwierig die Festlegung des richtigen Abstraktionsgrades ist, zeigt auch die Entwicklung der UML: In der ersten Version der UML war das Ziel gesetzt worden, Plattformunabhängigkeit der Modelle zu erreichen und diese dann auf die jeweiligen Plattformen in Form von Code zu transformieren. Im Laufe der Zeit stellte sich aber heraus, dass plattformrelevante Informationen auch im Modell aufgenommen werden müssen, um den Codegenerierungsprozess plattformabhängig steuern zu können. Um dieser (und anderer) Anforderung gerecht zu werden, wurden der UML2 die Profile hinzugefügt [75], die es erlauben, die UML an definierten Punkten zu erweitern.

Da die Meta-Modelle der abstrakten und konkreten Syntax wichtige Komponenten einer Sprache beschreiben, ist die Qualität und Ausdrucksstärke dieser Meta-Modelle von entscheidender Bedeutung. Um dieser Herausforderung zu begegnen, wurden deshalb in der Forschung der vergangenen Jahre verschiedene Muster entwickelt, die zur Strukturierung und Vereinfachung von Modellen beitragen [78] und dabei helfen, diese einfacher zu verstehen [51]. Ähnlich wie die Anwendung eines Design Pattern [42] helfen sprachbasierte Muster wie Powertyp, Materialisierung, Instanz-Spezialisierung, Deep Instantiation oder Vererbung (siehe Kapitel 2) außerdem, die Dokumentationen von Modelle (im Allgemeinen) zu erleichtern oder diese gänzlich zu ersetzen. Daneben beeinflussen sie mit ihrer Modellierungssemantik den Abstraktionsgrad eines Modells. Allerdings ist die Anwendung eines Sprachmusters auf ein bereits existierendes Modell nicht trivial, da sich jedes Sprachmuster auf viele konkrete Modellelemente auswirkt. Hinzu kommt, dass durch das Einführen oder Ändern eines Sprachmusters die Meta-Modelle (und damit die Sprache) einer Evolution unterzogen werden.

Solch eine Änderung der DSML wirkt sich nicht nur auf die Sprache selbst, sondern auch auf die in ihr modellierten Modelle aus. Dies führt dazu, dass Teile von Modellen oder ganze Modelle und die entsprechenden Werkzeuge, wie Editoren oder Code-Generatoren, [30, 100, 101] invalide in Bezug auf die neue Sprachversion werden. Um die in den Modellen enthalten Informationen wieder verwenden zu können, müssen die fehlerhaften Teile auf die neue Version des Meta-Modells angepasst werden. Diese Migration der Modelle muss in den meisten Fällen manuell erfolgen, was fehleranfällig, zeitaufwändig und damit auch kostenaufwändig ist. Deshalb haben Mens et al. [73] oder auch Favre

in [34] die Unterstützung von Meta-Modell Evolution gemeinsam mit entsprechender Modell Migration (häufig Koevolution genannt) als eine wichtige Herausforderung formuliert. In den letzten Jahren wurden einige Ansätze (siehe Kapitel 3) entwickelt, die sich dieser Herausforderung annehmen. Diese lassen sich in zwei große Gruppen einteilen. Zum einen gibt es den Ansatz zwei verschiedene Modellversionen miteinander zu vergleichen und daraus Unterschiede in den beiden Meta-Modellen abzuleiten. Aus diesen wird dann, meist manuell, eine Modelltransformation erstellt, die die Koevolution der entsprechenden Modelle vollzieht. Zum anderen hat sich ein Ansatz entwickelt, der Evolutionsoperatoren bereitstellt, die vom Benutzer gezielt verwendet werden, um das Meta-Modell schrittweise zu verändern. Dadurch treten im Gegensatz zum ersten Ansatz [94] keine Mehrdeutigkeiten in der Abfolge der Evolutionsschritte auf und müssen daher auch nicht aufgelöst werden. Folglich haben diese Ansätze die Möglichkeit bestimmte Semantiken in der Evolution abzubilden und leisten daher eine gewisse Abstraktion bezüglich der einfachen Modelländerungen. Beispielsweise lässt sich das Verschieben eines Attributes in diesem zweiten Ansatz durch einen (vielleicht gleichnamigen) Operator einfach abbilden. Im Gegenzug kann der erste Ansatz zwischen dem Verschieben und dem Löschen des Attributes zusammen mit einer Erstellung eines gleichnamigen Attributes an anderer Stelle nicht unterscheiden.

Ein weiteres Beispiel für das Auftreten einer komplexen Modellierungssemantik stellt der Einsatz eines sprachbasierten Musters dar. Trotz der vielen Ansätze und Werkzeuge in dem Bereich der Meta-Modell Evolution bietet keiner bzw. keines eine ausreichende Möglichkeit mit sprachbasierten Mustern in Modellen umzugehen. Lediglich für das Sprachmuster der Vererbung wurde in einigen Arbeiten (z.B. [56, 119]) eine entsprechende Strategie entwickelt, die es erlaubt, das Muster einzuführen oder bestimmte Eigenschaften des Musters zu ändern. Da die Sprachmuster weitreichende Auswirkungen auf verschiedene Modellelemente haben, sind die Änderungen, die sich während der Einführung und Änderung eines solchen Musters ergeben, komplex. Mit den bisherigen Methoden der aktuellen Forschung können diese Vorgänge nur schwer abgebildet werden.

Problemstellung: Aktuelle Ansätze in der Forschung haben sich mit der Evolution von Meta-Modellen bereits intensiv beschäftigt. Allerdings wurden die Einführung oder Änderungen von sprachbasierten Mustern dabei wenig bis gar nicht untersucht (Ausnahme bildet das Sprachmuster der Vererbung). Da die sprachbasierten Muster komplex sind und sie zumeist viele Modellelemente von verschiedenen Modellen beeinflussen, ist dieser Vorgang nicht trivial. Außerdem bildet jedes der verschiedenen Sprachmuster eine semantische Einheit. Diese Semantik kann durch die bisherigen Methoden im Bereich der Meta-Modell Evolution (siehe Kapitel 3) nicht einfach und eindeutig abgedeckt werden. Deshalb muss die Anpassung der invaliden Modellelemente, die im Zuge der Evolution durch den Umgang mit sprachbasierten Mustern erfolgt, bisher manuell vollzogen werden. Damit ist diese Art der Evolution eine potentielle Quelle für Fehler und zudem zeit- und kostenaufwändig, weshalb hierfür Aushilfe geschaffen werden sollte.

1.3 Evolution der Beispielsprache

Wie bereits gezeigt, erlaubt das im Beispiel 1.1 gezeigte Meta-Modell eines Prozesses, Knoten beliebiger Art (Prozess, Start, Stop) miteinander über eingehende und ausgehende Kontrollflüsse zu verbinden. Dies mag syntaktisch korrekt sein, semantisch gesehen ist das Setzen von Vorgängern bei Startknoten und Nachfolgern beim Stoppknoten jedoch wenig sinnvoll.

Um diese Nachlässigkeit beim Modellieren zu korrigieren, könnte man beispielsweise einen erweiterten Powertyp (detaillierte Erläuterungen zum Muster finden sich in Kapitel 2) verwenden. Wie dies am konkreten Beispiel aussieht ist in Abbildung 1-2 dargestellt. Der darin modellierte erweiterte Powertyp `NodeKind` hat zum einen den Vorteil, dass jede Instanz auch eine Spezialisierung des partitionierten Typs `Node` ist. Zum anderen definiert `NodeKind` Diskriminatorattribute (`supportsInCF` und `supportsOutCF`), die es erlauben zu steuern, welche Attribute (`incomingCF` bzw. `outgoingCF`) ein Konzept erbt. Folglich kann bei den Konzepten `Start` und `Stop` das Erben des eingehenden bzw. ausgehenden Kontrollflusses von `Node` verhindert werden, wodurch die Zuweisung bei einer entsprechenden Instanz nicht mehr möglich wäre.

Um das erweiterte Powertyp Muster im Modell einzuführen, müsste aber sowohl eine neue Meta-Ebene als auch ein neues Konzept mit Attributen erzeugt, das Konzept `Node` verschoben und Zuweisungen bei allen Spezialisierungen getätigt werden. Es zeigt sich also, dass bereits an diesem kleinen Beispiel viele Änderungen nötig sind, um das Sprachmuster in das vorhandene Modell zu integrieren. Daher ist es fraglich, ob eine manuelle Migration des Modells immer vollständig und korrekt durchgeführt werden kann.

Die gewünschte Situation wäre folglich, dass ein Werkzeug vorliegt, an dem der Benutzer lediglich die Strategie zur Einführung eines erweiterten Powertypen für `Node` auswählt. Alle restlichen Schritte werden dann mit Hilfe einiger Informationen des Benutzers (siehe unten) durchgeführt.

- Die Strategie für die Einführung eines erweiterten Powertypen würde also konkret so vorgeben, dass zunächst ein Name für eine neue Ebene (`ModelingLanguage`) gewählt wird.

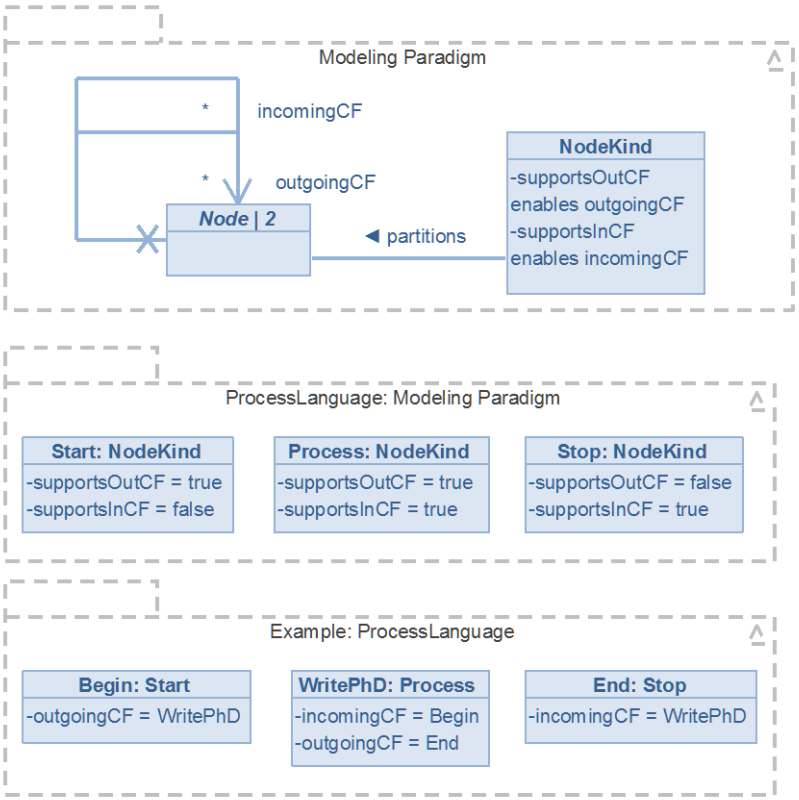

Abbildung 1-2 einfache Sprache zur Prozessmodellierung mit erweitertem Powertyp Muster

- Anschließend würde die neue Ebene erstellt, als Meta-Ebene von `ProcessLanguage` deklariert und das Konzept `Node` zur neuen Ebene verschoben werden.
- Danach müsste ein neuer Powertypen `NodeKind` (Name durch den Benutzer gewählt) erzeugt und die Diskriminatorattribute mitsamt den Zuweisungen bei `Start`, `Process` und `Stop` anlegt werden, um das in Abbildung 1-2 gezeigte Zielmodell zu erhalten.

Die vorgestellte Situation tritt in ähnlicher Form bei allen Änderungen auf, bei denen Sprachmuster eine Rolle spielen. Im Allgemeinen verlangen solche Anpassungen immer auch weitreichenden Migrationen von anderen betroffenen Modellelementen.

1.4 Herausforderungen

Die in Abschnitt 1.2 beschriebene Problemstellung führt zu drei Herausforderungen. Sie stellen die Richtlinien dieser Arbeit dar und bilden weiterhin die Grundlage für Anforderungen an bestehende Ansätze, die in Kapitel 3 dazu dienen diese zu evaluieren.

1.4.1 H1: Strategie zur Einführung und Änderung von sprachbasierten Entwurfsmustern

Die in Kapitel 2 vorgestellten sprachbasierten Entwurfsmuster stellen die am häufigsten eingesetzten Muster innerhalb der Meta-Modellierung dar. Um diese in ein existierendes Meta-Modell zu integrieren oder bestehenden Muster zu ändern, bedarf es einer Strategie, die angibt wie der Prozess dieser Änderung aussehen muss und welche Informationen dieser benötigt. Der Modellierungsprozess soll dafür stark unterstützt und geleitet werden. Die komplexe Sprachevolution und Semantik der verschiedenen Muster muss zu diesem Zweck auf elementare Änderungen im Modell abgebildet werden. Dabei spielt neben der Konformität hinsichtlich der Semantik für das jeweilige Muster auch der Einfluss anderer Muster eine wichtige Rolle. So führt zum Beispiel das Festlegen einer Generalisierung an einem Konzept dazu, dass die eventuell bestehende Instanziierung gelöscht werden muss (Regel C.9, Abschnitt 4.2).

Neben den Anpassungen am Meta-Modell sollen auch Änderungen, die nötig sind um andere dadurch beeinflusste Modelle zu migrieren, festgelegt und zusammen mit der Evolution ausgeführt werden. Die Strategie soll also einen konsistenten Zustand in einen anderen konsistenten Zustand überführen und dabei auch den Einfluss mehrerer sprachbasierter Entwurfsmuster beachten. Der Begriff Konsistenz bezieht sich dabei auf die in Abschnitt 4.2 definierten Begrifflichkeiten und Regeln. Ein weiterer Aspekt ist die Festlegung des Automatisierungsgrades der jeweiligen Strategie, d.h. es soll festgestellt werden, welche Informationen vom Benutzer zur Durchführung der Evolution und Koevolution geliefert werden müssen und welche aus den Informationen, die bereits im Modell vorhanden sind, berechnet werden können.

Wenn man also wie im Abschnitt 1.3 einen erweiterten Powertypen in ein bestehendes Meta-Modell einführen will, muss also konkret bekannt sein, welche Elemente durch die Einführung eines erweiterten Powertypen betroffen sind, wie und in welcher Abfolge diese zu ändern sind, damit das Modell anschließend syntaktisch valide und semantisch konform ist, und welche Informationen durch den Benutzer während dieser Migration bereitgestellt werden müssen.

1.4.2 H2: Validierung und Konformitätsprüfung von elementaren Modelländerungen

Die Grundlage für die Entwicklung von Strategien zur Einführung und Änderung sprachbasierter Entwurfsmuster stellt die Validierung und Konformitätsprüfung von elementaren Modelländerungen dar. Dabei müssen einfache Änderungen am Modell ausgewertet und entsprechende Ansätze gefunden werden, um die auftretenden syntaktischen Fehler und semantischen Konformitätsbrüche im Umgang mit sprachbasierten Entwurfsmustern zu erkennen. Daneben müssen Strategien entwickelt werden, die es erlauben, die Brüche in der Konsistenz oder der konformen Verwendung eines Musters automatisch aufzulösen. Da sich die verschiedenen Sprachmuster aus Kapitel 2 gegenseitig beeinflussen, muss immer die Konformität aller Muster gewährleistet werden.

Um die Herausforderung zu verdeutlichen, soll das Beispiel aus Abschnitt 1.3 im Nachfolgenden erweitert werden. Angenommen ein weiteres Konzept `And` wird vom Benutzer als Spezialisierung von `Node` erstellt. Da durch eine Instanziierung eines Powertypen eine Spezialisierung des partitionierten Typs impliziert wird, können Spezialisierungen von `Node` auch als Instanz von `NodeKind` modelliert werden. Konform zur Semantik des Powertyp Musters, muss folglich `And` zu einer Instanz von `NodeKind` und die Diskriminatorattribute (`supportsInCF` und `supportsOutCF`) müssen gesetzt werden. Letztgenannte geben an, ob ein Knoten eingehende oder ausgehende Kontrollflüsse besitzt. Daneben ist das Entfernen der Spezialisierung von `And` zu `Node` notwendig, wobei alle Zuweisungen der Instanzen von `And` nicht verändert werden sollen. Dem Benutzer muss also eine Möglichkeit dargelegt werden, wie er diese eventuelle Verletzung in der Konformität des Powertyp Musters auflösen kann.

1.4.3 H3: Unterstützung iterativer Evolution von Modellen

Sprachbasierte Muster wirken sich in der Regel auf viele Modellelemente aus. Deshalb ist die Einführung und Änderung eines Sprachmusters häufig mit vielen Änderungen an verschiedenen Modellelementen verbunden. Ferner existieren meist mehrere Möglichkeiten ein Meta-Modell dahingehend anzupassen, dass es neuen Anforderungen genügt. Aus diesen Gründen ist die Unterstützung einer iterativen Evolution von Modellen eine wichtige Herausforderung. Diese leitet sich unmittelbar aus den beiden ersten Herausforderungen ab, da die dort beschrieben Anpassungen der Modelle jeweils sehr komplex Änderungen nach sich ziehen.

Konkret muss demzufolge die Möglichkeit bestehen, die Evolution rückgängig zu machen oder diese erneut auszuführen (Undo/Redo Funktionalität). Daneben muss die Bewertung verschiedener alternativer Evolutionsstrategien durch den Modellierer unterstützt werden. Dabei ist die Aufzeichnung der Evolution, d.h. die Existenz einer Historie, ein integraler Bestandteil. Zusätzlich müssen die elementaren Auswirkungen der Evolution und Migration auf betroffenen Modellelemente dem Modellierer dargestellt werden, um den Vorgang der Evolution transparent zu gestalten.

Angewendet auf das Beispiel 1.3 würde dies bedeuten, dass die Anwendung der Strategie aus H1 Undo und Redo Funktionalität besitzt. Damit wäre es möglich, zunächst einen erweiterten Powertypen wie oben beschrieben einzuführen und die Auswirkungen auf das Modell zu analysieren. Falls die Evolution aus irgendwelchen Gründen zu einschneidend wäre, könnte man die Einführung rückgängig machen, eine Alternative wie zum Beispiel das Einführen einer Vererbungshierarchie durchführen und beide Evolutionen miteinander vergleichen.

1.5 Lösungsskizze

Der Umgang mit sprachbasierten Mustern in einer Modellierungsumgebung verursacht häufig Änderungen in der durch die Meta-Modelle beschriebenen Sprache. Diese Anpassungen führen zu Fehlern in der Sprache formulierten Modellen, die manuell beseitig werden müssen. Um dies zu vermeiden, wird im Folgenden dargelegt, wie das Ziel der Benutzerunterstützung in der Modellierung mit sprachbasierten Mustern erreicht werden soll.

Grundlegendes Konzept der Arbeit bildet die Bereitstellung verschiedener Operatoren, die auf Modelle oder deren Elemente angewendet werden können und dabei Modelltransformationen. Diese Bibliothek an Operatoren liefert die in H1 verlangte Strategie, die es erlaubt, sprachbasierte Muster in ein existierendes Meta-Modell einzuführen oder diese zu ändern. Jeder dieser Operatoren definiert einen Prozess, der die jeweilige Transformation der Modelle beschreibt und die Abbildung der Evolution auf einfache Änderungen an Modellelementen realisiert. Dieser Prozess beschreibt auch wie sich die jeweilige Evolution auf andere Muster, die bereits im Modell existieren, auswirkt. Neben der Abbildung auf elementare Änderungen im Modell definiert jeder Operator weiterhin, welche Informationen zu welchem Zeitpunkt im Prozess vorliegen müssen, damit eine spätere Ausführung möglich ist. Damit wird festgelegt, welche Daten von extern (also beispielsweise vom Benutzer) bereitgestellt werden müssen. Zudem berechnet ein Operator alle durch die Evolution betroffenen Elemente und passt diese während der Ausführung dem durch den Prozess festgelegtem Verhalten entsprechend an.

Eine wichtige Eigenschaft der Operatoren ist die Möglichkeit, sie komposit aufzubauen. Dabei werden die Modellelemente als elementare Operatoren bereitgestellt. Dadurch werden diese Operatoren die Grundlage für die Erstellung der komplex aufgebauten Operatoren der Sprachmuster und helfen somit mittels der Abstraktion, die durch diese Elementoperatoren stattfindet, diese einfacher zu beschreiben, was wiederum die Umsetzung von H1 erleichtert. Deshalb wird für jede Änderung einer Eigenschaft eines Modellelementes, die Inkonsistenzen nach sich ziehen kann, ein Operator bereitgestellt, der diese Inkonsistenzen auflöst. (H2). Dabei werden die Auswirkungen aller Sprachmuster berücksichtigt und die Änderungen konform zu ihrer Semantik durchgeführt.

Für die Umsetzung der Herausforderung H3 soll die Evolution eines Modells mittels der Operatoren aufgezeichnet werden, um so eine Historie zu erstellen, die den Verlauf der Evolution am Modell darstellt. Weiterhin wird die Funktionalität, den jeweiligen Evolutionsschritt rückgängig zu machen oder ihn wiederherzustellen, bereits im Operator verwurzelt. Dadurch wird es möglich, Änderungen in der Sprache ohne Gefahr des Verlustes von Elementen im Modell zu testen. Daneben werden alle Änderungen auf elementarster Ebene (also den Modellelementen) dargestellt, damit eine Abschätzung von verschiedenen alternativen Vorgehensweisen bei der Evolution eines Modells durch den Benutzer geschehen kann. Um dies und die sprachbasierten Muster an sich transparent zu gestalten, werden verschiedene Auswirkung der Muster auf Konzepte, Attribute und Zuweisungen bei den jeweiligen betroffenen Elementen visualisiert.

1.6 Beitrag der Arbeit

In diesem Abschnitt werden die wissenschaftlichen Beiträge dieser Arbeit dargestellt. Einige sind zwar in der Lösungsskizze bereits angedeutet, sollen aber hier zur Vollständigkeit ebenfalls noch einmal erwähnt werden.

Bereitstellung von Regeln und Definitionen eines Modellierungsparadigmas

Im Zuge dieser Arbeit wurde das Modellierungsparadigma von Volz [117] um Regeln und Definitionen (siehe Kapitel 4) erweitert und angepasst, die die Grundlage für die Validierung von Modellen schaffen. Die erstellten Definitionen beinhalten die formale Beschreibung der Auswirkungen der Sprachmuster und stellen Mengen bereit, die den uniformen Umgang mit Modellelementen innerhalb der Operatoren erleichtern und ebenfalls für die Benutzerunterstützung im Umgang mit sprachbasierten Mustern hilfreich sind (siehe Kapitel 7).

Operatoren zur konsistenten und konformen Ausführung von elementaren Änderungen

Für alle elementaren Änderungen im Modell, die zu Brüchen in der Konsistenz führen, wurden Operatoren erstellt, die diese beheben. Einige dieser Operatoren finden sich ebenfalls in der aktuellen Forschung (z.B. [56, 119]). Jedoch mussten diese dahingehend angepasst werden, dass die Auswirkungen aller sprachbasierten Muster berücksichtigt werden. Es wird zusätzlich vorgestellt, welche Aktionen am Modell dazu führen, dass ein entsprechender Operator verwendet werden kann, um die Koevolution an anderen Modellen auszuführen. Dabei wird für jede elementare Änderung neben der Konsistenz auch die Konformität hinsichtlich der Verwendung von sprachbasierten Mustern hergestellt.

Da alle Modellelemente Operatoren sind und Operatoren wiederum sich aus anderen zusammensetzen lassen, wurde im Zuge dieser Arbeit auch ein Ansatz bereitgestellt, der die einfache Definition von komplexen Operatoren erlaubt. Dadurch kann eine gewisse Abstraktion stattfinden. Operatoren bilden somit die Grundkomponenten aller Modelltransformationen.

Bibliothek an Operatoren zur (semi-)automatischen Einführung und Änderung von sprachbasierten Mustern

Weiterhin wurde eine Bibliothek an Operatoren bereitgestellt, die den Umgang mit sprachbasierten Mustern in Meta-Modellen erlaubt. Sie dienen sowohl der Einführung eines Musters in ein bestehendes Modell als auch deren Änderung. Die Operatoren der Bibliothek wurden zu weiten Teilen neu erstellt. Lediglich die Operatoren zum Sprachmuster der Vererbung existierten bereits und wurden dahingehend angepasst, dass andere sprachbasierte Muster im jeweiligen Operator ebenfalls berücksichtigt werden.

Visualisierung von Auswirkungen eines sprachbasierten Entwurfsmusters

Als weiterer Beitrag wird in der Arbeit gezeigt, wie die durch die Einführung oder Änderung eines sprachbasierten Musters verursachte Evolution visualisiert werden kann. Zusätzlich werden verschiedene Mengen von Modellelementen bereitgestellt, die den Umgang mit Sprachmustern erleichtern (siehe Abschnitt 4.2). Diese werden in der Umsetzung dazu verwendet, dem Benutzer verschiedene Sichten auf bestimmte Elemente zu bieten, die die Auswirkungen der sprachbasierten Muster auf die jeweiligen konkreten Modellelemente darstellen und die Sprachmuster damit transparent machen.

Prototypische Implementierung des Ansatzes

Neben der Beschreibung der Konzepte und Ansätze wird auch gezeigt, wie sich diese in ein Modellierungssystem integrieren lassen. Dazu wurde eine prototypische Implementierung als Erweiterung der *Model Workbench* Modellierungsumgebung im Rahmen dieser Arbeit angefertigt und darin die vorgestellten Ansätze evaluiert.

1.7 Struktur der Arbeit

Die vorliegende Arbeit gliedert sich in acht Kapitel. Das erste Kapitel beschäftigt sich mit der Beschreibung der Problemstellung und zeigt den Bedarf an Unterstützung für die Evolution von Meta-Modellen mit sprachbasierten Mustern auf. Im nachfolgenden Kapitel zwei werden die Sprachmuster Meta-Modellierung mit mehreren Ebenen, Clabject, Vererbung, Deep Instantiation, (erweiterter) Powertyp, Materialisierung und Instanz-Spezialisierung eingeführt und für jedes Muster ein Beispiel geliefert. Kapitel drei widmet sich der Vorstellung verwandter Arbeiten und zeigt auf, dass keiner der bisherigen Ansätze in der Lage ist, die aus den Herausforderungen abgeleiteten Anforderungen in Gänze zu erfüllen. Kapitel vier beschreibt Grundlagen, die in den darauffolgen Kapitel von Bedeutung sind. So wird darin das operatorenbasierte Linguistische Meta-Modell eingeführt und Regeln definiert, deren Einhalten für Konsistenz und Konformität hinsichtlich der Sprachmuster sorgen. Das fünfte Kapitel liefert für jede Änderung einer Eigenschaft eines Modellelements, die zu einer Verletzung der in Kapitel vier beschriebenen Grundlagen führt, einen Operator, der für die gültige Durchführung dieser Änderung zuständig ist. Aufbauend auf diesen elementaren Operatoren werden dann in Kapitel sechs komplexe Operatoren vorgestellt, die die Einführung und Änderung der sprachbasierten Muster unterstützen. Kapitel sieben zeigt, wie die vorgestellten Ansätze in einem Modellierungssystem umgesetzt werden können und beschreibt wie dabei der generelle Umgang mit Sprachmuster und eine iterative Evolution unterstützt werden. Das letzte Kapitel acht beinhaltet das Fazit dieser Arbeit zusammen mit den möglichen Anknüpfungspunkten zukünftiger Forschung im Ausblick.

2 Sprachbasierte Muster in der Meta-Modellierung

In diesem Kapitel werden alle für die Arbeit relevanten Sprachmuster vorgestellt, die in der aktuellen Forschung weitreichend eingesetzt werden (z.B. [6, 27, 59, 82, 117]). Diese unterscheiden sich von den in der Softwareentwicklung verwendeten Design Patterns [42] hinsichtlich ihres Bezugspunkts. Um dies zu erläutern soll zunächst das Muster der Orthogonalen Klassifikation vorgestellt werden, das die Unterscheidung dieser Muster ermöglicht. Die Beziehung zwischen Typ (Klasse) und Instanz (Objekt) ist in der Objektorientierung und Modellierung von zentraler Bedeutung. Ein Typ bzw. ein Klasse legt dabei zwei entscheidende Aspekte seiner Instanzen fest. Zum einen welche Attribute und Methoden diese zuweisen bzw. ausführen können und zum anderen die Struktur, die im Speicher angelegt wird, um Instanzen abzulegen. Folglich gibt es (nach Atkinson und Kühne [3]) zwei Arten der Instanziierung. Die ontologische Instanziierung[1], die inhaltliche Aspekte wie beispielsweise Attribute beschreibt und die linguistische Instanziierung [2], die die Speicherstruktur des Inhalts festlegt. Diese beiden Instanziierungen können voneinander getrennt werden. Dies hat zur Folge, dass Modelle nun zwei Meta-Modellen genügen müssen: einem ontologischen und einem linguistischen Meta-Modell (siehe Abbildung 2-1). Diese Struktur nennt man Orthogonale Klassifikation [3]. Diese Trennung von Inhalt und Struktur ähnlich wie z.B. in HTML erlaubt es, dank der uniformen Speicherstruktur, Inhalt ohne vorherigen Generierungsprozess zu interpretieren. Da die im Nachfolgenden präsentierten Muster im linguistischen (also sprachbezogenen) Meta-Modell verwurzelt sind, werden sie Sprachmuster oder

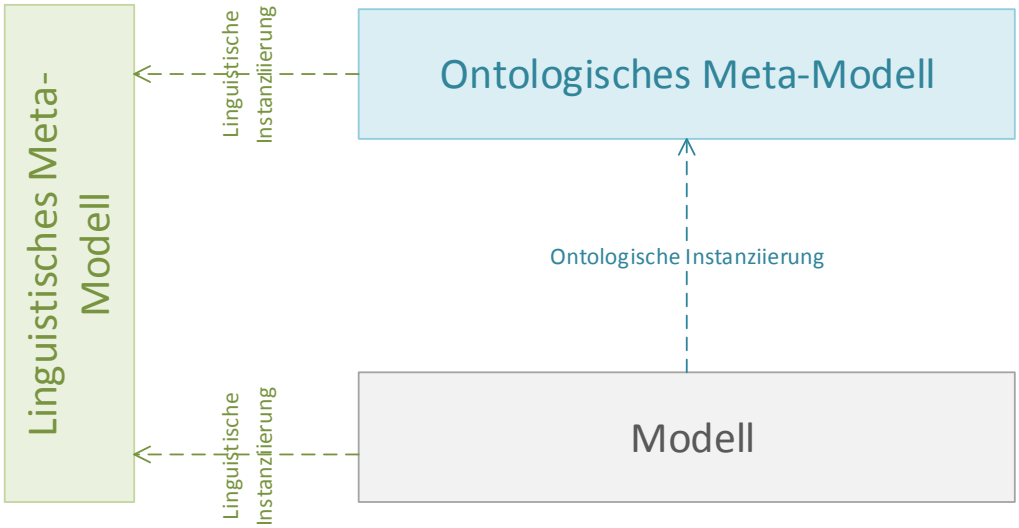

Abbildung 2-1 Orthogonale Klassifikation

[1] Wenn im Folgenden von Instanziierung gesprochen wird, dann ist immer die ontologische Instanziierung gemeint, falls nicht explizit etwas anderes erwähnt wird.

[2] Die Begrifflichkeiten ontologisch und linguistisch sind teilweise missverständlich, da die ontologische Instanziierung die im Modell verwendete Sprache festlegt. Andererseits definiert diese Meta-Ebene auch den inhaltlichen Rahmenbedingungen des Modells. Aufgrund der Tatsache, dass die beiden Definitionen nicht von grundlegender Bedeutung für diese Arbeit sind, werden hier die ursprünglichen Begriffe verwendet.

sprachbasierte Muster genannt. Die oben erwähnten Entwurfsmuster oder Design Patterns dagegen sind im ontologischen Meta-Modell enthalten.

2.1 Meta-Modellierung mit mehreren Ebenen und Clabjects

Um ein Modell formal zu beschreiben, benötigt man eine Sprache. Diese kann als Modell formuliert werden, das dann Meta-Modell genannt wird. Viele Programmiersprachen oder auch Modellierungssysteme beschränken die Programmierung bzw. (Meta-)Modellierung auf zwei Ebenen [67]. Dadurch müssen Anwender häufig dazu übergehen, die Semantik der Instanziierung manuell innerhalb der Sprache nachzubilden (z.B. [28, 98]). Um dieses Problem zu vermeiden, kann auch die Sprache, in der eine Modellierungssprache formuliert ist, formal als Modell ausgedrückt werden, wodurch ein Meta-Meta-Modell entsteht. Somit ergibt sich eine Hierarchie zwischen Meta-Ebenen (Abbildung 2-2), die prinzipiell nicht beschränkt ist. Dabei beschreibt jede darüber liegende Ebene (auch Sprach-Ebene genannt) wie die Ebene darunter (Instanz-Ebene) aufgebaut wird. Im Gegenzug ist jede Ebene Instanz der darüber liegenden Ebene, falls diese existiert. In der Forschung wird häufig der Meta-Meta-Ebene die Eigenschaft der Selbstbeschreibung zugeschrieben, um die möglichen Meta-Hierarchien zu beschränken. Ein Beispiel dafür ist der heute als Grundlage der Meta-Modellierung angesehene Standard Meta Object Facility (MOF) [85]. Die oberste Ebene besteht bei solchen Meta-Hierarchien im Wesentlichen aus den Bestandteilen Entität und Beziehung, wobei eine Beziehung wiederum als Entität ausgedrückt werden kann, die zwei andere Entitäten miteinander in Relation setzt. Diese sehr allgemeine Sprache ist zwar so abstrakt, dass sie immer als Grundlage zum Aufbau neuer Sprachen dienen kann, jedoch ist die verwendete Begrifflichkeit in manchen Anwendungsfeldern zu allgemein und muss in Form von weiteren Meta-Modellen verfeinert werden [64].

In den auf zwei Meta-Ebenen beschränkten Modellierungssystemen oder Programmiersprachen haben sich die Begriffe Typ (oder Klasse) für Elemente der Sprach-/Metaebene und Instanz (oder Objekt) für Elemente der Instanz-Ebene herausgebildet. Bei der Verwendung von mehreren Meta-Ebenen müssen diese meist streng voneinander getrennten Begriffe zusammengeführt werden, um einen uniformen Umgang mit den Elementen der Ebene unabhängig von der Position in der Meta-

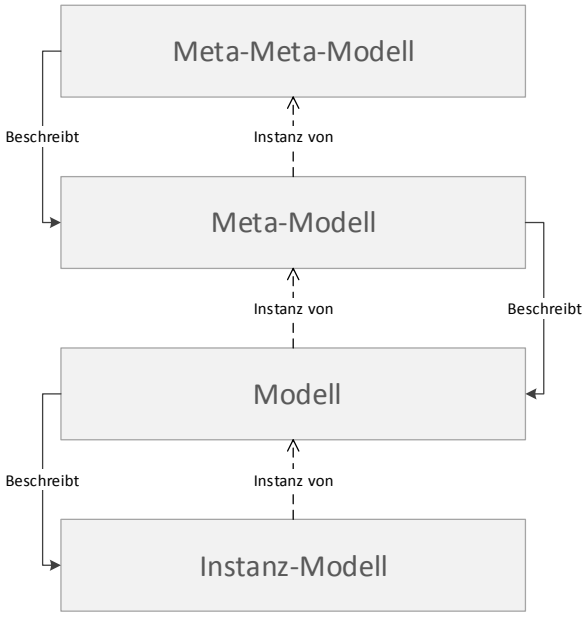

Abbildung 2-2 Meta-Modell Hierarchie mit vier Ebenen

Hierarchie zu ermöglichen [6]. Deshalb führt Atkinson [2] gemeinsam mit Kühne [4] das Muster des Clabjects ein (CLAss + obJECT). Jedes Clabject besitzt zwei verschiedene Facetten, eine Typ- und eine Instanzfacette. Die Typfacette kann Attribute definieren, während die Instanzfacette Zuweisungen an Attribute, die am instanziierten Typ deklariert sind, festlegt. Folglich besitzen Clabjects der obersten Ebene eine leere Instanzfacette und Clabjects der untersten Ebene eine leere Typfacette. Die Instanziierung ist also im Sinne dieser Definition eine Beziehung von der Instanzfacette der Instanz zur Typfacette des Typs. Neben dem Begriff „Clabject" wird für ein solches Element auch häufig Konzept (`Concept`) verwendet [117]. Dieser Terminus soll ebenfalls in dieser Arbeit verwendet werden.

Beispiel

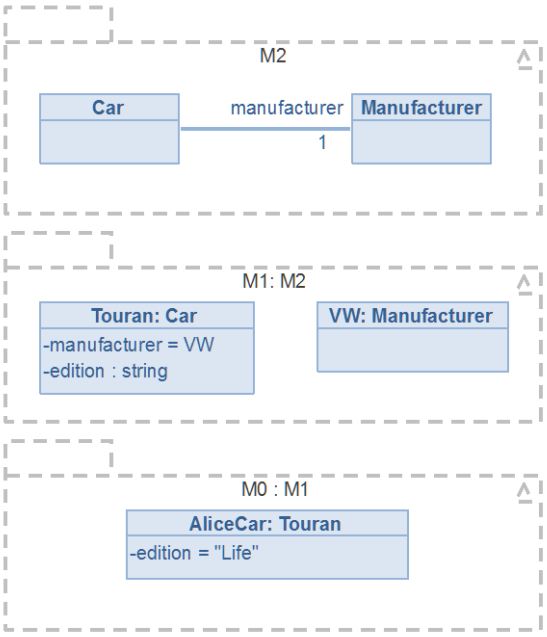

Abbildung 2-3 Beispiel einer Meta-Hierarchie mit drei Ebenen

In Abbildung 2-3 ist ein Beispiel für eine Meta-Hierarchie mit drei Ebenen dargestellt. Die Notation des Diagramms ist an die UML Notation angelehnt. Genauere Erläuterungen finden sich im Abschnitt 4.4. Die oberste Ebene `M2` beinhaltet zwei Konzepte: `Car` und `Manufacturer`, die durch eine Beziehung verbunden sind. Demzufolge soll dadurch ausgedrückt werden, dass Autos genau einen Hersteller besitzen. Aufgrund der Beziehung deklariert `Car` ein Attribut `manufacturer` in seiner Typfacette. Da `M2` die oberste Ebene darstellt, sind die Instanzfacetten von `Car` und `Manufacturer` leer. Ebene `M1` ist eine Instanz der Ebene `M2` und definiert ebenfalls zwei Konzepte `Touran` und `VW`. `Touran` ist eine Instanz von `Car` und hat das Attribut `manufacturer` mit dem Wert `VW` (Instanz von `Manufacturer`) in der Instanzfacette gesetzt, da der Touran von Volkswagen produziert wird. Daneben definiert `Touran` in der Typfacette ein eigenes Attribut `edition`. Dieses soll eine spezielle Edition eines Tourans darstellen. In der Ebene `M0`, die eine Instanz der Ebene `M1` ist, wurde mit `AliceCar` eine Instanz von `Touran` modelliert. Dieser Touran hat die spezielle Edition `„Life"`, was durch die entsprechende Zuweisung dargestellt ist.

2.2 Vererbung

Das Prinzip der Vererbung ist in der objektorientierten Welt [50] ein häufig verwendetes Sprachmuster. Sie steht für die „ist ein" (im Englischen „is a")[3] Beziehung zwischen zwei Klassen oder Entitätstypen [41]. Die erbende Klasse wird als Spezialisierung und die Oberklasse als Generalisierung bezeichnet. Die Vererbung wirkt sich auf die Struktur[4] einer Klasse (bzw. eines Konzeptes) aus. Zum einen werden die Attribute der Generalisierung an die Spezialisierung vererbt. Zum anderen gilt für Spezialisierungen das Substitutionsprinzip, d.h., dass eine Instanz der Spezialisierung an jeder Stelle zugelassen ist, an der eine Instanz der Generalisierung erwartet wird.

Im Kontext der Meta-Modellierung mit mehreren Ebenen muss die Definition etwas präzisiert werden, da, wie in Abschnitt 2.1 beschrieben, alle Konzepte sowohl Instanz- als auch Typfacette besitzen. Der üblichen Semantik der Vererbung folgend, wirkt sich diese also lediglich auf die Typfacette aus, die Instanzfacette bleibt dagegen unberührt. Folglich ist die Vererbung eine Beziehung zwischen den Typfacetten der entsprechenden Konzepte. Ein Muster, das die Erweiterung der Instanzfacette ermöglicht, wird in Abschnitt 2.6 vorgestellt.

Beispiel

Abbildung 2-4 zeigt ein Modell, in dem das Sprachmuster der Vererbung verwendet wird. In der verwendeten Sprache soll ausgedrückt werden, dass in einem Krankenhaus Personen existieren. Diese besitzen einen Namen und teilen sich in die Untergruppen der Ärzte und Patienten auf. Um dies zu modellieren, wurde auf der Ebene `Language` eine Generalisierung `Person` mitsamt zweier Spezialisierungen `Physician` und `Patient` definiert. `Person` besitzt ein Attribut `name`, welches an beide Spezialisierungen vererbt wird. Weiterhin existiert auf `Language` noch ein Konzept `Hospital`,

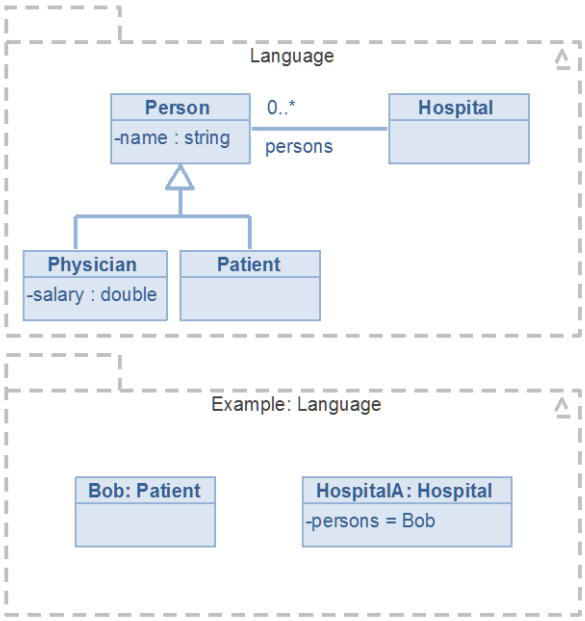

Abbildung 2-4 Beispiel einer Vererbung

[3] Die Verwendung der „ist ein" Beziehung ist in der natürlichen Sprache nicht eindeutig. So wird sie zusätzlich auch zur Beschreibung einer Instanziierung verwendet. In diesem Fall bildet sie dann aber eine Beziehung von Instanz- zur Typfacette zweier Konzepte [41].

[4] Neben den Auswirkungen auf die Struktur eines Konzeptes wird auch dessen Verhalten beeinflusst. Dieser Aspekt soll aber hier nicht weiter beleuchtet werden.

das eine Beziehung zu `Person` besitzt, die durch das Attribut `persons` am Konzept `Hospital` dargestellt wird. Die Ebene `Example` ist eine Instanz von `Language` und zeigt ein kleines Beispielmodell, in dem Bob ein Patient im Krankenhaus „HospitalA" ist. Dazu wurden die Konzepte `Bob` und `HospitalA` definiert. Da `Bob` eine Instanz von `Patient` ist, kann `Bob` einerseits das von `Person` gerbte Attribut `name` auf „Bob" setzen und andererseits `HospitalA Bob` als Wert für das Attribut `persons` verwenden (Substitutionsprinzip).

2.3 Deep Instantiation

Allgemein gilt, dass ein Konzept `A` ein anderes Konzept `B` instanziieren kann, wenn die entsprechenden Ebenen eine direkte Ebenen-Instanziierung eingehen. In der Meta-Modellierung mit mehreren Ebenen ist diese strikte Regel jedoch ein Problem, da häufig Konzepte in höheren Meta-Ebenen definiert werden und diese erst mehrere Ebenen darunter instanziiert werden sollen [6]. Ähnlich verhält es sich mit der Zuweisung von Attributen. Häufig will man die Zuweisung an ein Attribut nicht bei den direkten Instanzen setzen, sondern erst bei Instanzen dieser Instanzen eine Wert festlegen.

Um diese Probleme zu lösen, haben Atkinson und Kühne [5, 6, 68] das Deep Instantiation Sprachmuster eingeführt. Dieses definiert für jedes Modellelement einen Zähler (auch Potenz genannt), der mit jeder Instanziierung verringert wird. Bei Konzepten bezieht sich dies auf die Überschreitung von Ebenen-Instanziierungen, während bei Attributen die Instanziierung von Konzepten den Zähler beeinflusst. Wenn der Deep Instantiation Zähler 0 erreicht hat, kann das Konzept instanziiert oder das Attribut gesetzt werden.

Beispiel zur Deep Instantiation von Konzepten

Ein Beispiel findet sich in Abschnitt 5.3.4.

Beispiel zur Deep Instantiation von Attributen

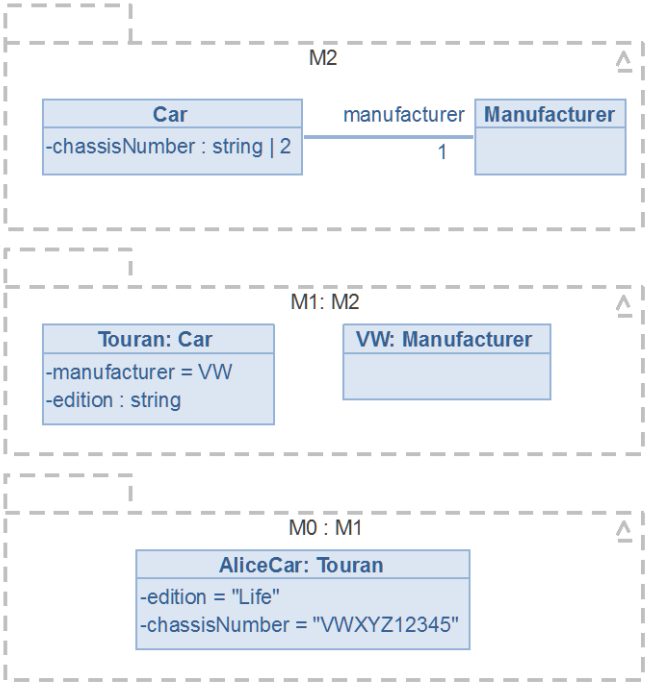

Abbildung 2-5 Beispiel mit Deep Instantiation für ein Attribut

In Abbildung 2-5 ist eine Erweiterung des Beispiels aus Abschnitt 2.1 dargestellt. Die einzige Erweiterung wurde auf der Ebene `M2` hinzugefügt. `Car` definiert nun ein Attribut `chassisNumber`, dass

die Fahrgestellnummer eines Autos darstellt. Da diese zwar zu jedem Auto gehört, aber auch erst bei jeder Instanz gesetzt werden darf/kann, wurde der Deep Instantiation Zähler von `chassisNumber` auf `2` gesetzt. Dadurch kann `AliceCar`, das auf der Ebene `M0` liegt, die Fahrgestellnummer setzten, da es Instanz von `Touran` ist, dieses wiederum eine Instanz von `Car` darstellt.

2.4 Powertype und Erweiterter Powertyp

Bei der Abstraktion eines konkreten Sachverhaltes in ein Modell wird häufig durch die Klassenbildung oder Generalisierung die Komplexität des Sachverhaltes verringert. Beide Beziehungen werden häufig durch die „ist ein" („is a") Sprechweise in der natürlichen Sprache gekennzeichnet [41]. Dennoch haben beide Abstraktionsmethoden eine unterschiedliche Semantik. Beispielsweise deutet der Ausdruck „ein Student ist eine Person" auf eine Generalisierung vom Konzept Student zum Konzept Person hin. Im Gegensatz dazu „sind Studenten eine Personengruppe", was einer Instanziierung vom Konzept Student zum Konzept Personengruppe gleich kommt [41]. Diese zwei verschiedenen Sichtweisen auf einen Studenten können in der Modellierung nur schwer abgebildet werden.

Powertyp

Deshalb führt Odell ([82] sowie Kapitel 23 in [83]) das Muster des Powertyps ein, um solch einer Problemstellung zu begegnen. Die wichtigsten Bestandteile sind in Abbildung 2-6 [5] dargestellt. Zunächst gibt es zwei verschiedene Konzepte. Der partitionierte Typ (im Beispiel `Person`), der die Generalisierung darstellt, wird durch den Powertyp (im Beispiel Personengruppe) in verschiedene Gruppen partitioniert. Daher werden beide auch durch die spezielle `partitions` Beziehung verbunden. Die Besonderheit dieses Musters ist die Verbindung von Instanziierung und Vererbung. Eine Instanz des Powertyps (im Beispiel `Student`) ist gleichzeitig eine Spezialisierung des partitionierten Typs, wodurch die duale Sichtweise auf die Powertyp-Instanz realisiert wird. Folglich ist die Voraussetzung für dieses Muster das Clabject Sprachmuster, da die Typfacette der Powertyp Instanz generalisiert wird und die Instanzfacette einen instanziierten Typ zugewiesen wird.

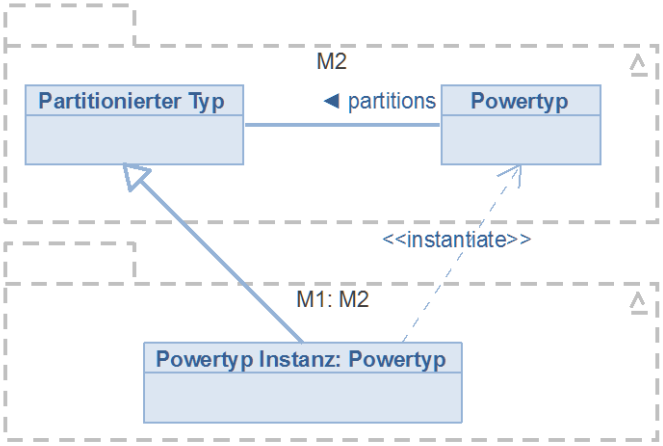

Abbildung 2-6 Darstellung des Powertyp Musters

[5] Der Pfeil für die Instanziierung ist nur in diesem Beispiel vorhanden. In allen anderen Modellen steht der instanziierte Typ eines Konzepts nach dem Doppelpunkt, der nach dem Namen folgt. Ähnlich verhält es sich bei der Generalisierung. Diese ist zwar implizit im Muster enthalten, wird aber in der Regel in Diagrammen nicht dargestellt.

Erweiterter Powertyp

Das Sprachmuster des erweiterten Powertyps wurde von Jablonski, Volz und Dornstauder in [59, 117, 118] vorgestellt. Es erweitert den Powertyp um Diskriminatorattribute, die eine Verbindung zu den Attributen des partitionierten Typs definieren. Für jedes Attribut am partitionierten Typ muss dabei ein entsprechendes Attribut am Powertyp definiert werden.[6] Mittels der Diskriminatorattribute lässt sich an einer Powertyp Instanz festlegen, ob ein Attribut vom partitionierten Typ geerbt wird oder nicht. Dadurch besteht die Möglichkeit Attribute, die nicht allen Spezialisierungen gemein sind, dennoch bei der Generalisierung zu definieren und durch die entsprechenden Werte der Diskriminatorattribute, die vererbten Attribute an der Instanz festzulegen. Dadurch können tiefe Vererbungshierarchien, die häufig komplex sind [121], vermieden werden.

Beispiel

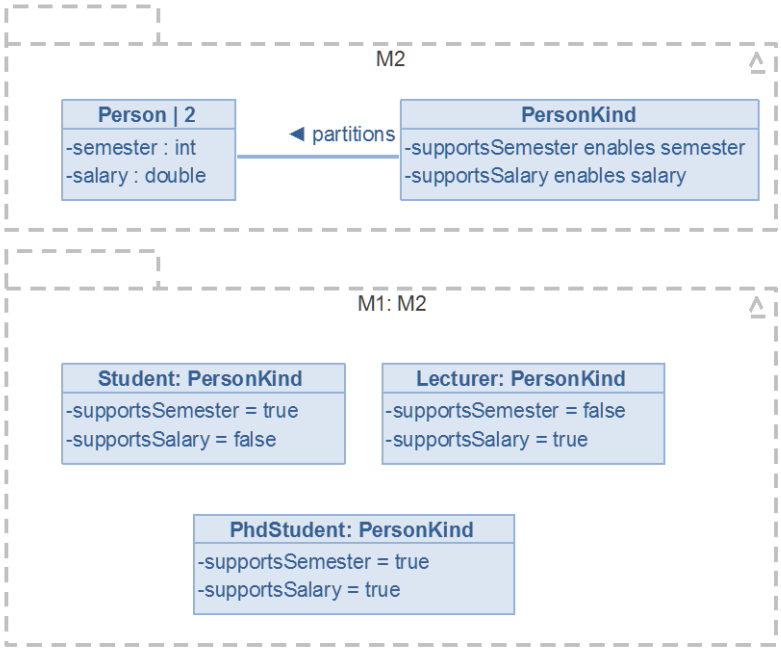

Abbildung 2-7 Beispiel eines erweiterten Powertyps

Um die Funktionsweise des erweiterten Powertyps zu verdeutlichen, soll ein Beispiel dienen, das in Abbildung 2-7 dargestellt ist. Auf der Ebene `M2` ist dabei ein partitionierter Typ `Person` und eine Powertyp `PersonKind`, der für eine Personengruppe steht, modelliert. `Person` besitzt zwei Attribute `semester` und `salary`, während `PersonKind` für jedes der beiden Attribute ein Diskriminatorattribut definiert. Die beiden Attribute von `Person` sollen angeben, in welchem Semester sich die Person befindet bzw. wie viel die Person verdient. Die Ebene `M1` beinhaltet drei Instanzen des Powertyps `PersonKind`, die damit implizit Spezialisierungen von `Person` darstellen. Die drei Konzepte sollen Studenten (`Student`), Dozenten (`Lecturer`) und wissenschaftliche Mitarbeiter (`PhDStudent`), die als Promotionsstudent eingeschrieben sind, darstellen. Da Studenten nicht bezahlt werden, besitzen sie kein Gehalt. Deshalb ist das Diskriminatorattribut `supportsSalary` auf `false` gesetzt, wodurch `salary` nicht an `Student` vererbt wird. Das Semester hingegen wird durch die Zuweisung von `true` für `supportsSemester` an `Student` weitergegeben. Auf der anderen Seite ist das Semester bei einem Dozenten nicht von Bedeutung, wohingegen das Gehalt geerbt werden soll. Folglich sind auch die

[6] Da ein Attribut des partitionierten Typs ohne ein Diskriminatorattribut an jede Powertyp-Instanz vererbt wird, entspricht dieses Verhalten dem Standardwert `true` für das Diskriminatorattribut. Deshalb stellt diese Bedingung keine Einschränkung dar.

jeweiligen Diskriminatorattribute gesetzt. Ein wissenschaftlicher Mitarbeiter, der Promotionsstudent ist, stellt eine Besonderheit dar, da er einerseits bezahlt wird und andererseits sich in einem Studiensemester befindet. Deshalb müssen beim Konzept `PhdStudent` beide Diskriminatorattribute auf `true` gesetzt sein.

2.5 Materialisierung

Materialisierung ist ein Sprachmuster, das es erlaubt mit Hilfe einer Zuweisung dynamisch Attribute an einem Konzept zu erstellen. Das Muster wurde von Pirotte in [87] eingeführt und in weiteren Arbeiten [26, 27] weiter untersucht. Das ursprünglich präsentierte Muster besitzt eine ähnliche Semantik wie das Powertyp Muster [67], deshalb soll sich der Begriff der Materialisierung für diese Arbeit (ähnlich wie Volz ([117], Kapitel 3.6) auf die dynamische Erzeugung von Attributen beschränken, was der T3 Attributpropagierung aus [87] entspricht.

Die Materialisierung stellt eine spezielle Beziehung zwischen Konzepten dar. Diese Beziehung ist eine Assoziation zwischen zwei Konzepten `A` und `C` und hat spezielle Eigenschaften, die sich auf die materialisierten Attribute auswirken. Dabei definiert Konzept `A` ein Attribut `rel` vom Typ `C` (und damit eine Beziehung). Alle Zuweisungen an dieses Attribut `rel` führen dann dazu, dass ein neues Attribut bei der Instanz von `A` materialisiert wird. Deshalb muss zusätzlich noch die Multiplizität, Kardinalität und der Name für die materialisierten Attribute festgelegt werden. Während die ersten beiden Eigenschaften direkt bei `rel` definiert werden[7], wird für den Namen ein Attribut bei `C` bestimmt, dessen Werte bei einer Instanz von `C` den Namen des materialisierten Attributes festlegen.

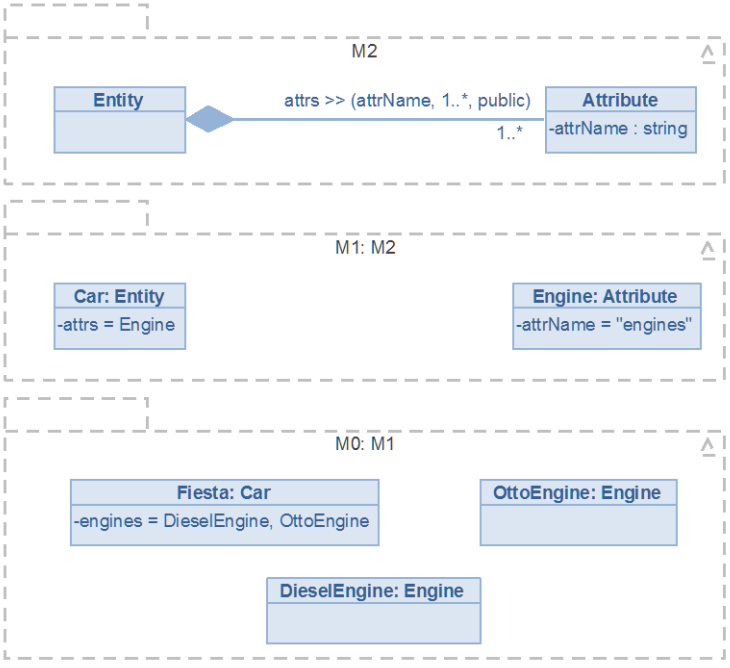

Abbildung 2-8 Beispiel mit Materialisierung

[7] Die Multiplizität und Sichtbarkeit ist aufgrund dessen für alle materialisierten Attribute eines Konzeptes gleich. Das Muster könnte dahingehend erweitert werden, dass diese Eigenschaften ebenfalls bei den Instanzen von `C` erst festgelegt würden. Dadurch würde aber eine Diskrepanz zur orthogonalen Klassifikation auftreten, da die jeweiligen Attribute von `C` als Typ die Domäne der Multiplizität bzw. Sichtbarkeit haben müsste.

Beispiel

Das Beispiel in Abbildung 2-8 zeigt eine exemplarische Anwendung des Materialisierungsmusters. In der Ebene `M2` liegen zwei Konzepte `Entity` und `Attribute`. `Entity` definiert die Materialisierungsbeziehung zu `Attribute` durch das Attribut (die Beziehung) `attrs`. Dabei legt es das Attribut `attrName` bei `Attribute` als das namengebende Attribut und die Werte `1..*` für die Multiplizität bzw. `public` für die Sichtbarkeit der materialisierten Attribute fest. Ebene `M1` definiert mit `Car` eine Instanz von `Entity`, die `attrs` auf eine Instanz von Attribute (`Engine`) gesetzt hat. Dadurch entsteht bei `Entity` ein materialisiertes Attribut `engines` (Wert von `attrName` bei `Engine`) mit der Multiplizität `1..*` und der Sichtbarkeit `public`. Aufgrund dessen kann das Konzept `Fiesta` als Instanz von `Car` auf der Ebene `M0` die beiden Instanzen von `Engine` `OttoEngine` und `DieselEngine` für das materialisierte Attribut `engines` zuweisen. Durch die Verwendung der Materialisierung lässt sich also auf jeder der drei Meta-Ebene eine Verbindung zwischen den Konzepten herstellen:

- Auf `M2` wird mittels `attrs` festhalten, dass Entitäten (`Entity`) Attribute (`Attribute`) besitzen.
- `M1` beschreibt durch die Zuweisung von `Car` an `attrs` mit dem Wert `Engine`, dass Autos einen Motor haben.
- `M0` definiert letztlich mit Hilfe der Zuweisung von `Fiesta` an `engines`, dass ein Fiesta sowohl mit Otto als auch Dieselmotor verfügbar ist`.`

Diese Verbindung auf jeder Meta-Ebene zwischen den entsprechenden Konzepten stellt den Vorteil der Materialisierung gegenüber anderen Varianten bei der Modellierung eines solchen Szenarios dar. ([117], Kapitel 3.6)

2.6 Instanz-Spezialisierung

Wie bereits im Abschnitt 2.2 erwähnt bezieht sich das Sprachmuster der Vererbung lediglich auf die Typfacette eine Clabject. Dennoch ist die Spezialisierung der Instanzfacette in vielen Szenarien sinnvoll. Deshalb stellt Volz ([117], Kapitel 3.5) das Sprachmuster der Instanz-Spezialisierung vor, welches durch unsere Forschungsgruppe in [60] weiter verfeinert wurde. Das Sprachmuster lehnt sich an die prototypische Vererbung an, die man aus Sprachen wie ECMAScript[8] kennt ([36], Kapitel 9) und auch von Lieberman in [71] beschrieben wird. Auch Pirotte et al. zeigen in [27] ein ähnliches Verhalten im Muster der Materialisierung, jedoch wird die Verbindung dort nicht auf Instanz, sondern auf Typ-Ebene festgelegt und wirkt sich dann auf die Instanzen aus. Allgemein betrachtet bildet die Instanz-Spezialisierung also eine Beziehung zwischen den beiden Instanzfacetten der beteiligten Konzepte.

Der Grundgedanke bei der Instanz-Spezialisierung ist das Erben von Zuweisungen, die am Prototyp (die Generalisierung der Instanzfacette) an instanziierte Attribute (siehe Definition 4.21) erstellt wurden. Diese geerbten Zuweisungen können bei der Instanz-Spezialisierung (i.A.) überschrieben werden. Da jedoch dieses Verhalten nicht immer genügt oder passend ist, erweitert Volz das Muster ([117], Kapitel 3.5) um die Möglichkeit, am Prototyp das Überschreibungsverhalten[9] der Instanz-Spezialisierungen zu

[8] Häufig wird diese Sprache auch JavaScript genannt. Der Begriff ist allerdings eine eingetragene Marke von Sun Microsystems und wird daher nicht für die standardisierte Sprache verwendet. Alternativ zum Begriff ECMAScript verwendet man heutzutage das Akronym JS um die Sprache zu benennen. [36]

[9] Volz suggeriert in Abschnitt 3.5 aus [117], dass das Überschreibungsverhalten am Attribut des instanziierten Typs festgelegt wird, im weiteren Verlauf der Arbeit wird jedoch das hier beschriebene Verhalten verwendet.

steuern. Dazu wird bei jeder Zuweisung am Prototyp ein Überschreibungstyp gewählt. Die verschiedenen Überschreibungstypen sind folgende:

- **Typ 0** (`forbidden`): Die Überschreibung des Wertes des Prototyps ist bei den Instanz-Spezialisierungen verboten.
- **Typ 1** (`normal`, Standardwert): Instanz-Spezialisierungen können den Wert mit einem beliebigen Wert (innerhalb des Attributtyps) überschreiben.
- **Typ 2** (`limited`): Der Prototyp legt die Domäne für die Werte der Zuweisungen an den Instanz-Spezialisierungen fest, d.h. jeder Wert einer Instanz-Spezialisierung ist eine Teilmenge des Wertes des Prototyps.
- **Typ 3** (`append`): Der Wert der Instanz-Spezialisierung wird dem Wert des Prototyps nachgestellt, um den eigentlichen Wert für das Attribut an der Instanz-Spezialisierung zu erhalten.
- **Typ 4** (`prepend`): Der Wert der Instanz-Spezialisierung wird dem Wert des Prototyps vorangestellt, um den eigentlichen Wert für das Attribut an der Instanz-Spezialisierung zu erhalten.

Für die Instanz-Spezialisierung muss das Substitutionsprinzip auch für die Instanzfacette erweitert werden: An jeder Stelle, an der eine Instanz eines Konzeptes erwartet wird, kann auch eine Instanz-Spezialisierung dieser Instanz stehen. Eine formale Betrachtung des Substitutionsprinzips findet sich in Abschnitt 4.2.

Beispiel

In Abbildung 2-9 ist ein Mobilfunktarif modelliert. Dazu wurde auf Ebene `M1` ein Konzept `MobilePlan` erstellt, das für den Tarif steht und einen Namen (`name`) sowie verschiedene Leistungen (`services`) besitzt. Jeder Tarif besitzt genau einen Anbieter (Attribut `provider` vom Typ `Provider`) und kann in mehreren Mobilfunknetzen (Attribut `networks` vom Typ `NetworkProvider`) angeboten werden. Auf der Ebene `M0` wurde ein beispielhaftes Modell dieser Sprache erstellt, das das Muster der Instanz-Spezialisierung verwendet. Der fiktive Mobilfunkanbieter `Cheap` (Instanz von `Provider`) stellt darin zwei Tarife zur Verfügung. Der `Base` (Instanz von `MobilePlan`) Tarif ist dabei der Prototyp (`<<concreteUseOf>>`) des `Special` Tarifes. Das Konzept Base definiert verschiedene Zuweisungen samt Überschreibungstypen:

- Attribut `name` wird mit dem Wert **„Base“** belegt und kann beliebig von jeder Instanz-Spezialisierung überschrieben werden (Typ 1: `normal`)
- Im Basistarif ist eine Flatrate für das unbegrenzte Telefonieren enthalten. Folglich ist für das Attribut `services` der Wert **„CallFlat“** gesetzt, der von jeder Instanz-Spezialisierung um weitere Werte erweitert werden kann (Typ 3: `append`)
- Der Anbieter des Tarifes ist `Cheap`, was durch die Zuweisung an das Attribut `provider` umgesetzt ist. Dieser Wert darf von Instanz-Spezialisierungen nicht überschrieben werden (Typ 0: `forbidden`)
- Der Tarif wird in zwei verschiedenen Mobilfunknetzen angeboten. Deshalb besitzt das Attribut `networks` die Werte `O2` und `D1` (Instanzen von `NetworkProvider`). Da diese Zuweisung den Überschreibungstyp 2 (`limited`) definiert, muss der Wert der Zuweisung bei einer Instanz-Spezialisierung eine Teilmenge von `{O2, D1}` sein.

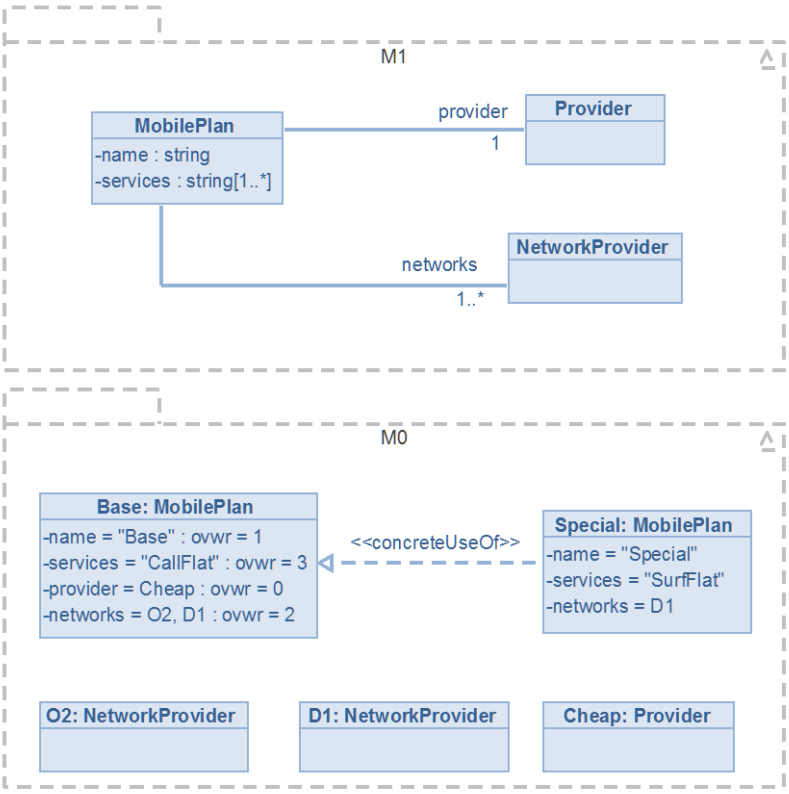

Abbildung 2-9 Beispiel einer Instanz-Spezialisierung

Die Instanz-Spezialisierung `Special` stellt einen Spezialtarif dar, der noch eine Internetfaltrate anbietet und lediglich als Netz D1 bereitstellt. Demzufolge überschreibt `Special` `name` mit einem neuen Namen (`„Special“`) und setzt den Wert `„SurfFlat“` bei `services`, wodurch implizit durch den Überschreibungstyp 3 der Wert `{„CallFlat“, „SurfFlat“}` angenommen wird. Das Attribut `networks` erhält den Wert `D1` (Teilmenge des Wertes von `Base`), während `provider` nicht zugewiesen werden darf und dadurch die Zuweisung von `Base` mit dem Wert `Cheap` geerbt wird.

Unterschied zur Definition in [117]

- Die letzten beiden Überschreibungstypen 3 und 4 sollen (anders als bei Volz) generisch betrachtet werden und auch z.B. auf Mengen anwendbar sein. In diesem Beispiel sind dann beide Typen gleich, da Mengen keine Ordnung besitzen.
- Volz nennt den Überschreibungstyp 2 in [117] „restricted“. Da in vielen Domänen (z.B. SQL) dieser Begriff für ein Verbot steht, wurde dieser Typ hier limited benannt, um die Beschränkung des Wertebereiches zu kennzeichnen.
- Als Standardverhalten wurde der Typ 1 ähnlich wie in ECMAScript gewählt. Volz definiert das Standardverhalten jedoch unterschiedlich. (siehe [117]: Abschnitt 3.5, 6.11.5, 6.12.9, 8.7.2 AZ.2)
- Auf die Definition des Überschreibungswertes „overwrite“ wurde verzichtet, da es fragwürdig erscheint, beim Prototyp einen Wert für ein Attribut festzulegen und gleichzeitig zu definieren, dass dieser Wert später durch eine Zuweisung in der Instanz-Spezialisierung vom Prototyp entfernt wird. Dadurch beeinflusst das Verhalten der Instanz-Spezialisierung den Prototyp. Allgemein ist in diesem Fall die Deklaration eines optionalen Attributes sinnvoller, das auch das Fehlen einer Zuweisung am Prototyp zulässt.

3 Verwandte Arbeiten

„The only constant is change" ([21], Abschnitt 1.1.3) ist ein häufig verwendetes Zitat, das zeigt, dass Veränderung ein inhärenter Vorgang ist. Dies gilt demnach gleichermaßen für Software und deren Entwicklung. Um diesen Fakt zu verdeutlichen, spricht Favre in [34] vom 3D software space. In diesem steht die Dimension der Meta-Modell Evolution orthogonal zur Produktentwicklungsdimension, welche den Softwareentwicklungsprozess (zum Beispiel das Wasserfallmodell) selbst enthält und der so genannten Repräsentationsdimension, die verschiedene Sichten auf eine Dimension widerspiegelt. Der Umgang mit einer Evolution eines Softwaresystems ist immer mit Kosten verbunden und daher ist eine Unterstützung essentiell. Deshalb wurden in den letzten Jahren viele Ansätze entwickelt, die dieser Herausforderung begegnen.

Dieses Kapitel widmet sich den aktuellen Ansätzen in der Forschung, die sich mit der Evolution eines Schemas geschäftigen und damit relevant für diese Arbeit sind. Dabei werden zunächst im Abschnitt 3.1 verwandte Gebiete untersucht, in denen Software Evolution auftritt. Anschließend werden die Arbeiten im Bereich der Meta-Modell Evolution im Abschnitt 3.2 vorgestellt und mit Hilfe von dort definierten Anforderungen evaluiert.

3.1 Software Evolution in verwandten Bereichen

Die Arbeiten im Bereich der Meta-Modell Evolution wurden von vielen Ansätzen in anderen Gebieten beeinflusst. Deshalb werden in diesem Abschnitt einige Arbeiten im Bereich der Schema und Ontologie Evolution sowie das Forschungsgebiet der Model Managements vorgestellt. Letztgenanntes ist dabei besonders für die Ansätze im Bereich der Meta-Modell Evolution von Bedeutung, da es die Grundlage für die Ansätze des Meta-Modell Matchings bildet.

3.1.1 Schema Evolution

Bereits seit einigen Jahrzehnten beschäftigt sich die Forschung mit dem Bereich der Schema Evolution. Darin wird untersucht, inwiefern Änderungen an einem Schema sich auf Instanzen auswirken und wie ein Migrationsprozess solcher Elemente unterstützt werden kann. Wenn sich auch die grundlegenden Ansätze ähneln, gibt es doch je nach verwendeten Paradigma Unterschiede in den Umsetzungen.

Prinzipiell gibt es nach Hartung [49] drei verschiedene Gütekriterien für die Unterstützung einer Schema Evolution. Neben einer Vollständigkeit, die eine vielfältige Bibliothek an Schemaänderungen verlangt, soll auch die zusätzliche Information, die durch den Benutzer bereitgestellt werden muss, klein gehalten werden. Weiterhin stellt die Transparenz der Evolution ein wichtiges Kriterium dar, wobei mit diesem Begriff ausgedrückt werden soll, dass Änderungen am Schema sich nach Möglichkeit nicht auf die Verfügbarkeit und Leistung der entsprechenden Datenbank (z.B. relational) auswirken sollen. Dies beinhaltet, dass andere Anwendungen und Benutzer so wenig wie möglich von der Schema Evolution betroffen sind. Die jeweiligen Gütekriterien lassen sich leicht auch auf andere Ansätze, wie zum Beispiel die Meta-Modell Evolution übertragen.

Relationale Ansätze

Die Relationalen Ansätze beschäftigen sich, wie der Name schon sagt, mit Änderungen an einem Schema einer relationalen Datenbank. Diese ist aus Relationen bzw. Datenbank-Tabellen aufgebaut, die durch Fremdschlüssel miteinander verknüpft werden [33]. Die Instanzen eines Schemas werden

üblicherweise als Tupel bezeichnet. Die Konsistenz der jeweiligen Tupel wird durch das Datenbankmanagementsystem überprüft und kann durch verschiedene Integritätsbedingungen von Benutzerseite konfiguriert werden. Durch die Einfachheit des verwendeten Paradigmas werden nahezu keine Sprachmuster verwendet, die in der Meta-Modellierung Anwendung finden. Wie oben bereits erwähnt werden zwei Meta-Ebene betrachtet. Selbst die Vererbung, die während der Modellierungsphase (Entity-Relationship Modellierung [33]) häufig Anwendung findet, wird durch den Vorgang der Transformation meist in einfache Relationen aufgelöst.

Die Arbeiten im Bereich der Schema Evolution lassen sich grundlegend in zwei Gruppen unterteilen. Während einige Ansätze sich darauf konzentrieren Unterstützung für die manuelle Migration nach einer Änderung am Schema zu bieten, liefern andere Bibliotheken bzw. Taxonomien, um eine gezielte Evolution durchzuführen. Ein Beispiel für die Unterstützung der manuellen Migration stellt Ronström in [91] vor. Nachdem durch den Benutzer Änderungen am Schema durchgeführt wurden, werden anschließend die vorhandenen Tupel ins neue Schema kopiert und mit Hilfe von Triggern synchron gehalten. Wenn die Elemente des neuen Schemas bestimmten Konsistenzbedingungen genügen und damit die Migration korrekt verlaufen ist, werden die entsprechenden alten Tabellen entfernt und dadurch die darin enthalten Daten ebenfalls gelöscht. Erst wenn dieser Vorgang abgeschlossen ist, können neue Transaktionen auf den Tabellen ausgeführt werden.

Die Ansätze der zweiten Kategorie, stellen für gewisse Änderungen am Schema bestimmte Operatoren bereit, die gezielt durch den Anwender genutzt werden können. So zeigen Shneiderman und Thomas in [103] 15 Operatoren, die simultan sowohl Schema als auch die jeweiligen Tupel evolvieren. Ein weiterer Repräsentant dieses Ansatzes findet sich im PRISM System [25]. Dieses definiert eine Sprache, um Schema Änderungen auszudrücken und Werkzeuge, die die Auswirkungen der Schema Evolution untersuchen. Neben den betroffenen Daten werden vom System auch Queries und invalide Integritätsbedingungen [24], die durch die Evolution beeinflusst werden, während der Migration angepasst.

Aboulsamh und Davies präsentieren in [1] einen Ansatz, um modellgetrieben eine Evolution an Informationssystemen durchzuführen. Da der vorgestellte Ansatz am Beispiel eines ORMs (Object Relational Mapping) dargestellt wird und die resultierenden Änderungen relationaler Natur sind, wurde dieser Ansatz hier eingegliedert. Die Grundlage einer Evolution eines Informationssystems im vorgestellten Ansatz bilden verschieden Evolutionsmodelle, die dazu dienen, Transformationen in SQL zu erzeugen. Das verwendete Meta-Modell ist ein Erweiterung des UML Meta-Modells und erlaubt es Evolutionsschritte abzuspeichern. Diese werden im ersten Schritt des Generierungsprozesses in eine grafische Repräsentation und anschließend in eine textuelle Syntax transformiert. Mit Hilfe dieser Syntax werden dann SQL Statements erstellt, die die Evolution durchführen.

In [112] präsentieren Terwillinger et.al. mit MoDEF eine ähnliche Lösung. MoDEF ist eine Erweiterung für Visual Studio, die es erlaubt, im Microsoft Entity Framework Artefakte eines objekt-relationalen Mappings (ORM) zu evolvieren. Dabei wird für Änderungen am Client Modell ein Skript generiert, dass die notwendigen Änderungen am Store Modell vornimmt.

XML Schema Evolution

Die Extensible Markup Language (XML) ist eine Sprache, mit der strukturiert Daten innerhalb von Textdateien abgelegt werden können. Um die Struktur der Daten zugänglich zu machen, kann ein Schema für ein XML Dokument referenziert werden. Dabei gibt es prinzipiell zwei Möglichkeiten das Schema auszudrücken: XML Schema und Document Type Definition (DTD). Heutzutage ist die Verwendung von XML Schema aufgrund diverser Vorteile üblich. Dennoch gibt es auch viele Ansätze, die sich mit der Evolution einer DTD beschäftigen. So unterstützt das XEM System (XML Evolution

Manager) eine vollständige Menge an Operatoren zum gezielten Anpassen einer DTD [111]. Der Ansatz von Lämmel und Lohmann in [70] präsentiert eine Klassifikation (erhaltend, erweiternd, reduzierend) von Transformationsänderungen einer DTD hinsichtlich der Änderung der Struktur eines XML Dokuments. Zusätzlich stellen sie eine Menge an Operationen vor, die neben dem Schema auch die jeweiligen XML Instanzen via XSLT migrieren.

Im Bereich der Evolution eines XML Schemas finden sich ähnliche Ergebnisse. Moro et.al. stellen in [76] eine Taxonomie an einfachen und komplexen Änderungen eines Schemas vor. Dabei zeigen sie für jede Klasse an Änderungen auf, welche Auswirkungen die Evolution auf die Validierung und Query Evaluation haben. Zusätzlich liefern sie ebenfalls Empfehlungen, um XML-Query so zu erstellen, dass sie robust gegenüber Änderungen am Schema sind.

Ein weiterer Ansatz findet sich im X-Evolution System [46]. Die webbasierte Anwendung setzt auf ein kommerzielles Datenbanksystem auf und unterstützt die Evolution eines XML Schemas. Um die Änderungen durchzuführen, kann sowohl eine baumartige grafische Repräsentation eines XML Schemas als auch eine deklarative DSL verwendet werden. Nach jeder primitiven Änderung am Schema (Einfügen, Löschen und Ändern) folgt eine inkrementelle Validierungsphase, in der schrittweise die Inhalte des XML Dokuments migriert werden [47]. Der Migrationsprozess erfolgt dabei semi-automatisch und ist nach [74] minimal im Sinne des Benutzeraufwandes.

Objekt-orientierte Ansätze

Anders als bei relationalen liegt bei objekt-orientierten Datenbanken ein anderes Modell zu Grunde. Sie speichern Daten nicht als Relationen, sondern persistieren Objekte direkt [7]. Die beiden Paradigmen unterscheiden sich grundsätzlich und lassen sich nicht auf einfache Weise vereinen [11]. Dennoch finden objekt-orientierte Datenbanken in den meisten Informationssystemen, trotz ihrer Nähe zu den heutzutage häufig verwendeten objekt-orientierten Programmiersprachen, selten Anwendung [1].

Bedingt durch das Ausgangsmodell finden sich in der Forschung zur Schema Evolution von objekt-orientieren Datenbanken auch verschiedene Sprachmuster, die während der Änderung ebenfalls umgewandelt werden. So zeigen Nguyen und Rieu in [79], wie Änderungen an kompositen Objekten vollzogen werden können. Dafür stellen sie eine Bibliothek an Operatoren bereit, in der sich auch Repräsentanten für die Änderungen einer Vererbung wiederfinden. Ein weiterer Ansatz für den Umgang mit der Änderung am Schema findet sich im ORION System [9]. Durch eine Bibliothek von einfachen Operatoren können vom Benutzer Anpassungen am Schema erfolgen, die bestimmte Invarianten erhalten. Die vorgestellten Operatoren sind nach den Autoren Banerjee et.al. vollständig und folglich nicht erweiterbar. ORION unterstützt ebenfalls die Änderung von Vererbungshierarchien und kompositen Objekten.

Etwas anders widmet sich die O2 Datenbank der Herausforderung der Schema Evolution [35]. Im System werden zwar ebenfalls primitive Operatoren bereitgestellt, diese können aber durch benutzerspezifische Konvertierungs- und Migrationsfunktionen erweitert werden. Dabei werden ebenfalls die Komposition und das Sprachmuster der Vererbung berücksichtigt. Die jeweiligen Änderungen innerhalb des Schemas werden mit Hilfe einer DSL ausgedrückt.

Anders als die bisher vorgestellten Ansätze werden in [113] von Tresch und Scholl nicht nur zwei Meta-Ebenen betrachtet, sondern die Evolutionsszenarien von *data-objects*, *schema-objects* und *meta-schema objects* untersucht. Dabei werden die Objekte der verschiedenen Ebenen uniform behandelt. Für die Änderungen am COCOON genannten Objektmodell stellen Tresch und Scholl eine Algebra genannt COOL vor. Diese dient neben der Erstellung von Anfragen auch dazu, Änderungen an Objekten durchzuführen. Weiterhin beinhaltet COOL eine Reihe von generischen

Evolutionsoperatoren, die gezielt abhängige Objekte migrieren. Die vorgestellte Taxonomie von Änderungen am Schema ist vollständig und kann deshalb jede Form der Evolution durchführen [113].

3.1.2 Evolution von Ontologien

Ontologien werden dazu verwendet, um Begrifflichkeiten bzw. Informationen strukturiert abzuspeichern. Im Zeitalter des Semantic Web dienen sie dazu, Wissen für den Computer verständlich abzulegen. Durch ihre ähnliche Struktur sind viele Ansätze zu Unterstützung der Evolution von Ontologien durch die Schema-Evolution geprägt [89]. Dennoch besitzen die Ansätze durch die zugrunde liegenden Paradigmen auch Unterschiede. So besitzen Ontologien anders als Datenbanken nicht immer eine strenge Trennung zwischen Typ- und Instanz-Ebene. Vielmehr liegt die jeweilige Zuordnung in der Sichtweise des jeweiligen Elements innerhalb einer Ontologie [81]. Diese Sichtweise findet sich, wenn auch in einer anderen Form, auch in den im Kapitel 2 vorgestellten Sprachmustern wieder. Ein Clabject (oder Konzept) zum Beispiel steht auch für ein Element, das je nach Sichtweise (oder entsprechender Meta-Ebene) ein Typ oder eine Instanz sein kann. Demzufolge lassen sich auch in Ontologien mehrere Meta-Ebenen abbilden, wenn wir davon ausgehen, dass die Instanziierung eine Ebene festlegt: Zum Beispiel kann das (imaginäre) Buch „Meta-Modellierung leicht gemacht" zum einen eine Instanz für den Typ „Buch" sein, aber auf der anderen Seite auch als Typ für die jeweiligen konkreten Exemplare dienen. Durch diese und andere Unterschiede müssen die Ansätze im Bereich der Ontologie Evolution weiteren Herausforderungen im Vergleich zur Schema Evolution begegnen.

Neben diesen Unterschieden in den beiden Ansätzen zeigen Noy und Klein in [81] auch, warum ihrer Meinung nach Versionierung und Evolution im Bereich der Ontologien nicht voneinander getrennt betrachtet werden können. Sie klassifizieren zudem welche verschiedenen Kategorien an Änderungen es für Ontologien geben kann: *Instance-data preservation, Ontology perservation, Consequence-Preservation*. Weiterhin definieren sie einfache Operatoren zur Änderung von Ontologien, die in die verschiedenen Klassen einsortiert werden. Diese werden wiederum in einer Ontologie CHAO (Change and Annotation Ontology) gespeichert, die die einzelnen Änderungen zwischen zwei Versionen festhält [80]. Auch im Bereich der Ontologie Evolution werden Operatoren verwendet, um gezielt Änderungen zu beschreiben oder durchzuführen. So beinhaltet das KAON Framework [110] eine Reihe an einfachen und auch kompositen Operatoren, die Änderungen an einer Ontologie ausdrücken.

Hartung stellt in [48] ein Framework OnEX (Ontology Evolution Explorer) vor, mit dessen Hilfe Evolutionen von Ontologien bewertet werden können. Dabei werden die Auswirkungen auf spezifische Elemente durch die Evolution gezeigt. Als Grundlage zur Erkennung solcher Änderungen zeigt Hartung außerdem einen Algorithmus, der die Änderungen zwischen zwei verschiedenen Versionen einer Ontologie bestimmt. Während der Durchführung dieses DIFFs (siehe Anschnitt 3.1.3) werden komplexe Operationen wie das Teilen oder Zusammenfügen von Elementen in der Ontologie erkannt.

3.1.3 Model Management

Das Forschungsgebiet des Model Managements beschäftigt sich damit, wie verschiedene Modelle, die durch ein Mapping miteinander verbunden sind, während einer Evolution synchronisiert werden können. Der Begriff Modell im Sinne des Models Management ist sehr weit gefasst. So sind zum Beispiel XML Schemata, Datenbank Schemata, ein Schema einer Webseite oder auch Schnittstellen objekt-orientierter Sprachen unter dieser Definition ein Modell [12]. Aufgrund des allgemeinen Modellbegriffs lassen sich viele Ergebnisse der Forschung dieses Bereichs auch auf andere Bereiche wie die Schema Evolution anwenden [13, 14]. Um den oben genannten Herausforderung in einer uniformen Art und Weise zu begegnen, werden sowohl für die unterschiedlichen Modellarten als auch für die Mappings dazwischen Datenmodelle bereitgestellt. Auf diesen Datenmodellen werden

zusätzlich algebraische Operatoren definiert, die Gemeinsamkeiten (MATCH) oder Unterschiede (DIFF) von Modellen erkennen oder zwei Instanzen von Modellen zusammenführen (MERGE). Eine Implementierung des Ansatzes findet sich im Rondo System [72]. Darin können die verschiedenen Operatoren des Model Managements zu neue benutzerspezifischen Operatoren zusammengesetzt werden. Dazu werden primitive und abgeleitete Operatoren bereitgestellt, die als Grundlage der benutzerspezifischen Operatoren dienen.

3.1.4 Fazit

Die Arbeiten in den verwandten Bereichen zeigen, dass eine Schemaevolution eine Herausforderung in der Softwareentwicklung darstellt. Weiterhin wird deutlich, dass die vorgestellten Gebiete viele gemeinsame Grundkonzepte besitzen, sich aber in der spezifischen Umsetzung der Lösungen unterscheiden. Diese Unterschiede gehen aus den zugrundeliegenden Paradigmen (Relationen, Objektorientierung, XML, …) hervor und dienen zumeist der Verbesserung einer Migrationsstrategie. Deshalb werden die vorgestellten Ansätze zwar als Grundlage für die Meta-Model Evolution verwendet, sie können allerdings nicht die spezifischen Gegebenheiten eines Meta-Modells vollständig abbilden. Beispielsweise betrachteten alle in diesem Abschnitt vorgestellten Bereiche lediglich zwei Meta-Ebenen und liefern nur Unterstützung für das Muster der Vererbung.

Gemeinsam ist allen Bereichen (bis auf den Bereich des Model Managements), dass die beiden Konzepte des Matchings bzw. die Bereitstellung einer Bibliothek an Operatoren verwendet werden, um Unterstützung für eine Evolution zu bieten. Dies zeigt, dass diese beiden Lösungsansätze sich für die prinzipiellen Umgang mit der Evolution sehr gut eignen. Dennoch besitzen beide Ansätze auch Schwachstellen, denen begegnet werden muss. So treten beim Matching verschiedener Schemata Mehrdeutigkeiten bei der Erkennung komplexer Evolution auf. Andererseits entstehen bei der Bereitstellung von Operatoren häufig große und für den Benutzer unübersichtliche Bibliotheken, die dadurch nur schwer einzusetzen sind.

3.2 Meta-Modell Evolution

Meta-Modelle legen die Struktur der jeweiligen Instanz-Modelle fest. Darum kann die Änderung eines Meta-Modells zur Folge haben, dass Instanz-Modelle ungültig werden. Um diese invaliden Modelle wieder nutzbar zu machen, ist eine Migration (auch Koevolution genannt) von Nöten. Die Unterstützung dieses Vorgangs sowie die Aufzeichnung der Evolution stellen zwei große Herausforderungen im Bereich der Meta-Modell Evolution dar. Die verschiedenen Arbeiten werden häufig anhand des Umgangs mit diesen beiden Herausforderungen klassifiziert. So können nach Herrmannsdörfer und Kögel [55] zustandsbasierte, änderungsbasierte und operationsbasierte Ansätze zur Aufzeichnung der Evolution von Metamodellen unterschieden werden. Dabei unterscheiden sich zustandsorientieren dadurch von den änderungsbasierten Ansätzen, dass sie unterschiedliche Versionen von Modellen speichern und daraus Unterschiede ableiten (Implementierung des DIFF Operators aus dem Model Management Bereich), während die letztgenannten die Änderungen eines Modells aufzeichnen, während sie auftreten. Die operationsbasierten Ansätze bilden einen Spezialfall der änderungsbasierten; insofern, als dass sie die aufgezeichneten Änderungen mit Hilfe einer Menge von vordefinierten Operationen aufzeichnen.

Als zweite Klassifikation dient der Umgang mit der Koevolution von invaliden Instanz-Modellen (Abschnitt 4.7 aus [52] oder auch [94]). Während einige Ansätze sich darauf spezialisieren, eine manuelle Erstellung von Migrationsstrategien zu unterstützen, versuchen andere Arbeiten aus verschiedenen Versionen (Meta-Modell Match) des Modells Änderungen abzuleiten, die an den

Instanz-Modellen durchgeführt werden müssen und wieder andere stellen eine Bibliothek an Operatoren bereit, die eine Evolutionssemantik definieren und gezielt Evolution und Koevolution ausführen. Da diese letzte Klasse die beiden Evolutionen simultan ausführt, wird auch häufig von *coupled evolution* (z.B. [56, 90]) gesprochen. Die im Folgenden vorgestellten Ansätze werden auf Grundlage der zweiten Klassifikation in Abschnitte eingeteilt, da diese Ordnung aufgrund des Fokus der Arbeit am sinnvollsten ist.

Für den Vergleich der nachfolgend vorgestellten Ansätzen sollen zunächst Anforderungen aus der Herausforderungen H1 - H3 aus Abschnitt 1.4 abgleitet werden:

Anforderung 1 Modularität und Wiederverwendung von Evolutionswissen: Um Redundanzen und Aufwand zu vermeiden, soll nach Möglichkeit die Evolutionsstrategie modular aufbaubar sein und somit Evolutionswissen wiederverwendbar machen. (notwendig für H1, H3)

Anforderung 2 Benutzerunterstützung bei der iterativen Evolution: Zumeist gibt es viele verschiedene Varianten, um ein Modell auszudrücken und damit auch zu ändern. Deshalb soll diese Anforderung festhalten, ob der Benutzer bei der schrittweisen Änderung am Modell unterstützt wird und ob einzelne Schritte in der Evolution rückgängig gemacht und erneut ausgeführt werden können. Weiterhin beinhaltet diese Anforderung wie Evolution aufgezeichnet und dem Benutzer zugänglich gemacht wird, damit dieser Bewertungen der einzelnen Evolutionsschritte vollziehen kann. (notwendig für H3)

Anforderung 3 Unterstützung der Evolution von allen in Kapitel 2 vorgestellten sprachbasierten Mustern: Der Umgang mit Sprachmustern ist am konkreten Modell teilweise komplex, da sich die verschiedenen Muster auf viele Modellelemente auswirken und sich auch gegenseitig beeinflussen. Daher soll diese Anforderung untersuchen, wie gut ein Ansatz die in Kapitel 2 vorgestellten Muster unterstützt und ob er Strategien für die mit ihnen einhergehende Evolution bereitstellt. (notwendig für H1, H2)

3.2.1 Manuelle Spezifikation der Migration

Die Arbeiten im Bereich der manuellen Spezifikation der Migration liefern Sprachen, um Modell-Transformationen durch den Benutzer auszudrücken. Dabei erstellt der Modellierer eine Migrationsstrategie zumeist nicht von Grund auf, da häufig ein Standardverhalten unterstützt wird. Aufgrund der Vielfältigkeit der Modell-Evolutionen müssen jedoch viele Änderungen am Standardverhalten durch den Modellierer spezifiziert werden, wodurch ein, im Vergleich zu anderen Ansätzen, erhöhter Aufwand entsteht. Andererseits ist dadurch der größte Teil der Evolutionsstrategie durch den Modellierer erstellt worden, wodurch die Auswirkungen für sie bzw. ihn einfacher ersichtlich sind.

Domain evolution framework

Das domain evolution framework wird von Sprinkle in [105] eingeführt. Das Werkzeug basiert auf dem Generic Modeling Environment (GME), in dem Modelle in Form von XMI persistiert werden. Für die Evolution der jeweiligen Modelle stellt das System eine DSL bereit, die es erlaubt, Unterschiede zwischen verschiedenen Versionen von Modellen auszudrücken. Neben der textuellen DSL liefert Sprinkle zusammen mit Karsai in [106] auch eine grafische Sprache, die den gleichen Zweck erfüllt. Die Strategie zur Migration der einzelnen Modelle wird durch den Modellierer festgelegt und mit Hilfe von XSL Transformationen propagiert.

Bezogen auf die oben genannten Anforderungen zeigt sich:

- Anforderung 1 ist aufgrund der Tatsache, dass die Transformationen für die Migration im Wesentlichen durch den Modellierer von Grund auf erstellt werden müssen, nicht erfüllt.
- Anforderung 2 ist erfüllt. Dadurch, dass der Modellierer das Wissen über die Evolution bereitstellt, ist die Notwendigkeit die konkreten Änderungen an verschiedenen Modellen darzustellen abgeschwächt. Durch die Verwendung von XSL sind beide Versionen des Modells zugänglich und damit ein Rücksetzen oder Wiederherstellen trivial.
- Anforderung 3 ist nicht erfüllt. Da GME als Grundlage des Ansatzes bis zu 3 Meta-Ebenen und lediglich das Sprachmuster der Vererbung unterstützt, liegen die anderen Sprachmuster nicht im Fokus dieses Systems. Zusätzlich schränkt der Ansatz der manuellen Spezifikation der Migration die Unterstützung der Evolution von Sprachmustern von vornherein sehr stark ein.

Epsilon Flock

Rose zeigt in [93] einen weiteren Ansatz, um die manuelle Spezifikation der Modell Migration zu unterstützen. Dafür definiert Rose eine eigene textuelle Modellierungsnotation genannt Epsilon HUTN (Human-Usable Textual Notation) [95], die bekannten Objekt Repräsentationen wie JSON ähnelt (siehe Listing 3-1). Alle konformen Modelle werden in XMI transformiert und gespeichert. Die in Epsilon HUTN beschrieben Modelle sind interoperabel mit EMF. Um die Evolution durchzuführen, wird neben der Repräsentation der Modelle auch eine Sprache Epsilon Flock geliefert, die für die Migration verantwortlich ist. Sie stellt eine regelbasierte Transformationssprache (siehe Listing 3-2) dar, die die Vermischung von deklarativen und imperativen Blöcken erlaubt [92].

```
1  FamilyPackage "families" {
2     Family "The Smiths" {
3        nuclear: true
4        name: "The Smiths"
5        averageAge: 25.7
6        numberOfPets: 2
7        address: "120 Main Street", "37 University Road"
8     }
9  }
```

Listing 3-1 Epsilon HUTN Notation (entnommen aus [95]*)*

Zusätzlich dazu unterstützt Epsilon Flock gewisse Routinen, wie das Kopieren von validen Elementen von der alten Modellversion in die neue. Diese werden als Standardverhalten bei der initialen Evolution verwendet. Für die Migration der anderen invaliden Elemente muss der Modellierer inkrementell die Transformation anpassen, bis diese soweit verfeinert wurde, dass die Evolution vollständig ist.

```
1  migrate <originalType> (to <evolvedType>)?
2  (when (:<eolExpression>)|({<eolStatement>+}))?
3    <eolStatement>*
4  }
5
6  delete <originalType>
7  (when (:<eolExpression>)|({<eolStatement>+}))?
```

Listing 3-2 konkrete Syntax der Epsilon Flock Migrations- und Löschregeln (entnommen aus [92]*)*

Die Anforderungen von oben werden durch Epsilon Flock wie folgt erfüllt:

- Anforderung 1 ist nicht erfüllt. Zwar werden gewisse Routinen verwendet, um häufig auftretende Muster (Kopieren der Inhalte) in ein Standardverhalten auszulagern, aber diese müssen zumeist stark angepasst werden. Somit muss der Modellierer den größten Teil der Migration eigenständig erstellen.
- Anforderung 2 ist ebenso wie bei Sprinkles Domain evolution framework erfüllt. Auch hier liegt die Begründung in der grundsätzlichen Ausrichtung des Ansatzes, da auch hier der Modellierer den größten Teil des Evolutionswissens bereitstellt.
- Anforderung 3 ist nicht erfüllt, da für die Sprachmuster keine Unterstützung durch das System geboten wird.

Andere Ansätze

Häufig referenzieren nicht nur Modelle sondern auch andere Artefakt wie zum Beispiel Transformationen Meta-Modelle. Demzufolge wirkt sich die Änderung eines Meta-Modells auch auf diese entsprechend aus. Hoisl et.al. [57] liefern einen Ansatz um Mappings zwischen Modellen, Transformationen und Meta-Modellen zu definieren, damit diese bei Änderungen synchronisiert werden können.

Ein weiterer Ansatz der besonders für die Anforderung 2 interessant ist, wird in [37] vorgestellt. Darin werden die Rollen von Meta-Modellierer und Modellierer getrennt, in dem Sinne, dass der Meta-Modellierer die Evolution des Meta-Modells vorgibt und Empfehlungen für die Migration für den Modellierer hinterlegt, der die Anpassung entsprechend durchführt.

In [29] präsentieren Demuth et al. einen Ansatz, der die freie Evolution unterstützt. Dabei geben sie keine Restriktionen bzw. Hilfestellungen während der Evolution vor, sondern validieren fortlaufend Bedingungen, die durch das Meta-Modell oder anderen Konsistenzbedingungen (Constraints) vorgegeben sind. Wenn eine Verletzung vorliegt, werden dem Modellierer automatisch Vorschläge unterbreitet, wie die jeweiligen Brüche aufgelöst werden könnten.

3.2.2 Meta-Modell Matching

Ansätze im Bereich des Meta-Modell Matchings verwenden eine Vorgehensweise, die auf den Ansätzen beruht, die bereits im Abschnitt des Model Managements beschrieben wurden. Aus zwei Versionen eines Modells werden Unterschiede ermittelt und in einem entsprechenden Modell gespeichert. Auf Basis dieser Unterschiede wird anschließend einen Migrationsstrategie generiert, die sich um die Koevolution der invaliden Elemente kümmert. Beim Erstellen dieser Strategie müssen Mehrdeutigkeiten [94] des Evolutionsablaufes aufgelöst werden, die zwar für die Evolution des Meta-Modells äquivalent sind, sich jedoch in der Migration unterscheiden. Beispielsweise verwenden das Verschieben eines Attributes zur Generalisierung und das Löschen des Attributes bei der Spezialisierung mit anschließendem Erstellen eines gleichnamigen Attributes bei der Generalisierung die gleichen elementaren Änderungen. Im ersten Fall bleiben die Werte der Attribute unberührt, während der zweite Fall ein Entfernen aller Zuweisungen verlangt.

Gruschko et al.

Die Forschungsgruppe um Boris Gruschko beschäftigt sich mit der Evolution von Ecore/EMF [109] oder MOF/UML (Meta)-Modellen [44]. Dafür liefern sie in [10, 45] eine Klassifikation von Änderungen. Dabei unterscheiden sie Änderungen, die Modell Instanzen nicht beeinflussen (*non-breaking changes*), Änderungen, deren Migration automatisch erstellt werden kann (*breaking and resolvable changes*), und Änderungen, die Benutzerinformationen zur Migration benötigen (*breaking and non-resolveable changes*). Zusätzlich zur Klassifikation liefern sie einen Prozess (siehe Abbildung 3-1), der beschreibt,

wie die Evolution abläuft und dabei darstellt, wie jede Klasse an Änderungen migriert wird. Nachdem von zwei Versionen eines Meta-Modells die entsprechenden Unterschiede extrahiert wurden, werden diese Änderungen in die Klassen eingeteilt. Je nachdem welche Klasse vorliegt, wird dann eine Migrationsstrategie gewählt. In [16, 17] wird die Klassifikation der Änderungen auch auf MOF angewendet und diese in OCL (Object Constraint Language) formalisiert. Auf Basis der Klassifikation und der Änderungen am Meta-Modell wird eine Implementierung vorgestellt, die die Auswirkung der Evolution am Meta-Modell auf die Modell Instanzen visualisiert. Damit können Benutzer semiautomatisch eine Bewertung der Evolution durchführen.

Bezogen auf die eingangs formulierten Anforderungen ergibt sich für den Ansatz folgende Bewertung:

- Anforderung 1 ist teilweise erfüllt, da für die ersten beiden Klassen an Änderungen, die entweder keiner Migration bedürfen oder auflösbar sind, vom System Strategien angeboten werden, um diese zu migrieren. Dadurch wird ein gewisser Teil des Wissens wiederverwendet. Dennoch müssen für die letzte Klasse eigene Strategien entwickelt werden, deren Erstellung zwar unterstützt wird, aber ein modularer Aufbau nicht möglich ist.
- Anforderung 2 ist erfüllt, da zum einen beide Versionen des Modells vorliegen und auch durch den Ansatz eine Unterstützung zur Bewertung der Evolution durch den Modellierer geboten wird.
- Anforderung 3 ist nicht erfüllt, da durch die verwendeten Umgebungen (Ecore, MOF) die Ebene-Hierarchie festgelegt ist und auch nur das Sprachmuster der Vererbung berücksichtigt werden kann.

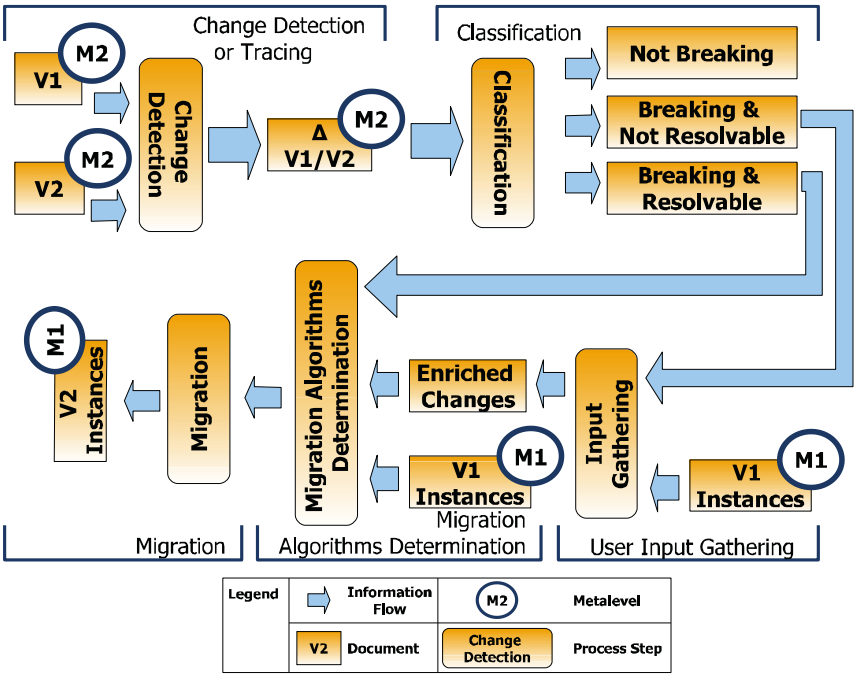

Abbildung 3-1 Evolutionsprozess des Ansatzes von Gruschko (entnommen aus [10]*)*

Cicchetti et al. / EMFMigrate

Die Arbeiten von Cicchetti, Ruscio et al. verwenden ebenfalls einen auf Meta-Modell Matching basierenden Ansatz. Dazu werden Änderungen nach der Klassifikation von Gruschko et al. eingeteilt. Allerdings versucht der Ansatz nicht nur einfache Modelländerungen zu erkennen, sondern leitet aus den initial ermittelten Unterschieden zweier Modelversionen komplexe Änderungen ab [18, 20]. Diese

bilden anschließend die Grundlage für Modell Transformationen, die die Migration durchführen. Die zu anfangs generierten Unterschiede zweier Modelle werden innerhalb eines Meta-Modells abgelegt [19]. Neben den Auswirkungen der Evolution eines Meta-Modells auf entsprechende Instanz-Modelle betrachtet Ruscio in [100] auch die Effekte auf eine Spezifikation der konkreten Syntax. Dafür verwendet er Transformationen, die die entsprechende Migration der in TCS (DSL zur Spezifikation einer konkreten textuellen Syntax) beschriebenen Syntax vollziehen. Eine Umsetzung des Ansatzes liefert Ruscio in Form des Tools EMFMigrate [101]. Neben dieser Migration einer konkreten Syntax beschäftigt sich die Forschungsgruppe um das Projekt EMFMigrate auch mit der Anpassung anderer Artefakte, die vom Meta-Modell abhängig sind [90].

Die eingangs formulierten Anforderungen werden durch den Ansatz wie folgt umgesetzt:

- Anforderung 1 ist erfüllt, da verschiedene Migrationsstrategien in Bibliotheken zur Verfügung gestellt werden. Zudem werden auch komplexe Evolutionsoperationen abgeleitet und unterstützt.
- Anforderung 2 ist ebenfalls erfüllt, da beide Versionen des Meta-Modells vorliegen und Unterschiede dem Modellierer visualisiert werden.
- Anforderung 3 ist nicht erfüllt, da die komplexen Evolutionsoperationen lediglich das Sprachmuster der Vererbung beinhalten. Für andere Sprachmuster wird keine Unterstützung geliefert und folglich muss die Migration durch den Modellierer manuell erfolgen.

Weitere Ansätze

In [114] stellen Vermolen et al. einen Ansatz vor, der ebenfalls auf die Technik des Meta-Modell Matchings beruht. Anders als die meisten Ansätze versucht dieser aus den primitiven Änderungen, die durch das Matching bzw. den DIFF entstehen, komplexe Evolutionsvorgänge abzuleiten. Dabei beziehen sie Abhängigkeiten zwischen verschiedenen Operationen ein und versuchen Überlagerungen, Auslöschung und Vermischungen von Änderungen zu erkennen, sowie mögliche Ausführungsreihenfolgen zu rekonstruieren. Der konkrete Vorgang sieht so aus, dass zunächst ein Evolutionsablauf (Trace genannt) ermittelt wird und daraus Operatoren (siehe nächster Abschnitt) abgeleitet werden.

3.2.3 Operationsbasierte Ansätze

Die in diesem Abschnitt vorgestellten Ansätze liefern Bibliotheken an Operatoren, die durch den Modellierer gezielt verwendet werden, um sowohl das Meta-Modell als auch die jeweiligen Modellinstanzen simultan zu ändern. Die jeweiligen Operatoren besitzen also eine konkrete Semantik, die durch die Änderung vorgegeben ist. Daher sind sie prädestiniert, komplexe Evolutionsschritte zu kapseln.

COPE / Edapt

COPE ist ein von der Forschungsgruppe um Markus Herrmannsdörfer entwickeltes System, das auf EMF aufsetzt und zu anfangs in Groovy implementiert wurde [53]. Später wurde es unter dem Namen Edapt in die Eclipse Incubation aufgenommen und nach Java migriert [31]. Der Hauptfokus des Ansatzes liegt darauf, Evolutionswissen wiederverwenden zu können und eine Automatisierung der Migration voranzutreiben [52]. COPE bzw. Edapt definiert eine Bibliothek an Operatoren [54], die zur gezielten simultanen Evolution von Meta-Modell und Modell verwendet wird. Die jeweiligen Abfolgen von Operatoren werden ebenfalls in einem Meta-Modell gespeichert [55]. Die Implementierung innerhalb des EMF Editors (siehe Abbildung 3-1) visualisiert neben der Historie der Evolution auch die

elementaren Änderungen am Meta-Modell und die verwendete Migrationsstrategie für die Instanz-Modelle.

Auf die Anforderungen bezogen ergibt sich folgendes Ergebnis:

- Anforderung 1 ist erfüllt, da durch die Operatoren das Evolutionswissen wiederverwendet werden kann. Zusätzlich lassen sich eigene Operatoren durch den Einsatz anderer bereits existierender Operatoren erstellen, wodurch die Modularität des Ansatzes zum Tragen kommt.
- Anforderung 2 ist erfüllt, da Undo/Redo Funktionalität unterstützt wird und ebenfalls eine Historie vorhanden ist, die anzeigt, welche konkreten Änderungen am Modell stattgefunden haben.
- Anforderung 3 ist nicht erfüllt, da durch die Schranken von EMF nur Modellhierarchien mit einer begrenzen Anzahl an Meta-Ebenen möglich sind und neben der Vererbung kein anderes Sprachmuster aus Kapitel 2 verwendet wird.

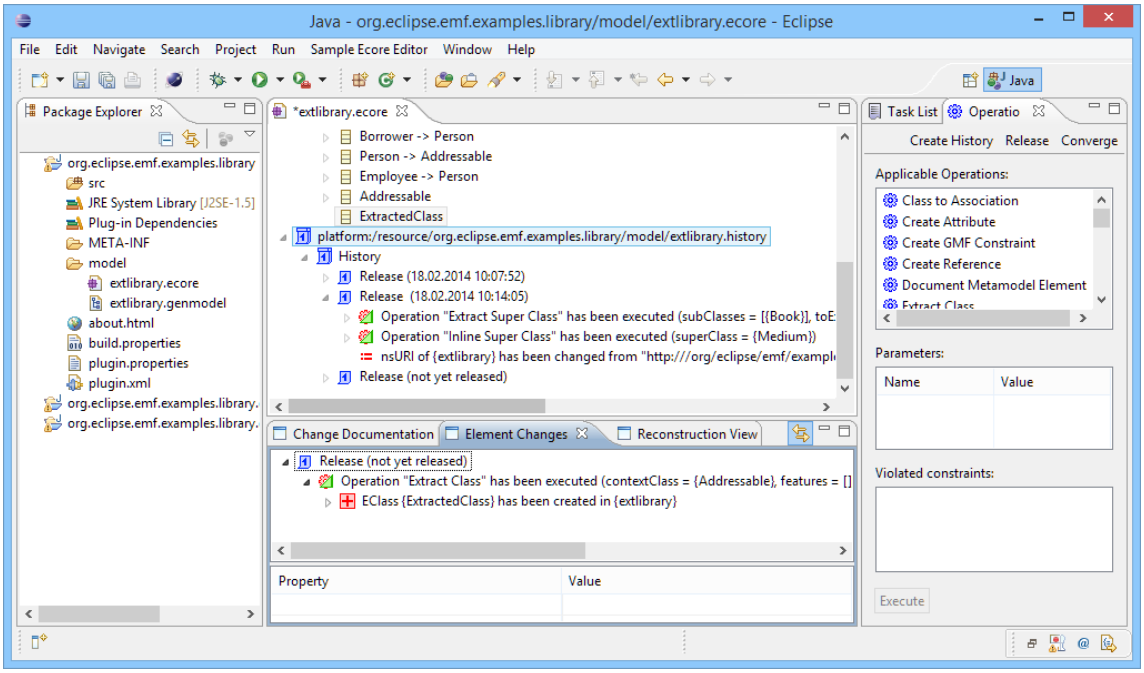

Abbildung 3-2 Screenshot des Edapt Systems

Wachsmuth

Wachsmuth stellt in [119] ebenfalls eine Bibliothek von Operatoren zur simultanen Evolution von Meta-Modellen und Instanz-Modellen vor. Dabei basiert seine Arbeit auf dem Modellierungsparadigma von MOF. Die vorgestellten Operatoren werden in verschiedene Kategorien eingeteilt und mit Hilfe von QVT (Query View Transformation) Modell-zu-Modell Transformationen beschrieben. Die vorgestellten Operatoren werden in [56] mit den Ansätzen von Herrmannsdörfer vermischt und bilden nach Meinung der Autoren ein hinsichtlich der Praxisrelevanz komplette Bibliothek an Operatoren.

Die zu Anfangs definierten Anforderungen erfüllt der Ansatz wie folgt:

- Anforderung 1 ist aufgrund der Verwendung von Operatoren erfüllt, da dadurch Transformationen wiederverwendet werden können.

- Anforderung 2 ist teilweise erfüllt. Zwar sind die jeweiligen Operatoren umkehrbar, es wird aber im Ansatz keine konkrete Unterstützung für eine Bewertung der Evolution durch den Benutzer geboten.
- Anforderung 3 ist nicht erfüllt, da lediglich für das Sprachmuster der Vererbung Operatoren bereitgestellt werden, die die Einführung und Änderungen unterstützen.

3.2.4 Fazit

Der in diesem Abschnitt vorgestellte Einblick in die vorhandenen Ansätze bezüglich der Evolution von Meta-Modellen in der aktuellen Forschung verdeutlicht, dass die Herausforderung bereits erkannt und ihr auch begegnet wurde. Dabei zeigt sich, dass die Benutzerunterstützung hinsichtlich der Evolution eines Meta-Modells im Vergleich zur manuell durchgeführten Migration wesentlich verbessert werden kann. Dies geschieht prinzipiell in drei verschieden Varianten. Während die in Abschnitt 3.2.1 vorgestellten Arbeiten versuchen die manuelle Migration zu vereinfachen, liefern die Ansätze im Bereich des Meta-Modell Matchings (Abschnitt 3.2.2) bereits Möglichkeiten die Migration der invaliden Modelle aus den Unterschieden der Meta-Modell Versionen abzuleiten. Dabei treten allerdings Mehrdeutigkeiten bei der Erkennung der Evolutionsabfolgen auf, die vom Benutzer aufgelöst werden müssen. Daneben beschränken sich diese Ansätze zumeist auf einfache Evolutionsänderungen, da komplexe Transformationssemantiken schwer zu rekonstruieren sind [114]. Die dritte Gruppe von Ansätzen (Abschnitt 3.2.3) begegnet diesem Problem, in dem sie eine Bibliothek an Operatoren bereitstellt, in der auch komplexe Evolutionstransformationen aufgenommen werden können. Allerdings entstehen dadurch meist große und unübersichtliche Bibliotheken, die die Benutzung erschweren.

Es spiegelt sich in der Betrachtung wieder, dass die Anforderungen, die zu eingangs formuliert wurden, von keinem System oder Ansatz vollständig erfüllt werden. Besonders die Unterstützung von sprachbasierten Mustern sticht dabei negativ heraus. Häufig wird zwar das Muster der Vererbung unterstützt, jedoch werden andere Muster und eine Kombination aus mehreren Mustern von der aktuellen Forschung in Bezug auf die Evolution nicht beleuchtet. Diese Tatsache ist dadurch geschuldet, dass die jeweiligen Modellierungssysteme, auf die die Ansätze aufbauen, neben der Vererbung kein anderes Sprachmuster unterstützen. In anderen Punkten jedoch bieten die bisherigen Arbeiten viele Konzepte, die den Umgang mit der Evolution erleichtern. So bietet beispielsweise der Operatoransatz eine Möglichkeit komplexe Evolutionstransformationen zu kapseln und damit Evolution gezielt auszuführen. Dem gewünschten Verhalten bezüglich der Unterstützung einer iterativen Evolution zusammen mit einer Wiederverwendbarkeit von Evolutionswissen kommt der Ansatz von Herrmannsdörfer im Edapt/COPE System am nächsten. Die Ergebnisse der Evaluierung des Abschnittes 3.2 wurde in der Tabelle 3-1 zusammengefasst.

Tabelle 3-1 Übersicht über die verschiedenen Ansätze im Bereich der Meta-Modell Evolution hinsichtlich der Anforderungen

Anforderung→ / Ansatz↓	Modularität/ Wiederverwendung	Iterative Evolution	Sprachmuster
Domain evolution framework	-	+	-
Epsilon Flock	-	+	-
Gruschko et al.	+/-	+	-
Cicchetti et al. / EMFMigrate	+	+	-
COPE / Edapt	+	+	-
Wachsmuth	+	+/-	-

4 Grundlagen

In diesem Kapitel werden die Grundlagen für die in Kapitel 2 beschriebenen Operatoren zum Umgang mit sprachbasierten Mustern vorgestellt. Dazu wird zunächst das operatorenbasierte linguistische Meta-Modell vorgestellt, das die Definition von beliebigen Meta-Hierarchien zulässt. Anschließend werden wichtige Begrifflichkeiten definiert und Regeln aufgestellt, die später in den Operatoren angewendet werden. In den letzten beiden Abschnitten werden noch wichtige Notationen der zur Beschreibung der Operatoren verwendeten Diagrammarten präsentiert.

4.1 Das operatorenbasierte linguistische Meta-Modell

Der folgende Abschnitt soll das operatorenbasierte linguistische Meta-Modell (LMM) einführen. Die Grundlagen des LMMs finden sich bei Volz [117]. Um eine tiefe Integration der Meta-Modell Evolution zu erreichen, wurde das LMM um den Operatorgedanken erweitert. Dies hat zur Folge, dass jedes Modellelement als Operator realisiert ist und damit alle Modelländerungen mittels deren Ausführung erfolgen können. Dadurch lassen sich komplexe Modelltransformationen aus diesen Basisoperatoren aufbauen. Dies kommt einer Realisierung des Ansatzes in einem Modellierungssystem zu Gute, da die Funktionalität der Modellelemente wiederverwendet werden kann und somit komplexe Änderungen durch Komposition einfacher Änderungen entstehen. Beispielhaft hierfür kann der *Move Attribute to Powertype Instance* Operator (Abschnitt 6.3.1) betrachtet werden. Dieser verwendet die jeweiligen Operatoren, um Attribute und Zuweisungen zu löschen, den Typ eines Attributes zu setzen und den Deep Instantiation Zähler zu inkrementieren.

In den folgenden Unterabschnitten wird für jedes Modellelement ein Klassendiagramm vorgestellt und dabei die Zusammenhänge mit anderen Elementen beschrieben. Es ist anzumerken, dass die folgenden Diagramme lediglich dazu gedacht sind, die Eigenschaften der jeweiligen Elemente zu verdeutlichen und nicht zwangsweise als exakte Vorlage zur Implementierung dienen.

4.1.1 Operator und Modellelemente

Abbildung 4-1 zeigt das Klassendiagramm des Operators zusammen mit der Vererbungshierarchie der Modellelemente. Operatoren bilden die Grundlage aller Änderungen eines Modells. Jeder Operator (Klasse `Operator`) definiert also eine Semantik einer Modelltransformation. Diese wird durch die Ausführung des Operators (`execute` Methode) ausgelöst. In den meisten Fällen ändert ein Operator nicht nur ein bestimmtes Modellelement, sondern migriert auch andere invalide gewordene Modellelemente (Koevolution). Deshalb kann die Ausführung eines Operators in verschiedenen Modi (Enumeration `ExecutionType`) durchgeführt werden. Der Regelfall ist die Ausführung der Evolution gemeinsam mit der Koevolution (Literal `ALL`). Dennoch gibt es Szenarien (siehe Kapitel 5), in denen beispielsweise die Änderungen am Modell schon durchgeführt wurden und nur noch die Koevolution durchgeführt werden soll. Deshalb ist es nötig, die Evolution (Literal `EVOLVE`) und Koevolution (Literal `CO_EVOLVE`) auch getrennt voneinander ausführen zu können.

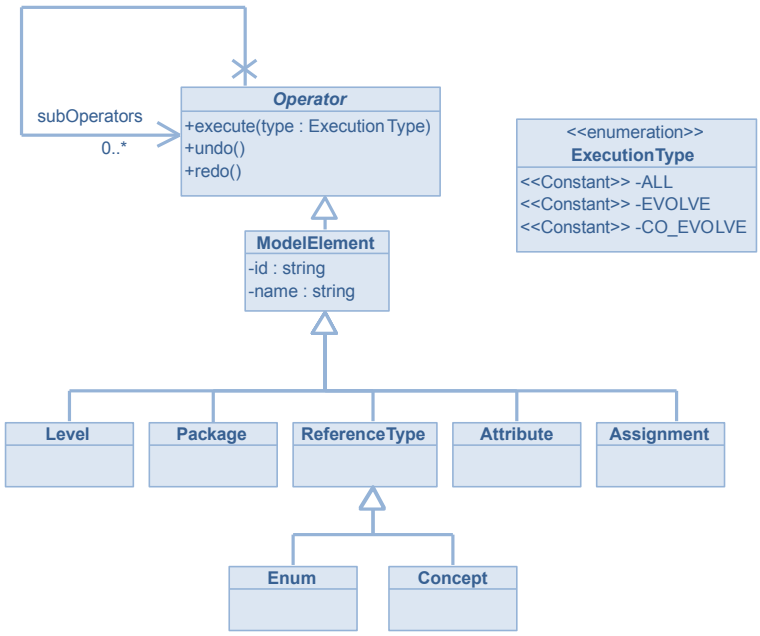

Abbildung 4-1 Vererbungshierarchie der Modellelemente des operatorenbasierten LMMs

Jeder Operator besitzt eine Methode, die seine Änderungen rückgängig macht (Methode **undo**) oder sie erneut ausführt (Methode **redo**). Operatoren können aus andern Operatoren aufgebaut sein (Attribut **subOperators**), die bei einer Ausführung des Operators ebenfalls ausgeführt werden. Konsequenterweise sind deshalb alle Modellelemente (**ModelElement**) ebenfalls als Operatoren realisiert. Dadurch werden Änderungen am Modell durch die Ausführung des jeweiligen Elementoperators durchgeführt. Aus Gründen der einfachen Identifikation besitzt jedes Modellelement zum einen eine eindeutige **id** und zum anderen einen Namen (Attribut **name**), der es unter allen Elementen, die zum gleichen Container gehören, ebenfalls eindeutig kennzeichnet. Der Aufbau der Containerhierarchie wird in den nachfolgenden Abschnitten erklärt.

4.1.2 (Meta-)Ebenen und Pakete

Ebenen (**Level**) bilden die grundlegenden Elemente einer Meta-Hierarchie. Neben einem Namen (**Level** ist eine Spezialisierung von **ModellElement**) können Ebenen einander instanziieren (**instanceOf**)[10]. Diese Beziehung ermöglicht es u.a. den Konzepten innerhalb der Instanz-Ebene, andere Konzepte der Typ Ebene zu instanziieren. Um Ebenen modular aufzubauen, besteht die Möglichkeit Ebenen logisch miteinander zu verschmelzen (**alignedWith**). Damit können Basis-Ebenen definiert werden, die dann von anderen Ebenen wiederverwendet werden können. Dementsprechend wird durch die beiden Beziehungen **instanceOf** und **aligendWith** das Erstellen von beliebigen Metahierarchien ermöglicht (siehe Abschnitt 2.1). Um innerhalb einer Ebene eine Struktur zu schaffen, existieren Pakete (**Package**), die wiederum Subpakete enthalten können.

[10] Die beschriebene Beziehung wird hier zwar **instanceOf** genannt, stimmt aber mit der **references** Beziehung aus ([117], Abschnitt 6.11.2) überein. Diese ist eine Generalisierung der strikten Instanziierung und erlaubt auch Spezialisierungen, die aufgrund einer Powertyp Instanziierung Meta-Ebenen überschreiten. Aufgrund einer intuitiven Nomenklatur, wird dennoch dieser Begriff hier verwendet, da die rein strikte Instanziierung ausgeblendet wird.

Innerhalb eines Paketes liegen Referenztypen, die entweder ein Konzept oder eine Enumeration darstellen und im nachfolgenden Abschnitt näher erläutert werden.

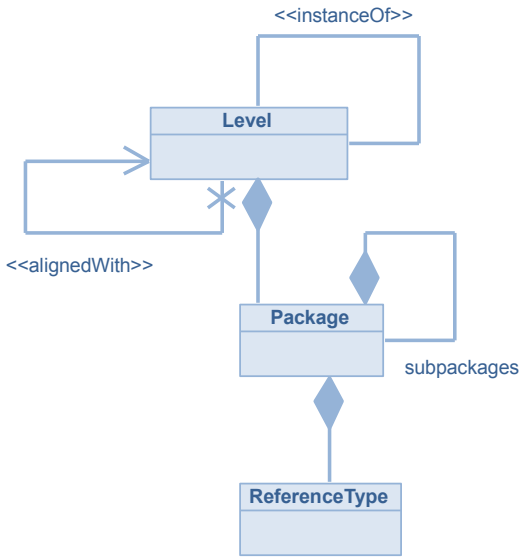

Abbildung 4-2 Ausschnitt aus der Containerhierarchie mit Ebenen, Paketen und Referenztypen

4.1.3 Konzepte und Enumerationen

Abbildung 4-3 zeigt einen Ausschnitt aus der Containerhierarchie des LMMs. Konzepte (`Concept`) und Enumerationen (`Enum`) sind beides Spezialisierungen der abstrakten Klasse `ReferenceType`[11]. Enumeration sind (ähnlich wie in Programmiersprachen) Typen mit einer endlichen und vorher festgelegten Anzahl an Werten (`literals`). Konzepte dagegen stellen das in Abschnitt 2.1 vorgestellte Muster des Clabjects dar. Sie haben verschiedene Beziehungen zu andern Konzepten, die für eine Vererbung (`<<extends>>`), eine Instanziierung(`<<instanceOf>>`), die Partitionierung (`<<partitions>>`) oder eine Instanz-Spezialisierung (`<<concreteUseOf>>`) stehen. Die Beziehungen sind jeweils von der (Instanz-)Spezialisierung, der Instanz oder dem Powertyp ausgehend. Die entsprechenden Sprachmuster Vererbung, Powertyp sowie Instanz-Spezialisierung werden in den Abschnitten 2.2, 2.4 bzw. 2.6 vorgestellt. Wenn eine Typ- und/oder eine Instanz-Spezialisierung verboten werden soll, kann ein Konzept final gesetzt werden (Wert `true` für die Eigenschaft `final`). Analog kann eine Instanziierung verhindert werden, indem das Konzept als abstrakt deklariert wird (Wert `true` für die Eigenschaft `abstract`). Weiterhin besitzt jedes Konzept einen Deep Instantiation Zähler (`diCounter`), der eine Instanziierung des Konzepts verzögern kann (siehe Abschnitt 2.3). Die formalen Definitionen dieser Eigenschaften befindet sich in Abschnitt 4.2.2.

[11] Bei Volz in [117] wird für Referenztypen eine Sichtbarkeit definiert. Da jedoch das LMM keine Methoden besitzt und die Definition nicht eindeutig ist (siehe [117] Abschnitt 6.11.1 und Abschnitt 7.4.7), wird an dieser Stelle darauf verzichtet

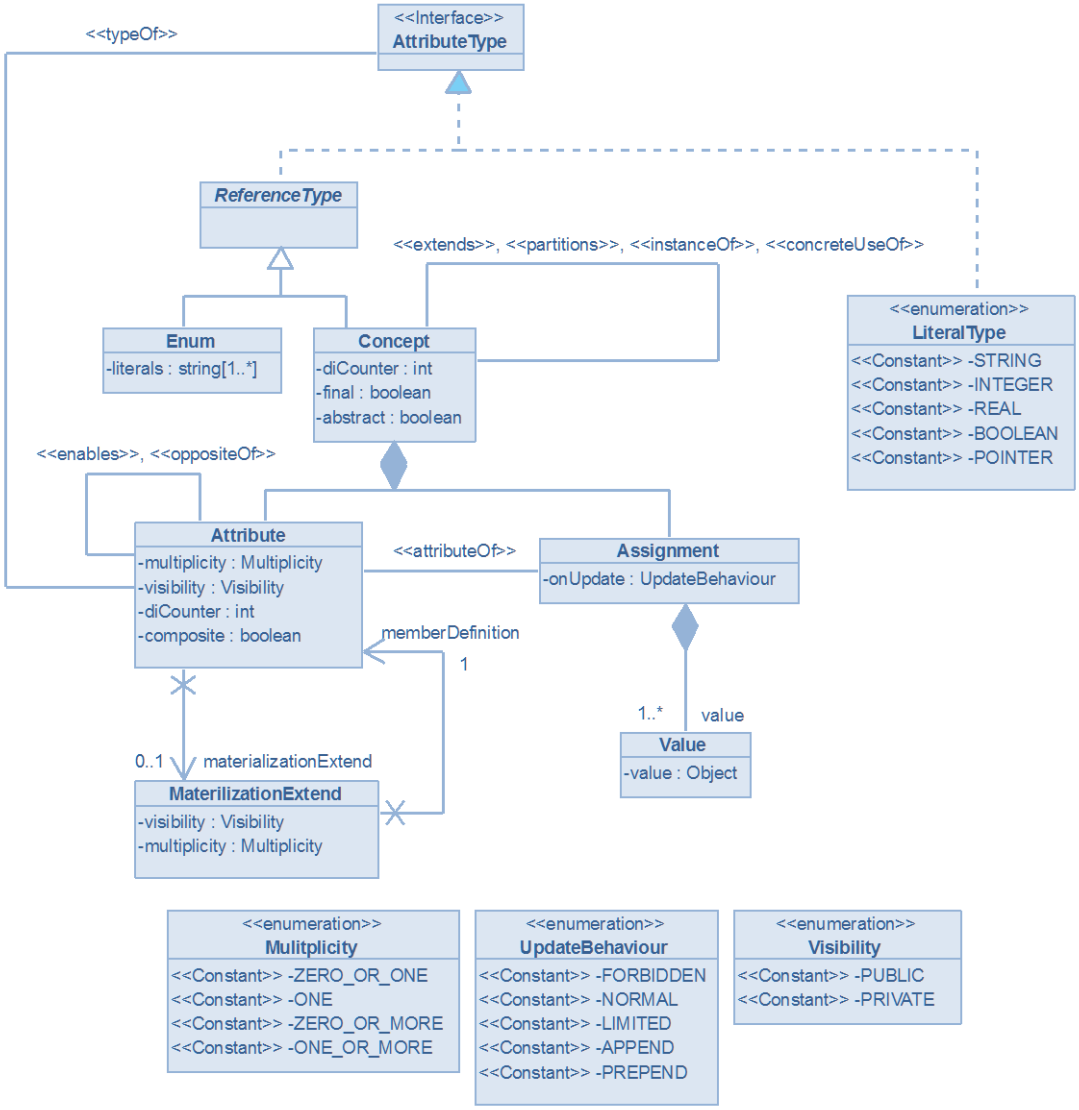

Abbildung 4-3 Ausschnitt aus der Containerhierarchie mit Konzepten, Enumerationen, Attributen und Zuweisungen

4.1.4 Attribute und Zuweisungen

Wie in Abbildung 4-3 dargestellt, werden Attribute und Zuweisungen an einem Konzept definiert. Attribute[12] besitzen immer einen Typ (`<<typeOf>>`), der literal (`LiteralType`) oder referentiell (`ReferenceType`) sein kann. Die möglichen literalen Typen `boolean` (Wahrheitswert), `integer` (Ganzzahl), `real` (Gleitkommazahl) und `string` (Zeichenkette) stehen für die üblichen elementaren Datentypen, wie sie auch häufig in modernen Programmiersprachen verwendet werden. Der literale Typ `pointer` repräsentiert beliebige interne oder externe Referenzen, die abgespeichert werden.

[12] In [117] besitzen Attribute eine weitere Eigenschaft (`final`). Diese verhindert das Überschreiben eines Wertes des Attributes durch eine Instanz-Spezialisierung. Da diese Entscheidung jedoch bereits an der Zuweisung am Prototyp getroffen wird und sie auf dieser Meta-Ebenen wenig sinnvoll erscheint, wurde hier die Eigenschaft nicht mit aufgenommen.

Zusätzlich zum Typ besitzen Attribute eine Multiplizität (`Multiplicity`) und eine Sichtbarkeit (`Visibility`). Mögliche Werte für die Sichtbarkeit[13] sind:

- `public`: das Attribut ist entsprechend überall dort sichtbar, wo nicht andere Regeln der Modellierungssemantik es verhindern.
- `private`: das Attribut ist nur innerhalb des umgebenden Konzeptes sichtbar und kann daher nur durch dessen Instanzen bestimmter Ordnung zugewiesen werden und ist z.B. in abgeleiteten Spezialisierungen nicht sichtbar und kann damit bei den jeweiligen Instanzen nicht gesetzt werden.

Die Multiplizität eines Attributes beeinflusst die Menge an Werten, die ihm zugewiesen werden kann. Die untere Grenze besagt, ob eine Zuweisung bei einem Konzept für das Attribut erstellt werden muss, falls das Attribut zuweisbar ist. Dabei steht der Wert `0` dafür, dass keine Zuweisung existieren muss und `1` für das Gegenteil. Attribute in der ersten Kategorie werden im Folgenden optional und diejenigen, die einen Wert zugewiesen bekommen müssen, obligatorisch genannt. Für die obere Grenze der Multiplizität gibt es ebenfalls zwei Werte, bei denen `1` dafür steht, dass das Attribut maximal einen Wert besitzen darf, während * definiert, dass mehrere Werte zugewiesen werden dürfen. Folglich werden Attribute mit maximal einem Wert einwertige Attribute und andere mehrwertige Attribute genannt. Aus den möglichen Kombinationen der beiden Grenzen ergibt sich die Menge {`0..1`, `1`, `0..*` und `1..*`} als Wertebereich der Multiplizität (siehe Enumeration `Multiplicity`).

Bidirektionale Beziehungen zwischen Konzepten lassen sich mit Hilfe der `<<oppositeOf>>` Beziehung zwischen den entsprechenden Attributen darstellen, während die `<<enables>>` Beziehung für die Deklaration eines Diskriminatorattributes eines erweiterten Powertyps steht (siehe Abschnitt 2.4). Zusätzlich besitzt jedes Attribut die Möglichkeit eine Materialisierungserweiterung (`MaterializationExtend`) zu definieren, wodurch das Sprachmuster der Materialisierung (Abschnitt 2.5) realisiert wird. Die Erweiterung legt dabei eine Attribut-Definition am gegenüberliegenden Konzept (`memberDefinition`) fest, deren Wert den Namen des materialisierten Attributes bestimmt. Die Multiplizität (`multiplicity`) und Sichtbarkeit (`visibility`) der materialisierten Attribute werden ebenfalls an der Materialisierungserweiterung definiert und besitzen die gleichen Wertebereiche wie die entsprechenden Eigenschaften des Modellelements. Jedes Attribut erlaubt die Definition eines Standardwertes, in der Form, dass eine Zuweisung an das Attribut am umgebenden Konzept des Attributes erstellt wird. Zuweisungen im Allgemeinen definieren für ein bestimmtes Attribut (`<<attributeOf>>`) Werte (`value`). Zusätzlich können sie das Überschreibungsverhalten für Instanz-Spezialisierungen (siehe Abschnitt 2.6) mit Hilfe der `onUpdate` Eigenschaft definieren.

4.2 Verwendete Begrifflichkeiten und Regeln

Im Folgenden finden sich wichtige Begriffe, die während der weiteren Erläuterungen häufig verwendet werden. Zunächst werden Mengen bzw. Relationen definiert, die die verschiedenen Beziehungen und Eigenschaften der Modellelemente formal festhalten. Daneben werden Konsistenzregeln definiert, die die Modellierungssemantik festlegen. Regeln, die in einer ähnlichen Form bereits in ([117], Kapitel 8) formuliert wurden, werden durch eine entsprechende Referenz auf den Abschnitt zusammen mit der dort verwendete Bezeichnung gekennzeichnet. Viele der Definitionen dieses Abschnitts finden sich in ähnlicher Form bereits im LMM. Dennoch wurden die Eigenschaften der Modellelemente, soweit sie

[13] Volz [117] definiert ähnliche Sichtbarkeitsstufen, wenngleich er dies unterschiedlich ausführt und teilweise eine Sichtbarkeit `protected` für Attribute verlangt ([117] Abschnitt 6.11 und Abschnitt 7.4.7).

für andere Definitionen von Bedeutung sind, als Relationen oder Mengen formal beschrieben. Sie dienen als Grundlage der Regeln und Begrifflichkeiten, die in den Kapiteln 5 und 6 später Anwendung finden.

Zunächst sollen wichtige Mengen definiert werden, die den Umgang mit den Modellelementen erleichtern:

- M als die Menge aller Modellelemente
- L als die Menge der Ebenen
- P als die Menge der Pakete
- R als die Menge der Referenztypen
- C als die Menge der Konzepte
- $T_{literal} = \{boolean,\ integer,\ string,\ real,\ pointer\}$ die Menge aller literalen Datentypen
- A als die Menge der Attribute
- V als die Menge der Zuweisungen
- W als die Menge aller Werte von Zuweisungen

Es gilt $C \subset R,\ C \subset W$ und $L,\ P,\ R,\ A,\ V \subset M$. Für die folgenden Regeln und Definitionen werden noch weitere Mengen benötigt, die Modelelemente mit einer speziellen Eigenschaft beinhalten:

- $Abstract = \{c \in C \mid c$ hat die `abstract` Eigenschaft auf `true` gesetzt$\}$ die Menge der abstrakten Konzepte
- $Final = \{c \in C \mid c$ hat die `final` Eigenschaft auf `true` gesetzt$\}$ die Menge der finalen Konzepte
- $Public = \{a \in A \mid a$ besitzt die Sichtbarkeitsstufe `public` $\}$ die Menge der öffentlichen Attribute
- $Private = \{a \in A \mid a$ besitzt die Sichtbarkeitsstufe `private`$\}$ die Menge der privaten Attribute

Um die verschiedenen Beziehungen der Modelelemente formal zu beschreiben, werden Relationen verwendet. Diese sind eine Teilmenge des kartesischen Produkts zweier Grundmengen G_1 und G_2, d.h. $R \subset G_1 \times G_2$. Aufgrund der einfachen Lesbarkeit wird für ein Tupel einer Relation auch die Infix-Notation verwendet, d.h. $(g_1, g_2) \in R \Leftrightarrow g_1\ R\ g_2$. Weiterhin soll eine abkürzende Schreibweise benutzt werden, wenn zwei Elemente über mehrere andere Elemente in Relation stehen. Seien $g_1,\ g_2\ \in G,\ n > 0\ und\ R \subset G \times G$ eine Relation, dann gilt

$$g_1\ R^n g_2 \Leftrightarrow \exists\ e_1, \ldots e_n\colon g_1 = e_1,\ g_2 = e_n \text{ und } e_1\ R\ e_2\ \ldots . e_{n-1}\ R\ e_n.$$

Um die nachfolgenden Relationen näher zu beschreiben, werden an dieser Stelle kurz einige wichtige Eigenschaften definiert: Sei $a,\ b,\ c \in G$ beliebig und $R \subset G \times G$ eine Relation. Dann heißt R

1. *asymmetrisch* $:\Leftrightarrow$ Falls $(a, b) \in R \Longrightarrow (b, a) \notin R$
2. *funktional*$:\Leftrightarrow$ Falls $(a, b) \in R \Longrightarrow \nexists\ d \neq b \in\ G\colon (a, d) \in R$
3. *azyklisch* $:\Leftrightarrow \forall\ n \in \mathbb{N}, e_1, \ldots e_n \in G$ mit $e_1 R\ e_2 R\ \ldots . R\ e_n$ gilt: $\forall i \neq j,\ i, j \in \{1..n\}\colon e_i \neq e_j$, d.h. die $e_i$sind paarweise verschieden.

4. *eineindeutig* $:\Leftrightarrow$ R und R^{-1} sind funktional, wobei R^{-1} die Umkehrrelation zu R ist. Für diese Relation gilt: $b\ R^{-1}\ a \Leftrightarrow a\ R\ b$.

Der Punkt 3 impliziert, dass R irreflexiv ($(a,a) \notin R$) ist. Als nächstes soll die Containerbeziehung der Modellelemente untereinander definiert werden.

Definition 4.1 *Containerbeziehung*

Seien $m_1, m_2 \in M,\ l \in L,\ r \in R$ und $In \subset M \times M,\ InLevel \subset R \times L$ zwei Relationen. Dann ist:

1. $m_1\ In\ m_2 :\Leftrightarrow m_2$ beinhaltet $m_1$im Sinne der Containerbeziehung
2. $r\ InLevel\ l :\Leftrightarrow \exists$ ein Paket $p \in P: r\ In\ p \wedge p\ In\ l$

4.2.1 Definitionen und Regeln in Bezug auf Ebenen

In dem nachfolgenden Abschnitt werden wichtige Definitionen und Regeln für Ebenen näher beschrieben. Die entsprechenden Eigenschaften einer Ebene wurde im Abschnitt 4.1.2 bereits vorgestellt. Als erstes wollen wir die Referenzen zwischen Ebenen charakterisieren.

Definition 4.2 *Logische Verschmelzung und Ebenen-Instanziierung*

Seien $l_1, l_2 \in L$ zwei Ebenen und $AlignedWith, LevelInstanceOf, InLevelRelation \subset L \times L$ drei Relationen. Dann ist

1. $(l_1, l_2) \in AlignedWith \Leftrightarrow\ l_1$ ist mit l_2 *logisch verschmolzen*: $\Leftrightarrow$ bei l_1 ist l_2 als Wert der `alignedWith` Eigenschaft gesetzt.
2. $(l_1, l_2) \in LevelInstanceOf \Leftrightarrow\ l_1$ *instanziiert* $l_2 :\Leftrightarrow l_1$ hat l_2 als Wert der `instanceOf` Eigenschaft gesetzt.
3. $(l_1, l_2) \in InLevelRelation :\Leftrightarrow\ (l_1, l_2) \in AlignedWith \vee\ (l_1, l_2) \in LevelInstanceOf$

Für diese Relationen sollen nun einige Regeln festgelegt werden, die die Relationen näher charakterisieren.

Regel E.1 Die Relation $InLevelRelation$ ist azyklisch. ([117], Abschnitt 8.4: LO.1)

Regel E.2 Die Relation $LevelinstanceOf$ ist asymmetrisch und funktional.

Regel E.3 Wenn $l_1, l_2 \in L$ mit $l_1\ AlignedWith\ l_2 \Longrightarrow\ (l_1, l_2) \notin InstanceOf$. ([117], Abschnitt 8.4: LO.2)

Regel E.4 Wenn $l_1, l_2 \in L$ mit $l_1\ InstanceOf\ l_2 \Longrightarrow\ (l_1, l_2) \notin AlignedWith$. ([117], Abschnitt 8.4: LO.2)

Bemerkung:

- Regel E.1 impliziert, dass die Relationen $AlignedWith$ und $LevelInstanceOf$ ebenfalls azyklisch sind: Angenommen es gibt ein Folge von Elementen $(e_i)_{i=1..n}, e_i \in L$ mit $e_1\ AlignedWith\ e_2 \ldots AlignedWith\ e_n$ und $e_n = e_1$, dann ist $\forall\ i = 1 \ldots n-1: (e_i, e_{i+1}) \in InLevelRelation$ und damit wäre $InLevelRelation$ zyklisch $\Longrightarrow$ Widerspruch. Für die Relation $LevelInstanceOf$ erfolgt ein Beweis analog.
- Regel E.3 und Regel E.4 besagen, dass eine Ebene entweder mit einer konkreten Ebene verschmolzen sein kann oder diese instanziiert. Beides ist ausgeschlossen.

Nachdem die grundlegenden Beziehungen zwischen Ebenen untereinander beschrieben wurden, wollen wir die Instanziierungsdistanz zweier Ebenen festlegen. Sie steht für die Anzahl der Instanziierungen, die auf dem Pfad zwischen zwei Ebenen abgelaufen werden müssen.

Definition 4.3 *Ebenen-Instanz n-ter Ordnung, Instanziierungsdistanz*

Seien $l_1, l_2 \in L$ zwei Ebenen. Falls $m \geq n \geq 0,\ e_1, \dots, e_m \in L,\ i_1, \dots i_n \in \{1 \dots m\}$ existieren, so dass gilt:

$\forall j = 1 \dots n: e_{i_{j-1}}\ LevelInstanceOf\ e_{i_j} \wedge \forall\ k\ \in \{1..m\}\ \setminus \{i_1, \dots i_n\} :\ e_{k-1} AlignedWith\ e_k.$

1. Dann heißt l_1 *Ebenen-Instanz n-ter Ordnung* von l_2 und n die *Instanziierungsdistanz* von l_1 zu l_2. In diesem Fall wird auch abkürzend die Notation $l_1\ LevelInstanceOf^n\ l_2$verwendet
2. Wenn $c_1, c_2 \in C$ zwei Konzepte mit $c_1\ InLevel\ l_1$ und $c_2\ Inlevel\ l_2$, dann hat c_1 die *Instanziierungsdistanz* n zu $c_2 :\Leftrightarrow l_1$ die Instanziierungsdistanz n zu l_2 besitzt.

Bemerkung:

- Anschaulich gesprochen steht eine Ebenen-Instanz n-ter Ordnung zu ihrer Ebene über genau n Instanziierungen und beliebig vielen Verschmelzungen in Relation. Wenn man die Relationen als Graph betrachtet, dann besitzt der Pfad also genau n Instanziierungen.
- Der zweite Punkt der Definition erleichtert den Umgang mit dem Begriff der Instanziierungsdistanz für Konzepte. Einfach gesagt beschreibt er, dass die Instanziierungsdistanz zweier Konzepte gleich der Instanziierungsdistanz ihrer umgebenden Ebenen ist. Dies ist wohldefiniert, da jedes Konzept in genau einem Paket liegt, das wiederum in genau einer Ebene definiert ist.

Die nun folgende Definition dient dazu, um eine Ebene zusammen mit den mit ihr verschmolzenen Ebenen uniform zu behandeln.

Definition 4.4 *Virtuelle Ebene*

Sei $l_1 \in L$ eine Ebene. Dann ist die *virtuelle Ebene* $VirtualLevel(l_1)$ von l_1 die transitive Hülle der $AlignedWith$ Relation bezüglich l_1, d.h.

$$VirtualLevel(l_1) = \{l_2 \in L |\ \exists n \geq 0: l_1\ AlignedWith^n\ l_2\} = \{l_2 \in L |\ l_1\ LevelInstanceOf^0\ l_2\}$$

Um mit dem Begriff der virtuellen Ebene einfach umzugehen, soll folgende Definition dienen:

Seien $c \in C$ ein Konzept und $l \in L$ eine Ebene. Dann ist $c\ InLevel\ VirtualLevel(l) :\Leftrightarrow \exists\ l_{virtual} \in VirtualLevel(l) \wedge c\ InLevel\ l_{virtual}$.

4.2.2 Definitionen und Regeln in Bezug auf Referenztypen und Konzepte

Dieser Abschnitt dient der Definition von Begrifflichkeiten und Regeln, die die Beziehungen zwischen Konzepten (bzw. Referenztypen) und deren Eigenschaften formal beschreiben. Dazu werden zunächst für alle im LMM vorhandenen Eigenschaften Relationen und Mengen definiert und darauf aufbauend weitere Begriffe zu definieren. Diese werden in den Operatoren der Kapitel 5 und 6 häufig verwendet und dienen einer einfachen Behandlung der beschriebenen Konzepte bzw. der entsprechenden Mengen der Konzepte, die die jeweilige Eigenschaft besitzen.

Als ersten sollen Beziehungen (Assoziationen) zwischen Konzepten definiert werden. Da diese durch referentielle Attribute abgebildet sind, müssen wir zunächst eine weitere Relation definieren. Diese

ordnet einem Attribut ihren Typ (z.B. ein Konzept) zu. Wenn klar ist, dass nicht Relationen zwischen Konzepten sondern die Assoziation gemeint ist, wird im Folgenden auch der Begriff Beziehung als Synonym verwendet.

Definition 4.5 *Attributtyp*

Seien $a \in A$ ein Attribut, $t \in R \cup T_{literal}$ ein Konzept und $IsOfType \in A \times (R \cup \mathrm{T_{literal}})$ eine Relation. Dann ist

$(a, t) \in IsOfType \Leftrightarrow t$ der Attributtyp von a ist $:\Leftrightarrow a$ hat t bei der `typeOf` Eigenschaft gesetzt.

Jetzt werden alle relevanten Relationen zwischen Konzepten beschrieben, die sich auch im LMM wiederfinden.

Definition 4.6 *Instanz, Typ-, Instanz-Spezialisierung, Powertyp, Assoziation*

Seien $c_1, c_2 \in C$ zwei Konzepte und $InstanceOf, Extends, ConcreteUseOf, Partitions, AssociatedWith, InRelation \subset C \times C$ Relationen. Dann ist

1. $(c_1, c_2) \in InstanceOf \Leftrightarrow c_1$ ist eine *Instanz* von $c_2 \Leftrightarrow c_2$ ist der *instanziierte Typ* von $c_1 : \Leftrightarrow c_1$ hat c_2 als Wert der `instanceOf` Eigenschaft gesetzt.
2. $(c_1, c_2) \in Extends \Leftrightarrow c_1$ ist eine *(Typ-)Spezialisierung* von $c_2 \Leftrightarrow c_2$ ist die *Generalisierung* von $c_1 :\Leftrightarrow c_1$ hat c_2 als Wert der `extends` Eigenschaft gesetzt.
3. $(c_1, c_2) \in ConcreteUseOf \Leftrightarrow c_1$ ist eine *Instanz-Spezialisierung* von $c_2 \Leftrightarrow c_2$ ist der *Prototyp* von $c_1 :\Leftrightarrow c_1$ hat c_2 als Wert der `concreteUseOf` Eigenschaft gesetzt.
4. $(c_1, c_2) \in Partitions \Leftrightarrow c_1$ ist der *Powertyp* von $c_2 \Leftrightarrow c_2$ ist der *partitionierte Typ* von $c_1 : \Leftrightarrow c_1$ hat c_2 als Wert der `partitions` Eigenschaft gesetzt.
5. $(c_1, c_2) \in AssociatedWith \Leftrightarrow c_1$ definiert eine *Assoziation (Beziehung)* zu $c_2 : \Leftrightarrow \exists$ ein Attribut $a \in A: a\ In\ c_1 \wedge a\ IsOfType\ c_2$.
6. $(c_1, c_2) \in InRelation :\Leftrightarrow (c_1, c_2) \in InstanceOf \vee (c_1, c_2) \in Extends \vee (c_1, c_2) \in ConcreteUseOf \vee (c_1, c_2) \in Partitions$

Für diese Relationen sollen nun Regeln festgelegt werden, die wichtige Eigenschaften der Relationen sowie den Zusammenhang mit finalen oder abstrakten Konzepten beschreiben.

Regel C.1 Die Relation $InRelation$ ist azyklisch. ([117], Abschnitt 8.6: ER. 2)

Regel C.2 Die in Definition 4.6 unter 1.- 4. vorgestellten Relationen sind asymmetrisch, intransitiv und funktional. ([117], Abschnitt 8.6: ER.3, ER.8)

Regel C.3 Die Relation $Partitions$ ist sogar eineindeutig. ([117], Abschnitt 8.6: ER. 9)

Regel C.4 Sei $c \in Abstract$ ein abstraktes Konzept $\Longrightarrow \nexists$ Konzept $c_{inst} \in C: c_{inst}\ InstanceOf\ c$ ([117], Abschnitt 8.6: ER.5) $\wedge \forall\ a \in A$ mit $a\ In\ c$ gilt: $a \in Public$ ([117], Abschnitt 8.7.2 AZ.2)

Regel C.5 Sei $c \in Final$ ein finales Konzept $\Longrightarrow \nexists$ Konzept $c_{ext} \in C: c_{ext}\ Extends\ c \vee c_{ext}\ Partitions\ c \vee c_{ext}\ ConcreteUseOf\ c$

Regel C.6 $\forall\ c \in Final \Longrightarrow c \notin Abstract$

Regel C.7 $\forall c \in Abstract \Longrightarrow c \notin Final$

Bemerkung:

- Regel C.1 impliziert, dass die Relationen $InstanceOf$, $Extends$, $ConcreteUseOf$ und $Partitions$ ebenfalls azyklisch sind. Der Beweis erfolgt analog zum Beweis in der Bemerkung zur Regel E.1.
- In Regel C.2 ist enthalten, dass jedes Konzept höchstens eine Generalisierung, einen Prototyp, einen instanziierten Typ sowie einen partitionierten Typ besitzt (Eigenschaft funktional der Relationen). Dadurch sind ähnliche Probleme [104], die bei einer Mehrfachvererbung auftreten können, ausgeschlossen.
- Regel C.3 legt fest, dass jeder partitionierte Typ auch höchstens einen (direkten) Powertyp besitzen kann.
- Die Regel C.5 beinhaltet ER.4 aus ([117], Abschnitt 8.6), erweitert diese jedoch um die Bedingung, dass ein finales Konzept auch nicht partitioniert werden darf. Dies ist insofern sinnvoll, da durch eine Partitionierung eine indirekte Spezialisierung der Powertyp-Instanzen entsteht.
- Die Eigenschaften abstrakt und final schließen sich gegenseitig aus, da ein Konzept, das sowohl abstrakt als auch final ist, weder Instanzen, Spezialisierungen, Instanz-Spezialisierung noch einen Powertyp haben kann. (Regel C.6 bzw. Regel C.7)

Seien $c_1, c_2 \in C$ zwei Konzepte.

Regel C.8 Wenn $c_1 InstanceOf\ c_2 \Longrightarrow \nexists$Konzept $c_3 \in C$: $c_1\ Extends\ c_3 \vee c_1\ ConcreteUseOf\ c_3$ ([117], Abschnitt 8.6: ER.6)

Regel C.9 Wenn $c_1 Extends\ c_2 \Longrightarrow \nexists$ Konzept $c_3 \in C$: $c_1\ InstanceOf\ c_3 \vee c_1\ Partitions\ c_3$

Regel C.10 Wenn $c_1 ConcreteUseOf\ c_2 \Longrightarrow \nexists$ Konzept $c_3 \in C$: $c_1\ InstanceOf\ c_3$ ([117], Abschnitt 8.6: ER.7)

Regel C.11 Wenn $c_1 Partitions\ c_2 \Longrightarrow \nexists$ Konzept $c_3 \in C$: $c_1\ Extends\ c_3$

Bemerkung:

- Regel C.9 enthält einen Teil der Bedingung ER.7 ([117], Abschnitt 8.6). Jedoch wurde neben einer Instanziierung auch eine Partitionierung für ein spezialisiertes Konzept verboten, da dies u.a. zu einer (impliziten) Mehrfachvererbung (siehe nachfolgendes Beispiel) oder zu einer Spezialisierung des partitionierten Typs durch den Powertyp führen kann. Analog gilt dies ebenfalls für Regel C.11.

Tabelle 4-1 Übersicht über mögliche Kombinationen von Relationen eines Konzepts

	InstanceOf	Extends	ConcreteUseOf	Partitions
InstanceOf	✗	✗	✗	✓
Extends	✗	✗	✓	✗
ConcreteUseOf	✗	✓	✗	✓
Partitions	✓	✗	✓	✗

Gegenbeispiel:

In diesem Beispiel soll gezeigt werden, wie durch eine Partitionierung zusammen mit einer gleichzeitigen Spezialisierung Mehrfachvererbung entstehen kann.

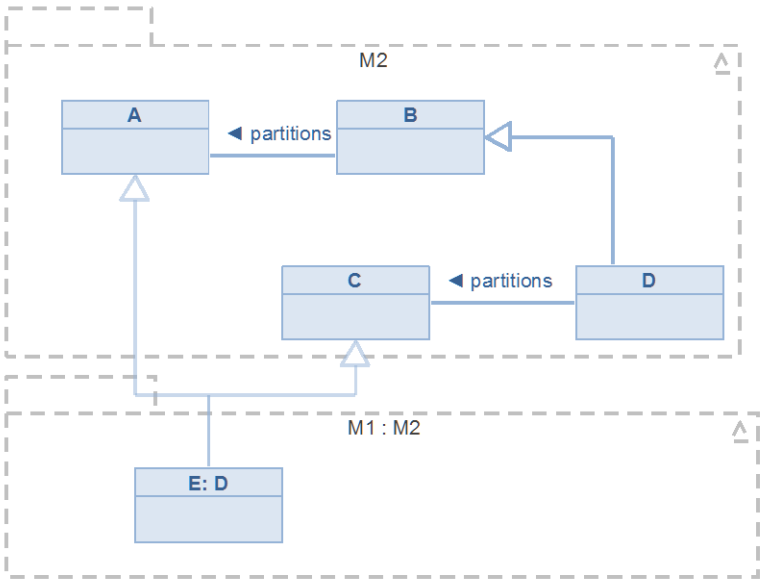

Abbildung 4-4 Beispiel einer Mehrfachvererbung durch Partitionierung und Spezialisierung

Abbildung 4-4 zeigt in der Ebene `M2` vier Konzepte, von denen `A` der partitionierte Typ zu `B` und `C` der partitionierte Typ zu `D` ist. Weiterhin ist `D` eine Spezialisierung von `B` und verstößt damit gegen Regel C.11 bzw. Regel C.9. Für eine Instanz `E` von `D` existiert implizit eine Spezialisierung zu den jeweiligen partitionierten Typen `A` bzw. `C`, die auch im Diagramm visualisiert ist. Dadurch entsteht eine Mehrfachvererbung, die zu bekannten Problemen führen kann [104].

Als nächstes werden nützliche Begriffe im Umgang mit Hierarchien von Partitionierungen und Spezialisierungen formal eingeführt.

Definition 4.7 *Abgeleiteter Powertyp, abgeleitete Spezialisierung, abgeleitete Generalisierung*

Seien $c_1, c_2 \in C$ zwei Konzepte. Dann ist

1. c_1 eine *abgeleitete Spezialisierung* von $c_2 \Leftrightarrow c_1\ Extends^* c_2 \Leftrightarrow\ c_2$ ist eine *abgeleitete Generalisierung* von $c_1 :\Leftrightarrow \exists\ n > 0,\ e_1, \ldots, e_n \in C$ mit $e_1 = c_1$ und $e_n = c_2$, so dass $\forall\ i = 1..n$ gilt: $e_{i-1}\ Extends\ e_i \vee (\exists$ ein Powertyp $p \in C: e_{i-1}\ InstanceOf\ p\ \wedge\ p\ Partitions\ e_i)$
2. c_1 eine *abgeleiteter Powertyp* von $c_2 \Leftrightarrow c_1\ Partitions^*\ c_2 :\Leftrightarrow \exists\ p \in C: p\ Partitions\ c_2 \wedge c_1\ Extends^*\ p$

Ein Konzept heißt *echte abgeleitete Generalisierung, abgeleitete Spezialisierung oder echter abgeleiteter Powertyp*, falls $c_1 \neq c_2$ gilt.

Bemerkung:

- Die Relation $Extends^*$bildet eine Halbordnung, da sie reflexiv, antisymmetrisch und transitiv ist.
- Die Bedingung $e_{i-1}\ Extends\ e_i \vee (\exists\ p \in C: e_{i-1}\ InstanceOf\ p\ \wedge\ p\ Partitions\ e_i)$ besagt, dass e_{i-1} eine Spezialisierung oder eine Instanz des Powertyps von e_i ist.

Die nächste Definition beschreibt zwei Funktionen, die als Hilfsmittel dienen, den Umgang mit Deep Instantiation Zählern von Konzepten und Attributen formal zu beschreiben. Mit Hilfe dieser Definition, kann anschließend festgelegt werden, wann ein Konzept in einer Ebene instanziierbar ist.

Definition 4.8 *Deep Instantiation Zähler*

Seien $a \in A$ ein Attribut und $c \in C$ ein Konzept. Dann ist

1. $di_{concept}: C \rightarrow \mathbb{N}^+$ die Funktion, die jedem Konzept seinen Deep Instantiation Zähler zuordnet.
2. $di_{attribute}: A \rightarrow \mathbb{N}^+$ die Funktion, die jedem Attribut seinen Deep Instantiation Zähler zuordnet.

Da die Funktion $di_{attribute}(a)$ häufig im Kontext der Instanzen n-ter Ordnung auftritt, wird sie auch häufig mit $n(a)$ bezeichnet.

Definition 4.9 *Instanziierbarkeit bezüglich einer Ebene*

Sei $c \in C$ ein Konzept und $l_1, l_2 \in L$ zwei Ebenen, so dass $c\ InLevel\ l_1$.

Dann heißt c *instanziierbar in* $l_2 :\Leftrightarrow l_2$ ist ein Ebenen-Instanz $di_{concept}(c)$-ter Ordnung von l_1

Durch die nachfolgenden Regeln werden alle Kandidaten für die jeweiligen Relationen der Konzepte festgelegt.

Seien $c_1, c_2, c_3 \in C$ drei Konzepte und $l_1, l_2 \in L$ zwei Ebenen mit $c_1\ InLevel\ l_1$, $c_2\ InLevel\ l_2$.
Weiterhin sei $d = di_{concept}(c_2)$.

Regel C.12 Wenn $c_1\ InstanceOf\ c_2 \Longrightarrow l_1 LevelInstanceOf^d\ l_2 \wedge c_2 \notin Abstract$. ([117], Abschnitt 8.5: LR.2, Abschnitt 8.6.4, Abschnitt 8.8.2: DI.3, Abschnitt 8.6: ER.1)

Regel C.13 Wenn $c_1 AssociatedWith\ c_2 \Longrightarrow \exists\ n < d \in \mathbb{N}_0$: $l_1\ LevelInstanceOf^n l_2$ ([117], Abschnitt 8.7.3)

Regel C.14 Wenn $c_1 Extends\ c_2 \Longrightarrow c_2 \notin Final \wedge \exists\ n < d \in \mathbb{N}_0: l_1\ LevelInstanceOf^n l_2$ ([117], Abschnitt 8.5: LR.3, Abschnitt 8.6.2)

Regel C.15 Wenn $c_1\ ConcreteUseOf\ c_2 \Longrightarrow c_2 \notin Final \wedge \exists\ n < d \in \mathbb{N}_0: l_1\ LevelInstanceOf^n l_2\ \wedge$ $\nexists$ keine Konzepte $p, c_3 \in C$: $c_2\ InstanceOf\ p\ Partitions^*\ c_3$ ([117], Abschnitt 8.5: LR.3, Abschnitt 8.6.4, Abschnitt 8.8.3: IS.2)

Regel C.16 Wenn $c_1 Partitions\ c_2 \Longrightarrow\ c_2 \notin Final \wedge \nexists$ Powertyp $p \neq c_1 \in C: p\ Partitions\ c_2 \wedge$ $\exists\ n \in \mathbb{N}_0: l_1\ LevelinstanceOf^n l_2$([117], Abschnitt 8.5: LR.3, Abschnitt 8.6.3)

Bemerkung:

- Regel C.12 besagt, dass jedes nicht abstrakte, instanziierbare Konzept einer Ebene, die durch die umgebende Ebene des Konzepts (nicht zwangsweise direkt) instanziiert wird, ein möglicher instanziierter Typ ist.
- Mögliche Konzepte, zu denen Assoziationen erstellt werden können oder die ein entsprechender Attributtyp sein können, besitzen eine Deep Instantiation Zähler, dessen Wert größer als die Instanziierungsdistanz der entsprechenden Ebene ist. (Regel C.13) Für den Fall $n = 0$ liegen diese Elemente in der gleichen virtuellen Ebene wie das Konzept c_1.
- Generalisierungen sind zusätzlich noch nicht final. (Regel C.14)

- Prototypen sind weiterhin auch noch keine Powertyp-Instanz. (Regel C.15)
- Für potentielle partitionierte Typen gelten die beiden Bedingungen für Generalisierungen ebenfalls, nur, dass alle Konzepte, die bereits einen Powertyp besitzen, nicht in dieser Menge enthalten sind. (Regel C.16)

Definition 4.10 *Instanzen n-ter Ordnung und abgeleiteter instanziierter Typ*

Seien $c_1, c_2 \in C$ zwei Konzepte und $n \in \mathbb{N}_0$ eine positive natürliche Zahl.

Dann ist c_1 eine *Instanz n-ter Ordnung* von $c_2 \iff c_1\ InstanceOf^n\ c_2 :\Leftrightarrow \exists\ m > n \geq 0,\ e_1, \dots e_m \in C$ mit $e_1 = c_1$ und $e_m = c_2$, so dass eine der folgenden Bedingungen gilt:

1. $n \geq 1$ und $\exists\ i_1, \dots, i_n \in \{1 \dots m\}, i_n = m$: $(\forall\ j = 1 \dots n$: $e_{i_j-1}\ InstanceOf\ e_{i_j}) \wedge (\forall k \in \{1..m\} \setminus \{i_1, \dots i_n\}$: $e_{k-1} ConcreteUseOf\ e_k)$
2. $n = 0, m > 1$ und $\forall\ j = 2 \dots m$: $e_{j-1}\ ConcreteUseOf\ e_j$
3. $n = 0$ und $m = 1$

Im Falle $n = 1$ heißt c_2 *abgeleiteter instanziierter* Typ von c_1.

Bemerkung:

- Punkt 1 in Definition 4.10 verlangt, dass jede Instanz n-ter ($n \geq 1$) Ordnung eines Konzeptes mit einer direkten Instanz in Relation steht. Anschließend unterliegt die Anordnung der Kanten im Pfad keinerlei Beschränkung, bis auf die Tatsache, dass genau n Instanziierungen vorliegen müssen.
- Punkt 3 in Definition 4.10 drückt aus, dass jedes Konzept Instanz 0-ter Ordnung von sich selbst ist.

Beispiel:

Seien $c_1, c_2, c_3 \in C$ drei Konzepte.

- Wenn c_2 eine Instanz-Spezialisierung von c_1 ist, dann sind c_1 und c_2 Instanzen 0-ter Ordnung von c_1.
- Wenn $c_3\ ConcreteUseOf\ c_2\ InstanceOf\ c_1$ gilt, dann sind c_2 und c_3 Instanzen 1-ter Ordnung von c_1.

Nun kann der Begriff der Ebenen-Abhängigkeit beschrieben werden, der wichtig ist, um ein Konzept innerhalb der Hierarchie der Ebenen zu verschieben (siehe Abschnitt 6.1.1 oder Abschnitt 6.1.3). Je nachdem wohin ein Konzept verschoben werden soll, sind grob gesagt die abwärts ebenen-abhängigen oder die aufwärts ebenen-abhängigen Referenztypen von Bedeutung. Anschaulich gesprochen beinhalten die beiden Mengen also alle Referenztypen, die bei einer Verschiebung mit verschoben werden müssen.

Definition 4.11 *Ebenen-Abhängigkeit*

Seien $r_1, r_2 \in R$ zwei Referenztypen und $l_1, l_2 \in L$ zwei Ebenen mit $r_1\ InLevel\ l_1$ und $r_2\ InLevel\ l_2$. Dann ist

r_1 *aufwärts ebenen-abhängig* zu $r_2 \Leftrightarrow r_2$ ist *abwärts ebenen-abhängig* zu $r_1 :\Leftrightarrow r_2 \in C$, $l_1 VirtualLevel(l_2)$ und eine der folgenden Aussagen zutrifft:

1. $\exists\ a \in A: a\ In\ r_2 \wedge a\ IsOfType\ r_1$
2. $r_1 \in C \wedge r_2\ Extends\ r_1$
3. $r_1 \in C \wedge r_2\ Partitions\ r_1$
4. $r_1 \in C \wedge r_2\ ConcreteUseOf\ r_1$

4.2.3 Definitionen und Regeln in Bezug auf Attribute

In diesen Abschnitt sollen wichtige Definitionen und Regeln für Attribute vorgestellt werden. Dazu werden zu Beginn die bereits im LMM formulierten Eigenschaften eines Attributes als Relationen oder Mengen beschrieben, bevor die im weiteren Verlauf der Arbeit verwendeten Begrifflichkeiten definiert werden, die auf diese Begriffe aufbauen. Die Definition des Attributtyps und der jeweiligen Kandidaten dafür, findet sich in Definition 4.5 bzw. Regel C.13. Die Sichtbarkeit eines Attributes wurde durch die Mengen $Public$ und $Private$ bereits definiert. Nun soll zunächst die Multiplizität und anschließend die Begriffe des optionalen, obligatorischen, einwertigen und mehrwertigen Attribut beschrieben werden.

Definition 4.12 *Multiplizität*

Seien $a \in A$ ein Attribut, $Multiplicity = \{1, 0..1, 0..*, 1..*\}$ die Menge an möglichen Werten der Multiplizität eines Attributes, $m \in Multiplicity$ eine Multiplizität und $HasMultiplicity \subset A \times Multiplicity$ eine Relation.

Dann ist $(a, m) \in HasMultiplicity :\Leftrightarrow a$ hat m als Wert der `multiplicity` Eigenschaft gesetzt.

Definition 4.13 *Optionale, obligatorische, einwertige und mehrwertige Attribute*

Seien $a,\ m,\ HasMultiplicity$ und $Multiplicity$ wie in Definition 4.12.

Dann heißt a

1. *optional* $:\Leftrightarrow a\ HasMultiplicity\ 0..1 \vee a\ HasMultiplicity\ 0..*$
2. *obligatorisch* $:\Leftrightarrow a\ HasMultiplicity\ 1 \vee a\ HasMultiplicity\ 1..*$
3. *einwertig* $:\Leftrightarrow a\ HasMultiplicity\ 1 \vee a\ HasMultiplicity\ 0..1$
4. *mehrwertig* $:\Leftrightarrow a\ HasMultiplicity\ 0..* \vee a\ HasMultiplicity\ 1..*$

Regel A.1 Die Relation $HasMultiplicity$ ist funktional.

Bemerkung:

- Weitere Regeln zur Multiplizität finden sich in Abschnitt 4.2.4.

Als Grundlage für weitere Definition soll als nächstes gezeigt werden, wie der Begriff des materialisierten Attributs formal greifbar wird.

Definition 4.14 *Attribut-Definition einer Materialisierungserweiterung*

Seien $a_{mat}, a_{definition} \in A$ zwei Attribute und $DefinesNameWith \subset A \times A$ eine Relation.

Dann ist $(a_{mat}, a_{definition}) \in DefinesNameWith :\Leftrightarrow$
a_{mat} besitzt eine Materialisierungserweiterung und diese hat $a_{definition}$ als Wert der `memberDefinition` Eigenschaft gesetzt.

In diesem Fall heißt $a_{definition}$ eine Attribut-Definition oder die Attribut-Definition von a_{mat}.

Regel A.2 Seien a_{mat}, $a_{definition}$ wie in Definition 4.14. Wenn
$a_{mat}\ DefinesNameWith\ a_{definition} \Longrightarrow$ (∃ ein Konzept $c \in C: a_{mat}\ IsOfType\ c) \wedge$
$n(a_{mat}) = n(a_{definition})$

Regel A.3 Sei $a_{definition}$ eine Attribut-Definition $\Longrightarrow\ a_{definition}\ IsOfType\ string \wedge$
$a_{definition}\ HasMultiplicity\ 1$

Bemerkung:

- Regel A.2 beschreibt, dass jedes Attribut mit einer Materialisierungserweiterung ein Konzept als Typ besitzen muss. Weiterhin muss der Deep Instantiation Zähler dieses Attributes mit dem der Attribut-Definition übereinstimmen.
- Durch Regel A.3 wird festgelegt, dass eine Attribut-Definition vom Typ `string` sein muss und eine Multiplizität von `1` besitzt. Dies ist notwendig, da jedes materialisierte Attribut genau einen Namen besitzen soll.

Definition 4.15 *Materialisierte Attribute eines Konzepts*

Seien $a, a_{definition} \in A$ zwei Attribute mit $a\ DefinesNameWith\ a_{definition}$, $c \in C$ ein Konzept, $name \in W$ ein Zuweisungswert und $MaterializedIn \subset A \times C$ eine Relation.

Dann ist $(a, c) \in\ MaterializedIn\ :\Leftrightarrow$ ∃ zwei Zuweisungen $v, v_{definition} \in V$ und ein Konzept $c_{definition} \in C$ so, dass folgende Bedingungen gelten:

1. $v\ In\ c \wedge\ v\ ToAttribute\ a \wedge v\ HasValue\ c_{definition}$
2. $v_{definition}\ ToAttribute\ a_{definition} \wedge v_{defintion}\ In\ c_{definition}$
3. $v_{definiton}\ HasValue\ name$, wobei $name$ der Name von a ist.

Bemerkung

- Ein Attribut ist dann an einem Konzept materialisiert, wenn es eine entsprechende Zuweisung für ein Attribut mit Materialisierungserweiterung an diesem Konzept gibt. Der Wert dieser Zuweisung muss ein Konzept sein, das eine Zuweisung an die Attribut-Definition mit dem Namen des Attributes als Wert definiert.

In der folgenden Definition sollen die speziellen Attribute eines erweiterten Powertyps beschrieben werden.

Definition 4.16 *Diskriminatorattribut*

Seien $p, c \in C$ zwei Konzepte und $a_p, a_c \in A$ zwei Attribute mit $a_p \; In \; p$ und $a_c \; VirtualIn \; C$. Weiterhin sei $Enables \subset A \times A$ eine Relation.

Dann ist $(a_p, a_c) \in Enables :\Leftrightarrow$ $a_p$hat a_c als Wert der `enables` Eigenschaft gesetzt.

Seien p, c, a_p und a_c wie in Definition 4.16.

Regel A.4 Wenn $a_p \; Enables \; a_c \Longrightarrow p \; Partitions \; c$ ([117], Abschnitt 8.8, PT.1)

Regel A.5 Wenn $a_p \; Enables \; a_c \Longrightarrow n(a_p) = 1 \wedge \; a_p \; IsOfType \; boolean \; \wedge a_p \in Public \wedge$ $a \; HasMultiplicity \; 1$

Regel A.6 Wenn $\exists \; a_p \; In \; P$: $a_p \; Enables \; a_c \Longrightarrow \forall$ Attribute $a_1 \in A$ mit $a_1 \; VirtualIn \; c$ gilt: $\exists$ ein Attribut $a_2 \in A$ mit $\; a_2 \; In \; P$ und $a_2 \; Enables \; a_1$

Bemerkung:

- Durch Regel A.5 wird ausgedrückt, dass jedes Diskriminatorattribut den Typ `boolean` besitzt, öffentlich ist, genau einen Wert besitzen muss und einen Deep Instantiation Zähler von `1` hat. Dies ist deshalb der Fall, damit bei allen direkten Instanzen des Powertyps ein Wert vorliegt, der festlegt, ob ein bestimmtes Attribut des partitionierten Typs geerbt wird.
- Regel A.6 besagt, dass ein erweiterter Powertyp für jedes virtuelle Attribut des partitionierten Typs ein Diskriminatorattribut definieren muss. Dadurch wird die Menge der Diskriminatorattribute bei einer Spezialisierung des Powertyps nicht größer. Dies ist für eine uniforme Behandlung aller Attribute am partitionierten Typ sinnvoll.

Um bidirektionale Beziehungen abzubilden, wird im Modell die `oppositeOf` Eigenschaft der jeweiligen Attribute verwendet. Diese soll nun formal definiert werden.

Definition 4.17 *Gegenüberliegendes Attribut*

Seien $a_1, a_2 \in A$ zwei Attribute und $OppositeOf \subset A \times A$ eine Relation.

Dann ist $(a_1, a_2) \in OppositeOf :\Leftrightarrow$ $a_1$hat a_2 als Wert der `oppositeOf` Eigenschaft gesetzt oder a_2 hat a_1 als Wert der `oppositeOf` Eigenschaft gesetzt.

Regel A.7 Offensichtlich ist die Relation $OppositeOf$ symmetrisch. Weiterhin ist sie funktional.

Regel A.8 Seien $a_1, a_2 \in A$ zwei Attribute mit $a_1 \; OppositeOf \; a_2 \Longrightarrow \exists$ zwei Konzepte $c_1, c_2 \in C$: $a_1 \; IsOfType \; c_1 \wedge a_2 \; IsOfType \; c_2 \wedge a_1 \; In \; c_2 \wedge a_2 In \; c_2$.

Regel A.9 Seien $a_1, a_2 \in A$ zwei Attribute mit $a_1 \; OppositeOf \; a_2 \Longrightarrow n(a_1) = n(a_2)$

Bemerkung:

- Die Regel A.8 drückt aus, dass jedes gegenüberliegende Attribut ein Konzept als Typ besitzt und an diesem das entsprechende andere Attribut definiert ist.

- Um die Zuweisungen der jeweiligen Instanzen der gegenüberliegenden Attribute setzen zu können, müssen beide Attribute einer bidirektionalen Beziehung den gleichen Deep Instantiation Zähler haben. Regel A.8 trägt dieser Tatsache Rechnung.

Um in den Operatoren einfach festzustellen, welche Instanzen eines Konzeptes ein bestimmtes Attribut setzen müssen bzw. gesetzt haben (können), ist die folgende Definition des Deep Instantiation Zählers eines Attributes bezüglich eines Konzepts hilfreich.

Definition 4.18 *Relativer Deep Instantiation Zähler eines Attributes bezüglich eines Konzeptes*

Seien $a \in A$ ein Attribut und $c_1, c_2 \in C$ zwei Konzepte.

Wenn ein Konzept $c_3 \in C$ und eine natürliche Zahl $m \in \mathbb{N}$ existiert mit $c_2\ InstanceOf^m\ c_3 \wedge c_3\ Extends^*\ c_1 \wedge (a\ In\ c_1 \vee a\ IsOfType\ c_1)$, dann ist

$$n_{rel}(a, c_2) = n(a) - m$$ der *relative Deep Instantiation Zähler* von a bezüglich c_2.

Wenn der Kontext (Konzept c_2) ersichtlich ist, wird auch statt $n_{rel}(a, c_2)$ nur $n(a)$ geschrieben.

Bemerkung:

- Wenn ein Konzept $c_1 \in C$ ein Attribut definiert oder Typ eines Attributes ist, dann ist der Deep Instantiation Zähler bezüglich jeder abgeleiteten Spezialisierung von c_1 gleich dem Deep Instantiation Zähler des Attributes.
- Die Bedingung $a\ In\ c_1$ ist immer im Kontext des Setzens von Attributen von Bedeutung, während die Bedingung $a\ IsOfType\ c_1$ für die Zuweisung von Instanzen an ein Attribut wichtig ist.

Definition 4.19 *Sichtbare Attribute einer echten abgeleiteten Generalisierung*

Seien $a \in Public$ ein Attribut, $c, g \in C$ zwei Konzepte mit $c \neq g \wedge c\ Extends^* g$ und $VisibleIn \subset A \times C$ eine Relation.

Dann ist a ein *sichtbares Attribut* von $c \Leftrightarrow a\ VisibleIn\ c :\Leftrightarrow$ falls eine der folgenden Bedingungen zutrifft

1. $\exists$ Diskriminatorattribut $d \in A$ und eine Zuweisung $v \in V$ mit: $d\ enables\ a \wedge v\ ToAttribute\ d\ \wedge v\ VirtualAssignedAt\ c \wedge v\ HasValue\ true$.
2. $\nexists$ Diskriminatorattribut $d \in A$ mit $d\ enables\ a$

Bemerkung:

- Die Definitionen zu den Relationen $ToAttribute$, $VirtualAssignedAt$ und $HasValue$ befinden sich im Abschnitt 4.2.4.
- Ein Attribut ist in einer echten abgeleiteten Spezialisierung dann sichtbar, wenn es öffentlich ist und das Diskriminatorattribut am Konzept virtuell `true` zugewiesen bekommt, falls das Diskriminatorattribut existiert.

Nachfolgend werden verschiedene Attributtypen für ein Konzept definiert, die die Sprachmuster auf Attribut-Ebene auflösen und damit bestimmen, welche Attribute an einem Konzept virtuell, instanziiert oder zuweisbar sind.

Definition 4.20 *Virtuelle Attribute eines Konzepts*

Seien $a \in A$ ein Attribut, $c \in C$ ein Konzept und $VirtualIn \subset A \times C$ eine Relation.

Dann ist a ein *virtuelles Attribut* von $c \Leftrightarrow a\ VirtualIn\ c\ :\Leftrightarrow$ eine der folgenden Bedingungen zutrifft

1. $a\ In\ c$ (direkt definiert)
2. $a\ MaterializedIn\ c$ (materialisiertes Attribut)
3. ∃ ein Konzept $g \in C, g \neq c$ mit $c\ Extends^*\ g \wedge (a\ In\ g \vee a\ MaterializedIn\ g) \wedge a\ VisibleIn\ c$ (sichtbares Attribut einer echten abgeleiteten Generalisierung)
4. ∃ ein Konzept $t \in C$ mit $c\ InstanceOf\ t \wedge a\ VirtualIn\ t \wedge n_{rel}(a, c) \geq 1$ (virtuelles Attribut des instanziierten Typs mit relativen Deep Instantiation Zähler größer gleich eins)

Bemerkung:

- Die Definition ist wohldefiniert, da $InstanceOf$ azyklisch ist und damit die Rekursion im Punkt 4 enden muss.
- Virtuelle und damit insbesondere materialisierte Attribute sind bei Instanz-Spezialisierungen nicht sichtbar.

Definition 4.21 *Instanziierte Attribute eines Konzeptes*

Seien $a \in A$ ein Attribut, $c \in C$ ein Konzept und $InstantiatedIn \subset A \times C$ eine Relation.

Dann ist a ein *instanziiertes Attribut* von $c \Leftrightarrow a\ InstantiatedIn\ c\ :\Leftrightarrow$ eine der folgenden Bedingungen zutrifft

1. ∃ ein Konzept $t \in C$ mit $c\ InstanceOf\ t \wedge a\ VirtualIn\ t$ (virtuelles Attribut des instanziierten Typs)
2. ∃ ein Konzept $t \in C$ mit $c\ InstanceOf\ t \wedge a\ InstantiatedIn\ t \wedge\ n_{rel}(a, c) \geq 0$ (instanziiertes Attribut des instanziierten Typs mit relativen Deep Instantiation Zähler größer gleich null)
3. ∃ ein Konzept $p \in C$ mit $c\ ConcreteUseOf\ p \wedge\ a\ InstantiatedIn\ p$ (instanziiertes Attribut des Prototyps)

Mit Hilfe der obigen Begrifflichkeit kann nun die Menge der zuweisbaren Attribute eines Konzepts definiert werden, die festlegt, welche Attribute an einem Konzept einen Wert erhalten dürfen.

Definition 4.22 *Zuweisbare Attribute eines Konzeptes*

Seien $a \in A$ ein Attribut, $c \in C$ ein Konzept und $AssignableAt \subset A \times C$ eine Relation.

Dann ist a ein *zuweisbares Attribut* von $c \Leftrightarrow a\ AssignableAt\ c\ :\Leftrightarrow$ eine der folgenden Bedingungen zutrifft

1. $a\ VirtualIn\ c$ (Standardwert)
2. $a\ InstantiatedIn\ c$

Die hier vorgestellten Definitionen der virtuellen und zuweisbaren Attribute unterscheiden sich von den entsprechenden Definitionen von Volz ([117], Abschnitt 8.7.4). Der entscheidende Punkt liegt darin, dass Volz in seiner Definition Attribute, die an einem instanziierten Typ definiert sind und einen Deep Instantiation Zähler größer eins besitzen, auch an abgeleitete Typ-Spezialisierungen des jeweiligen

Konzepts „vererbt". Instanz-Spezialisierungen dagegen erben zwar eine eventuelle Zuweisung (Standardwert), die Instanzen dieser Konzepte können jedoch das entsprechende Attribut nicht setzen, im Gegensatz zu den Instanzen der abgeleiteten Spezialisierungen. Dadurch unterscheidet sich die Menge der Instanzen n-ter Ordnung eines Konzepts **C**, die für ein Attribut, das als Typ **C** besitzt, zugewiesen werden dürfen, von der Menge der Konzepte, die ein Attribut von **C** zuweisen dürfen.

Um die Definitionen konform zum Substitutionsprinizp (siehe Kapitel 2) und der oben genannten Richtlinie zu formulieren, wurde daher eine strikte Trennung der Typ- und Instanzfacette vollzogen. Konkret bedeutet dies, dass durch eine Instanziierung ein Attribut instanziiert (daher auch der Name in Definition 4.21) und dann über die Instanzfacette weiter propagiert wird (Instanz-Spezialisierung). Folglich wird ein Attribut des instanziierten Typs nicht an die Typ-Spezialisierung weitervererbt, sondern an die Instanz-Spezialisierung als instanziiertes Attribut weitergereicht. Anschaulich wird also nur auf der „obersten" Ebene ein Attribut an eine Spezialisierung vererbt und nachdem es einmal instanziiert wurde, nur noch an die Instanzen bestimmter Ordnung weitergegeben. Insgesamt wird durch diese Festlegung keine Einschränkung vorgenommen, da die Typ- und Instanz-Spezialisierung sich nicht gegenseitig ausschließen.

Beispiel

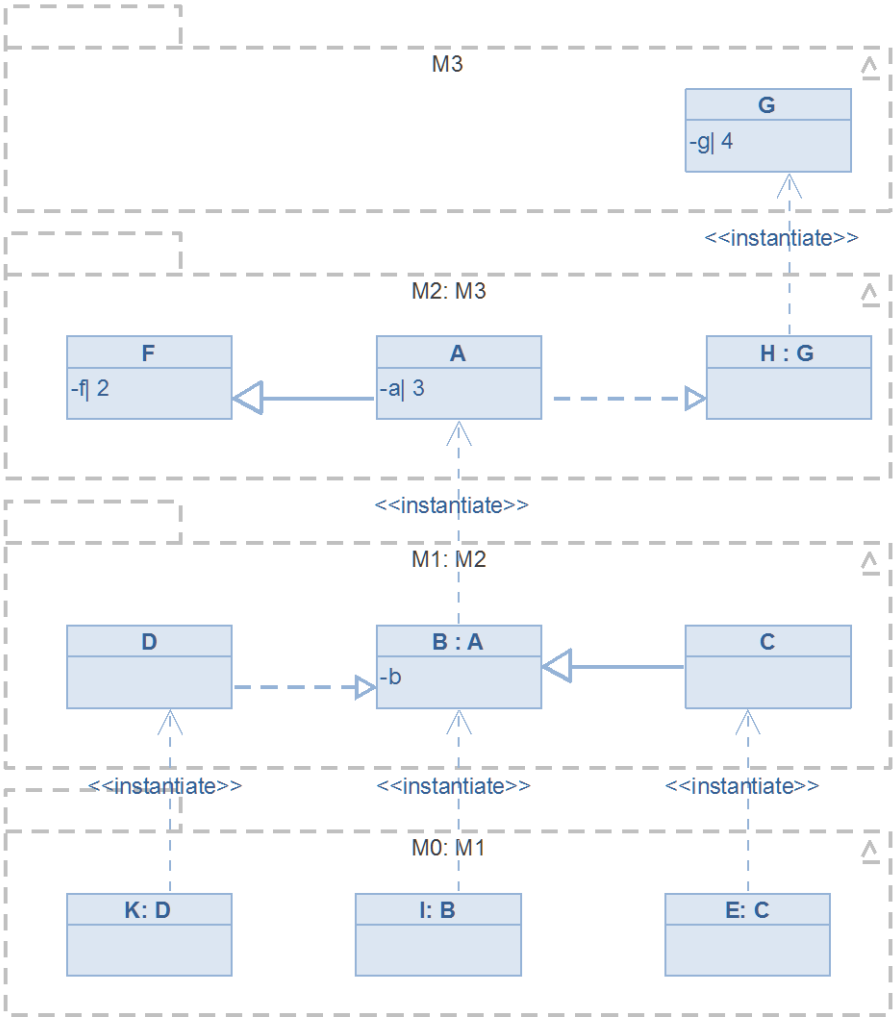

Abbildung 4-5 Beispiel zur Verdeutlichung der Attributbegriffe

Um die Begriffe virtuelles, instanziiertes und zuweisbares Attribut zu veranschaulichen, ist in Abbildung 4-5 eine Meta-Modellhierarchie abgebildet. Dabei wurde, anders als sonst, auch die Instanziierung als Pfeil im Diagramm visualisiert. Ansonsten wurde die Notation aus Abschnitt 4.4 verwendet. Im Nachfolgenden sollen nun für jedes Konzept die drei verschiedenen Mengen an Attributen dargestellt werden. Die jeweiligen „Potenzen" des Attributes geben den relativen Deep Instantiation Zähler des Attributes bezüglich des Konzeptes an. Die Menge der virtuellen Attribute für jedes einzelne Konzept findet sich in Tabelle 4-2.

Tabelle 4-2 Virtuelle Attribute der Konzepte im Beispiel

G	H	A	F	D	B	C	K	I	E
g^4	g^3	a^3, f^2	f^2	-	a^2, f, b	b	-	a	-

Für das Verständnis sollen exemplarisch die Mengen der virtuellen Attribute von **B** und **C** erläutert werden. Das Attribut **b** ist bei **B** direkt definiert, **a** wird über die Instanziierung zu **A** weitergegeben, da **a** einen Deep Instantiation Zähler von drei hat. Durch die Generalisierung von **F** zu **A**, wird das Attribut **f** an **A** vererbt und (mit der gleichen Begründung wie für **a**) bei **B** virtuell sichtbar. Das Attribut **g** wird nicht an **A** vererbt, da **A** eine Instanz-Spezialisierung von **H** darstellt, das wiederum eine Instanz vom umgebenden Konzept (**G**) von **g** ist. **C** dagegen ist eine Spezialisierung von **B** und erbt daher alle direkt an **B** definierten Attribute, also **b**.

Tabelle 4-3 Instanziierte Attribute der Konzepte des Beispiels

G	H	A	F	D	B	C	K	I	E
-	g^3	g^3	-	a^2, f, g^2	a^2, f, g^2	-	a, f^0, g	b^0, f^0, a, g	b^0

Tabelle 4-3 zeigt für jedes Konzept des Beispiels die jeweilige Menge an instanziierten Attributen. Dabei sind vor allem die Konzepte **K** und **E** interessant, um den Unterschied zwischen Instanzen einer Typ- und Instanz-Spezialisierung deutlich zu machen. Während **D** als Instanz-Spezialisierung von **B** alle instanziierten Attribute von **B** erhält, besitzt **C** keines dieser Attribute. Folglich besteht die Menge der instanziierten Attribute von **K** (als Instanz von **D**) aus **a**, **f** und **g**, während **E** als Instanz von **C** nur das virtuelle Attribut **b** des Konzeptes **C** instanziiert.

Tabelle 4-4 Zuweisbare Attribute der Konzepte des Beispiels

G	H	A	F	D	B	C	K	I	E
g^4	g^3	a^3, f^2, g^3	f^2	a^2, f, g^2	a^2, f, g^2,b	b	a, f^0, g	b^0, f^0, a, g	b^0

Die Menge der zuweisbaren Attribute sind letztlich die Vereinigung der beiden anderen Attributmengen. Die Resultate für das Beispiel finden sich in Tabelle 4-4. Die Menge der zuweisbaren Attribute setzt sich demnach aus den instanziierten Attributen und den virtuellen Attributen eines Konzeptes zusammen. Letztere dienen dem Setzen eines Standardwertes und werden an Instanz-Spezialisierungen nicht weitervererbt, da Instanzen dieser Konzepte das entsprechende Attribut nicht instanziieren und damit auch nicht zuweisen können. Folglich erbt eine Instanz-Spezialisierung nicht alle Zuweisungen des Prototyps, sondern nur diejenigen, die für instanziierte Attribute erstellt wurden. Um diesen Sachverhalt formal festzuhalten, wird im Folgenden Abschnitt der Begriff der virtuellen Zuweisungen eingeführt. Vorher wird jedoch eine weitere Relation benötigt, die eine Zuweisung mit dem entsprechenden Attribut verbindet

4.2.4 Definitionen und Regeln in Bezug auf Zuweisungen

In diesem Abschnitt befinden sich wichtige Begrifflichkeiten im Zusammenhang mit Zuweisungen. Dabei wurden sowohl für den Wert einer Zuweisung als auch für das Attribut Relationen definiert, die

die entsprechenden Eigenschaften des LMMs abbilden. Zusätzlich wird der Begriff der virtuellen Zuweisung formuliert, da dieser in den Kapiteln 5 und 6 benötigt wird.

Definition 4.23 *Attribut einer Zuweisung*

Seien $v \in \mathrm{V}$ eine Zuweisung, $a \in A$ ein Attribut und $ToAttribute \subset V \times A$ eine Relation.

Dann ist a das Attribut zu $v \Leftrightarrow v$ eine Zuweisung an a ist $\Leftrightarrow v\ ToAttribute\ a :\Leftrightarrow v$ hat a als Wert der `AttributeOf` Eigenschaft gesetzt.

Nun kann formal beschrieben werden, wann eine virtuelle Zuweisung an einem Konzept vorliegt. Dabei werden die verschiedenen Sprachmuster berücksichtigt.

Definition 4.24 *Virtuelle Zuweisung*

Seien $v \in \mathrm{V}$ eine Zuweisung, $c \in C$ ein Konzept und $VirtualAssignedAt \subset V \times C$ eine Relation. Weiterhin sei $a \in A$ ein Attribut mit $v\ ToAttribute\ a$.

Dann ist v eine *virtuelle Zuweisung* von $c \Leftrightarrow v\ VirtualAssignedAt\ c :\Leftrightarrow$ eine der folgenden Bedingungen zutrifft

1. $v\ In\ c$ (direkt definiert)
2. $\exists$ ein Konzept $p \in C$ mit $c\ ConcreteUseOf\ p \wedge\ a\ InstantiatedIn\ p$ (Zuweisung an ein instanziiertes Attribut des Prototyps)
3. $\exists$ ein Konzept $g \in C$ mit $c\ Extends^*\ g \wedge a\ VirtualIn\ g$ (virtuelles Attribut einer abgeleiteten Generalisierung, Standardwert)

Für die nachfolgend formulierten Regeln wird die Relation benötigt, die einer Zuweisung ihren Wert zuordnet. Diese soll nun definiert werden.

Definition 4.25 *Wert einer Zuweisung*

Seien $v \in \mathrm{V}$ eine Zuweisung, $w \in W$ ein Zuweisungswert, $a \in A$ ein Attribut und

$HasValue \subset V \times W$ eine Relation.

Dann ist $(v, w) \in HasValue :\Leftrightarrow v$ besitzt ein `Value` Element, das w als Wert der `value` Eigenschaft gesetzt hat.

Sei $v \in \mathrm{V}$ eine Zuweisung, $c_{val}, \mathrm{t} \in C$ zwei Konzepte, $a \in A$ ein Attribut mit $v\ ToAttribute\ a \wedge$ $a\ IsOfType\ t$ und $HasMultiplicity$ wie in Definition 4.12.

Regel Z.1 Wenn $v\ HasValue\ c_{val} \Longrightarrow \exists\ g \in C$: $c_{val}\ InstanceOf^{n(a)}\ g\ Extends^* t$, wobei $n(a) = di_{attribute}(a)$.

Regel Z.2 Wenn a einwertig ist $\Longrightarrow \forall$ Zuweisungen $z \in V$ mit $z\ ToAttribute\ a$: $\exists!\ w \in W$: $z\ HasValue\ w$

Regel Z.3 Wenn a obligatorisch ist $\Longrightarrow \forall$ Konzepte $c \in C$ mit $c\ Extends^*\ t$ und $\forall\ c_{inst} \in C$ mit $c_{inst} InstanceOf^{n(a)}\ c$ gilt: $\exists$ eine Zuweisung $z \in V$: $z\ ToAttribute\ a \wedge$ $z\ VirtualAssignedAt\ c_{inst}$

Weiterhin sei $a_{op} \in A$ ein Attribut und $c_{cont} \in C$ das Konzept mit $v\ In\ c_{cont}$.

Regel Z.4 Dann ist: $a\ AssignableAt\ c_{cont}$ ([117], Abschnitt 8.7.4)

Regel Z.5 Wenn $a\ OppositeOf\ a_{op} \wedge v\ HasValue\ c_{val} \Longrightarrow \exists\ v_{op} \in V$ mit $v_{op}\ ToAttribute\ a_{op} \wedge v_{op}\ In\ c_{val} \wedge\ v_{op}\ HasValue\ c_{cont}$

Bemerkung:

- Regel Z.1 legt den Wertebereich eines Attributs fest, dessen Typ ein Konzept ist. Er besteht aus der Menge der Instanzen n-ter Ordnung der abgeleiteten Spezialisierungen des Attributtyps, wobei n der Deep Instantiation Zähler des Attributes ist.
- Regel Z.2 besagt, dass einwertige Attribute höchstens einen Wert besitzen müssen.
- Regel Z.3 bringt zum Ausdruck, dass obligatorische Attribute bei jeder Instanz n-ter Ordnung des Attributtyps oder einer abgeleiteten Spezialisierung davon eine virtuelle Zuweisung besitzen müssen.
- Die zuweisbaren Attribute eines Konzepts bilden die Menge an Kandidaten für die `AttributeOf` Eigenschaft einer Zuweisung. (Regel Z.4)
- Die Regel Z.5 besagt, dass für jede Zuweisung an ein gegenüberliegendes Attribut eine entsprechende Zuweisung an das andere Attribut an jedem Konzept, das als Wert der Zuweisung gesetzt ist, existieren muss. Ferner müssen diese Zuweisungen das umgebende Konzept der Ausgangszuweisung als Wert besitzen.

4.3 Notation der Ablaufdiagramme

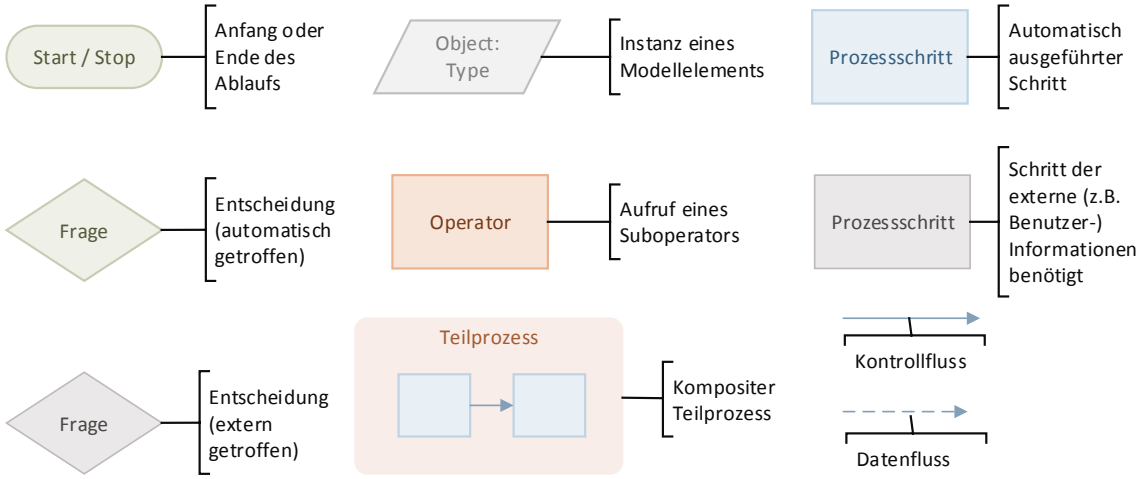

Abbildung 4-6 Notation der Ablaufdiagramme

Die Ablaufdiagramme, die dazu verwendet werden, um in den Kapiteln 5 und 6 den Ablauf eines Operators zu beschreiben, haben eine Notation, die einem Flussdiagramm ähnelt und in Abbildung 4-6 dargestellt ist. Dabei werden auf der linken Seite die jeweiligen Formen dargestellt und mit einer Legenden auf der rechten Seite beschrieben. Wie üblich gibt es zu jedem Ablauf einen Start und einen Stop Knoten. Weiterhin existieren Prozessschritte, Entscheidungen und Objekte, die Modellelemente oder Mengen solcher darstellen. Mengen von Modellelementen werden in einer ähnlichen Notation wie in Java mit generischen Parametern versehen, um die jeweiligen Modellelementtypen zu kennzeichnen. Die verschiedenen Knoten im Modell werden durch Kontrollflüsse verbunden. Dabei wird vereinbart, wenn eine Menge von Objekten in einen Prozess oder eine Entscheidung fließt, dass der Schritt bzw. die Entscheidung für jedes Element der Menge getroffen wird. Folglich werden

Schleifen, die auf der bloßen Iteration einer Menge beruhen, nicht visualisiert, um die Diagramme übersichtlich zu gestalten.

Für den Fall, dass ein Prozessschritt ein Objekt benötigt und nicht eindeutig ersichtlich ist, um welches es sich handelt, wurde ein Datenfluss vom Objekt zum Schritt gezeichnet. Neben der durch die verschiedenen Formen vorgegebenen Unterscheidung, hat auch die Farbe einen semantischen Wert. Alle roten Figuren benötigen Informationen, die z.B. durch den Benutzer oder eine äußeren Operator bereitgestellt werden können. Wenn ein Operator einen andern Operator verwendet, wird dies durch ein oranges Rechteck gekennzeichnet. Für Operatoren mit vielen Prozessschritten wurden zudem komposite Teilprozesse definiert, die den jeweiligen Bereich im Prozess umspannen und damit einen Leitfaden und eine grobe Übersicht bieten. Die jeweiligen Titel der Teilprozesse finden sich im dazugehörigen Text als Abschnittstitel wieder.

4.4 Notation der Beispieldiagramme

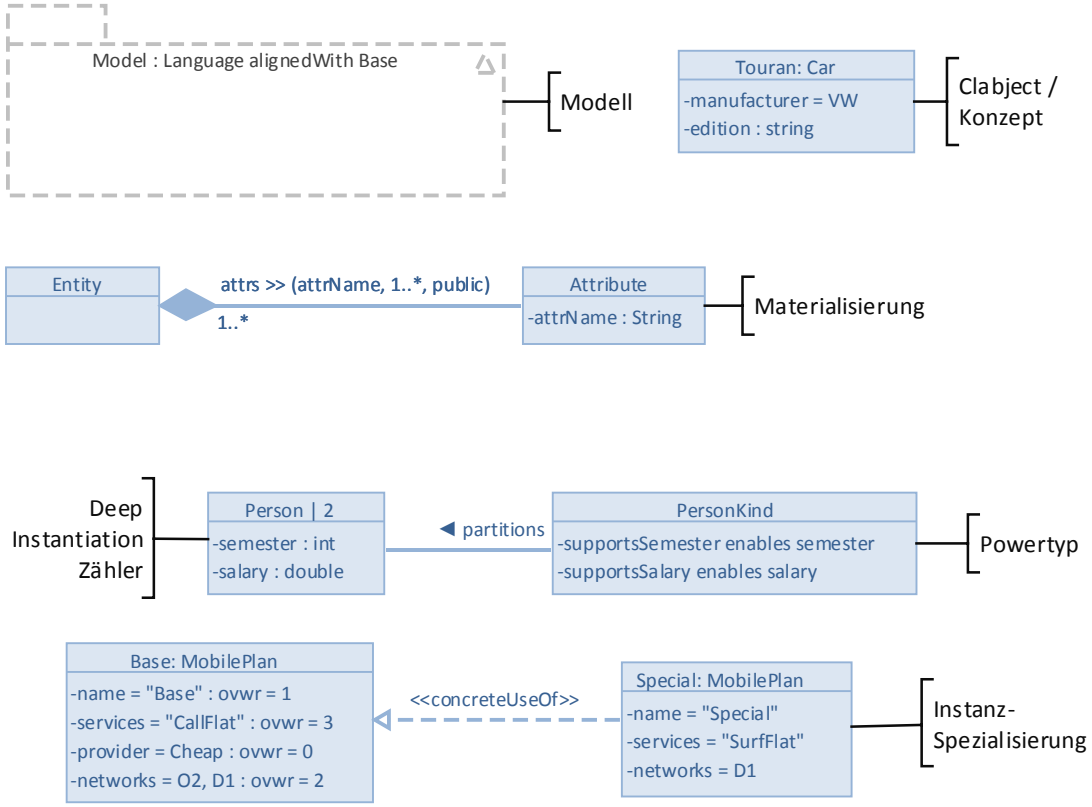

Abbildung 4-7 Notation der Sprachmuster

Um die Beispiele der Kapitel 5 und 6 einfach darzustellen, wurde zu jedem ein Diagramm erstellt. Diese zeigen zunächst eine Ausgangssituation, in der eine Meta-Modell Hierarchie abgebildet ist, und ein Resultat, das durch die Anwendung des jeweiligen Operators entsteht. Die verwendete Notation ähnelt stark der UML. Sie wurde allerdings um Notationen für die Sprachmuster erweitert. Die verschieden Notationen sind in Abbildung 4-7 dargestellt, wobei die jeweiligen Ausschnitte, die die Sprachmuster darstellen, durch Legenden näher bezeichnet wurden.

- Die verschiedenen Metaebenen wurden in einer paketähnlichen Darstellung visualisiert, wobei das Meta-Modell jeweils nach dem Doppelpunkt referenziert wird. Wenn Ebenen miteinander verschmolzen wurden, steht der Name der Ebene, deren Modellelemente importiert werden, nach dem Schlüsselwort `alignedWith`.

- Die Materialisierungserweiterung wurde bei der entsprechende Beziehung nach dem „>>“ als Tripel hinzugefügt. Dabei verweist der erste Eintrag auf die Attribut-Definition, während der zweite die Multiplizität und der dritte die Sichtbarkeit aller materialisierten Attribute festlegt.
- Für das Deep Instantiation Sprachmuster wird der Zähler des jeweiligen Konzeptes, wenn er größer als eins ist, nach einem „|“ dargestellt.
- Die für den Powertyp charakteristische `partitions` Beziehung wird durch ein Dreieck mit dem entsprechenden Text auf der Beziehung visualisiert.
- Die Instanz-Spezialisierung wird durch einen Pfeil mit der Beschriftung `<<concreteUseOf>>` dargestellt. Der Überschreibungstyp der Zuweisung am Prototyp wird als Wert für `ovwr` hinter dem Doppelpunkt angegeben.
- Die übliche Notation für Klassen in UML Diagrammen wurde für Konzepte adaptiert. Da diese sowohl Instanz- als auch Typfacette besitzen, werden sowohl Attributdefinitionen als auch Zuweisungen innerhalb des Bereiches für Attribute definiert.
- Der instanziierte Typ eines Konzeptes wird nach einem Doppelpunkt referenziert.

Die in den Diagrammen verwendeten Attribute haben zumeist keine explizit dargestellten Typen. Wenn jedoch der Attributtyp für die Anwendung eines Operators von Bedeutung ist, wird dieser wie in der UML üblich nach einem Doppelpunkt dargestellt.

Standardwerte werden in den Diagrammen nicht explizit mit aufgenommen. So wird eine Multiplizität von `0..1,` die Sichtbarkeit `public` und ein Deep Instantiation Zähler von eins nicht visualisiert. Weiterhin werden die Multiplizität von Attribut-Definitionen und Diskriminatorattributen sowie der Typ von letztgenannten ebenfalls nicht dargestellt, da diese immer den Wert `1` bzw. `boolean` besitzen.

5 Operatoren zur Ausführung von elementaren Änderungen

Diese Kapitel widmet sich der Vorstellung von elementaren Operatoren des LMMs. Sie dienen dazu die einfachen Änderungen an einem Modellelement so durchzuführen, dass nach der Ausführung ein strukturell valider und zu den Sprachmustern konformer Zustand entsteht. Dahingehend stellt diese Bibliothek ein Novum dar, welches in keinem bisherigen Ansatz (Kapitel 2) in dieser Vollständigkeit existiert. Im Wesentlichen werden für alle Eigenschaft eines Modellelementes des operatorenbasierten LMMs Methoden zum Ändern bzw. Setzen präsentiert. Daraus resultiert auch die Benennung der jeweiligen Operatoren (`Set <<Eigenschaft>>`). Zusätzlich werden für jedes Modellelement Operatoren zum Entfernen bereitgestellt, die die entsprechenden Abhängigkeiten zu anderen Elementen auflösen, bevor das jeweilige Element entfernt wird. Auf die Vorstellung von Operatoren zum Erstellen von Elementen wurde verzichtet, da das bloße Anlegen eines Modellelements keine Inkonsistenz erzeugt und für das Setzen der entsprechenden Eigenschaften bereits Operatoren vorhanden sind, die in diesem Fall genutzt werden können.

Die Operatoren wurden nach dem betroffenen Modellelement in Abschnitte unterteilt. Jeder Abschnitt hat dabei den gleichen Aufbau. Zunächst wird kurz erläutert (Auswirkung), was der Operator im groben Zügen am Modell ändert. Danach folgt eine ausführliche Beschreibung, die den Ablauf schrittweise erklärt. Dabei ist stets ein Ablaufdiagramm dargestellt, um den Überblick während des Lesens zu verbessern. Die verwendete Notation für diesen Diagrammtyp wird in Abschnitt 4.3 beschrieben. Eine Besonderheit der in diesem Kapitel verwendeten Ablaufdiagramme liegt darin, dass in jedem die elementare Änderung des Operators durch einen separaten Teilschritt hervorgehoben ist. Neben der Beschreibung des Ablaufes wird für einige Operatoren auch ein Beispiel vorgestellt, das nochmal die exemplarische Anwendung dieses Operators an einem Modell zeigen soll. Für die Darstellung dieser Modelle wurde eine eigene Notation verwendet, die an die UML Notation angelegt ist, Sprachmuster unterstützt und im Abschnitt 4.4 näher erläutert wird.

Generell kann durch die Ausführung eines Operators ein anderer Suboperator aufgerufen werden. Ein solches Verhalten ist aufgrund der Modularität und der Vermeidung von Redundanzen wünschenswert. Dadurch können jedoch Rekursionen und Zyklen innerhalb der Aufrufe entstehen (Bsp: *Set Value* Operator). Letztgenannte müssen entsprechend erkannt und aufgelöst werden. Aufgrund der Tatsache, dass die hier beschriebenen Operatoren ein Konzept und keine direkte Implementierungsvorlage darstellen sollen und um die Übersichtlichkeit der Abläufe nicht zu gefährden, wurde auf die explizite Behandlung von Zyklen innerhalb der Operatoren verzichtet. Die Behandlung dieses Problems kann auf verschiedene Arten erfolgen und ist implementierungsspezifisch. Eine mögliche Lösungsvariante soll an dieser Stelle skizziert werden. Prinzipiell empfiehlt sich eine externe Behandlung der Zyklen, da diese auch zwischen unterschiedlichen Operatoren auftreten können (Bsp: *Set Value* → *Delete Concept* → *Set Value*). Folglich sieht eine mögliche Lösung so aus, dass bei jedem Aufruf eines Operators alle ausgeführten Suboperationen (parametrierter Operator) gemerkt werden. Vor jedem Aufruf einer weiteren Suboperation wird dann überprüft, ob diese bereits ausgeführt wurde. Falls dies zutrifft, wird die Ausführung abgebrochen. Die Gleichheit zwischen zwei Operationen ist dann gegeben, wenn beide denselben Operator verwenden und am (semantisch) gleichen Modellelement aufgerufen werden.

Neben der gezielten Änderung eines Modells können die in diesem Kapitel vorgestellten Operatoren auch zur Auflösung von Konsistenz- oder Konformitätsbrüchen dienen, die während der Validierung eines Modells auftreten. Dies kann wie folgt umgesetzt werden: Während der Evolution des Modells (ohne Verwendung der Operatoren) treten elementare Änderungen auf. Diese können dazu führen, dass ein Konsistenz- oder Konformitätsbruch entsteht, der durch die Validierung erkannt wird. In diesem Fall kann dem Benutzer dennoch bei der Migration unterstützt werden, dahingehend, dass der jeweilige Operator als Migrationslösung vorgeschlagen wird, damit dieser die Koevolution (`ExecutionType Co_EVOLVE,` siehe Abschnitt 4.1.1) durchführt. Dabei ist zu beachten, dass die elementare Änderung nicht erneut ausgeführt wird. Ebenso ist von Bedeutung, dass die Berechnungen des Operators auf der Version des Modells vor der elementaren Änderung stattfinden. Da verschiedene elementare Änderungen sich beeinflussen oder auslöschen können, kann in diesem Fall jeweils nur die letztausgeführte Änderung unterstützt werden.

5.1 Ebene (Level)

Ebenen (Beschreibung in Abschnitt 4.1.2 bzw. 4.2.1) bilden die grundlegende Struktur der Meta-Hierarchie und werden durch das LMM Element `Level` abgebildet. Die folgenden Operatoren dienen der Änderung einer Ebene. Sie setzen sowohl die instanziierte Ebene als auch die logisch verschmolzenen Ebenen oder entfernen eine Ebene samt ihres Inhalts.

5.1.1 Set Aligned Level

Auswirkung

Der Operator setzt die Verschmelzung mit einer Ebene und entfernt dabei alle invaliden Spezialisierungen, Partitionierungen, Instanz-Spezialisierungen und Assoziationen zur vormals definierten Basis-Ebene.

Ablauf

Auswahl der Ebenen

Abbildung 5-1 zeigt den Ablauf des *Set Aligned Level* Operators. Dieser wird an einer Ebene `ToChange` aufgerufen. Im ersten Schritt werden alle Ebenen `ToRemove`, die bisher mit `ToChange` verschmolzen sind, bestimmt. Danach wird die Menge aller Ebenen `ToSet` ausgewählt, die künftig eine logische Verschmelzung mit `ToChange` eingehen sollen. Diejenigen Ebenen von `ToSet`, die auch in `ToRemove` enthalten sind, werden aus der zweiten Menge entfernt, wodurch `ToRemove` alle diejenigen Ebenen enthält, die nach der Ausführung des Operators keine Verschmelzung zu `ToChange` mehr eingehen.

Elementare Änderung

Anschließend werden alle Elemente aus `ToSet` logisch mit `ToChange` durch das Setzen der `alignedWith` Referenz verschmolzen.

Generalisierung, Prototyp und partitionierten Typ entfernen

Nachdem die elementare Änderung erfolgt ist, ermittelt der Operator alle Konzepte von `ToChange`, da diese nun invalide Referenzen (`extends`, `partitions`, `concreteUseOf` oder eine Beziehung) auf ein Konzept, das in einer Ebene von `ToRemove` liegt, besitzen können. Deshalb wird für jedes Konzept in `ToChange` die Generalisierung `Base` bestimmt und untersucht, ob sie zum einen existiert und zum anderen in einer Ebene aus `ToRemove` liegt. Ist dies der Fall wird die Generalisierung mittels des *Delete Specialization* Operators gelöscht. Analog dazu wird im nächsten Schritt der partitionierte Typ ermittelt und für den Fall, dass dieser in einer Ebene von `ToRemove` liegt, wird die Referenz mit Hilfe des *Delete Partition* Operators entfernt. Anschließend wird ein eventuell vorhandener Prototyp `Prototype`

bestimmt und falls dieser in **ToRemove** liegt, wird die Instanz-Spezialisierung durch den *Delete Concrete Use Of* Operator aufgelöst.

Beziehungen entfernen

Zuletzt werden alle Beziehungen, die nun invalide sind, ebenfalls migriert. Dazu werden für jedes Konzept in **ToChange** alle Attribute bestimmt und es wird untersucht, ob der Typ des Attributes ein Konzept ist, das in einer Ebene von **ToRemove** liegt. Ist dies der Fall muss das Attribut mittels des *Delete Attribute* Operators gelöscht werden. Nach der Migration der Beziehungen endet der Operator.

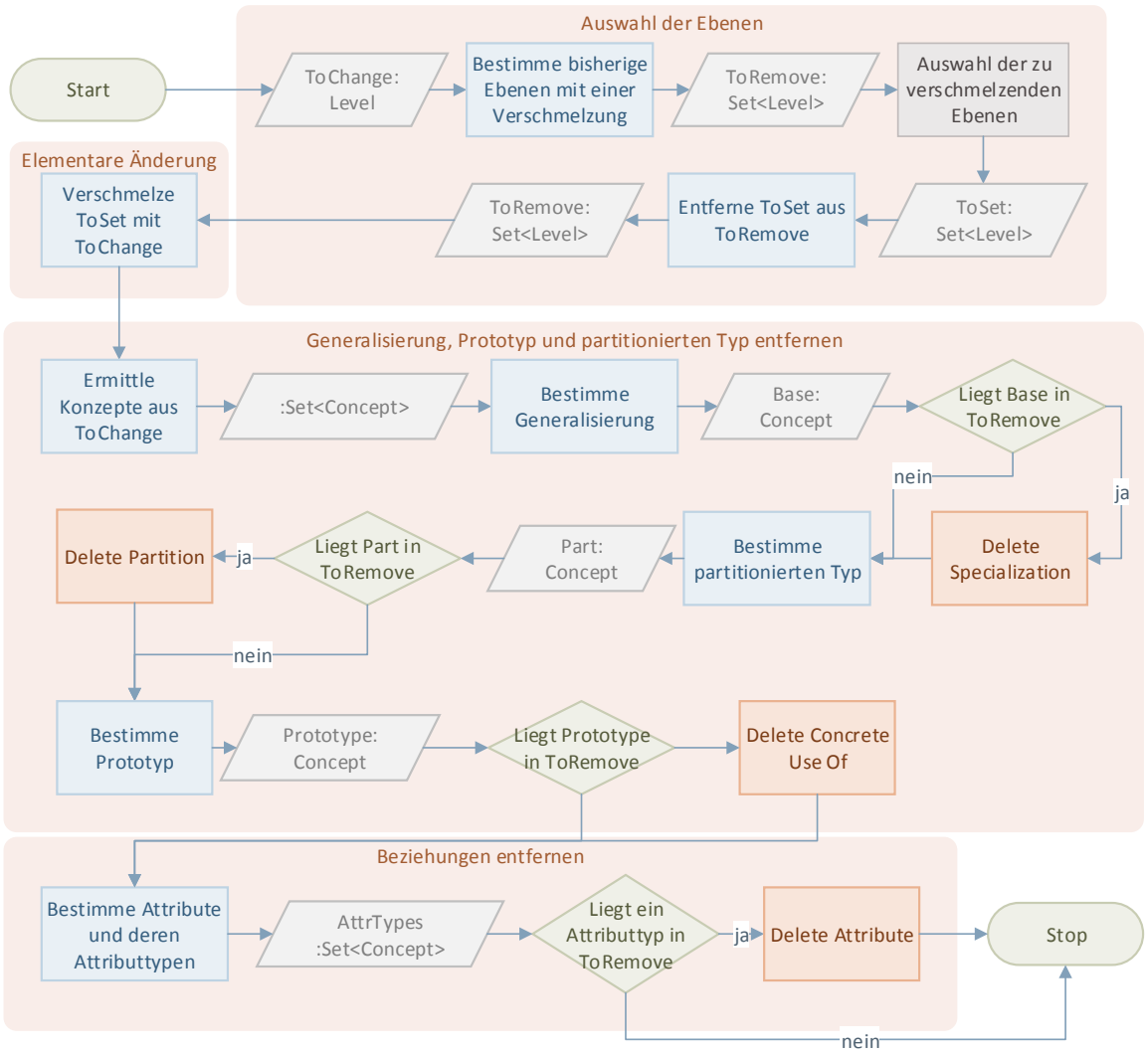

Abbildung 5-1 Ablauf des Set Aligned Level Operators

Beispiel

Der Operator findet Anwendung im Beispiel des *Move Type to Lower Level* Operators in Abschnitt 6.1.3.

5.1.2 Set Instantiated Level

Auswirkung

Die Ausführung des Operators entfernt die Instanziierung zu einer Ebene und entfernt alle Konzept-Instanziierungen, die dadurch ungültig sind.

Ablauf

Instanziierten Typ entfernen

Der Ablauf des Operators ist in Abbildung 5-2 dargestellt. Zunächst wird der Operator an einer Ebene `ToChange` aufgerufen. Im ersten Schritt wird die bisherige instanziierte Ebene `ToRemove` ermittelt und danach die neue Ebene `ToSet` gewählt. Wenn `ToRemove` existiert und ungleich `ToSet` ist, werden alle Konzepte aus `ToChange` bestimmt. Da der jeweilige instanziierte Typ in der Ebene lag, zu der die Referenz gelöscht wurde, oder in einer in der Hierarchie noch weiter oben gelegenen Ebene (siehe Regel C.12), ist diese Referenz nun ebenfalls ungültig. Deshalb wird diese mit Hilfe des *Delete Instantiation* Operators entfernt.

Elementare Änderung

Im letzten Schritt setzt der Operator die Referenz auf die neue instanziierte Ebene `ToSet` und endet anschließend.

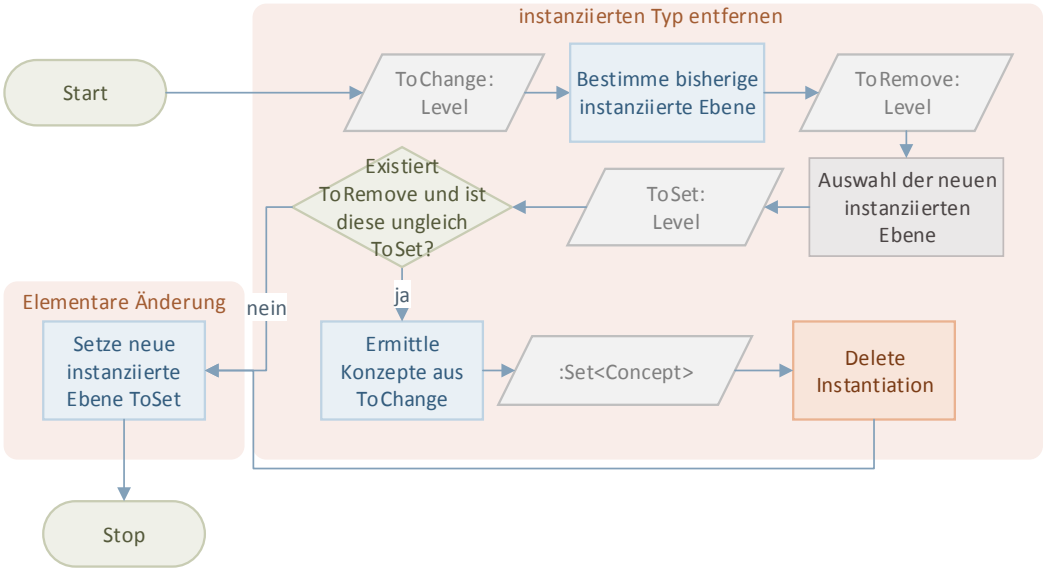

Abbildung 5-2 Ablauf des Set Instantiated Level Operators

5.1.3 Delete Level

Auswirkung

Durch den Operator wird eine Ebene mit allen Paketen, die in ihr liegen, gelöscht.

Ablauf

Abhängige Ebenen migrieren

Der Operator (Abbildung 5-3) wird an einer Ebene `ToDelete` aufgerufen, die gelöscht werden soll. Dazu werden zunächst alle Ebenen (`Instances`) bestimmt, die Instanzen von `ToDelete` sind. An diesen wird anschließend der *Set Instantiated Level* Operator aufgerufen (Auswahl eine leeren Wertes), wodurch alle Instanziierungen von Konzepten innerhalb einer dieser Ebenen zu einem Konzept aus `ToDelete` entfernt werden. Im nächsten Schritt ermittelt der Operator alle Ebenen, die `ToDelete` erweitern, d.h. eine `alignedWith` Referenz zu `ToDelete` besitzen. Für jedes Element dieser Menge (`Aligned`) wird der *Set Aligned Level* Operator aufgerufen, wobei alle bisher gesetzten Ebenen bis auf `ToDelete` im Operator ausgewählt werden. Dadurch werden alle Spezialisierungen, Partitionierungen, Instanz-Spezialisierungen und Beziehungen zu Konzepten in `ToDelete` migriert.

Pakete löschen

Im nächsten Schritt werden alle Pakete berechnet, die in `ToDelete` liegen. Diese werden anschließend mit Hilfe des *Delete Package* Operators gelöscht.

Elementare Änderung

Bevor der Operator beendet ist, wird im letzten Schritt noch die Ebene `ToDelete` entfernt.

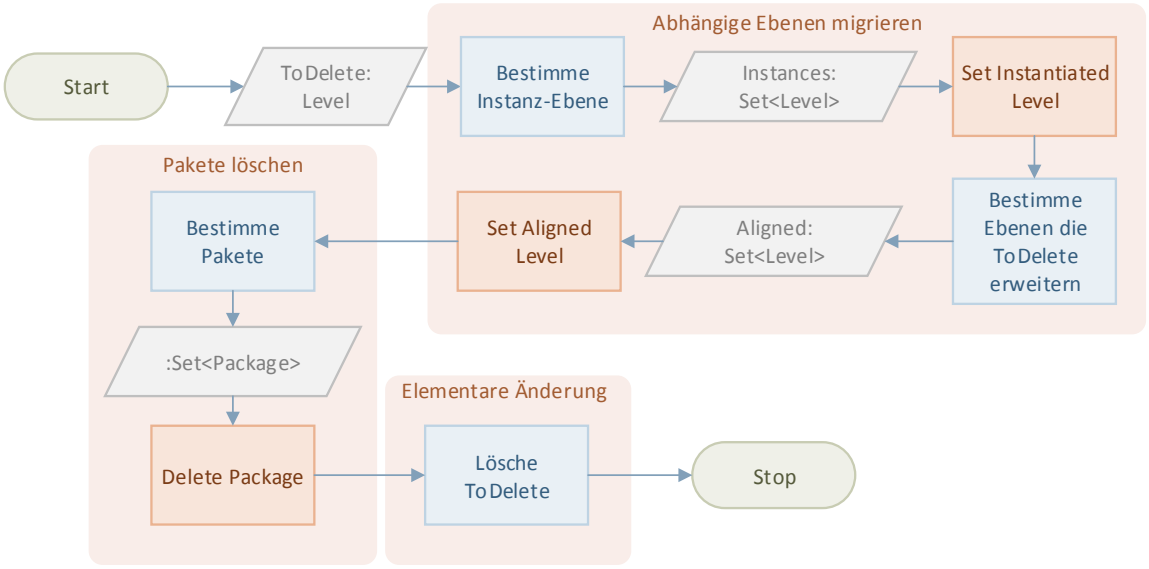

Abbildung 5-3 Ablauf des Delete Level Operators

5.2 Paket (Package)

Da Pakete (Beschreibung in Abschnitt 4.1.2) lediglich der Strukturierung innerhalb einer Ebene dienen und deshalb nur einen Namen besitzen, führen die meisten Änderungen an einem Paket zu keiner Inkonsistenz. Einzig beim Löschen eines Paketes, muss darauf geachtet werden, dass alle Inhalte ebenfalls entfernt werden.

5.2.1 Delete Package

Auswirkung

Der Operator löscht ein Paket und alle Referenztypen (Konzepte und Enumerationen), die darin liegen.

Ablauf

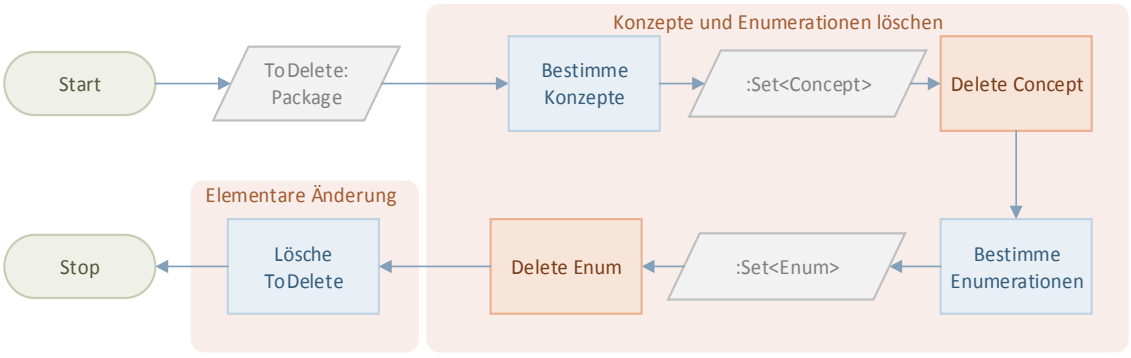

Abbildung 5-4 Ablauf des Delete Package Operators

Konzepte und Enumerationen löschen

Der Ablauf des Operators ist in Abbildung 5-4 dargestellt. Zu Beginn wird ein Paket `ToDelete` übergeben, das im Folgenden gelöscht werden soll. Dazu werden zunächst alle Konzepte ermittelt, die anschließend durch den *Delete Concept* Operator gelöscht werden. Im Anschluss werden noch alle Enumerationen von `ToDelete` bestimmt und danach mit Hilfe des *Delete Enum* Operators entfernt.

Elementare Änderung

Im letzten Schritt wird noch das Paket entfernt und der Operator ist beendet.

5.3 Konzept (Concept)

Ein Konzept stellt die Umsetzung des Clabjects Musters (Abschnitt 2.1) innerhalb des LMMs dar. Es kann daher Beziehungen, eine Instanziierung, eine (Instanz-)Spezialisierung oder eine Partitionierung zu anderen Konzepten definieren. Der nachfolgende Abschnitt stellt Operatoren zur Änderung eines Konzeptes vor, die neben der elementaren Änderung auch weitere Migrationen nach sich ziehen. Anders als bei den Eigenschaften eines Attributes zum Beispiel, sind für das Modelelement „Konzept" explizite Operatoren zum Entfernen von Referenzen (Instanziierung, Spezialisierung, Instanz-Spezialisierung und Partitionierung) definiert, um die Operatoren zum Setzen der Referenzen übersichtlicher und verständlicher zu gestalten. Ein weiterer Grund besteht darin, dass das bloße Entfernen einer dieser Referenzen eine häufig verwendete Modelländerung ist und daher die Kapselung in einem eigenen Operator auch aus benutzerspezifischen Standpunkten sinnvoll ist. Ähnlich verhält es sich mit den Operatoren zur inkrementellen Erhöhung oder Verringerung des Deep Instantiation Zähler (Abschnitt 5.3.4 und 5.3.3).

Für die Operatoren *Set Concept Abstract* und *Set Concept Final* wurde aufgrund der Tatsache, dass die möglichen Werte `true` oder `false` sind, auf eine Auswahl des Wertes innerhalb des Operators verzichtet. Die vorgestellten Operatoren, zeigen lediglich den Fall, wenn ein Konzept final oder abstrakt gesetzt wird, da nur dann Inkonsistenzen auftreten können. Der jeweilige andere Wert (`false`) kann bei einem Konzept elementar gesetzt werden und bedarf keiner weiteren Anpassung innerhalb des Modells.

5.3.1 Set Concept Abstract

Auswirkung

Der Aufruf des Operators setzt ein Konzept abstrakt und löscht die Instanzen oder verschiebt diese zu einer Spezialisierung.

Ablauf

Instanz-Migration

Der Operator wird an einem nicht finalen Konzept `ToChange` aufgerufen (siehe Regel C.6 bzw. Regel C.7). Im ersten Schritt werden die direkten Instanzen (`Instances`) von `ToChange` ermittelt. Wenn diese nicht existieren, kann mit der Bestimmung der privaten Attribute fortgefahren werden. Anderenfalls muss die Instanziierung migriert werden, da diese nach der Ausführung des Operators nicht mehr valide ist. Deshalb werden zunächst alle nicht abstrakten Spezialisierungen von `ToChange` bestimmt, um eventuell die Instanziierung von `Instances` umzubiegen. Für den Fall, dass jedoch keine Spezialisierung existiert oder die Änderung der Instanziierung nicht gewählt wurde, werden die Instanziierungen von `Instances` zu `ToChange` gelöscht (Operator *Delete Instantiation*) (Regel C.4). Wenn andererseits Spezialisierungen vorhanden sind und eine Verschiebung stattfinden soll, dann wird zunächst eine Spezialisierung `NewType` gewählt, die den neuen Typ der Instanzen von `ToChange`

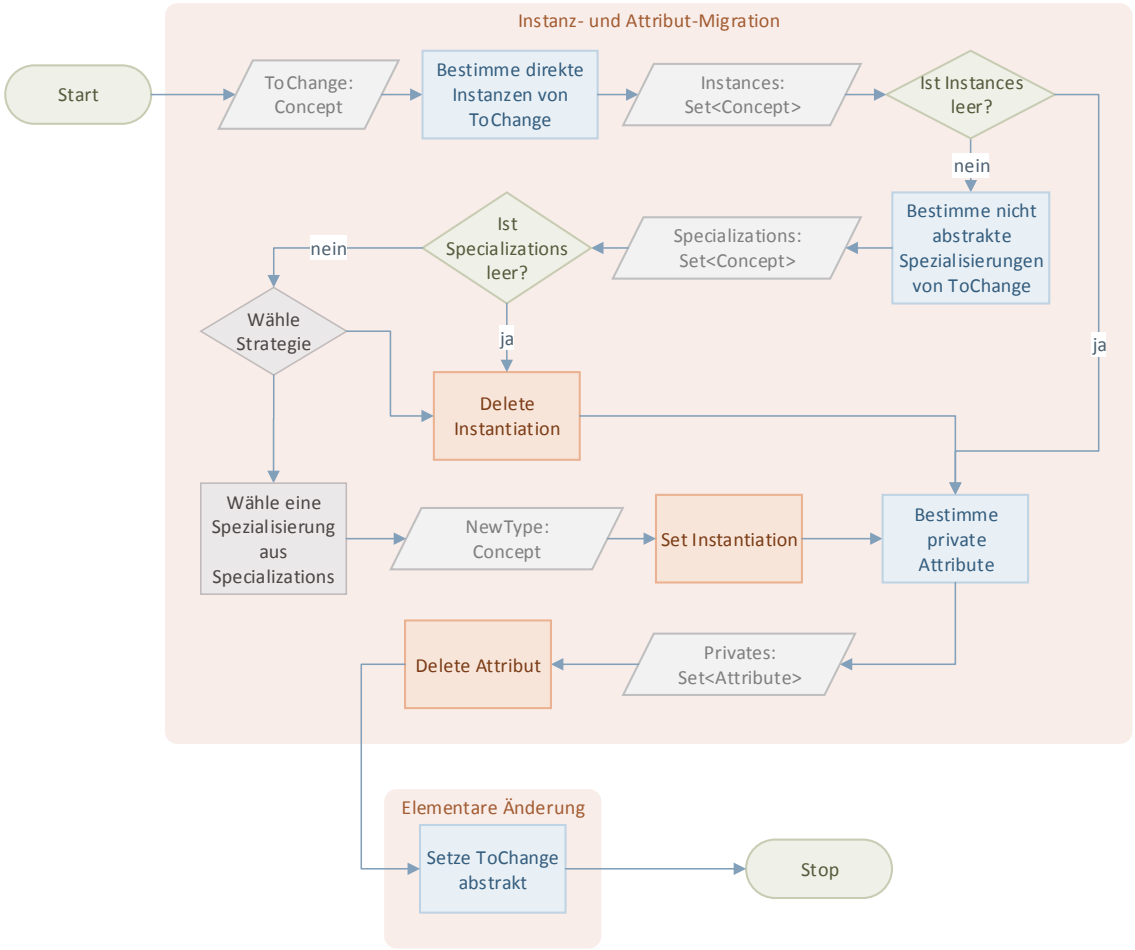

Abbildung 5-5 Ablauf des Set Concept Abstract Operators

darstellt. Dieser wird dann mittels *Set Instantiation* Operator bei allen Elementen aus `Instances` gesetzt. Als nächstes werden alle privaten Attribute (`Privates`) von `ToChange` ermittelt. Diese sind bei einem abstrakten Konzept von keinerlei Nutzen und daher auch nach Regel C.4 verboten. Deshalb werden sie im nächsten Schritt mit Hilfe des *Delete Attribute* Operators entfernt.

Elementare Änderung

Im letzten Schritt kann `ToChange` auf abstrakt gesetzt werden, wodurch der Operator beendet wird.

5.3.2 Set Concept Final

Auswirkung

Durch den Operator wird ein Konzept als final deklariert. Dabei werden alle Instanz- Spezialisierungen, Spezialisierungen sowie eine mögliche Partitionierung entfernt.

Ablauf

Spezialisierung und Powertyp migrieren

Initial wird der Operator (Abbildung 5-6) an einem nicht abstrakten Konzept `ToChange` aufgerufen (siehe Regel C.6 bzw. Regel C.7). Als erstes werden von `ToChange` alle direkten Spezialisierungen `Specializations` bestimmt. Diese sind nach der Ausführung des Operators nicht mehr gültig und folglich wird die Spezialisierung entfernt (*Delete Specialization*). Da `ToChange` als finales Konzept auch

kein partitionierter Typ sein darf (Regel C.5), wird im nächsten Schritt ein eventuell vorhandener Powertyp bestimmt und anschließend die Referenz mit Hilfe des *Delete Partition* Operators entfernt. Danach werden noch die Instanz-Spezialisierungen von **ToChange** ermittelt und die Referenz durch den *Delete Concrete Use Of* Operator entfernt.

Elementare Änderung

Zum Abschluss wird **ToChange** final gesetzt und der Operator ist beendet.

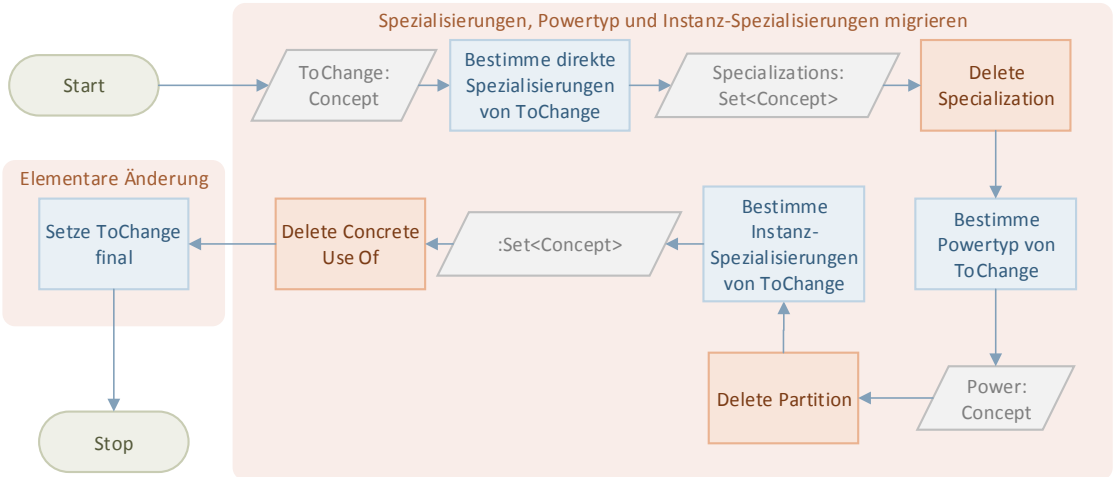

Abbildung 5-6 Ablauf des Set Concept Abstract Operators

5.3.3 Increment DI Counter of Concept

Auswirkung

Der Operator erhöht den Deep Instantiation Zähler des übergebenen Konzeptes um eins. Instanzen, die dadurch keine Instanziierung mehr eingehen dürfen, werden migriert.

Ablauf

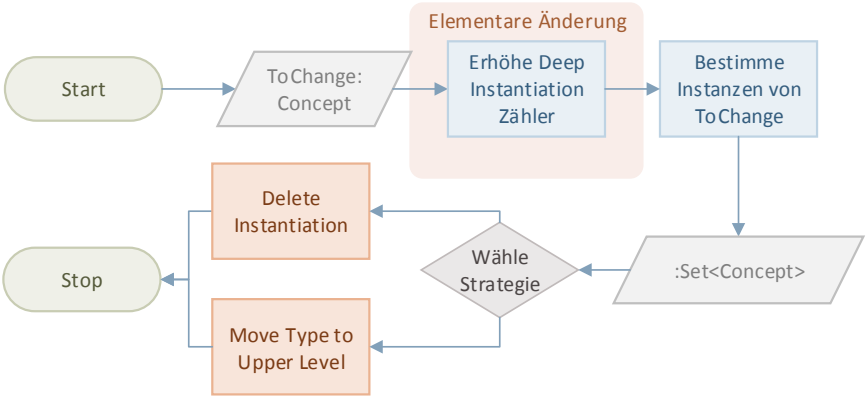

Abbildung 5-7 Ablauf der Increment DI Counter of Concept Operators

Elementare Änderung

Der Operator wird zunächst an einem Konzept **ToChange** aufgerufen, von dem danach der Deep Instantiation Zähler um eins erhöht wird.

Anschließend werden alle Instanzen von **ToChange** gesucht, da diese nach Ablauf des Operators nicht mehr gültig sind. Deshalb muss für jede Instanz die Auswahl getroffen werden, ob entweder das

Konzept eine Ebene nach unten verschoben oder die Instanziierung durch den *Delete Instantiation* Operator gelöscht wird, womit der Operator endet.

5.3.4 Decrement DI Counter of Concept

Auswirkung

Die Ausführung des Operators verringert den Deep Instantiation Zähler eines Konzepts um eins und migriert die invalide gewordenen Instanzen.

Ablauf

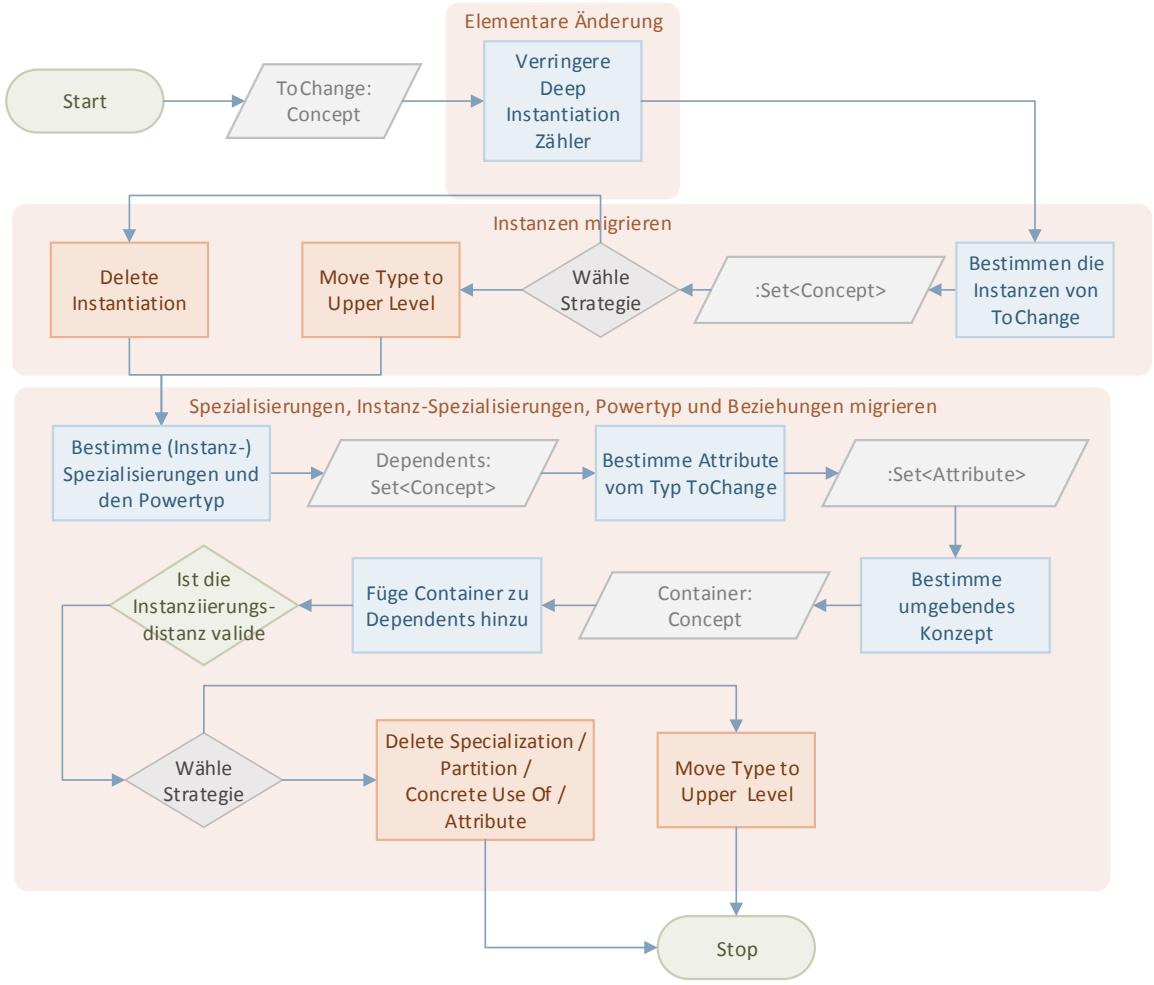

Abbildung 5-8 Ablauf des Decrement DI Counter Operators

Elementare Änderung

Der Operator wird an einem Konzept **ToChange**, dessen Deep Instantiation Zähler größer als eins ist, aufgerufen. Danach wird der Deep Instantiation Zähler von **ToChange** um eins verringert.

Instanzen migrieren

Anschließend werden alle Instanzen berechnet, da die jeweilige Instanziierung nun ungültig ist. Für jede dieser Instanzen gibt es zwei Alternativen: Entweder wird die Instanziierung mit Hilfe des *Delete Instantiation* Operators gelöscht oder der *Move Type to Upper Level* Operator wird dazu verwendet, die entsprechende Instanz eine Meta-Ebene nach oben zu verschieben, da in diesem Fall die Instanziierung wieder gültig wäre.

Spezialisierungen, Instanz-Spezialisierungen, Powertyp und Beziehungen migrieren

Da die Änderung des Deep Instantiation Zählers auch die Menge an validen Spezialisierungen, Instanz-Spezialisierungen, Powertypen und Beziehungen eines Konzeptes beeinflusst, müssen die nun invalide gewordenen Konzepte ebenfalls migriert werden. Falls ein Konzept vor der Ausführung ein Powertyp eine Spezialisierung oder eine Instanz-Spezialisierung war, das in einer Ebene definiert wurde, deren Instanziierungsdistanz zur Ebene von `ToChange` um eins kleiner als der Deep Instantiation Zahler von `ToChange` ist, ist dieses nun invalide. Entsprechend werden diese Konzepte zunächst ermittelt (`Dependets`). Gleichermaßen sind Konzepte betroffen, die eine Beziehung zu `ToChange` definieren und in einer Ebene mit der oben genannten Eigenschaft liegen. Deshalb werden zunächst alle Attribute, die `ToChange` als Typ besitzen, ermittelt und anschließend die umgebenden Konzepte bestimmt und zur Menge `Dependents` hinzugefügt. Falls die Instanziierungsdistanz (siehe Definition 4.3) nicht der entsprechenden Regel (Regel C.12 bis Regel C.16) entspricht, muss diese Inkonsistenz behoben werden. Dazu kann entweder die entsprechende Verbindung mittels des jeweiligen Operators (*Delete Partition, Delete Concrete Use Of, Delete Specialization* bzw. *Delete Attribute*) aufgehoben werden oder die invaliden Konzepte müssen auf die nächst höhere Ebene verschoben werden (Operator *Move Type to Upper Level*). Nachdem die Migration vollständig ist, wird der Operator beendet.

Beispiel

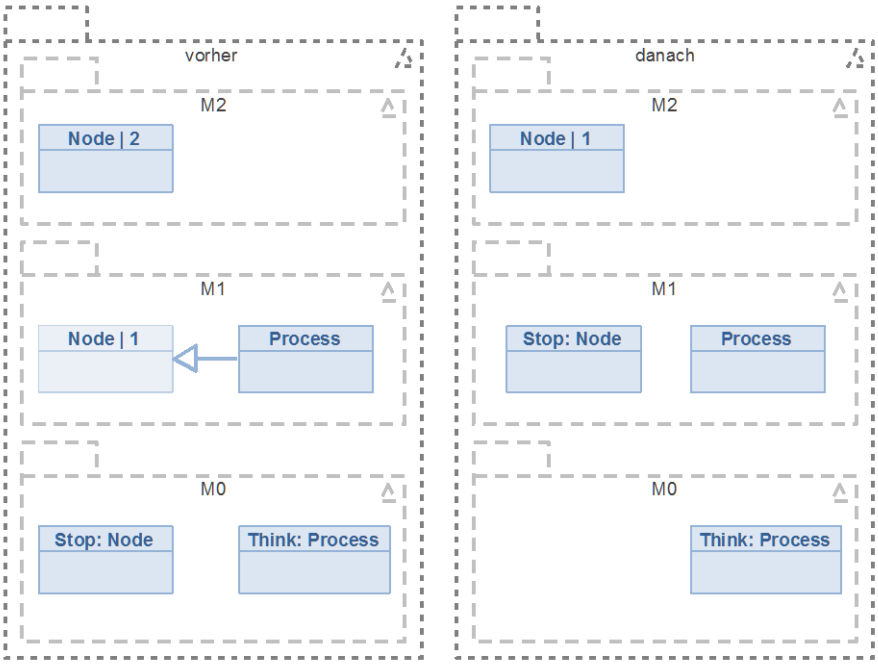

Abbildung 5-9 Beispiel vor (links) und nach (rechts) der Ausführung des Decrement DI Counter of Concept Operators

Auf der linken Seite der Abbildung 5-9 ist eine einfache Meta-Hierarchie gegeben. Auf `M2` wird ein Konzept `Node`[14] mit einem Deep Instantiation Zähler von zwei definiert. In der Ebene `M1` wird `Node` mit dem Konzept `Process` erweitert. Da durch den Deep Instantiation Zähler die Instanziierung verzögert

[14] Die Verwendung der `M2` Ebene in diesem Beispiel ist nicht zwangsweise erforderlich, da `Node` auch auf `M1` definiert werden könnte. Allerdings können solche Situationen dann auftreten, wenn `Node` mit einem anderen Konzept in Beziehung stehen würde, das beispielsweise ein partitionierter Typ ist und damit mindestens zwei Ebene über `M0` liegen müsste. Ein ähnliches Beispiel findet sich im Abschnitt 6.3.1 für das Konzept `Department`.

wurde, kann **Node** auf **M1** spezialisiert werden (Regel C.14). Um dies zu visualisieren wurde **Node** in der Ebene **M1** mit entsprechend verringerten Deep Instantiation Zähler hervorgehoben dargestellt. Auf der untersten Meta-Ebene **M0** wurden zwei Konzept definiert: **Stop** als Instanz von **Node** und **Think** als Instanz von **Process**.

Nun soll der *Decrement DI Counter of Concept* Operator auf **Node** angewendet werden. Zu Beginn, wird der Wert des Deep Instantiation Zählers um eins auf den neuen Wert eins verringert. Anschließend werden alle Instanzen von **Node** berechnet, was die Menge {**Stop**} als Ergebnis liefert. Für alle Elemente dieser Menge, also nur für **Stop**, wird dann entschieden, ob die Instanziierung gelöscht oder das Konzept verschoben werden soll. In unserem Fall entscheiden wir uns für die letzte Variante und verschieben **Stop** nach **M1**. Als nächstes wird nach allen Spezialisierungen, Instanz-Spezialisierungen und Powertypen zusammen mit den Konzepten, die eine Beziehung zu **Node** eingehen, gesucht. Da **M1** die Instanziierungsdistanz eins zu **M2** besitzt, muss **Process** migriert werden. Dies erfolgt in diesem Beispiel mit dem Löschen der Spezialisierung durch den entsprechenden Operator. Das resultierende Meta-Modell ist in Abbildung 5-9 auf der rechten Seite zu sehen.

5.3.5 Set Deep Instantiation Counter of Concept

Auswirkung

Durch den Operator wird der Deep Instantiation Zähler eines Konzeptes gesetzt. Davon betroffene Konzepte werden durch den Operator entsprechend angepasst.

Ablauf

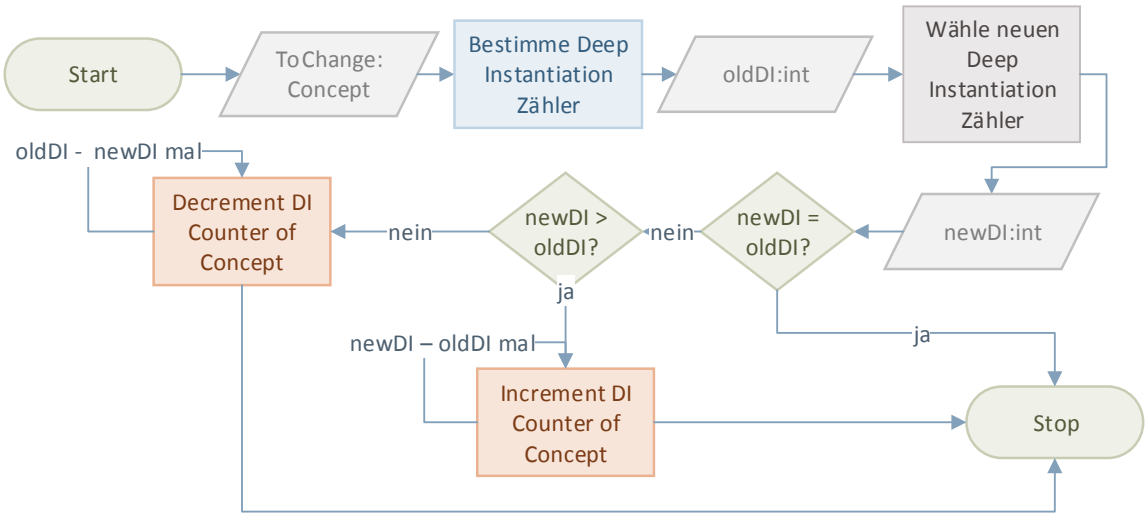

Abbildung 5-10 Ablauf des Set Deep Instantiation Counter Operators

Der Ablauf des Operators ist in Abbildung 5-10 dargestellt. Er wird zu Beginn an einem Konzept **ToChange** aufgerufen, von dem im ersten Schritt der bisherige Deep Instantiation Zähler **oldDI** ermittelt wird. Danach wird der neue Deep Instantiation Zähler **newDI** ausgewählt. Wenn beide Zähler gleich sind, wird der Operator beendet. Anderenfalls wird, falls **newDI** größer als **oldDI** ist, der *Increment DI Counter of Concept* Operator (**newDI-oldDI**)-mal aufgerufen, wodurch schrittweise alle abhängigen Konzepte migriert werden. Wenn **newDI** kleiner als **oldDI** ist, wird entsprechend der *Decrement DI Counter of Concept* Operator **oldDI-newDI**-mal aufgerufen, bevor der Operator endet.

5.3.6 Subroutine: Lösche Zuweisungen bei Instanzen

In diesem Abschnitt soll eine Subroutine vorgestellt werden, die in den nachfolgenden Operatoren häufig Anwendung findet. Um dabei die Diagramme übersichtlich zu gestalten und die Beschreibung nicht redundant zu erstellen, wurde dieser Teilschritt in diesen Abschnitt ausgelagert.

Auswirkung

Die Subroutine löscht alle Zuweisungen, die von Instanzen eines übergebenen Konzeptes **`Base`** an Attribute von **`Type`**, die nicht bei **`SuperType`** vorliegen, erstellt wurden.

Ablauf

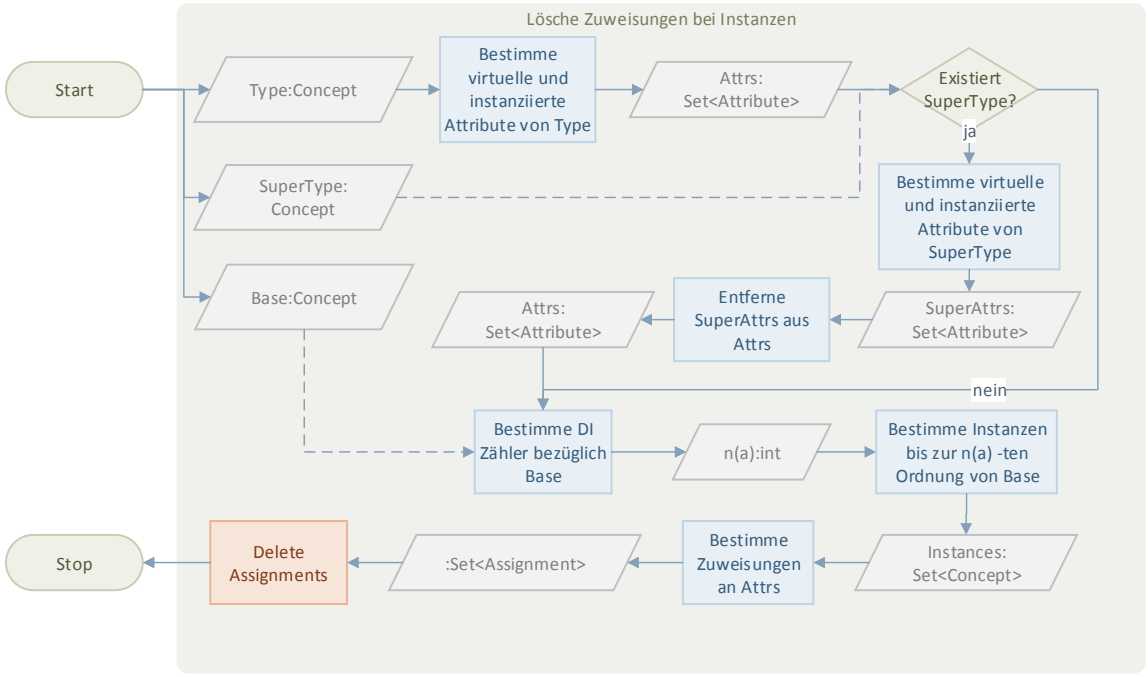

Abbildung 5-11 Ablauf der Subroutine zum Löschen von Zuweisungen bei Instanzen

Die Subroutine (Abbildung 5-11) wird mit drei Konzepten initialisiert (wobei **`SuperType`** optional ist):

- **`Type`** und einer abgeleiteten Generalisierung davon (**`SuperType`**), die dazu dienen die Menge an Attribute festzulegen, deren Zuweisungen gelöscht werden
- und **`Base`**, dessen Instanzen n(a)-ter Ordnung die Zuweisungen gesetzt haben.

Zunächst werden im ersten Schritt alle virtuellen und instanziierten Attribute mit relativen Deep Instantiation Zähler größer als null von **`Type`** (**`Attrs`**) berechnet. Dies entspricht Punkt 1 und 2 in Definition 4.21, wobei in diesem Fall der relative Zähler bezüglich **`Type`** und nicht bezüglich der Instanz berechnet wird, da die konkrete Instanz in diesem Fall (noch) nicht vorliegt. Punkt 3 kann nicht auftreten, da Instanzen n-ter Ordnung immer eine direkte Instanz von **`Type`** als Bezugspunkt haben (siehe Definition 4.10). Für den Fall, dass **`SuperType`** als Parameter gesetzt wurde, wird als nächstes von diesem Konzept ebenfalls die Menge an virtuellen und instanziierten Attributen **`SuperAttrs`** bestimmt und anschließend aus **`Attrs`** entfernt. Die resultierende Menge an Attributen (ebenfalls **`Attrs`** genannt) ist also **`Attrs`** \ **`SuperAttrs`**.[15] Anschaulich betrachtet berechnet die Subroutine also

[15] Wenn **`SuperType`** keine abgeleitete Generalisierung von **`Type`** darstellt, bleibt die Menge von **`Attrs`** unverändert.

alle Attribute, die in einer Vererbungshierarchie (inklusive eventueller partitionierter Typen) in Konzepten zwischen `SuperType` und `Type` definiert wurden. Nachdem nun die Attribute ermittelt wurden, wird für jedes der Deep Instantiation Zähler (`n(a)`) bezüglich `Base`[16] berechnet (Definition 4.18), damit die maximale Ordnung `n(a)` der Instanzen festgelegt ist. Anschließend werden alle Instanzen bis zur `n(a)`-ten Ordnung (Definition 4.10) ermittelt, um danach die Zuweisungen an `Attrs` zu bestimmen[17]. Im letzten Schritt werden diese Zuweisungen noch mit Hilfe des *Delete Assignment* Operators entfernt und die Ausführung der Subroutine endet.

5.3.7 Subroutine: Zuweisungen bereinigen

Auswirkung

Die Subroutine berechnet alle abgeleiteten Generalisierungen eines Konzepts und bestimmt alle Attribute, die als Typ ein solches Konzept besitzen. Alle Zuweisungen an diese Attribute, die einen Wert besitzen, der in der Menge der übergebenen Konzepte liegt, werden von diesem Wert bereinigt.

Ablauf

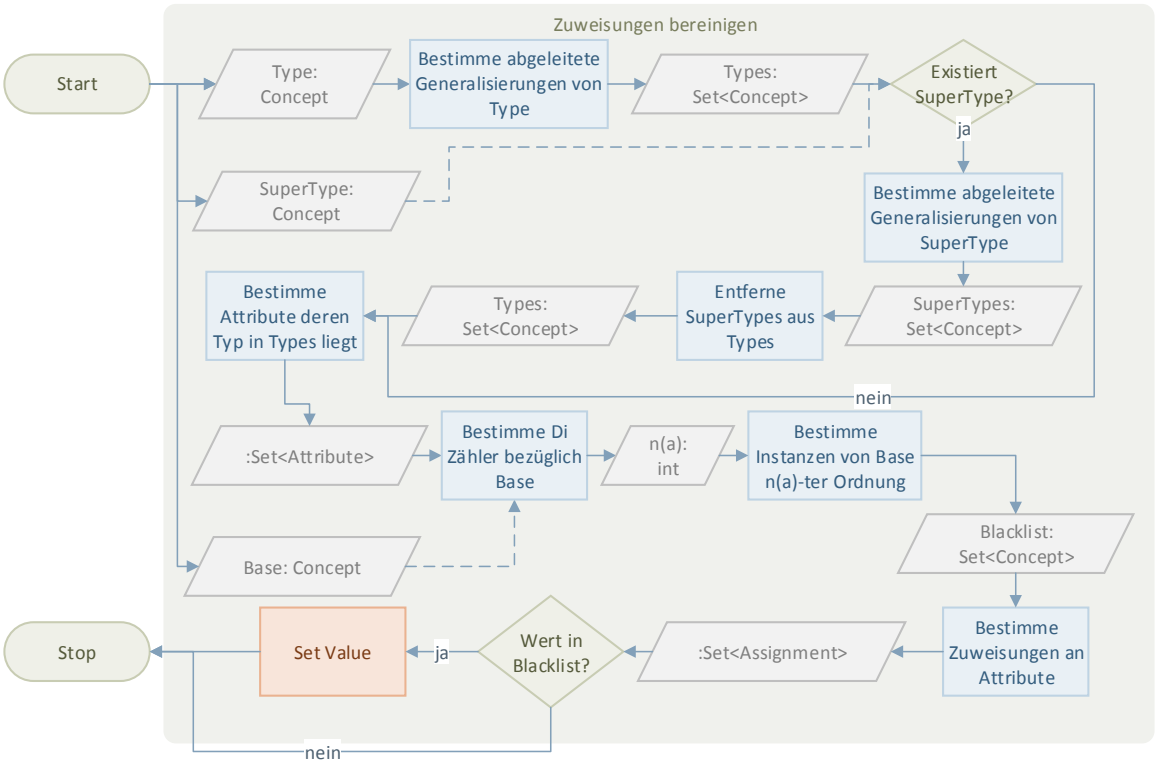

Abbildung 5-12 Ablauf der Subroutine zum Bereinigen der Zuweisungen

Die Subroutine wird mit drei Parametern aufgerufen. Während `Type` und `Base` obligatorisch sind, ist der Parameter `SuperType`, der eine abgeleitete Generalisierung von `Type` darstellt, optional. `Type` wird für die Berechnung der Zuweisungen, die bereinigt werden sollen, benötigt und `Base` dient der Berechnung der Menge an Werten, die durch die Subroutine bei Zuweisungen entfernt wird. Als erster Schritt ermittelt die Subroutine alle abgeleiteten Generalisierungen (`Types`) von `Type`, da alle Attribute,

[16] In diesem Fall ist das umgebende Konzept des Attributs der Bezugspunkt für die Berechnung.

[17] Da Instanzen, die eine kleinere Ordnung als der Deep Instantiation Zähler des Attributes besitzen, ebenfalls das Attribut im Sinne eines Standardwertes setzen können, müssen alle Instanzen mit einer Ordnung kleiner als `n(a)` bestimmt werden.

die einen Typ in dieser Menge besitzen, potentiell ungültige Werte zugewiesen bekommen haben. Wenn der Parameter `SuperType` existiert, werden zunächst auch von diesem die abgeleiteten Generalisierungen bestimmt und anschließend von der Menge `Types`[18] entfernt. Der Parameter dient also dazu bestimmte Bereiche (von `Type` bis `SuperType`) in einer Vererbungshierarchie für die Subroutine festzulegen. Anschließend werden alle Attribute, deren Attributtyp ein Element aus `Types` ist, bestimmt. Als nächstes ermittelt der Operator für alle diese Attribute den Deep Instantiation Zähler bezüglich `Base`[19] (`n(a)`) und berechnet alle Instanzen `n(a)`-ter Ordnung (`Blacklist`). Danach werden alle Zuweisungen an das Attribut ermittelt und dahingehend überprüft, ob ein Wert aus `Blacklist` zugewiesen wurde. Ist dies der Fall wird der *Set Value* Operator aufgerufen, um die Zuweisung zu bereinigen. Nachdem alle Zuweisungen migriert wurden, endet die Subroutine.

5.3.8 Subroutine: Powertyp-Instanz migrieren

Auswirkung

Die Subroutine löscht alle Zuweisungen, die an einer Powertyp-Instanz (oder deren Instanzen) an Attribute des partitionierten Typs gesetzt wurden. Außerdem wird das Substitutionsprinzip für alle Instanzen der Powertyp-Instanz zum partitionierten Typ aufgelöst, indem Zuweisungen bereinigt werden.

Ablauf

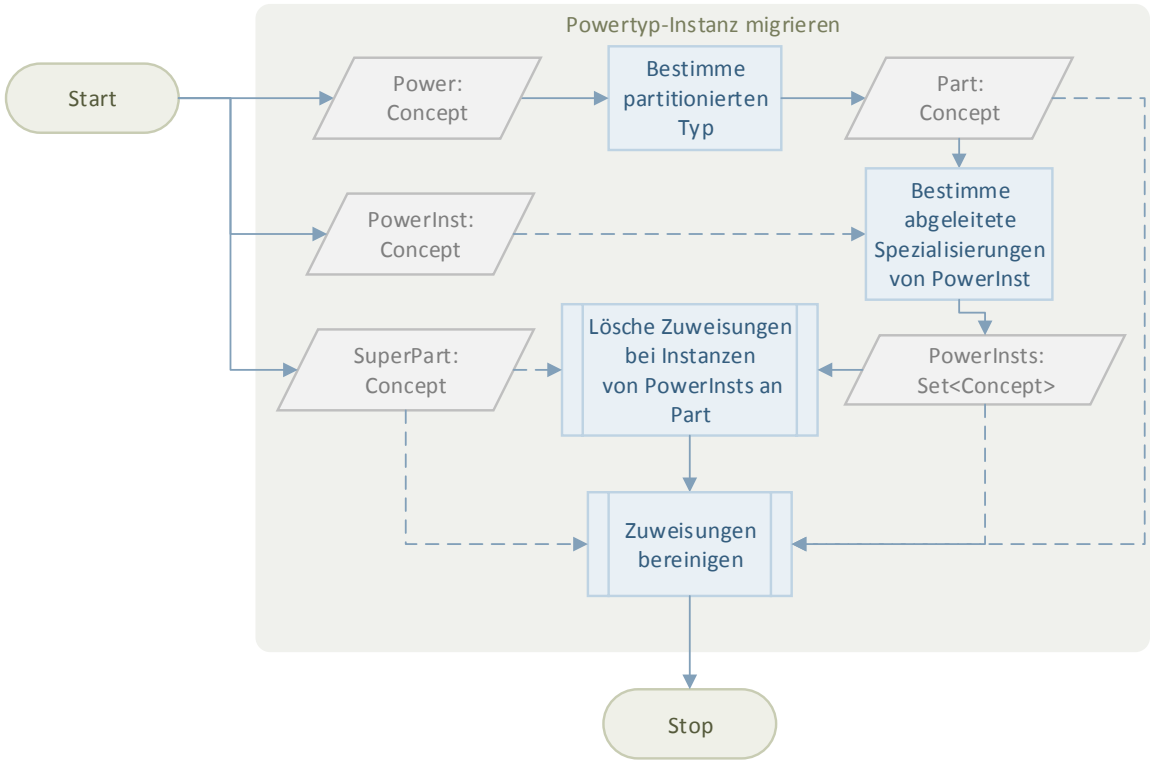

Abbildung 5-13 Ablauf der Powertyp-Instanz migrieren Subroutine

Zu Beginn wird die Subroutine mit drei Konzepten `PowerInst`, dem Powertyp `Power` und einer eventuellen abgeleiteten Generalisierung des partitionierten Typs `SuperPart` initialisiert. Im ersten

[18] Wenn kein Parameter `SuperType` übergeben wird oder dieser keine abgeleitete Generalisierung von `Type` darstellt, bleibt die Menge `Types` unverändert.

[19] In diesem Fall ist der Typ des Attributs der Bezugspunkt für die Berechnung.

Schritt wird zunächst der partitionierte Typ `Part` gesucht. Anschießend werden alle abgeleiteten Spezialisierungen (`PowerInsts`) von `PowerInst` bestimmt. Für jedes Element aus `PowerInsts` (inkl. `PowerInst`) wird die Subroutine aus Abschnitt 5.3.6[20] initialisiert. Dabei wird `Part` als Parameter `Type` und `PowerInst` als Parameter `Base` gesetzt, während der Parameter `SuperType` mit `SuperPart` initialisiert wird (falls dieser nicht leer ist). Damit werden alle Zuweisungen der Instanzen `n(a)`-ter Ordnung von `PowerInsts` an ein Attribut von `Part`, das nicht bei `SuperPart` vorliegt, gelöscht, wobei `n(a)` den jeweilige Wert des Deep Instantiation Zählers eines (virtuellen) Attributes von `Part` bezüglich des jeweiligen Elementes aus `PowerInsts` darstellt. Nachdem die innere Subroutine beendet ist, müssen Zuweisungen, deren Attribut als Typ eine abgeleitete Generalisierung von `Part` besitzt, untersucht werden, ob Instanzen einer gewissen Ordnung eines Elementes aus `PowerInsts` zugewiesen wurden. Deshalb wird die Subroutine aus Abschnitt 5.3.7 ausgeführt, die als `Type` Parameter `Part`, `SuperPart` als `SuperType` Parameter und das jeweilige Element aus `PowerInsts` als Parameter `Base` erhält. Nachdem die innere Subroutine die Zuweisungen bereinigt hat und beendet ist, wird auch die äußere Subroutine gestoppt.

5.3.9 Delete Instantiation

Auswirkung

Durch die Anwendung des Operators wird die Instanziierung eines Konzeptes zu einem anderen Konzept gelöscht. Dabei werden alle ungültigen Zuweisungen und Zuweisungswerte entfernt. Falls es sich beim ehemals instanziierten Konzept um einen Powertypen handelt, werden zusätzlich invalide Zuweisungen an Attribute des partitionierten Typs gelöscht.

Ablauf

Zuweisung und Powertyp Instanz migrieren

Am Anfang wird der Operator (Abbildung 5-14) an einem Konzept `Base` aufgerufen. Optional kann entweder ein weiteres Konzept `SuperType` übergeben werden, das eine abgeleitete Generalisierung des instanziierten Konzepts darstellt, oder `SuperPart`, das eine abgeleitete Generalisierung des partitionierten Typs vom instanziierten Konzept (Powertyp) darstellt. Diese Parameter werden später verwendet, wenn der Operator in anderen Operatoren zur Anwendung kommt. Beim einem manuellen Aufruf (durch einen Benutzer) des Operators bleiben diese Parameter ungesetzt. Im ersten Schritt wird das Konzept `Type` bestimmt, welches von `Base` instanziiert wird. Durch das Entfernen der Instanziierung werden bestimmte Zuweisungen an Attribute von `Type` ungültig. Um diese zu entfernen, wird die Subroutine aus Abschnitt 5.3.6 verwendet. Dabei werden `Type`, `Base` und `SuperType` als Parameter für die Initialisierung der gleichnamigen Parameter der Subroutine verwendet. Die Subroutine berechnet zunächst die ungültigen Zuweisungen durch `Base` oder dessen Instanz-Spezialisierungen und löscht diese anschließend.

Nachdem die Subroutine beendet ist, werden alle ungültigen Werte einer Zuweisung entfernt. Dazu werden `Base` zusammen mit `Type` und `SuperType` als gleichnamiger Parameter der Subroutine *Zuweisungen bereinigen* aus Abschnitt 5.3.7 übergeben. In der Subroutine werden zunächst alle Attribute vom Typ `Type` (oder einer abgeleiteten Generalisierung davon) ermittelt, die eine Zuweisung besitzen, deren Werte in der Menge der Instanzen `n(a)`-ter Ordnung von `Base` liegen. Diese Werte sind nach der Ausführung des Operators ungültig und müssen aus der Zuweisung entfernt werden. Nachdem die Subroutine beendet ist, wird überprüft, ob `Type` ein abgeleiteter Powertyp ist und der

[20] Da in diesem Fall `Base` eine Instanz eines Powertyps ist und damit eine Instanz-Spezialisierung nicht möglich ist (Regel C.15), besteht die Menge der Instanzen 0-ter Ordnung lediglich aus `Base`.

`SuperType` Parameter leer ist, da sonst die Spezialisierung zum partitionierten Typ nicht migriert werden muss. Wenn die Bedingung zutrifft, wird der abgeleitete Powertyp `Power` bestimmt und als Parameter `Power` zusammen mit `SuperPart` als gleichnamiger Parameter und `Base` als Parameter `PowerInst` der Subroutine aus Abschnitt 5.3.8[21] übergeben, damit diese die Instanzen des Powertyps migriert.

Elementare Änderung

Daran anschließend und für den Fall, dass `Type` kein Powertyp ist, wird nun die Instanziierung zwischen `Base` und `Type` gelöscht und der Operator endet.

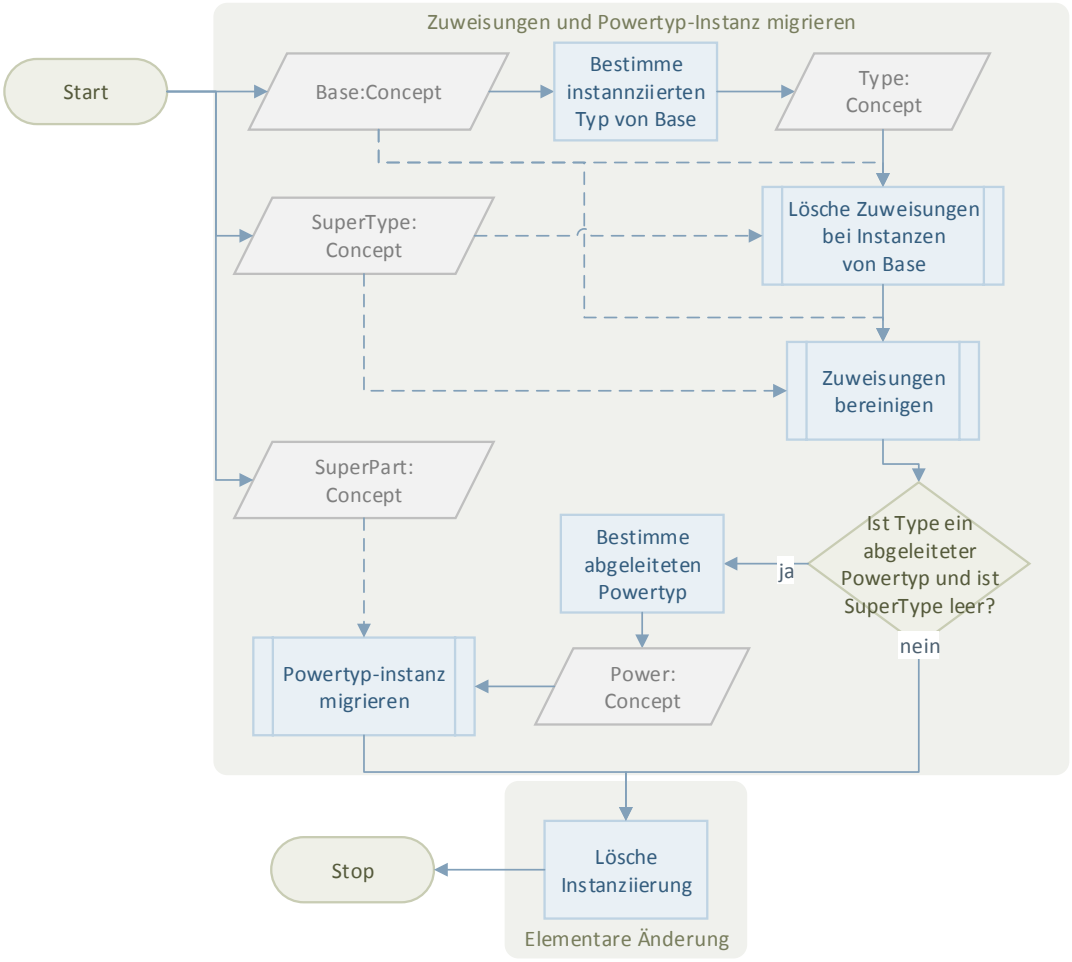

Abbildung 5-14 Ablauf der Delete Instantiation Operators

Beispiel

In Abbildung 5-15 ist auf der linken Seite ein einfaches Modell dargestelt, das ein kleines Unternehmen abbildet. Die Ebene `M2` wird (bis auf einige Ausnahmen) bereits im Beispiel des *Move Attribute to Powertype Instance* Operators beschrieben. Die einzige Änderung, die vorgenommen wurde, besteht darin, dass nun noch ein Attribut `workedHours` bei `Employee` definiert wurde und dass der Powertyp `EmployeeKind` nun noch ein Diskriminatorattribut für `workedHours` (`supportsWH`) besitzt.

[21] Da in diesem Fall `Base` eine Instanz eines Powertyps ist und damit eine Instanz-Spezialisierung nicht möglich ist (Regel C.15) besteht die Menge der Instanzen 0-ter Ordnung lediglich aus `Base.`

Auf der Ebene `M1` wurden zwei Konzepte definiert, `Cleaning` und `Manager`, die beide Instanzen von `EmployeeKind` sind. Erst genanntes Konzept steht für das Reinigungspersonal, während `Manager` einen Manager im Unternehmen modellieren soll. Da Manager ein Festgehalt besitzen (hier nicht modelliert), ist bei ihnen die Anzahl der gearbeiteten Stunden irrelevant. Folglich haben sie `supportsWH` auf `false` gesetzt, wohingegen `Cleaning` das Attribut `workedHours` erbt, da der entsprechende Wert `true` ist. Weiterhin erben beide Konzepte `worksFor` von `Employee`, da sie die entsprechenden Werte für das Diskriminatorattribut `supportsWF` gesetzt haben. In der Ebene `M0` ist ein kleines Beispielmodell für diese Sprache zu finden. Es wurden die Konzepte `Alice` und `Bob` modelliert, von denen erstgenannte ein Manager (`Alice` ist Instanz von `Manager`) ist und der andere zum Reinigungspersonal (`Bob` ist Instanz von `Cleaning`) zählt. Beide arbeiten in der Büroabteilung, was durch das Konzept `Office` (Instanz von `Department`) und die entsprechenden Werte für die Attribute `worksFor` bei `Alice` und `Bob` sowie `staff` bei `Office` modelliert wurde. Zudem hat `Bob` in diesem Monat bereits 55 Stunden für das Unternehmen gearbeitet, da `workedFor` den Wert `55` bei `Bob` besitzt.

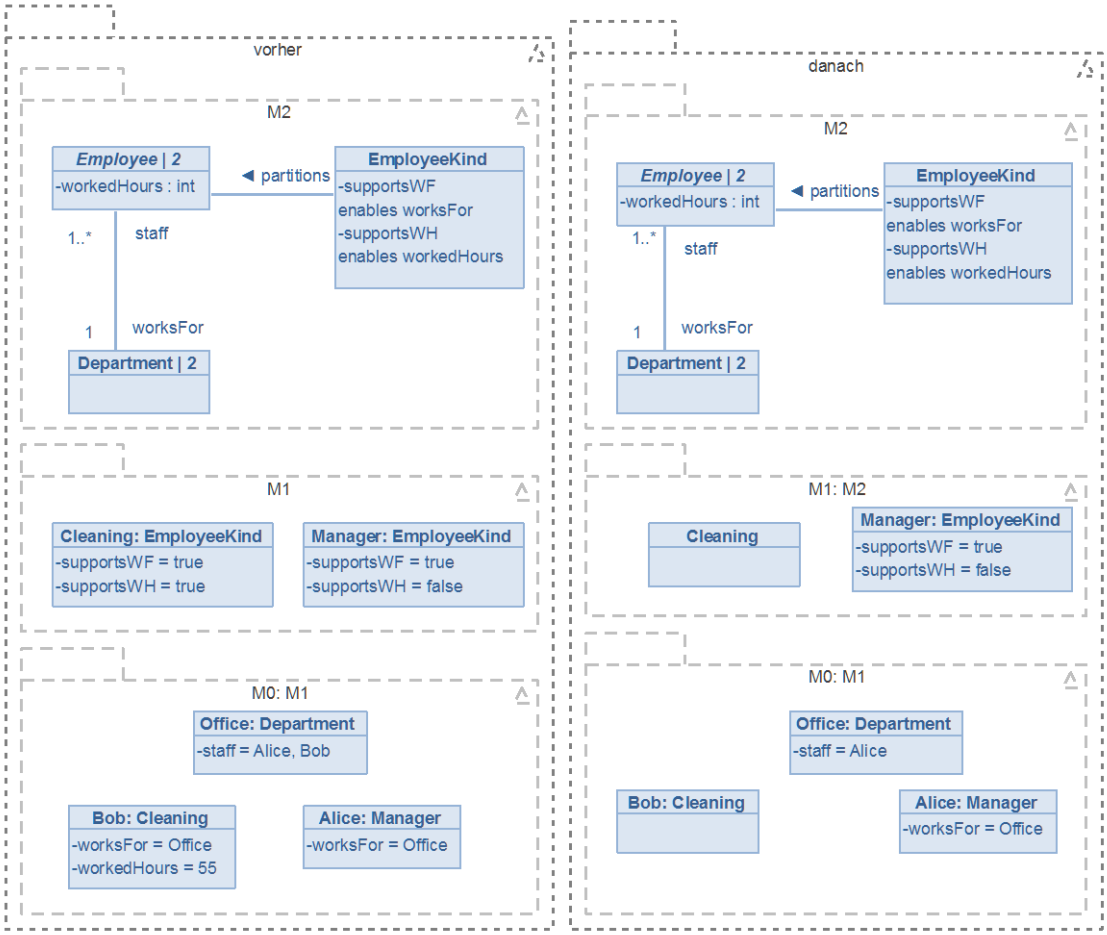

Abbildung 5-15 Beispiel vor (links) und nach (rechts) der Ausführung des Delete Instantiation Operators

Nun soll sich im Beispiel die modellierte Domäne ändern: Die Reinigung wird ausgelagert und soll nun von einem externen Unternehmen, das in diesem Fall die Mitarbeiter dieser Abteilung übernimmt, durchgeführt werden. Folglich sind nun alle Personen des Reinigungspersonals keine Mitarbeiter mehr und die Instanziierung von `Cleaning` zu `EmployeeKind` soll mit Hilfe des *Delete Instantiation* Operators gelöscht werden. Nachdem der Operator an `Cleaning` aufgerufen wurde (die Parameter `SuperType` und `SuperPart` sind nicht gesetzt), wird der instanziierte Typ, in diesem Fall `EmployeeKind`, bestimmt. Im nächsten Schritt werden die Zuweisungen von `Cleaning` (und eventuell

dessen Instanzen) zu Attributen des Typs `EmployeeKind` durch das Aufrufen der Subroutine aus Abschnitt 5.3.6 gelöscht. Dabei wird der `Type` Parameter der Subroutine mit `EmployeeKind` und der `Base` Parameter mit `Cleaning` initialisiert. Dadurch werden zunächst die Diskriminatorattribute `supportsWF` und `supportsWH` berechnet. Da diese einen Deep Instantiation Zähler von `0` bezüglich `Cleaning` haben[22], werden alle Instanzen 0-ter Ordnung von diesem Konzept berechnet. Aufgrund der Tatsache, dass `Cleaning` keine Instanz-Spezialisierungen besitzt, besteht diese Menge nur aus `Cleaning` selbst. Folglich werden im nächsten Schritt die beiden Zuweisungen von `Cleaning` ermittelt und anschließend durch den *Delete Assignment* Operator gelöscht, womit die Ausführung der Subroutine endet. Danach wird die Subroutine zum *Bereinigen der Zuweisungen* aus Abschnitt 5.3.7 parametriert (`EmployeeKind` als `Type`, `Cleaning` als `Base`). Da `EmployeeKind` keine weiteren abgeleiteten Generalisierungen besitzt und kein Attribut vom Typ `EmployeeKind` existiert, werden in der Subroutine keine Zuweisungen bereinigt und sie stoppt.

Da `EmployeeKind` ein Powertyp ist, wird als nächstes die Subroutine zur Migration der Powertyp-Instanzen (Abschnitt 5.3.8) ausgeführt. Dabei ist `Cleaning` der Aktualparameter für `PowerInst` und `EmployeeKind` der für `Power`. Zu Beginn bestimmt die Subroutine `Employee` als partitionierter Typ. Danach werden alle abgeleiteten Spezialisierungen von `Cleaning` ermittelt. Da diese Menge lediglich aus `Cleaning` besteht, wird die Subroutine zum Löschen der Zuweisungen aus Abschnitt 5.3.6 nur für dieses Konzept aufgerufen. Diese bestimmt zunächst alle virtuellen Attribute von `Employee`, also `workedHours` und `worksFor`. Weil diese jeweils eine Deep Instantiation Zähler von `1` haben und `Cleaning` eine Instanz des Powertyps (und damit eine Spezialisierung von `Employee`[23]) ist, müssen alle Instanzen, deren Ordnung kleiner als eins ist, von `Cleaning` berechnet werden. Die Menge besteht also aus allen Instanzen 0-ter Ordnung (= {`Cleaning`}) und den Instanzen 1-ter Ordnung (= {`Bob`}). Im nächsten Schritt der inneren Subroutine werden alle Zuweisungen von `Cleaning` und `Bob` an die Attribute `worksFor` und `workedHours` gesucht, um diese zu entfernen. Die beiden Zuweisungen bei `Bob` werden folglich im nächsten Schritt durch den *Delete Assignment* Operator gelöscht und die Subroutine zum Löschen der Zuweisungen stoppt.

Nun wird für alle Elemente aus `PowerInsts` die Subroutine aus Abschnitt 5.3.7 zur Bereinigung der Zuweisungen aufgerufen. Für unser Beispiel entsteht also eine Parametrierung mit `Employee` als `Type` und `Cleaning` als `Base` Parameter. Die innere Subroutine berechnet zunächst alle abgeleiteten Generalisierungen von `Employee`. Da außer `Employee` keine weitere abgeleitete Generalisierung existiert, werden alle Attribute vom Typ `Employee` im nächsten Schritt der inneren Subroutine ermittelt. Das führt zum Ergebnis, dass `staff` ermittelt wird und alle Instanzen 1-ter Ordnung von `Cleaning` (`Blacklist` besteht damit nur aus `Bob`) berechnet werden. Im nächsten Schritt wird die Zuweisung von `Office` an `staff` bestimmt. Nun werden die Werte dieser Zuweisung untersucht, ob diese in der Menge `Blacklist` liegen. Dies trifft auf `Bob` zu, weshalb dieser Wert von der Zuweisung entfernt wird. Daran anschließend ist die innere Subroutine zum Bereinigen der Zuweisungen und die äußere zum Migrieren der Powertyp-Instanzen beendet. Im letzten Schritt entfernt der Operator die Instanziierung von `Cleaning` zu `EmployeeKind` und das resultierende Modell (Abbildung 5-15 auf der rechten Seite) entsteht.

[22] Da der Deep Instantiation Zähler bei der Definition der Attribute eins beträgt und `Cleaning` eine Instanz des Konzeptes `EmployeeKind` ist, an dem die Attribute definiert wurden, beträgt der Wert des relativen Zählers null.
[23] Durch die Spezialisierung ist der Deep Instantiation Zähler bezüglich `Cleaning` gleich dem Deep Instantiation Zähler des Attributes.

5.3.10 Delete Specialization

Auswirkung

Der Operator entfernt eine Spezialisierung eines Konzepts zu einer Generalisierung. Dabei werden alle ungültigen Zuweisungen migriert.

Ablauf

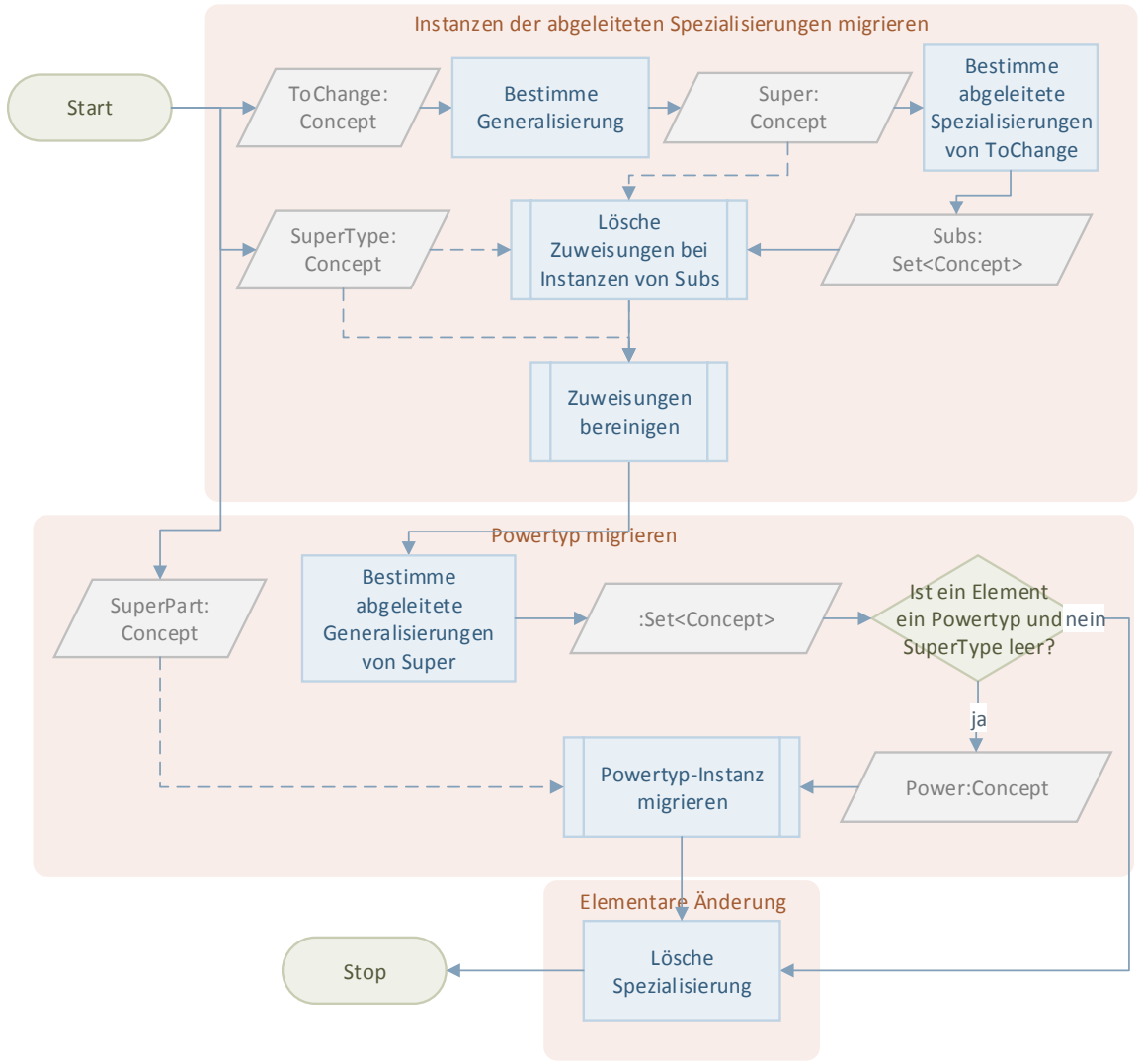

Abbildung 5-16 Ablauf des Delete Specialization Operators

Instanzen der abgeleiteten Spezialisierungen migrieren

Der Operator erhält drei Parameter: **ToChange** als zu änderndes Konzept, sowie **SuperType** und **SuperPart**, die beide optional sind und bei einem manuellen Aufruf (z.B. durch den Benutzer) leer sind. **SuperType** stellt dabei eine abgeleitete Generalisierung von **ToChange** dar, während **SuperPart** eine abgeleitete Generalisierung eines möglichen partitionierten Typs von **ToChange** ist. Zu Beginn werden vom übergebenen Konzept **ToChange** die Generalisierung **Super** bestimmt und alle abgeleiteten Spezialisierungen **Subs** von **ToChange** berechnet. Danach wird die Subroutine aus Abschnitt 5.3.6 für jedes Element aus **Subs** inklusive **ToChange** (Parameter **Base**) mit **Super** als Parameter **Type** und **SuperType** als gleichnamiger Parameter initialisiert und ausgeführt. Dadurch werden alle Zuweisungen bei allen Instanzen (bis zu einer gewissen Ordnung) von **Subs** bzw. **ToChange** gelöscht, die ein Attribut von **Super** zuweisen, welches bei **SuperType** nicht vorhanden ist.

Durch das Löschen der Vererbung von `ToChange` zu `Super` geht das Substitutionsprinzip für `ToChange` und dessen abgeleiteten Spezialisierungen verloren. Deshalb dürfen bei allen Attributen, deren Typ `Super` oder eine abgeleitete Generalisierung davon (unterhalb von `SuperType` in der Vererbungshierarchie) ist, keine Instanzen von diesen Konzepten mehr zugewiesen werden. Um diese Inkonsistenz zu beheben, wird die Subroutine aus Abschnitt 5.3.7 für jedes Elemente aus `Subs` (inkl. `ToChange`) ausgeführt (mit `Super` als Parameter `Type`, `SuperType` als gleichnamiger Parameter und das jeweilige Element aus `Subs` als `Base` Parameter).

Powertyp migrieren[24]

Daran anschließend muss festgestellt werden, ob `ToChange` ein abgeleiteter Powertyp ist (Definition 4.7). Dies ist dann der Fall, wenn `Super` oder eine abgeleitete Generalisierung davon ein Powertyp ist. Wenn dem so ist und `SuperType` nicht übergeben wurde, hat das Entfernen der Spezialisierung zur Folge, dass die Instanzen von Elementen aus `Subs` und damit auch die von `ToChange` (`Instances`) keine Powertyp-Instanzen mehr darstellen, d.h. den partitionierten Typ nicht mehr erweitern. Dadurch werden Zuweisungen an Attribute des partitionierten Typs invalide. Deshalb werden zunächst alle abgeleiteten Generalisierungen[25] von `Super` ermittelt. Wenn eine davon ein Powertyp (dieses Element so im Folgenden `Power` heißen) ist, wird die Subroutine aus Abschnitt 5.3.8 für jedes Element aus `Instances` aufgerufen (mit `Power` als Parameter `Power`, dem jeweiligen Element aus `Instances` als Parameter `PowerInst` und `SuperPart` als gleichnamiger Parameter). Damit werden bei allen Instanzen von `Instances` die Zuweisungen an alle Attribute von `Part`, die bei `SuperPart` nicht vorhanden sind, gelöscht. Außerdem werden alle Zuweisungen, die diese Konzepte als Wert gesetzt haben, bereinigt, falls der Typ des entsprechenden Attributes zwischen `Part` (eingeschlossen) und `SuperPart` (ausgeschlossen) in der Vererbungshierarchie liegt.

Elementare Änderung

Nach der Ausführung der Subroutine wird im letzten Schritt noch die Spezialisierung von `ToChange` zu `Super` entfernt und die Ausführung des Operators ist beendet.

Beispiel

Abbildung 5-17 zeigt auf der linken Seite ein einfaches Modell eines Unternehmens. Auf der Ebene `M1` wurden fünf Konzepte erstellt. Dabei stellen die Konzepte `Person`, `Employee`, `Manager`, `GeneralManager` und `Department` Personen, Mitarbeiter, Manager, Geschäftsführer und Abteilungen des Unternehmens dar. Die Konzepte `Person` und `Employee` definieren jeweils ein Attribut, welche zum einen den Namen (`name`) einer Person und zum anderen das Gehalt (`salary`) eines Mitarbeiters darstellen. Konzept `GeneralManager` ist eine Spezialisierung von `Manager`, das eine Spezialisierung von `Employee` ist, welches wiederum `Person` spezialisiert. Außerdem besitzt `Department` über das Attribut `employees` eine Beziehung zu `Employee`. In der Ebene `M0` wurde ein Beispielmodell für die durch `M1` definierte Sprache erstellt. Darin ist `Alice` eine Instanz von `Manager` und besitzt den Namen **„Alice“** (Zuweisung an `name`) und ein Gehalt von `100 000` € (Zuweisung an `salary`) jährlich. Weiterhin besitzen `Bob` als Instanz von `Employee` den Namen **„Bob**“ sowie ein Gehalt von `50 000` € und `Semor` als Instanz von `GeneralManager` den Namen **„Semor“** und ein Gehalt von

[24] Dieser Teilschritt stimmt im Wesentlichen mit dem gleichnamigen Teilschritt aus Abschnitt 5.3.11 (Operator *Delete Partition*) überein und könnte für die Umsetzung des Operators in einem Modellierungssystem ebenfalls als Subroutine ausgelagert werden. Um die Subroutinen nicht zu weit zu verschachteln, wird an dieser Stelle darauf verzichtet.

[25] Die abgeleiteten Generalisierungen wurden bereits im vorherigen Schritt in der Subroutine 5.3.7 ermittelt und können an dieser Stelle wiederverwendet werden. Um die Funktionsweise klar zu verdeutlichen, wurde der Schritt jedoch erneut ins Diagramm mit aufgenommen.

`120 000` €. Zusätzlich wurde noch eine Instanz von `Department` (`Sales`) modelliert, das als Mitarbeiter `Alice`, `Bob` und `Semor` hat (Zuweisung an `employees`).

Im Nachfolgenden soll die Spezialisierung von `Manager` zu `Employee` mit Hilfe des *Delete Specialization* Operators entfernt werden (`SuperPart` und `SuperType` Parameter bleiben leer). Dazu wird zunächst die Generalisierung von `Manager` ermittelt, was als Ergebnis `Employee` liefert. Danach werden alle abgeleiteten Spezialisierungen von `Manager` (`Subs` = {`Manager`, `GeneralManager`}) berechnet. Als nächstes wird die Subroutine aus Abschnitt 5.3.6, mit `Employee` als `Type` Parameter, jeweils für `Manager` und `GeneralManager` (als `Base` Parameter) ausgeführt. Diese berechnet zunächst alle virtuellen und instanziierten Attribute von `Employee`, also `name` und `salary` und berechnet danach für beide den Deep Instantiation Zähler bezüglich `Manager` bzw. `GeneralManager`. Da `salary` und `name` bei einer abgeleiteten Generalisierung von `Manager` bzw. `GeneralManager` definiert wurden und beide eine Deep Instantiation Zähler von `1` haben, ist die maximale Ordnung aller zu untersuchende Instanzen ebenfalls eins. Folglich besteht die Menge `Instances` für `Manager` aus `Manager` (Instanzen 0-ter Ordnung) und `Alice` (Instanzen 1-ter Ordnung), während im Durchlauf für `GeneralManager` das Konzept selbst sowie `Semor` in der Menge enthalten sind. In jeder Ausführung der Subroutine werden für beiden Konzepte die Zuweisungen an `name` und `salary` gesucht. Danach werden die beiden Zuweisungen von `Alice` bzw. `Semor` an die beiden Attribute gelöscht (`Manager` und `GeneralManager` besitzen keine Zuweisungen an die Attribute), und die Subroutine endet.

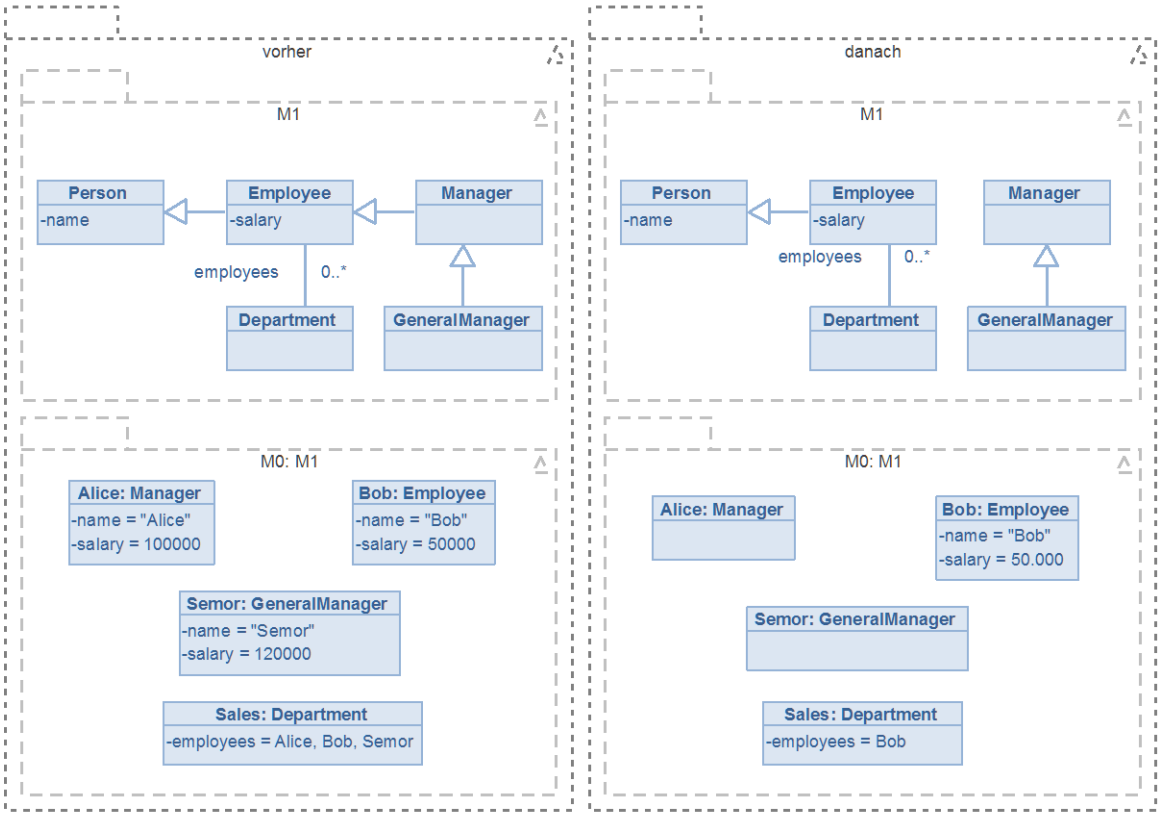

Abbildung 5-17 Unternehmensmodell vor (links) und nach (rechts) der Ausführung des Delete Specialization Operators

Als nächstes werden alle Instanzen bestimmter Ordnung von `Manager` und `GeneralManager` durch die Subroutine zum Bereinigen der Zuweisungen (Abschnitt 5.3.7) als Zuweisungswerte an Attribute vom Typ `Employee` oder `Person` entfernt. Demzufolge werden die beiden Konzepte (`Manager`,

GeneralManager) jeweils als **Base** zusammen mit **Employee** als **Type** Parameter übergeben. Anschließend werden alle Attribute, die als Typ **Employee** oder **Person** haben ermittelt. Dies sind in diesem Beispiel **employees** am Konzept **Department**. Anschließend werden die Instanzen 1-ter Ordnung[26] von **Manager** ({**Alice**}) bzw. **GeneralManager** ({**Semor**}) ermittelt. Danach wird die Zuweisung bei **Sales** (Instanz von **Department**) gesucht und um den Wert **Alice** und **Semor** bereinigt. Da weder **Person** noch **Employee** ein Powertyp sind, wird als letztes noch die Spezialisierung von **Manager** zu **Employee** gelöscht und das in Abbildung 5-17 auf der rechten Seite gezeigte Modell entsteht.

5.3.11 Delete Partition

Auswirkung

Durch den Operator wird die **partitions** Beziehung des Powertyps zum partitionierten Typ gelöscht. Zusätzlich wird die implizite Spezialisierung der Instanzen des Powertyps zum partitionierten Typ aufgelöst und invalide Zuweisungen angepasst.

Ablauf

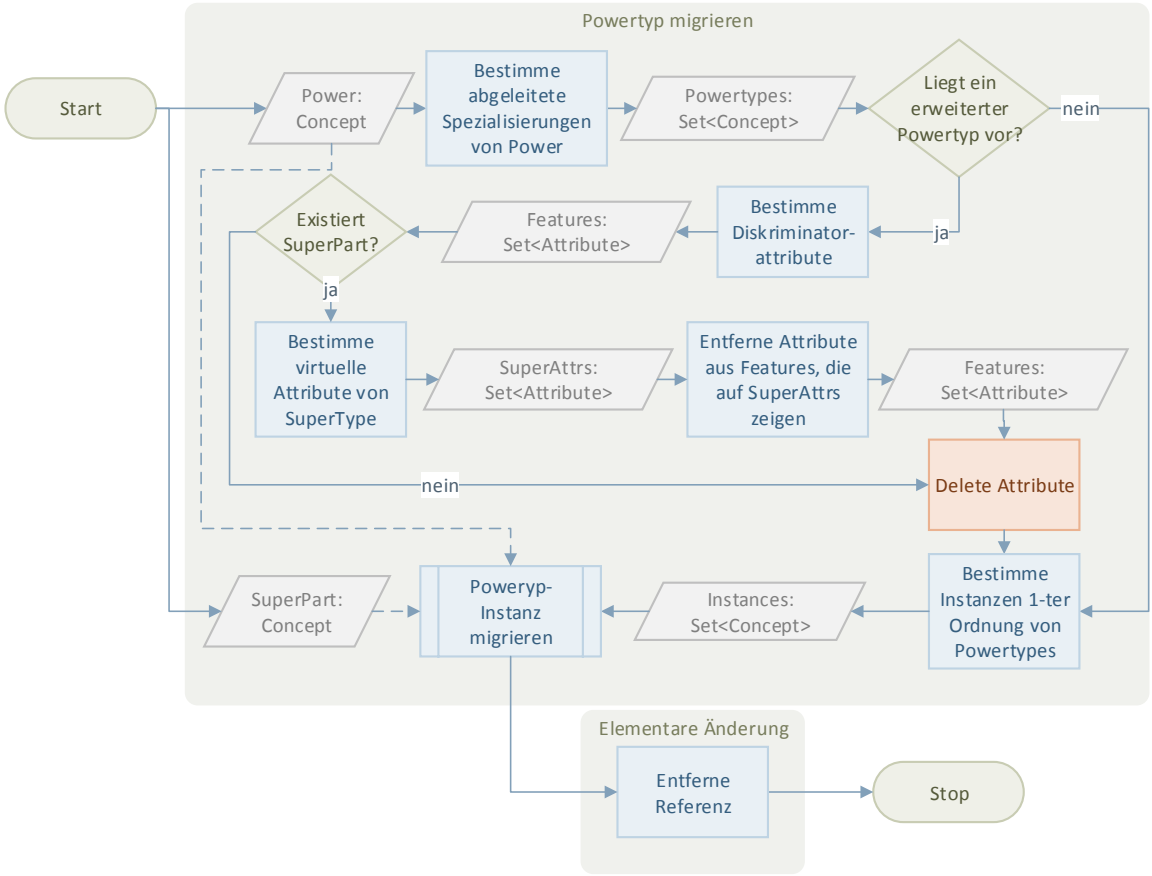

Abbildung 5-18 Ablauf des Delete Partition Operators

Powertyp migrieren

Der Operator wird an einem Powertyp **Power** aufgerufen. Optional kann noch ein zweiter Parameter **SuperPart** übergeben werden, der eine abgeleitete Generalisierung des partitionierten Typs von

[26] Da **Manager** und **Semor** eine abgeleitete Spezialisierung vom umgebenden Konzept von **employees** (**Employee**) sind, wird entsteht als relativer Wert der am Attribut definierte Deep Instantiation Zähler.

Power darstellt. Bei einem manuellen Aufruf des Operators (z.B. durch den Benutzer) bleibt dieser Parameter leer. Als erstes werden alle abgeleiteten Spezialisierungen des Powertyps (Powertypes) bestimmt. Falls Power (oder ein anderes Element aus Powertypes) ein erweiterter Powertyp ist, müssen die Diskriminatorattribute (Features) gelöscht werden. Deshalb werden in diesem Fall zunächst alle Diskriminatorattribute gesucht. Wenn SuperPart existiert, sind nicht alle Diskriminatorattribute ungültig, da diejenigen, die auf virtuelle Attribute von SuperPart zeigen, erhalten bleiben sollen. Demzufolge werden in diesem Fall zunächst alle virtuellen Attribute von SuperPart gesucht und im nächsten Schritt die entsprechenden Diskriminatorattribute aus Features entfernt, damit sie später nicht gelöscht werden. Anschließend werden alle (verblieben) Attribute aus Features durch den *Delete Attribute* Operator gelöscht. Danach ermittelt der Operator alle Instanzen 1-ter Ordnung (Instances) jedes Elements aus Powertypes, um anschließend für alle die Subroutine zur Migration einer Powertyp-Instanz aus Abschnitt 5.3.8 aufzurufen. Dabei wird neben der konkreten Instanz als PowerInst Parameter auch Power und SuperPart als gleichnamiger Parameter der Subroutine übergeben. Durch Die Subroutine wird die Spezialisierung der Powertyp-Instanzen zum partitionierten Typ aufgelöst, da diese nach dem Löschen der partitions Beziehung nicht mehr besteht. Deshalb werden alle Zuweisungen an Attribute von Part bei allen Instanzen der Powertyp-Instanzen (Instances) entfernt und Zuweisungen um ungültige Werte bereinigt, die dadurch entstehen, dass das Substitutionsprinzip einer Powertyp-Instanz zum partitionierten Typ nicht mehr gilt.

Elementare Änderung

Bevor der Operator beendet ist, muss noch im letzten Schritt die partitions Referenz von Power zu Part entfernt werden.

Beispiel

In Abbildung 5-19 ist auf der linken Seite ein Beispiel dargestellt, das dem Beispiel aus Abschnitt 2.4 zum erweiterten Powertyp Muster ähnelt. Die Beschreibung kann demzufolge in großen Teilen dort gefunden werden. Allerdings wurden auch einige Gegebenheiten ergänzt. So definiert der Powertyp PersonKind auf M2 nun zusätzlich ein Attribut description, das zur Beschreibung des jeweiligen Konzepts dient. Weiterhin wurde dieses Attribut bei allen Instanzen von PersonKind auf der M1 Ebene gesetzt. Außerdem existiert nun noch eine Ebene M0, in der ein Konzept Alice als Instanz von PhDStudent modelliert ist, bei dem die Attribute semester und salary gesetzt sind. Dadurch kommt zum Ausdruck, dass Alice im 17. Semester an der Universität ist und ein monatliches Gehalt von 3937,21 € erhält.

Nun soll der *Delete Partition* Operator am Konzept PersonKind aufgerufen werden. Der Parameter SuperPart bleibt, da es sich um einen manuellen Aufruf handelt, ungesetzt. Da PersonKind keine abgeleiteten Spezialisierungen besitzt, wird zunächst festgestellt, dass das Konzept ein erweiterter Powertyp ist. Deshalb werden als nächstes die Diskriminatorattribute supportsSemester und supportsSalary gesucht und anschließend gelöscht. Dadurch werden die Zuweisungen bei den Powertyp-Instanzen Student, Lecturer und PhDStudent an die Diskriminatorattribute ebenfalls entfernt. Danach werden Student, Lecturer und PhDStudent als Instanzen 1-ter Ordnung ermittelt, um für jedes die Subroutine aus Abschnitt 5.3.8 mit PersonKind als Power Parameter auszuführen. Da in diesem Beispiel nur PhDStudent eine Instanz besitzt, bleiben die Subroutinen für Lecturer und Student ohne Folgen. Für PhDStudent wird zunächst der partitionierte Typ von PersonKind also Person bestimmt. Da keine weiteren abgeleiteten Spezialisierungen von PhDStudent existieren, wird die innere Subroutine zum *Löschen der Zuweisungen bei Instanzen* aus Abschnitt 5.3.6 nur für PhDStudent und Person als Type Parameter initialisiert. Anschließend werden in der inneren Subroutine mit salary und semester alle virtuellen Attribute von Person berechnet und es werden

alle Instanzen 1-ter (= `Alice`) und 0-ter Ordnung (=`Person`) gesucht.[27] Danach werden die Zuweisungen von `Alice` an `salary` und `semester` gelöscht und da `Person` keine Zuweisungen definiert, endet die innere Subroutine. Als nächstes wird in der Subroutine zur Migration der Powertyp-Instanzen die Subroutine zum Bereinigen der Zuweisungen ausgeführt. Aufgrund der Tatsachen, dass `Person` keine abgeleiteten Generalisierungen besitzt und keine Attribute vom Typ `Person` existieren, stoppt diese innere und die äußere Subroutine. Im letzten Schritt des Operators wird die `partitions` Beziehung von `PersonKind` zu `Person` entfernt und das rechte Modell aus Abbildung 5-19 entsteht.

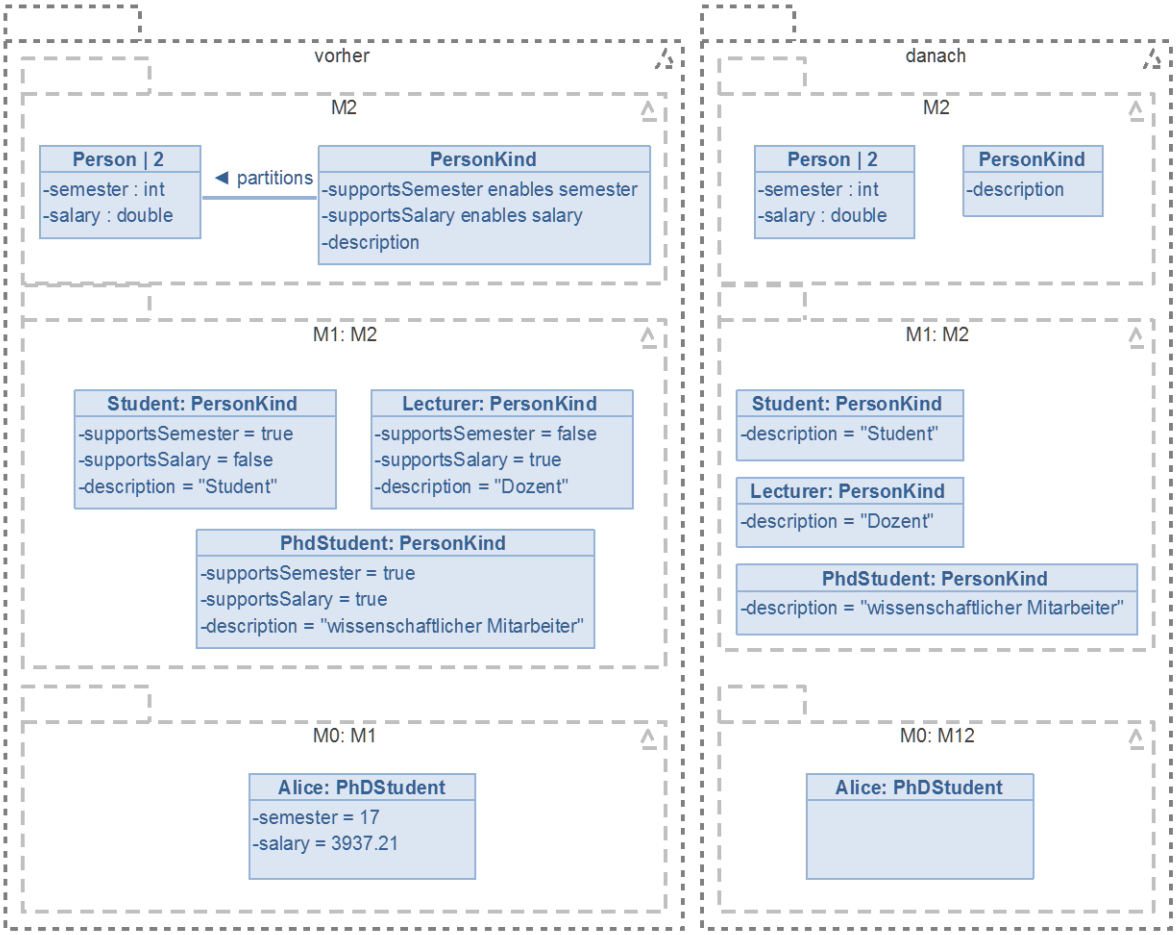

Abbildung 5-19 Beispiel vor (links) und nach (rechts) der Ausführung des Delete Partition Operators

5.3.12 Delete Concrete Use Of

Auswirkung

Die Ausführung des Operators an einem Konzept bewirkt, dass die Instanz-Spezialisierung gelöscht, die Instanziierung zum Typen des Prototyps hergestellt und die Werte für die Zuweisungen des Konzepts berechnet und neu gesetzt werden.

[27] `PhDStudent` ist eine Powertyp-Instanz und daher eine Spezialisierung von `Person`. Damit ist der relative Deep Instantiation Zähler von `salary` und `semester` bezüglich `PhDStudent` eins.

Ablauf

Instanz-Spezialisierung migrieren

Am Anfang wird der Operator (Abbildung 5-20) an einem Konzept **`Base`** aufgerufen, von dem der Prototyp entfernt werden soll. Dazu muss dieser zunächst im darauffolgenden Schritt berechnet werden. Das resultierende Konzept **`Prototype`** wird dann verwendet, um dessen abgeleiteten instanziierten Typ (**`Type`**) festzustellen. Dabei wird für den Fall, dass **`Prototype`** keine direkte Instanziierung besitzt der abgeleitete Typ (instanziierten Typ des Prototyps von **`Prototype`**, usw.) berechnet. Anschließend wird die Instanziierung von **`Base`** zu **`Type`** erstellt.

Elementare Änderung

Als nächstes wird die Instanz-Spezialisierung von **`Base`** zu **`Prototype`** entfernt.

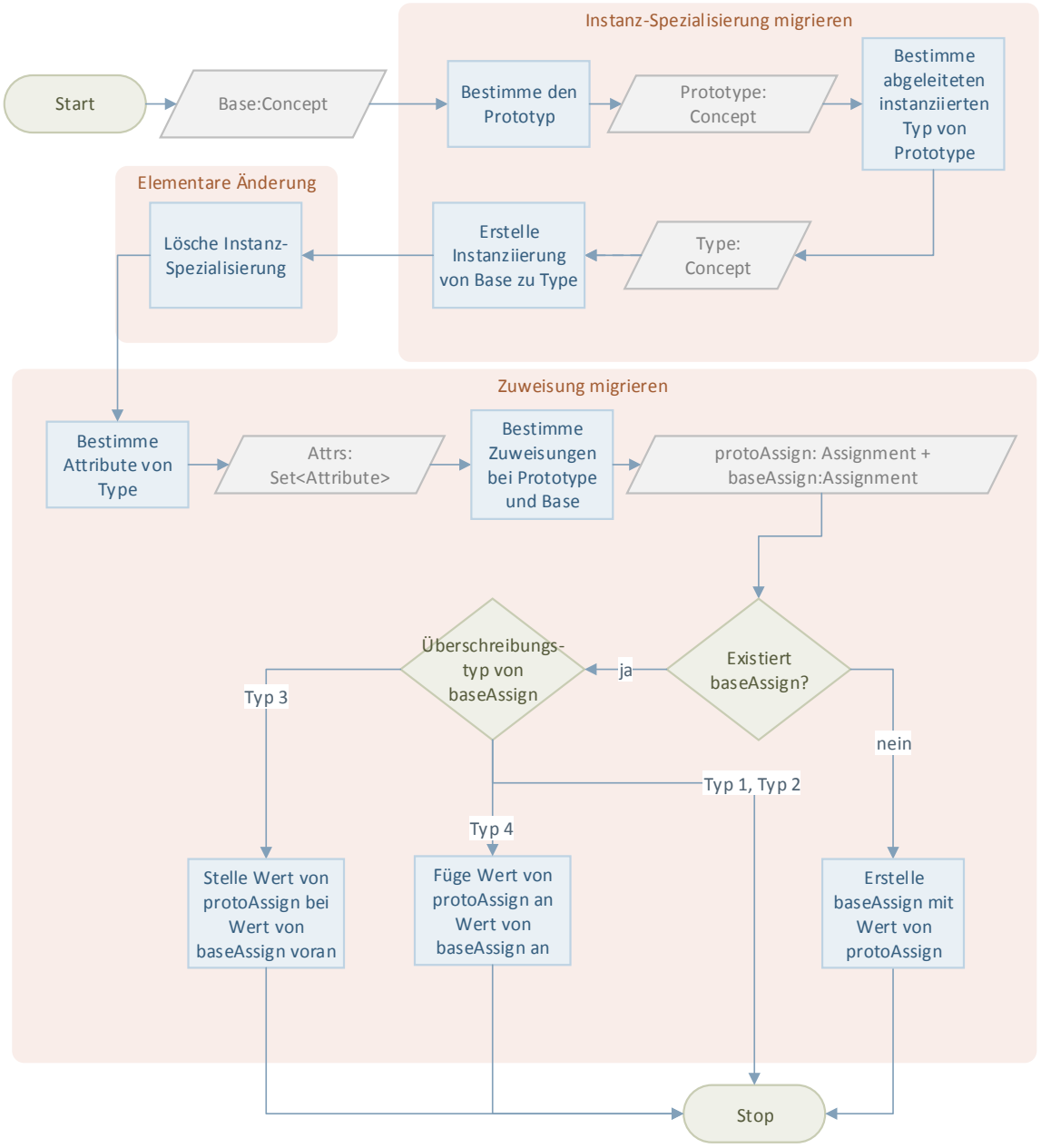

Abbildung 5-20 Ablauf des Delete Concrete Use Of Operators

Zuweisung migrieren

Im darauffolgenden Schritt werden alle Attribute von `Type` ermittelt, um danach alle Zuweisungen von `Base` (`baseAssign`) und `Prototype` (`protoAssign`) an diese Attribute zu bestimmen. Falls eine Zuweisung für ein Attribut beim `Prototype` existiert, jedoch bei `Base` nicht, muss eine neue Zuweisung mit dem Wert von `protoAssign` bei `Base` erstellt werden, da vorher der Wert durch die Instanz-Spezialisierung vererbt wurde.

Wenn die Zuweisung `baseAssign` existiert, muss nach dem Überschreibungstyp unterschieden werden. Typ 0 (`forbidden`) kann in diesem Fall nicht auftreten, da dies bedeuten würde, dass man ein Attribut nicht überschreiben darf und damit auch keine Zuweisung `baseAssign` einer Instanz-Spezialisierung existieren kann. Für die Überschreibungstypen 1 (`normal`) und 2 (`limited`) muss keine Änderung erfolgen, da der Wert ohnehin entweder beliebig definiert wurde oder jeder der Werte von `baseAssign` in der Domäne von `protoAssign` lag. Wenn der Typ 3 (`append`) vorliegt, wurde ein Basiswert durch den Wert von `baseAssign` erweitert. Folglich muss der neue Wert von `baseAssign` den Wert von `protoAssign` voranstellen, um den korrekten Wert zu erlangen. Ähnlich verhält es sich bei Typ 4 (`prepend`), nur dass in diesem Fall der Wert von `protoAssign` nicht voran gestellt, sondern hinten angehängt werden muss. Nachdem alle Zuweisungen angepasst wurden endet der Operator.

Beispiel

Als Beispiel soll die Umkehr des Beispiels des *Set Concrete Use Of* Operators dienen. Die Beschreibung der Modelldomäne findet sich in Abschnitt 5.3.17. Dennoch ist das Modell in Abbildung 5-21 noch einmal umgekehrt dargestellt, um den Lesefluss nicht zu behindern und die wenigen Unterschiede korrekt darzustellen. Auf der linken Seite der Abbildung ist auf `M0` eine Instanz-Spezialisierung von `IbizaStyle` zum Prototyp `Ibiza` dargestellt. Die Ebene `M1` ist dagegen unverändert.

Nun soll der Prototyp von `IbizaStyle` mit Hilfe des *Delete Concrete Use Of* Operators entfernt werden. Dazu wird zunächst vom Operator `Ibiza` als Prototyp ermittelt. Von diesem wird dann das instanziierte Konzept (also der Typ von `Ibiza`) bestimmt (`Car`). Im nachfolgenden Schritt wird die Instanziierung von `IbizaStyle` zu `Car` erstellt und anschließend die Instanz-Spezialisierung zu `Ibiza` gelöscht.

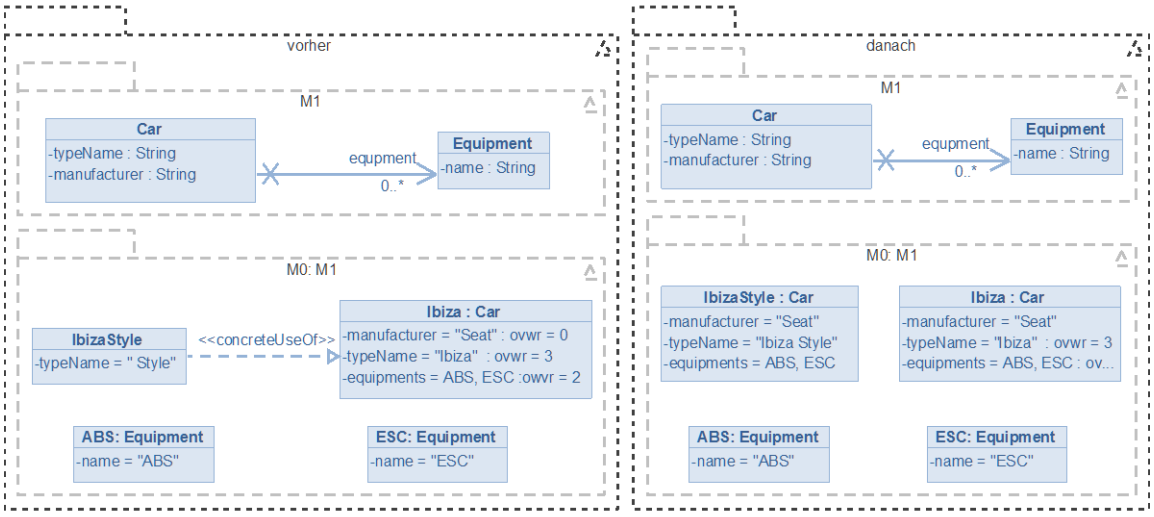

Abbildung 5-21 Beispiel vor (links) und nach (rechts) der Ausführung des Delete Concrete Use Of Operators

Danach werden alle Attribute von `Car` ermittelt: `manufacturer`, `typeName` und `euipment`. Anschließend müssen alle Zuweisungen von `Ibiza` und `IbizaStyle` an das jeweilige Attribut

bestimmt werden, um die Werte der neuen Zuweisungen bei **IbizaSyle** festzulegen. Da **manufacturer** und **equipment** bei **IbizaStyle** vorher nicht zugewiesen wurden, werden die Werte der Zuweisungen von **Ibiza**, also **„Seat“** bzw. **ABS, ESC**, übernommen. Für das Attribut **typeName** muss nun der Überschreibungstyp, der bei **Ibiza** definiert wurde, untersucht werden. Da Typ 3 (**append**) vorliegt, bedeutet dies, dass der Wert der Zuweisung von **IbizaStyle** an den Wert der Zuweisung von **Ibiza** angehängt wurde, um den endgültigen Wert für das Attribut bei **IbizaStyle** zu erhalten. Folglich muss der Wert **„Ibiza“** nun bei **„ Style“** vorangestellt werden, um den korrekten Wert (**„Ibiza Style“**) der Zuweisung für **IbizaStyle** zu erhalten. Danach ist die Änderung des Modells vollständig und endet mit dem in Abbildung 5-21 auf der rechten Seite gezeigten Resultat.

5.3.13 Subroutine: Obligatorische Attribute setzen

Auswirkung

Die Subroutine setzt alle obligatorischen Attribute eines übergebenen Typs für alle Instanzen des Konzepts **Base**.

Ablauf

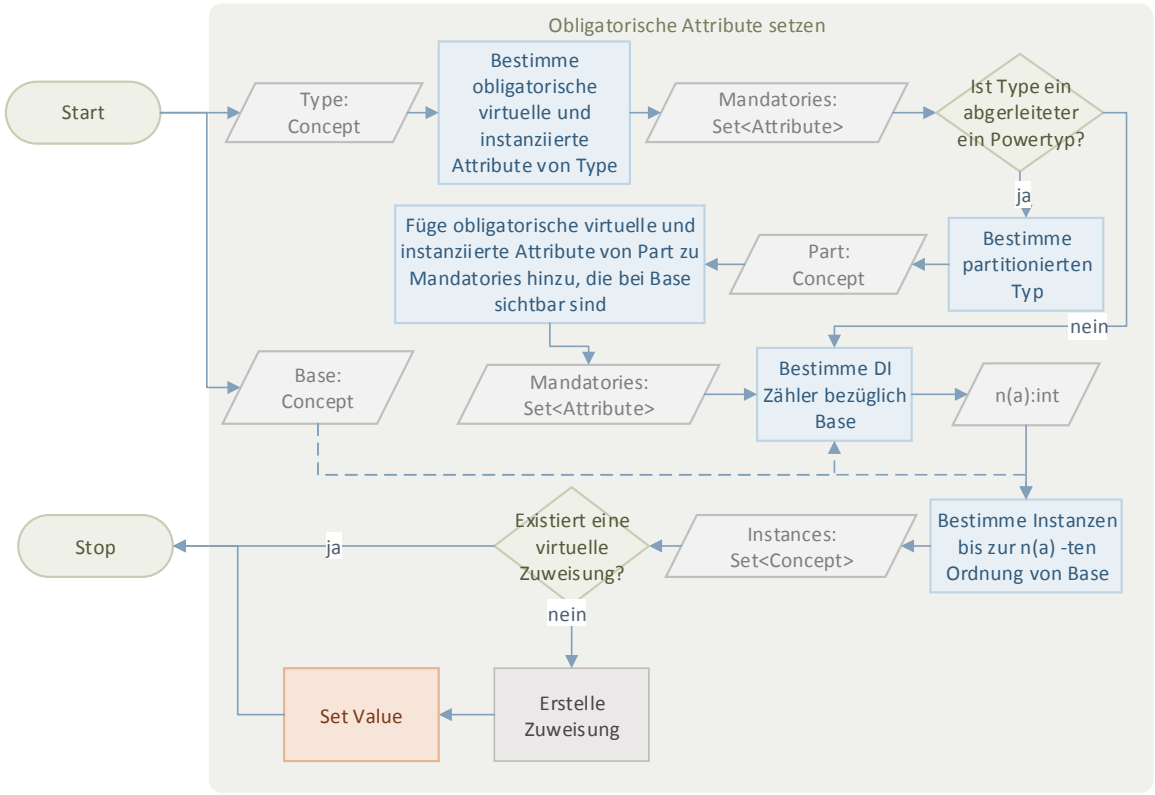

Abbildung 5-22 Ablauf der Subroutine zum Setzen der obligatorischen Attribute

Die Subroutine wird mit zwei Parametern aufgerufen: **Type**, der die Menge an obligatorischen Attributen festlegt, und **Base**, dessen Instanzen die Attribute setzen müssen. Im ersten Schritt werden alle virtuellen und instanziierten[28] Attribute von **Type** bestimmt, die eine Multiplizität von **1** oder **1..*** haben und einen relativen Deep Instantiation Zähler größer als null bezüglich **Type** haben

[28] Dies entspricht Punkt 1 und 2 in Definition 4.21. Punkt 3 tritt nicht auf, da die Subroutine im Kontext der Instanz-Spezialisierung nicht verwendet wird.

(`Mandatories`). Danach wird überprüft, ob `Type` ein abgeleiteter Powertyp ist. Dies ist deshalb von Belang, weil dann auch der partitionierte Typ obligatorische Attribute besitzen kann, die eine Instanz setzen muss. Entsprechend wird in diesem Fall zunächst der partitionierte Typ `Part` bestimmt, bevor im nächsten Schritt alle obligatorischen Attribute zur Menge `Mandatories` hinzugefügt werden, die bei `Base` sichtbar (Definition 4.19) sind. Anschließend wird für jedes dieser Attribute der Deep Instantiation Zähler bezüglich `Base` (`n(a)`)[29] ermittelt, um danach die maximale Ordnung der Instanzen zu kennen, die das Attribut setzen müssen. Damit werden anschließend alle Instanzen bis zur `n(a)`-ten Ordnung berechnet. Diese Menge `Instances` wird dazu verwendet, um zu überprüfen, ob jede Instanz `n(a)`-ter Ordnung das Attribut virtuell zugewiesen (Definition 4.24) hat. Wenn ein Konzept, dessen Ordnung kleiner als `n(a)` ist, ein Attribut aus `Mandatories` gesetzt hat, fungiert die Zuweisung als Standardwert für alle Instanzen `n(a)`-ter Ordnung von `Base`, die von diesem Konzept abstammen (in Hinsicht auf die Instanziierung). Wenn keine virtuelle Zuweisung existiert, muss ein Wert (oder ein Standardwert) für die jeweilige Instanz `n(a)`-ter Ordnung gewählt werden. Deshalb wird in diesem Fall eine Zuweisung für das entsprechende Attribut aus `Mandatories` an einem Konzept aus `Instances` (externe Auswahl, z.B. durch Benutzer) erstellt und anschließend der *Set Value* Operator aufgerufen. Nachdem alle Instanzen einen Wert besitzen endet die Ausführung der Subroutine.

5.3.14 Set Instantiation

Auswirkung

Der Operator setzt einen neuen instanziierten Typ und migriert eine eventuell vorhandene Generalisierung, einen Prototyp, oder eine bisherige Instanziierung.

Ablauf

Prototyp migrieren

Initial wird der Operator an einem Konzept `ToChange` aufgerufen, von dem der instanziierte Typ neu gesetzt werden soll. Deshalb wird im ersten Schritt zunächst der neue Typ aus der Menge aller instanziierbaren Konzepte (Regel C.12) gewählt. Danach wird zunächst der Prototyp (`Prototype`) von `ToChange` bestimmt. Falls dieser existiert, muss dieser mit Hilfe des *Delete Concrete Use* Of Operators entfernt werden, da sich einen Instanziierung und eine Instanz-Spezialisierung gegenseitig ausschließen (Regel C.8 bzw. Regel C.10).

Instanziierten Typ migrieren

Da durch den *Delete Concrete Use* Of Operator der ehemalige abgeleitete instanziierte Typ von `ToChange` nun als instanziierter Typ von `ToChange` gesetzt ist, genügt es im nächsten Schritt lediglich den direkt instanziierten Typ zu bestimmen (anstelle des abgeleiteten). Wenn dieser existiert (z.B. in dem Fall, dass `Prototype` existierte), muss diese Instanziierung zunächst migriert werden. Wenn der neu gewählte Typ (`Type`) eine abgeleitete Spezialisierung des alten Typs (`OldType`) darstellt, entsteht aufgrund des Substitutionsprinizps und der Attributvererbung keine Inkonsistenz bei Zuweisungen und der Operator kann direkt mit den elementaren Änderungen fortfahren. Wenn dies jedoch nicht

[29] Wenn der Deep Instantiation Zähler des Attributes bei der Definition eins ist, werden also alle Instanzen der 0-ten Ordnung von `ToChange` bestimmt, was der Menge `ToChange` und dessen Instanz-Spezialisierungen entspricht.

der Fall ist, muss zunächst untersucht werden, ob `Type` und `OldType` eine gemeinsame abgeleitete Generalisierung besitzen.[30]

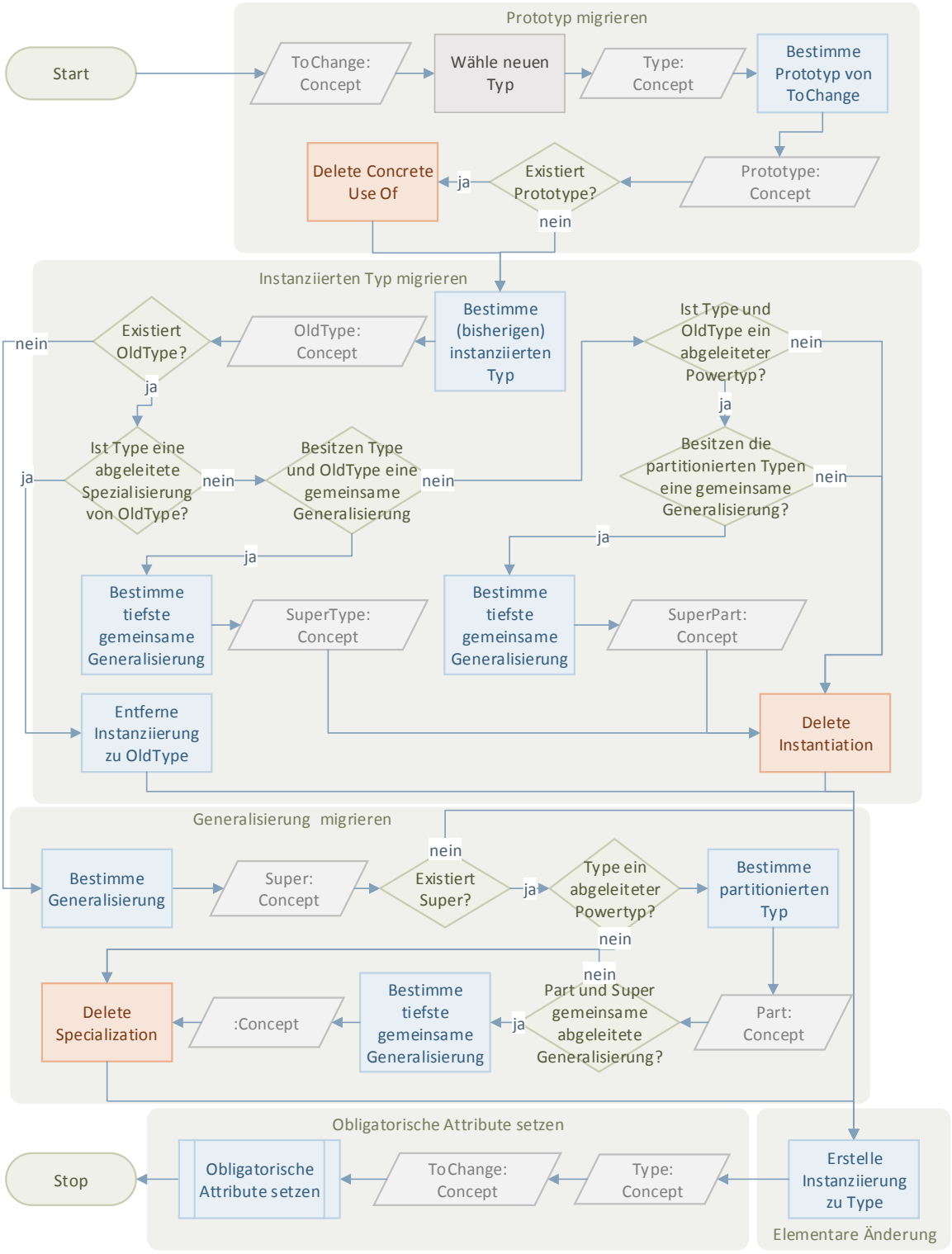

Abbildung 5-23 Ablauf des Set Instantiation Operators

[30] Dies ist beispielsweise dann der Fall, wenn `Type` eine Generalisierung von `OldType` ist oder ein Geschwisterknoten in der gleichen Vererbungshierarchie ist.

Wenn dies zutrifft, sind nicht alle Zuweisungen ungültig, da alle Zuweisungen an Attribute der gemeinsamen abgeleiteten Generalisierung weiterhin gültig sind und auch das Substitutionsprinzip von `ToChange` für diese Generalisierung weiterhin gilt. Folglich wird im nächsten Schritt, die in der Vererbungshierarchie tiefste gemeinsame abgeleitete Generalisierung `SuperType` gesucht. Algorithmisch lässt sich dies wie folgt berechnen: Für ein Konzept werden zunächst alle abgeleiteten Generalisierungen bestimmt. Für das zweite Konzept wird dann schrittweise (beginnen mit dem Konzept selbst) untersucht, ob das aktuell untersuchte Konzept, in der Menge der abgeleiteten Generalisierungen vom ersten Konzept liegt. Ist dies der Fall, ist `SuperType` gefunden. Anderenfalls wird das aktuelle Element durch die direkte Generalisierung bzw. den partitionierten Typ ersetzt und mit ihm wie oben beschrieben weiterverfahren. `SuperType` ist also diejenige gemeinsame abgeleitete Generalisierung, zu der sowohl `Type` als auch `OldType` die kürzeste Distanz [31] in der Vererbungshierarchie besitzt.

Nachdem `SuperType` ermittelt wurde, wird dieses dem *Delete Instantiation* Operator als gleichnamiger Parameter zusammen mit `ToChange` übergeben. Der Operator ermittelt dann mit Hilfe der Subroutine aus Abschnitt 5.3.6 (`ToChange` als `Base` Parameter, `OldType` als `Type` Parameter und `SuperType` als gleichnamiger Parameter) alle Zuweisungen an (evtl. virtuelle oder instanziierte) Attribute, die zwar bei `OldType` vorhanden waren, bei `Type` aber nicht mehr existieren. Da diese Zuweisungen bei allen Instanzen einer gewissen Ordnung von `ToChange` nicht mehr gültig sind, werden diese entfernt. Alle Instanzen n-ter Ordnung von `ToChange` dürfen nach der Ausführung des Operators bei Attributen deren Typ ein Konzept ist, das in der Vererbungshierarchie zwischen `OldType` (inklusive) und `SuperType` (exklusive) liegt, als Wert nicht mehr zugewiesen werden. Deshalb wird die Subroutine aus Abschnitt 5.3.7 zum Bereinigen dieser Zuweisungen im inneren Operator ausgeführt, die diese Inkonsistenz beseitigt. Wenn die Subroutine beendet ist, wird die Instanziierung von `ToChange` zu `OldType` gelöscht, da der `SuperType` Parameter nicht leer ist. Nachdem der *Delete Instantiation* Operator beendet ist, wird die elementare Änderung ausgeführt.

Falls keine gemeinsame abgeleitete Generalisierung von `Type` und `OldType` bestand, wird zunächst untersucht, ob beide Konzepte ein abgeleiteter Powertyp sind, und ob ihre partitionierten Typen eine gemeinsame Generalisierung besitzen. Wenn beide Bedingungen zutreffen, müssen nicht alle Zuweisungen an Attribute des partitionierten Typs (Standardverhalten des *Delete Instantiation* Operators) bei allen Instanzen von `ToChange` entfernt werden. Deshalb wird in diesem Fall zunächst die tiefste gemeinsame abgeleitete Generalisierung der beiden partitionierten Typen bestimmt und diese dem *Delete Instantiation* Operator als `SuperPart` Parameter übergeben. Dadurch wird die Instanziierung zu `OldType` (alter Powertyp) komplett und die Spezialisierung zum partitionierten Typ teilweise migriert. Wenn eine der beiden Bedingungen nicht zutrifft, dann wird der *Delete Instantiation* Operator ohne zusätzliche Parameter aufgerufen, wodurch die komplette Instanziierung von `ToChange` zu `OldType` zusammen mit einer eventuellen Spezialisierung zu dem partitionierten Typ von `OldType` aufgelöst werden. Danach schließt sich das Setzen der neuen Instanziierung (elementare Änderung) an.

Generalisierung migrieren

Falls oben festgestellt wurde, dass bisher kein instanziierter Typ `OldType` für `ToChange` existierte, besteht die Möglichkeit, dass `ToChange` eine Generalisierung besitzt. Da sich eine Instanziierung und

[31] Wenn man die Konzepte als Knoten und die Spezialisierung als Kanten eines gerichteten Graphs betrachtet, dann ist diese Distanz die Länge des Pfades zwischen den Knoten.

eine Spezialisierung ausschließen (Regel C.8 bzw. Regel C.9), muss in diesem Fall zunächst die Generalisierung `Super` von `ToChange` festgestellt werden. Falls diese existiert, wird zunächst untersuch, ob `Type` ein abgeleiteter Powertyp ist. Wenn dem so ist, wird der entsprechende partitionierte Typ `Part` bestimmt und überprüft, ob `Part` und `Super` eine gemeinsame abgeleitete Generalisierung [32] besitzen. Falls dieses existiert, wird die tiefste gemeinsame abgeleitete Generalisierung berechnet und diese dem *Delete Specialization* Operator als Parameter `SuperType` übergeben, damit dieser nicht alle Zuweisungen entfernt. In allen anderen Fällen, wird der Operator ohne diesen Parameter aufgerufen, wodurch die Spezialisierung komplett aufgelöst wird. Daran anschießend kann die neue Instanziierung hergestellt werden.

Elementare Änderung

Nachdem alle Beziehungen (Instanziierung, Instanz-Spezialisierung und Spezialisierung), die die neue Instanziierung verhindern, von `ToChange` migriert wurden, kann als nächstes die Instanziierung zum neuen Typ `Type` erstellt werden.

Obligatorische Attribute setzen

Im nächsten Schritt wird die Subroutine aus Abschnitt 5.3.13 dazu verwendet, um bei allen Instanzen von `ToChange` alle obligatorischen Attribute von `Type` zu setzen, falls diese noch keinen Wert besitzen. Dabei wird `Type` als gleichnamiger Parameter und `ToChange` als `Base` Parameter übergeben. Nachdem die Subroutine endet, wird auch der Operator beendet.

Beispiel

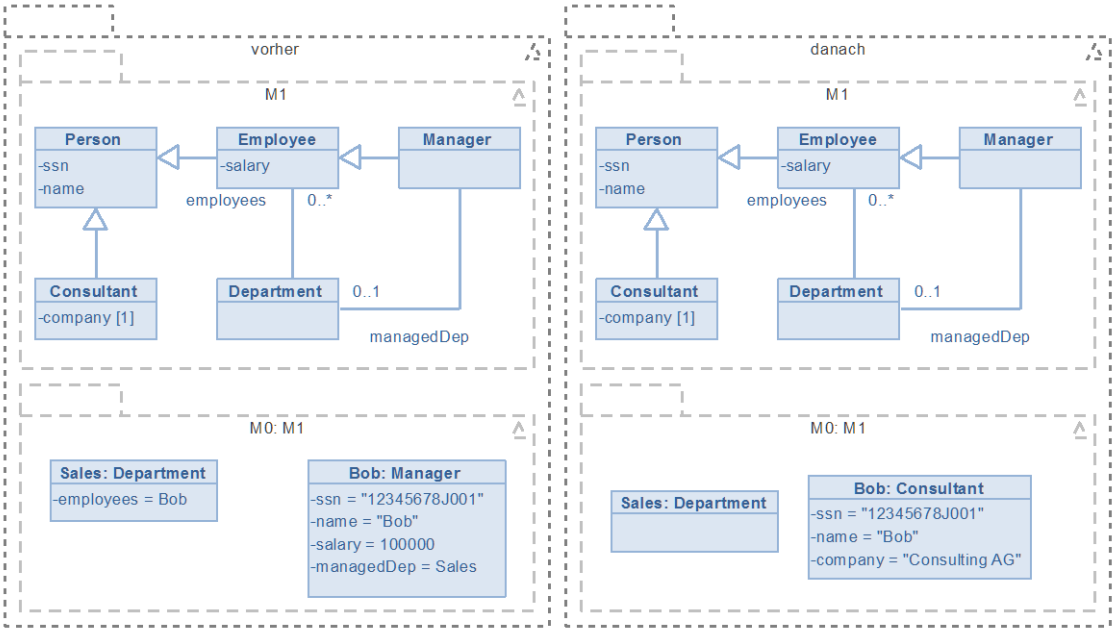

Abbildung 5-24 Beispiel vor (links) und nach (rechts) der Ausführung des Operators

In Abbildung 5-24 ist auf der linken Seite ein (sehr einfaches) Modell dargestellt, das ein Unternehmen beschreibt. Auf Ebene `M1` wurde eine Vererbungshierarchie definiert. Die Wurzel dieser Hierarchie stellt das Konzept `Person` dar, welches die Attribute `ssn` (Sozialversicherungsnummer) und `name` besitzt.

[32] Für den Fall, dass `Part` eine abgeleitete Spezialisierung von `Super` ist, wird dabei `Super` als tiefste gemeinsame Generalisierung ermittelt. Folglich führt der *Delete Specialization* Operator lediglich die elementare Änderung durch. Ähnlich wie vorher, könnte hier ebenfalls eine vorherige Abfrage den Aufruf des Operators verhindern. Aus Übersichtlichkeitsgründen wurde an dieser Stelle allerdings darauf verzichtet.

Weiterhin wurden `Employee` und `Consultant` (Mitarbeiter und Berater des Unternehmens) als Spezialisierungen modelliert, wobei `Employee` das Attribut `salary` (Jahresgehalt) und `Consultant` das Attribut `company` (Consulting Unternehmen) mit der Multiplizität `1` definiert. `Employee` wiederrum besitzt eine weitere Spezialisierung `Manager`, die den Leiter einer Abteilung (Konzept `Department`) darstellt und folglich das Attribut `managedDep` besitzt (Beziehung zu `Department`). Außerdem definiert `Department` über das Attribut `employees` noch eine Beziehung zu `Employee`, die einer Abteilung Mitarbeiter zuordnet. Die Ebene `M0` definiert zwei Konzepte: `Bob` als Instanz von `Manager` und `Sales` (Verkaufsabteilung) als Instanz von `Department`. Die Abteilung `Sales` besitzt `Bob` als Mitarbeiter, wie aus der Zuweisung für `employees` deutlich wird. `Bob` wiederrum ist Leiter der Abteilung `Sales` (Zuweisung an `managedDep`), hat die Sozialversicherungsnummer `12345678J001` (Zuweisung an `ssn`), den Namen **„**`Bob`**“** (Zuweisung an `name`) und ein Jahresgehalt von `100000` € (Zuweisung an `salary`).

Im Beispiel soll `Bob` aus dem Unternehmen ausscheiden und zu einer Consulting Firma („Consulting AG“) wechseln. Um dies umzusetzen wird die Instanziierung von `Bob` mit Hilfe des *Set Instantiation* Operators neu gesetzt. Dazu wählen wir zunächst den neuen Typ von `Bob` nämlich `Consultant`. Da `Bob` keinen Prototyp besitzt, wird als nächstes `Manager` als bisheriger instanziierter Typ bestimmt. Nun wird untersucht, ob `Consultant` eine abgeleitet Spezialisierung von `Manager` ist. Weil dies nicht zutrifft, wird festgestellt, dass `Consultant` und `Manager` eine gemeinsame Generalisierung besitzen. Dementsprechend wird die tiefste gemeinsame Generalisierung bestimmt. Dazu werden zunächst alle abgeleiteten Generalisierungen von `Manager` berechnet, was als Ergebnis die Menge {`Manager`, `Employee`, `Person`} liefert. Nun wird für `Consultant` schrittweise jede Generalisierung überprüft, ob sie in dieser Menge liegt. Da `Consultant` nicht enthalten ist, wird mit dessen direkter Generalisierung das Vorgehen wiederholt. Aufgrund dessen, dass `Person` in der Menge liegt, ist `Person` die tiefste gemeinsame Generalisierung. Anschließend wird `Person` als `SuperType` Parameter dem *Delete Instantiation* Operator übergeben. Folglich, bleiben alle Zuweisungen an Attribute von `Person` bei `Bob` auch nach dem Operator erhalten und die Subroutine aus Abschnitt 5.3.6 entfernt lediglich die Zuweisungen für `salary` und `managedDep`. Dies wird durch die Parametrierung (`Manager` als `Type`, `Person` als `SuperType` und `Bob` als `Base` Parameter) der Subroutine erreicht. Danach wird die Subroutine aus Abschnitt 5.3.7 verwendet, um alle Zuweisungen an ein Attribut, dessen Typ zwischen `Manager` (eingeschlossen) und `Person` (ausgeschlossen) in der Vererbungshierarchie liegt, vom Wert `Bob` zu bereinigen. Es wird also zunächst das Attribut `employees` vom Typ `Employee` ermittelt. Danach werden alle Instanzen 0-ter Ordnung von `Bob` ermittelt. Da `Bob` keine Instanz-Spezialisierung besitzt, besteht die Ergebnismenge lediglich aus dem Konzept selbst. Nun wird die Zuweisung bei `Sales` untersucht. Da dort lediglich `Bob` zugewiesen wurde, wird die Zuweisung innerhalb des *Set Value* Operators durch den *Delete Assignment* Operator gelöscht. Bevor der *Delete Instantiation* Operator endet, wird noch die Instanziierung von `Bob` zu `Manager` entfernt. Anschließend erstellt der *Set Instantiation* Operator eine Instanziierung von `Bob` zu `Consultant`.

Als nächstes wird die Subroutine zum Setzen der obligatorischen Attribute (Abschnitt 5.3.13) mit `Consultant` als `Type` und `Bob` als `Base` Parameter ausgeführt. Dabei werden im ersten Schritt alle virtuellen und instanziierten Attribute von `Consultant` bestimmt, die obligatorisch sind und einen relativen Deep Instantiation Zähler größer als null bezüglich `Consultant` haben. Diese müssen bei `Bob` gesetzt werden. In diesem Beispiel trifft dies auf `company` zu, von dem im nächsten Schritt der Deep Instantiation Zähler bezüglich `Bob` ermittelt wird. Da `Bob` (nun) eine Instanz von `Consultant` ist und `company` einen Deep Instantiation Zähler von eins definiert, liegt der Wert der relativen Deep Instantiation Zählers in diesem Fall bei `0`. Folglich werden im nächsten Schritt alle Instanzen 0-ter Ordnung von `Bob` ermittelt. Wie oben bereits erwähnt, besteht diese Menge nur aus `Bob`. Da wir für `Bob` nun einen Wert zuweisen müssen, wählen wir den Wert **„**`Consulting AG`**“** für das Attribut

company und die Ausführung des Operators endet mit dem in Abbildung 5-24 auf der rechten Seite dargestellten Modell.

5.3.15 Set Specialization

Auswirkung

Der Operator setzt die Generalisierung des übergebenen Konzeptes neu und migriert eine eventuelle Instanziierung oder Partitionierung. Außerdem werden alle obligatorischen Attribute gesetzt.

Ablauf

Instanziierten Typ migrieren

Initial wird der Operator (Abbildung 5-25) an einem Konzept ToChange aufgerufen, das eine Generalisierung erhalten soll. Diese neue Generalisierung (Super) wird im ersten Schritt aus den möglichen Kandidaten (Regel C.14) gewählt. Da eine Spezialisierung keinen instanziierten Typ besitzen darf (Regel C.9), muss dieser (Type) zunächst ermittelt werden. Falls ein instanziierter Typ existiert, unterscheidet der Operator, ob es sich um einen abgeleiteten Powertyp handelt, da in diesem Fall eine Spezialisierung zum partitionierten Typ bereits besteht und die Migration dieser Spezialisierung analog zum Vorgehen im Teilschritt *Generalisierung migrieren* erfolgt. Falls kein Powertyp vorliegt, wird der *Delete Instantiation* Operator verwendet, um die Instanziierung zu entfernen. Danach kann direkt mit der Bestimmung eines eventuellen partitionierten Typs von ToChange fortgefahren werden. Im Falle, dass Type ein abgeleiteter Powertyp ist, kann der *Delete Instantiation* Operator nicht angewendet werden, da er auch die Spezialisierung zum partitionierten Typ auflöst. Dies kann dazu führen, dass zu viele Zuweisungen gelöscht oder angepasst werden, da die neue Generalisierung Super eventuell Attribute mit dem partitionierten Typ teilt. Es wird also in diesem Fall zunächst unterschieden, ob es sich um einen erweiterten Powertyp handelt. Ist dem so, wird die Instanziierung zu Type aufgelöst, indem die Subroutine aus Abschnitt 5.3.6 (ToChange als Base, Type als Type Parameter und der SuperType Parameter bleibt leer) die Zuweisungen an alle Attribute von Type bei Instanzen von ToChange entfernt. Danach werden durch die Subroutine aus Abschnitt 5.3.7 (ToChange als Base, Type als Type Parameter und der SuperType Parameter bleibt leer) Zuweisungen, deren Attribut eine abgeleitete Generalisierung von Type besitzt, bereinigt.

Subroutine: Zuweisungen löschen – erweiterter Powertyp

Für den Fall, dass ein erweiterter Powertyp vorliegt, muss anders vorgegangen werden, da die Subroutine aus Abschnitt 5.3.6 ebenfalls Zuweisungen an Diskriminatorattribute entfernt. Dies wiederum könnte dazu führen, dass Zuweisungen an das entsprechende Attribut des partitionierten Typs bei Instanzen von ToChange gelöscht würden. Da diese Attribute aber eventuell ebenfalls an der neuen Generalisierung Super als virtuelle Attribute vorliegen, würden durch diesen Schritt gültige Zuweisungen entfernt. Deshalb müssen alle Zuweisungen an Diskriminatorattribute elementar entfernt werden. Dazu soll die in Abbildung 5-11 gezeigte Subroutine[33] verwendet werden: Die Subroutine erhält als Parameter Type und ToChange. Im ersten Schritt werden alle virtuellen Attribute von Type bestimmt. Für jedes dieser Attribute wird der Deep Instantiation Zähler bezüglich ToChange (n(a)) bestimmt, damit die maximale Ordnung der Instanzen, die eine Zuweisung an das Attribut besitzen, festgelegt ist.

[33] Im Zuge ein Umsetzung in einem Modellierungssystem könnte diese Subroutine mit der aus Abschnitt 5.3.6 verschmolzen werden, indem ein weiterer Parameter steuert, ob für die Diskriminatorattribute das elementare Löschen oder der *Delete Assignment* Operator verwendet wird. Auf diese Möglichkeit wurde hier verzichtet, um die Parameterzahl der Subroutinen klein zu halten und damit, die Verständlichkeit der Abläufe zu wahren.

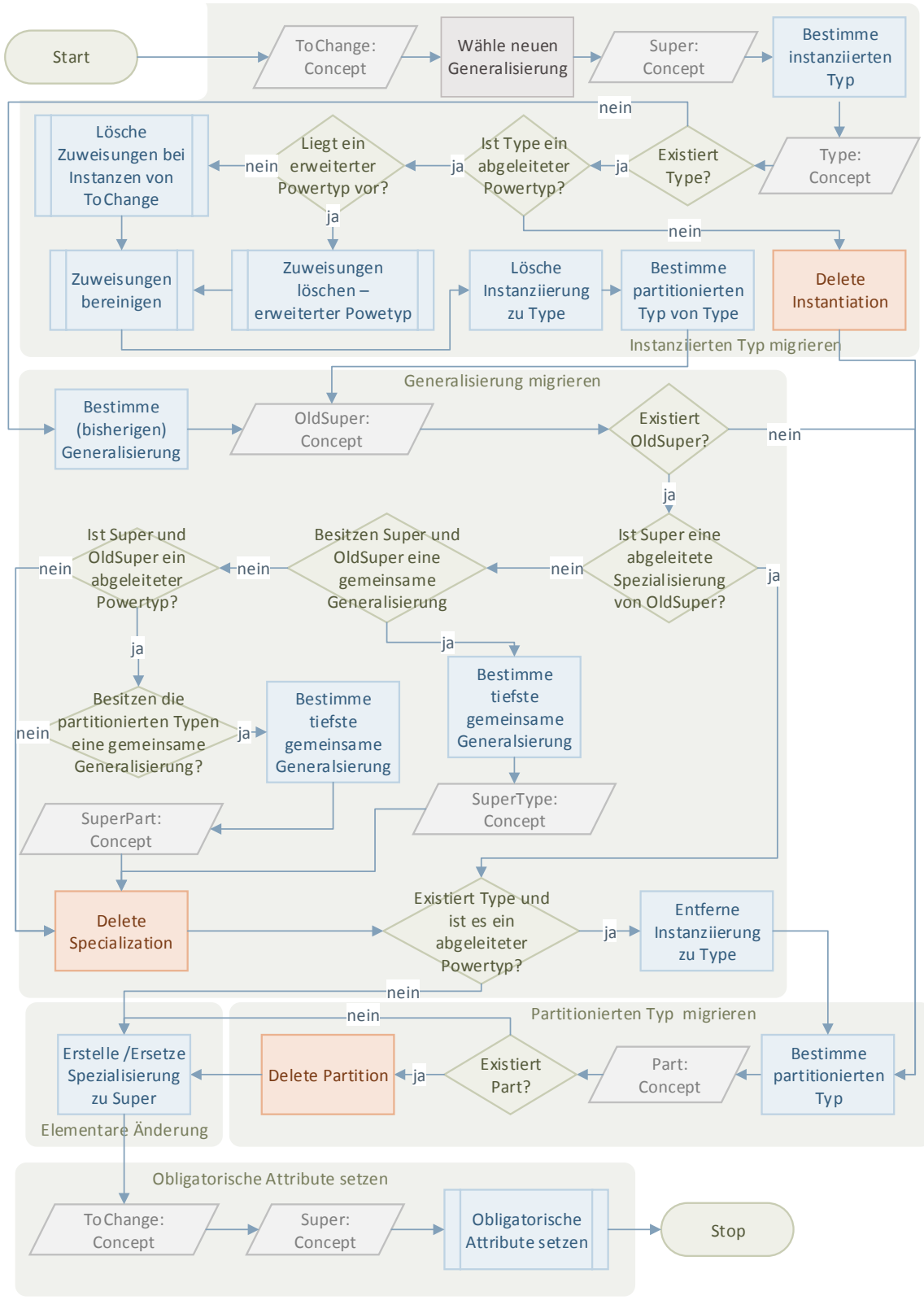

Abbildung 5-25 Ablauf des Set Specialization Operators

Anschließend werden alle Instanzen bis zu `n(a)`-ten Ordnung berechnet, damit danach alle Zuweisungen dieser Instanzen an das Attribut bestimmt werden können. Wenn die Zuweisungen an ein Diskriminatorattribut von **Type** gemacht wurden, wird diese Zuweisung elementar gelöscht. Wenn dies nicht der Fall ist, wird der *Delete Assignment* Operator verwendet, um die Zuweisung zu entfernen.

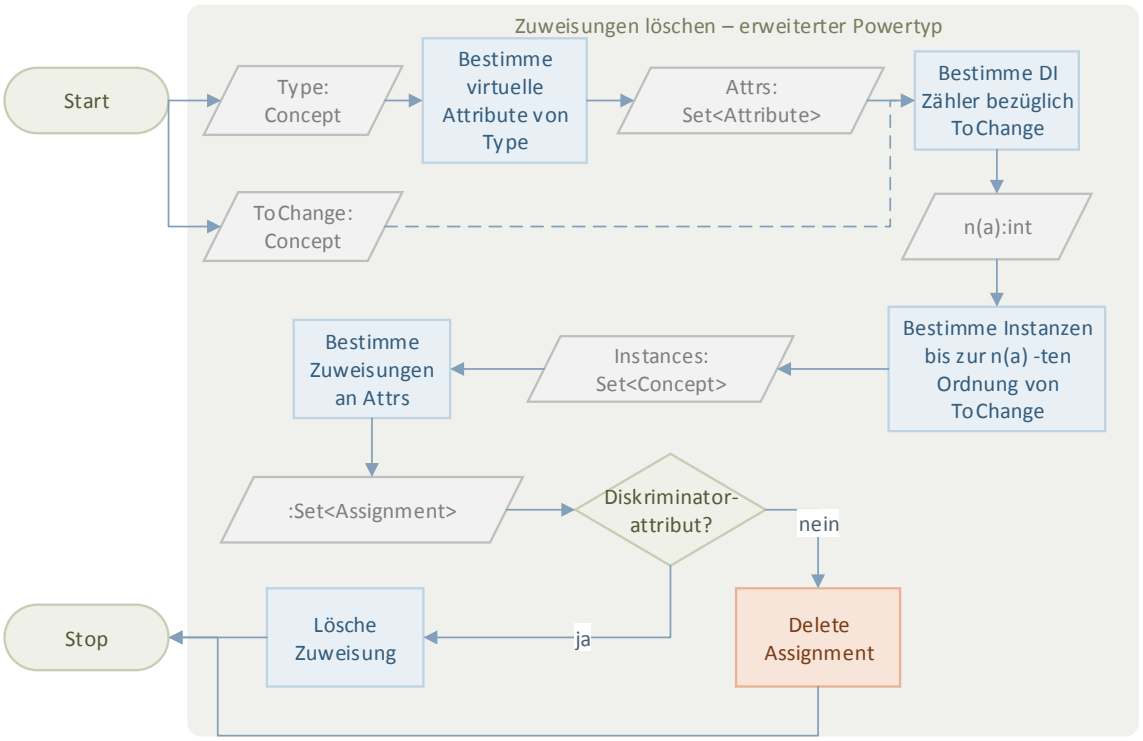

Abbildung 5-26 Ablauf der Subroutine zum Löschen der Zuweisungen im Falle eines erweiterten Powertyps

Nachdem die Subroutine endet, müssen auch im Falle, dass **Type** ein abgeleiteter erweiterter Powertyp ist, wie oben bereits beschrieben, die Zuweisungen bereinigt werden. Anschließend wird in beiden Fällen der partitionierte Typ bestimmt und mit ihm wie mit einer möglichen Generalisierung (**OldSuper**, siehe nächster Teilschritt) verfahren. Der partitionierte Typ, nimmt also dann die Rolle von **OldSuper** an.

Generalisierung migrieren

Falls kein instanziierter Typ für **ToChange** existiert, ist es möglich, dass eine Spezialisierung besteht. Diese (**OldSuper**) wird also zunächst bestimmt. Wenn **OldSuper** nicht existiert, kann direkt ein möglicher partitionierter Typ von **ToChange** bestimmt werden. Anderenfalls muss untersucht werden, ob **Super** eine abgeleitete Spezialisierung von **OldSuper** ist. Weil in diesem Fall keine Attribute wegfallen und auch das Substitutionsprinzip erhalten bleibt, kann mit der Unterscheidung, ob **Type** existiert und ein abgeleiteter Powertyp ist, fortgefahren werden. Für den Fall, dass **Super** die bisherige Generalisierung **OldSuper** nicht erweitert, muss untersucht werden, ob beide eine gemeinsame Generalisierung besitzen. Wenn dem nicht so ist, wird zunächst überprüft, ob **OldSuper** und **Super** ein abgeleiteter Powertyp sind. Ist dies der Fall und besitzen die jeweiligen partitionierten Typen von **Super** und **OldSuper** eine gemeinsame abgeleitete Generalisierung, wird die tiefste gemeinsame abgeleitete Generalisierung[34] von den beiden partitionierten Typen bestimmt. Diese wird dann als **SuperPart** Parameter dem *Delete Specialization* Operator übergeben, damit dieser die Attribute entfernt, die beim partitionierten Typ von **OldSuper** virtuell oder instanziiert waren und nun beim partitionierten Typ von **Super** nicht mehr vorhanden sind. Wenn keine der beiden Bedingungen zutrifft, wird die komplette Spezialisierung mit Hilfe des Standardverhaltens des *Delete Specialization* Operators migriert. Wenn jedoch eine gemeinsame Generalisierung besteht, müssen nicht alle Zuweisungen gelöscht bzw. bereinigt werden. Folglich wird dann die tiefste gemeinsame abgeleitete

[34] Eine Vorgehensweise zur Berechnung findet sich im Teilschritt „Instanziierten Typ migrieren“ des *Set Instantiation* Operators im Abschnitt 5.3.14.

Generalisierung (`SuperType`) zwischen `OldSuper` und `Super` gesucht. Diese wird dann dem *Delete Specialization* Operator als Parameter `SuperType` übergeben, damit alle Zuweisungen an Attribute, die an einem Konzept definiert wurden, das in der Vererbungshierarchie zwischen `SuperType` (ausgeschlossen) und `OldSuper` (eingeschlossen) liegt, entfernt werden. Weiterhin werden dadurch alle Zuweisungen bereinigt, an denen Instanzen von `ToChange` nicht mehr gesetzt werden dürfen. Danach (und im Fall, dass `Super` eine abgeleitete Spezialisierung von `OldSuper` ist) wird, wenn `OldSuper` ein partitionierter Typ und damit `Type` ein abgeleiteter Powertyp ist, die Instanziierung zu `Type` und dadurch die Spezialisierung von `ToChange` zu `OldSuper` entfernt[35].

Partitionierter Typ migrieren

Wenn `ToChange` keine Generalisierung oder (k)einen instanziierten Typ besaß, dann könnte `ToChange` ein Powertyp sein. Deshalb wird als nächstes der partitionierte Typ `Part` von `ToChange` ermittelt und für den Fall, dass `Part` existiert, wird dieser mit Hilfe des *Delete Partition* Operators (Abschnitt 5.3.11) aufgelöst.

Elementare Änderung

Anschließend wird die neue Spezialisierung von `ToChange` zu `Super` erstellt. Falls `Super` eine abgeleitete Spezialisierung von `OldSuper` ist, wird die alte Spezialisierung durch die neue ersetzt.

Obligatorische Attribute setzen

Danach werden mit Hilfe der Subroutine aus Abschnitt 5.3.13 alle obligatorischen Attribute bei allen Instanzen von `ToChange` gesetzt. Dazu wird `ToChange` als `Base` und `Super` als `Type` Parameter übergeben. Anschließend ist die die Ausführung des Operators beendet.

Beispiel

Abbildung 5-27 zeigt auf der linken Seite ein kleines Unternehmensmodell vor der Ausführung des Operators. Darin ist auf Ebene `M1` eine Vererbungshierarchie modelliert, in der das Konzept `Person` die Wurzel darstellt. Es definiert zwei Attribute, die den Namen (`name`) und die Sozialversicherungsnummer (`ssn`) einer Person darstellen. Weiterhin wurde eine Spezialisierung in Form von `Employee` definiert, die wiederum durch `Permanent` (Festangestellter) und `Non-Permanent` (befristet angestellte Mitarbeiter) spezialisiert wird. Festangestellte besitzen ein Jahresgehalt (`salary`), während bei befristet angestellten Mitarbeitern der Stundenlohn (`hourlyRate`) festgehalten wird. Eine erneute Spezialisierung der Festangestellten stellt das Reinigungspersonal dar, was durch die entsprechende Vererbung der Konzepte `Permanent` und `Cleaning` zum Ausdruck gebracht wird. Außerdem wurde noch ein Konzept `Department` modelliert, das eine Abteilung darstellt und das Attribut `employees` besitzt, um alle Festangestellten abzuspeichern. Ebene `M0` definiert ein kleines Beispielmodell, in dem die Verkaufsabteilung (`Sales` als Instanz von `Department`) einen festangestellten Mitarbeiter (Zuweisung an `employees` mit dem Wert `Bob`) besitzt. `Bob` wiederum ist eine Instanz von `Cleaning` und gehört damit zum Reinigungspersonal.

Nun soll das Reinigungspersonal fortan nicht mehr festangestellt sein. Deshalb wird der *Set Specialization* Operator am Konzept `Cleaning` aufgerufen und im ersten Schritt `Non-Permanent` als neue Generalisierung gewählt. Da `Cleaning` keinen instanziierten Typ besitzt, wird als nächstes die alte Generalisierung `Permanent` bestimmt. Aufgrund der Tatsache, dass `Non-Permanent` zwar keine Spezialisierung von `Permanent` darstellt, aber beide Konzepte eine gemeinsame Generalisierung besitzen, wird im nächsten Schritt die tiefste gemeinsame abgeleitete Generalisierung berechnet. Das

[35] Die elementare Änderung des *Delete Specialization* Operator löscht eine Spezialisierung, da diese aber im Fall eines Powertyps nicht explizit vorhanden ist, muss an dieser Stelle die Instanziierung entfernt werden.

Resultat `Employee` wird anschließend dem *Delete Specialization* Operator übergeben. Dieser wiederum verwendet die Subroutine, um alle Zuweisungen zu löschen, die an Attribute von `Permanent` gemacht wurden, da `Employee` die direkte Generalisierung von `Permanent` ist. Entsprechend werden alle Zuweisungen von allen Instanzen bis zur 1-ten Ordnung[36] von `Cleaning` an `salary` entfernt. Weil `Cleaning` keine Zuweisung definiert, wird also lediglich die Zuweisung von `Bob` an `salary` gelöscht und die Subroutine endet. Danach berechnet der Operator alle Instanzen 1-ter Ordnung von `Cleaning`, was in diesem Fall, wie bereits gezeigt, die Menge {`Bob`} liefert. Diese wird verwendet, um in der Subroutine aus Abschnitt 5.3.7 alle Attribute, deren Typ in der Vererbungshierarchie zwischen `Permanent` und `Employee` liegt, zu bereinigen. Demzufolge werden für `employees` (Attributtyp ist `Permanent`) alle Zuweisungen bestimmt und anschließend überprüft, ob `Bob` (Menge aller Instanzen 1-ter Ordnung) zugewiesen wurde. Da dies bei `Sales` der Fall ist, wird `Bob` als Wert und damit die Zuweisung entfernt und der *Delete Specialization* Operator endet.

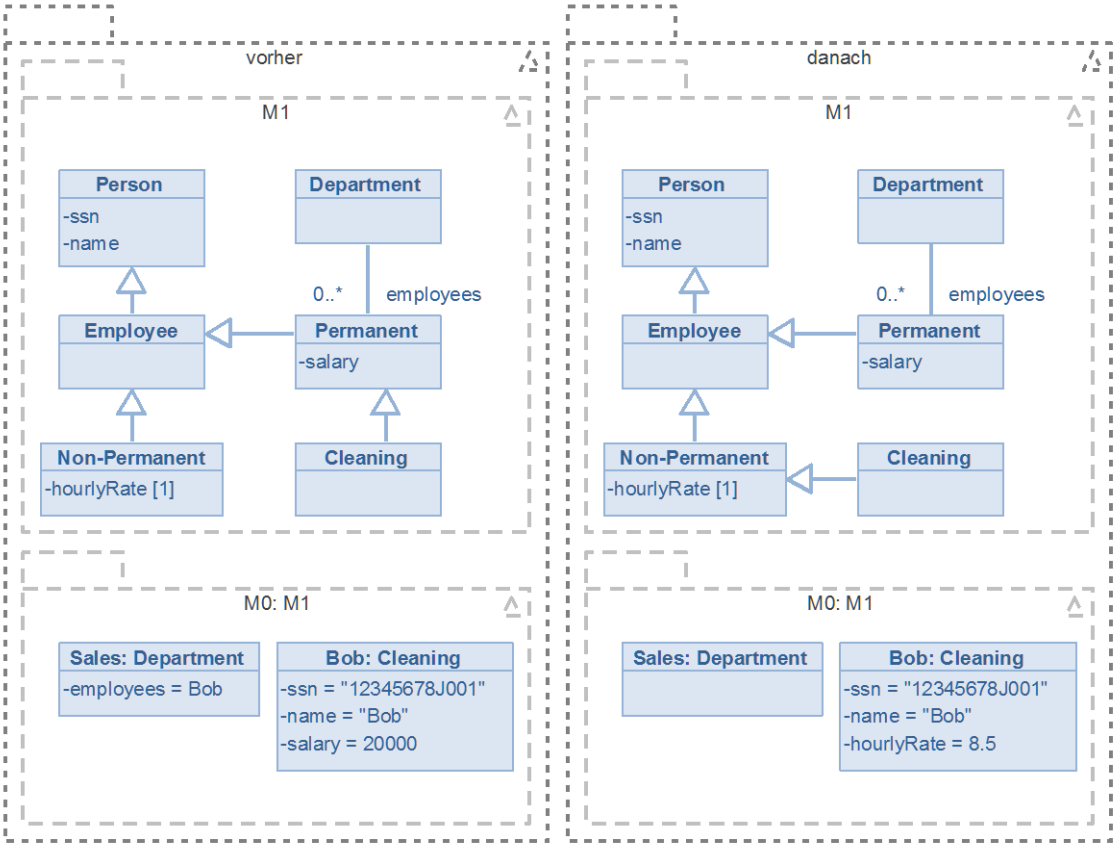

Abbildung 5-27 Beispiel vor (links) und nach (rechts) der Ausführung des Set Specialization Operators

Als nächstes wird, da keine Powertyp Instanziierung vorliegt, die Spezialisierung von `Cleaning` zu `Permanent` entfernt und anschließend einen neue zu `Non-Permanent` erstellt. Im letzten Teilschritt werden alle obligatorischen Attribute von `Non-Permanent` (= `hourlyRate`) ermittelt und für alle Instanzen 1-ter Ordnung von `Cleaning` ein entsprechender Wert ausgewählt. In unserem Beispiel erhält `Bob` eine Zuweisung an `hourlyRate` mit den von uns gewählten Wert `8.5`. Nachdem dieser

[36] Da `salary` einen Deep Instantiation Zähler von eins definiert und die Spezialisierung von `Cleaning` zu `Permanent` den relativen Deep Instantiation Zähler bezüglich `Cleaning` nicht verringert, müssen alle Instanzen bis zur 1-ten Ordnung ermittelt werden.

Schritt beendet ist, endet auch der Operator mit dem in Abbildung 5-27 auf der rechten Seite gezeigtem Resultat.

5.3.16 Set Partition

Auswirkung

Der Operator setzt einen partitionierten Typ am übergebenen Konzept und migriert eine Generalisierung, falls diese vorher bestand.

Ablauf

Generalisierung migrieren

Wie in Abbildung 5-28 gezeigt ist, wird der Operator an einem Konzept `ToChange` aufgerufen, für das ein neuer partitionierter Typ gesetzt werden soll. Dazu wird dieser (im Folgenden `Part` genannt) zunächst aus der Menge aller Kandidaten (Regel C.16) gewählt. Falls `ToChange` eine Generalisierung besitzt, muss diese migriert werden, da sich eine Partitionierung und eine Spezialisierung ausschließen (Regel C.9 bzw. Regel C.11). Deshalb wird die Generalisierung zunächst ermittelt. Für den Fall, dass sie existiert, wird durch den *Delete Specialization* Operator die Spezialisierung von `ToChange` aufgelöst.

Partitionierten Typ migrieren

Danach muss der alte partitionierte Typ `OldPart` migriert werden, weshalb dieser bestimmt wird. Wenn er nicht existiert oder `Part` eine abgeleitete Spezialisierung davon ist, kann direkt mit der elementaren Änderung fortgefahren werden. Für den Fall, dass beide Bedingungen nicht zutreffen, muss noch untersucht werden, ob `Part` und `OldPart` eine gemeinsame Generalisierung besitzen. Wenn nicht wird mit Hilfe des *Delete Partition* Operators die Partitionierung von `ToChange` zu `OldSuper` aufgelöst. Falls jedoch eine gemeinsame Generalisierung existiert, wird die tiefste gemeinsame abgeleitete Generalisierung [37] beider Konzepte bestimmt und diese als `SuperPart` Parameter dem *Delete Partition* Operator übergeben, damit dieser ungültige Diskriminatorattribute entfernt und invalide Zuweisungen bereinigt oder löscht.

Elementare Änderung

Als nächstes wird dann `Part` als neuer partitionierter Typ bei `ToChange` gesetzt, wodurch `ToChange` zum Powertyp wird. Falls `Part` eine abgeleitete Spezialisierung von `OldPart` ist, wird an dieser Stelle die alte Partitionierung durch die neue ersetzt.

Obligatorische Attribute setzen bzw. erstellen

Danach wird entschieden, ob `ToChange` ein erweiterter Powertyp ist. Wenn dem so ist, werden alle virtuellen Attribute von `Part` bestimmt und festgestellt, ob sie ein Diskriminatorattribut besitzen (Regel A.6). Falls dies nicht der Fall ist, wird für jedes ein entsprechendes Diskriminatorattribut erstellt und dabei der *Set Enables* Operator verwendet. Da durch das Setzen der Partitionierung alle Instanzen von `ToChange` nun auch Spezialisierungen von `Part` sind, müssen die Instanzen dieser Instanzen alle obligatorischen Attribute von `Part` zuweisen. Deshalb wird die Subroutine aus Abschnitt 5.3.13 verwendet, um diese Aufgabe zu erledigen. Im Anschluss ist die Ausführung des Operators beendet.

[37] Eine Vorgehensweise zur Berechnung findet sich im Teilschritt „Instanziierten Typ migrieren" des *Set Instantiation* Operators im Abschnitt 5.3.14.

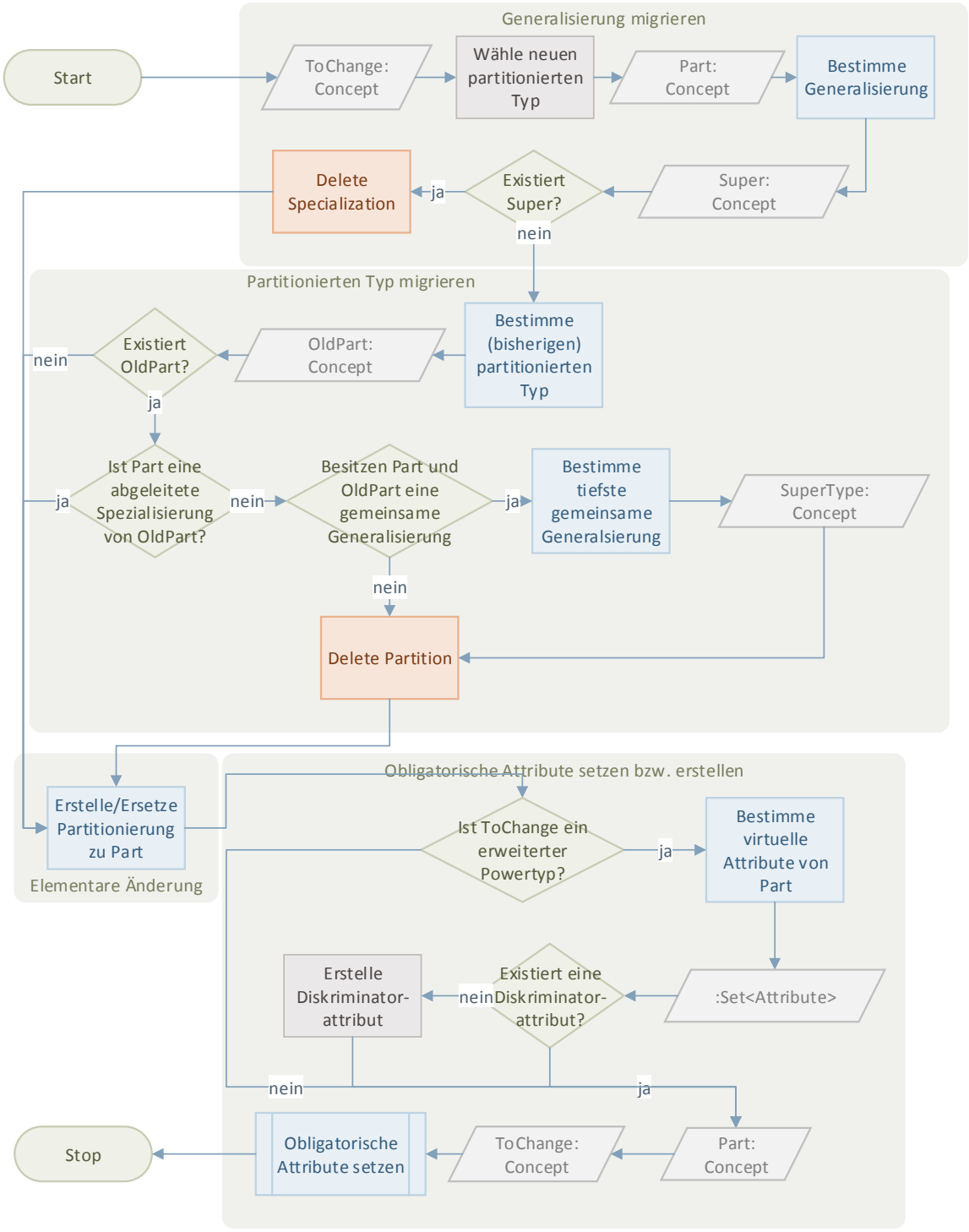

Abbildung 5-28 Ablauf des Set Partition Operators

Beispiel

In Abbildung 5-29 ist auf der linken Seite eine einfache Meta-Hierarchie dargestellt. Auf Ebene `M2` ist das Konzept `AbstractNode` (Deep Instantiation Zähler von `2`) als partitionierter Typ vom Powertyp `NodeKind` definiert worden. Der partitionierte Typ definiert ein Attribut `id`, das ein Diskriminatorattribut beim `NodeKind` (`supportsID`) besitzt. Weiterhin wurde eine Spezialisierung von `AbstractNode` in Form von `Node` (Deep Instantiation Zähler von `2`) modelliert, die ebenfalls ein Attribut `name` deklariert. Eine solche Konstellation ist ungewöhnlich, da Spezialisierungen des partitionierten Typs in der Regel als Instanzen des Powertyps abgebildet werden. In diesem Fall ist die

Generalisierung **AbstractNode** jedoch ein abstraktes Konzept, das als Grundgerüst für **Node** fungiert. Allerdings wurde die Partitionierung (eventuell durch einen Modellierungsfehler) von **NodeKind** zu **AbstractNode** gesetzt. Ebene **M1** definiert ein Konzept **Process**, das eine Instanz von **NodeKind** ist und das Diskriminatorattribut auf **true** gesetzt hat und damit das Attribut **id** von **AbstractNode** erbt. Deshalb kann die Instanz von **Process** auf **M0** (**Start**) das Attribut (mit dem Wert **„1“**) setzen.

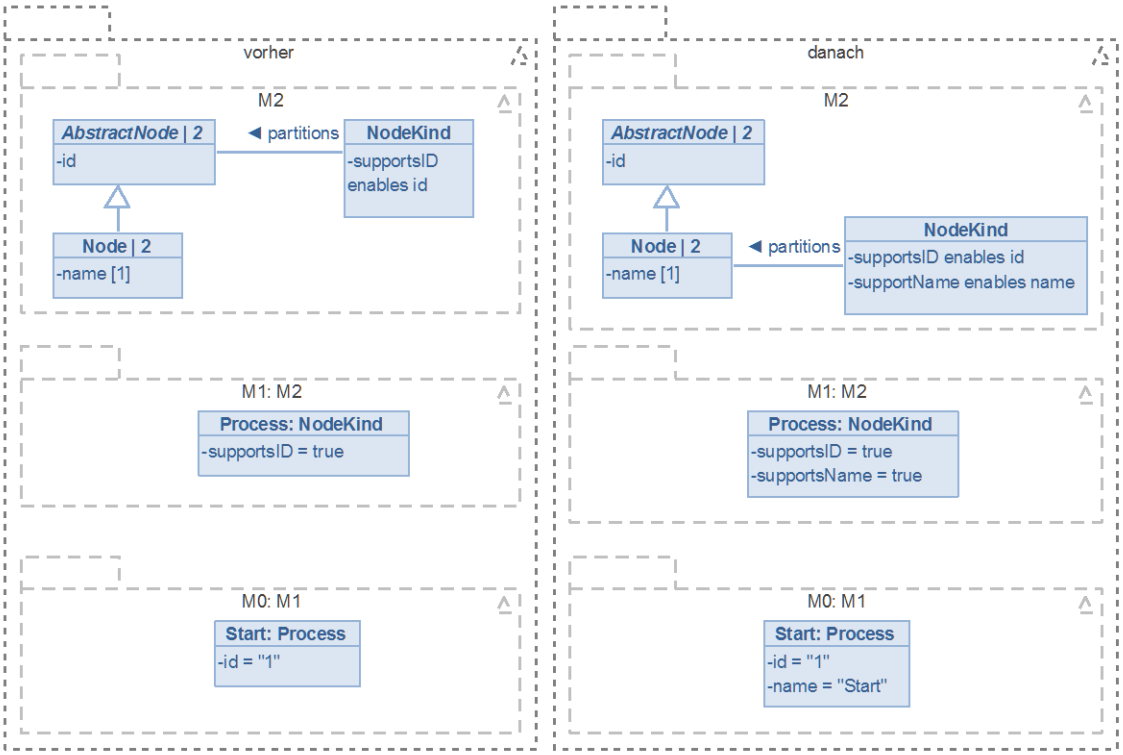

Abbildung 5-29 Beispiel vor (links) und nach (rechts) der Anwendung des Set Partition Operators

Um die Nachlässigkeit beim Modellieren zu beheben, soll die Partitionierung von **NodeKind** auf **Node** geändert werden. Deshalb wird der *Set Partition* Operator an **NodeKind** aufgerufen. Im ersten Schritt wird **Node** als neuer partitionierter Typ gewählt. Da **NodeKind** keine Generalisierung besitzt, wird als nächstes mit **AbstractNode** der bisherige partitionierte Typ bestimmt. Aufgrund der Tatsache, dass **Node** eine Spezialisierung von **AbstractNode** ist, wird als nächstes die Partitionierung von **NodeKind** ersetzt, sodass sie nun auf **Node** zeigt. Anschließend wird ein neues Diskriminatorattribut erzeugt, da ein erweiterter Powertyp vorliegt und **name** (Attribut des neuen partitionierten Typs) keines besitzt. Danach werden alle obligatorischen Attribute gesetzt. Da **Process** einen Instanz von **NodeKind** ist, muss das neue Diskriminatorattribut **supportsName** gesetzt werden. Es erhält den Wert **true**, wodurch **Process** das Attribut **name** erbt. Weil **NodeKind** ein Powertyp ist, müssen auch noch alle obligatorischen Attribute von **Node** gesetzt werden. Da **supportsName** bei **Process** auf **true** gesetzt ist und damit **name** bei **Process** sichtbar ist, muss **Start name** einen Wert zuweisen. Im Beispiel wird der Wert **„Start“** gewählt. Anschließend ist der Operator beendet und das in Abbildung 5-29 auf der rechten Seite gezeigte Modell entsteht.

5.3.17 Set Concrete Use Of

Auswirkung

Der Operator setzt einen (neuen) Prototyp am aufgerufenen Konzept und migriert den bisherigen Prototyp oder einen bisherigen instanziierten Typ.

Ablauf

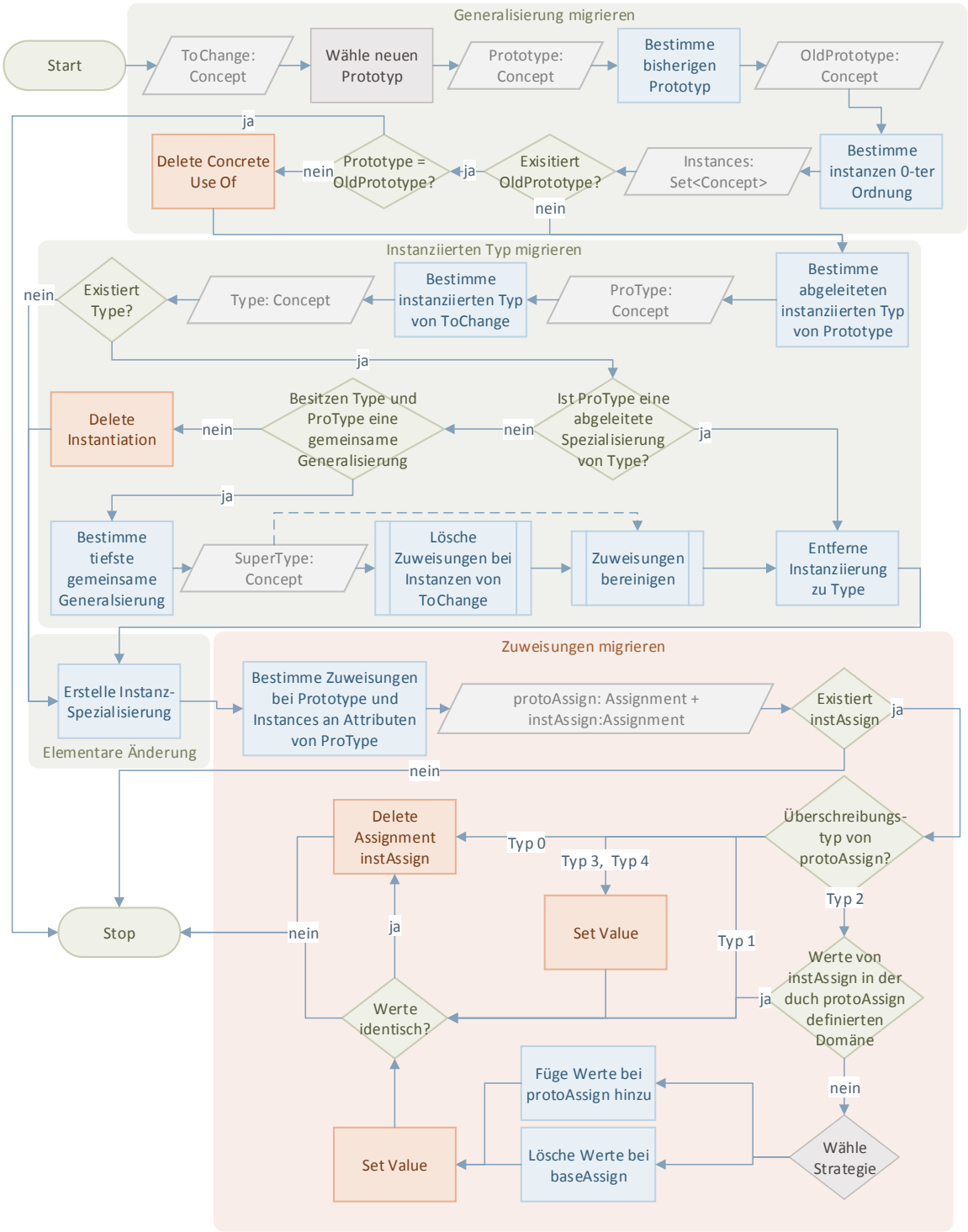

Abbildung 5-30 Ablauf des Set Concrete Use Of Operators

Prototyp migrieren

Der *Set Concrete Use Of* Operator, dessen Ablauf in Abbildung 5-30 gezeigt ist, wird mit einem Konzept `ToChange` initialisiert. Danach wird der neue Prototyp (`Prototype`) aus der Menge an möglichen Kandidaten (Regel C.15) gewählt und anschließend werden alle Instanzen 0-ter Ordnung (`Instances`) von `ToChange` bestimmt. Im nächsten Schritt ermittelt der Operator den bisherigen Prototyp `OldPrototype` von `ToChange`. Falls dieser existiert und ungleich `Prototype` ist, wird mit Hilfe des *Delete Concrete Use* Of Operators die Instanz-Spezialisierung durch eine Instanziierung zum instanziierten Typ des Prototyps ersetzt. Wenn `OldPrototype` und `Prototype` gleich sind, muss der Operator keine Änderungen vollziehen und kann beendet werden.

Instanziierten Typ migrieren

Als nächstes wird der abgeleitete instanziierte Typ von `Prototype` (`ProType`) ermittelt und anschließend der instanziierte Typ `Type` von `ToChange` bestimmt. Falls dieser nicht existiert, kann die elementare Änderung als nächstes ausgeführt werden. Wenn jedoch `Type` existiert und `ProType` eine abgeleitete Spezialisierung davon ist, wird lediglich die Instanziierung entfernt und anschließend ebenfalls die elementare Änderung vollzogen. Für den Fall, dass `ProType` keine abgeleitete Spezialisierung von `Type` ist, muss noch untersucht werden, ob die beiden Konzepte eine gemeinsame abgeleitete Generalisierung besitzen. Wenn dem so ist, müssen bestimmte Zuweisungen entfernt werden, die durch die spätere (implizite) Instanziierung von `ToChange` zu `ProType` nicht mehr gelten. Um dies zu vollziehen, wird zunächst die tiefste gemeinsame abgeleitete Generalisierung [38] (`SuperType`) von `Type` und `ProType` ermittelt, damit diese als `SuperType` Parameter zusammen mit `ToChange` als `Base`, sowie `Type` als gleichnamiger Parameter der Subroutine zum Löschen der Zuweisungen (Abschnitt 5.3.6) übergeben werden kann. Neben den ungültigen Zuweisungen muss auch das Substitutionsprinzip von `ToChange` zu `Type` angepasst werden. Dazu wird `ToChange` als `Base` Parameter zusammen mit `Type` und `SuperType` als gleichnamige Parameter der Subroutine aus Abschnitt 5.3.7 übergeben. Danach wird noch die Instanziierung von `ToChange` zu `Type` entfernt und es schließt sich in diesem Fall ebenfalls die Durchführung der elementaren Änderungen an. Falls keine gemeinsame abgeleitete Generalisierung zwischen `Type` und `ProType` existiert, wird der *Delete Instantiation* Operator verwendet, um die Instanziierung aufzuheben.

Elementare Änderung

Danach wird die Instanz-Spezialisierung von `ToChange` zu `Prototype` hergestellt.

Zuweisungswerte festlegen

Durch die spezielle Semantik der Instanz-Spezialisierung werden Zuweisungen an instanziierte Attribute des Prototyps vererbt. Deshalb müssen alte Zuweisungen von `ToChange` entsprechend angepasst werden. Dazu werden zu jedem Attribut von `ProType` die Zuweisung durch `Prototype` (`protoAssign`) und jedes Element aus `Instances` (beinhaltet `ToChange` und alle Instanz-Spezialisierungen davon) bestimmt. Jedes dieser Paare an Zuweisungen (`protoAssign` ist die Zuweisung beim Prototyp und `instAssign` die Zuweisung bei der jeweiligen Instanz 0-ter Ordnung) wird nun durch den Operator, je nachdem welcher Überschreibungstyp die Prototyp-Zuweisung hat, angepasst. Wenn allerdings `instAssign` nicht existiert, muss keine Änderung vorgenommen werden, da der Wert vom jeweiligen direkten Prototyp der Instanz geerbt wird. Für den Fall, dass der Überschreibungstyp 0 (`forbidden`) vorliegt und dadurch eine Überschreibung verboten ist, muss die Zuweisung durch den *Delete Assignment* Operator entfernt werden. Wenn Typ 1 (`normal`) vorliegt,

[38] Eine Vorgehensweise zur Berechnung findet sich im Teilschritt „Instanziierten Typ migrieren" des *Set Instantiation* Operators im Abschnitt 5.3.14.

muss keinerlei Anpassung geschehen, da hier eine beliebige Überschreibung des Attributes zugelassen ist. Hat eine Zuweisung den Typ 2 (**limited**) muss zunächst unterschieden werden, ob der Wert von **instAssign** eine Teilmenge des Wertes von **protoAssign** darstellt, da nun die letztgenannte Zuweisung als Domäne für die erste fungiert. Ist dies der Fall, müssen keine Anpassungen an **instAssign** geschehen. Falls nicht können die Werte, die bei **instAssign** angenommen werden und bei **protoAssign** nicht vorhanden sind, entweder bei **instAssign** gelöscht oder bei **protoAssign** hinzugefügt werden. Die letztgenannte Alternative entspricht dabei einer Ausweitung der Domäne. In beiden Fällen wird dann der *Set Value* Operator verwendet, um die entsprechende Zuweisung anzupassen.

Wenn die Zuweisung bei **protoAssign** den Überschreibungstyp 3 (**append**) oder 4 (**prepend**) annimmt, kann ein Element aus **Instances protoAssign** um seine Werte erweitern. Deshalb wird der Wert von **instAssign** mit Hilfe des *Set Value* Operators entsprechend durch den Benutzer angepasst. Zum Schluss wird noch (bis auf den Fall für Typ 0) überprüft, ob die Werte von **instAssign** und **protoAssign** nun gleich sind. Ist dies der Fall, so wird **instAssign** gelöscht, da die Zuweisung dann vom Prototyp geerbt wird. Nachdem alle Zuweisungen der Instanzen 0-ter Ordnung von **ToChange** angepasst wurden, ist die Ausführung des Operators beendet.

Beispiel

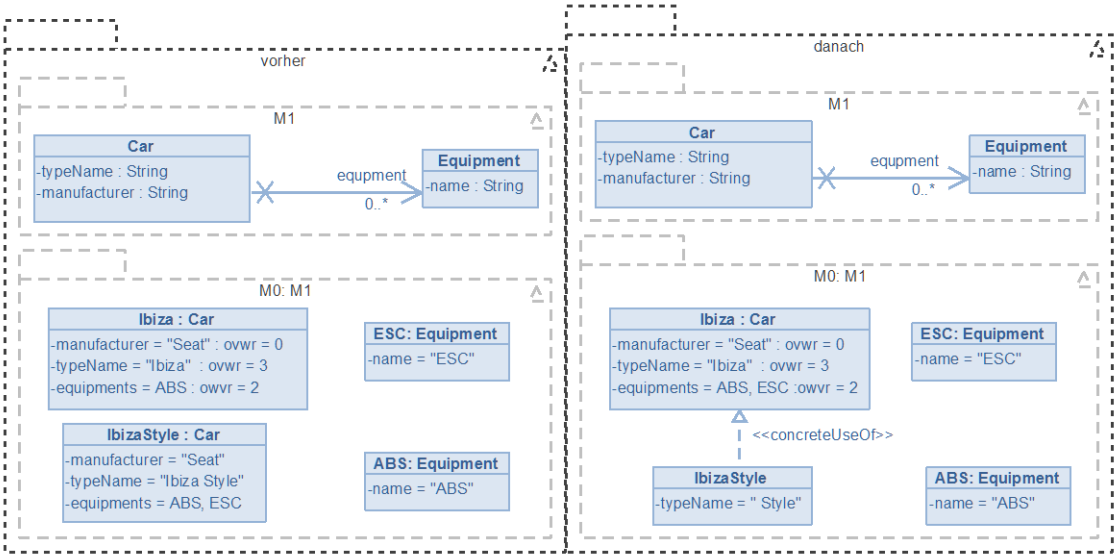

Abbildung 5-31 Beispiel vor (links) und nach (rechts) der Ausführung des Set Concrete Use Of Operators

In Abbildung 5-31 ist links ein ähnliches Modell wie im Beispiel des *Extract Prototype* Operators gegeben. Daher findet sich die Beschreibung des dargestellten Sachverhaltes auch größten Teils im Abschnitt 6.4.1 und es soll hier nur auf die Unterschiede eingegangen werden. Auf **M0** ist das Konzept **Ibiza** gegeben und das Konzept **IbizaReference** nicht vorhanden. **Ibiza** hat als Hersteller (Attribut **manufacturer**) den Wert **„Seat“** (Überschreibungstyp 0, **forbidden**) und als Typbezeichnung (**typeName**) **„Ibiza“** (Überschreibungstyp 3, **append**). Weiterhin besitzt es als Ausstattung (Attribut **equipment**) ABS, was durch das gleichnamige Konzept und die entsprechende Zuweisung (Überschreibungstyp 2, **limited**) modelliert wurde.[39]

[39] Die in diesem Beispiel verwendete Ausstattung der modellierten Marken, muss nicht mit der Ausstattung der real existierenden Marken übereinstimmen.

Da der Ibiza Style eine Variante des Ibiza darstellt, sollen nun die dazugehörigen Konzepte mittels des *Set Concrete Use Of* Operators eine Instanz-Spezialisierung eingehen. Dazu wird am Konzept `IbizaStyle` der Operator aufgerufen und im ersten Schritt `Ibiza` als neuer Prototyp gewählt. Anschließend werden alle Instanzen 0-ter Ordnung von `IbizaStyle` bestimmt, was als Ergebnis nur das Konzept selbst als Menge `Instances` liefert, da keine weiteren Instanz-Spezialisierungen existieren. Da kein bisheriger Prototyp bei `IbizaStyle` gesetzt war, wird als nächstes der instanziierte Typ von `Ibiza` und anschließend von `IbizaStyle` bestimmt, was in diesem Fall bei beiden das Konzept `Car` ist. Da `Car` eine abgeleitete Spezialisierung von sich selbst ist, wird danach die Instanziierung von `IbizaStyle` zu `Car` gelöscht und die Instanz-Spezialisierung (im Diagramm durch den `<<concreteUseOf>>` Pfeil visualisiert) hergestellt. Anschließend werden alle Attribute von `Car`, dies sind `manufaturer`, `typeName` und `equipment` zusammen mit den Zuweisungen von `Ibiza` und `IbizaStyle` bestimmt. Da der Wert von `manufacturer` nicht überschreiben werden darf, wird die Zuweisung von `IbizaStyle` an das Attribut gelöscht. Das Attribut `typeName` hat am Prototyp `Ibiza` den Überschreibungstyp 3, daher wird der Wert an dieser Stelle bei `IbizaStyle` auf **„ Style“** gesetzt. Beim letzten Attribut `equipment` liegt der Überschreibungstyp 2 bei `Ibiza` vor, was zur Folge hat, dass `Ibiza` die Domäne des Attributs für `IbizaStyle` festlegt. Da `ESC` bei `Ibiza` nicht gesetzt wurde, liegt es auch nicht in der Domäne. In diesem Beispiel wird die Domäne erweitert und folglich der Wert `ESC` bei `Ibiza` für das Attribut `equipment` hinzuzufügen. Weil nun die Zuweisungen an `equipment` bei `Ibiza` und `IbizaStyle` den gleichen Wert haben, wird die Zuweisung bei `IbizaStyle` gelöscht, da der Wert durch die Instanz-Spezialisierung ohnehin vererbt wird. Das Resultat des Operatoraufrufes findet sich in Abbildung 5-31 auf der rechten Seite.

5.3.18 Delete Concept

Auswirkung

Der Operator entfernt ein Konzept, migriert alle abhängigen Konzepte und löscht alle Attribute und Zuweisungen, die das Konzept deklariert hatte.

Ablauf

Abhängige Konzepte migrieren

Der Operator (Abbildung 5-32) wir an einem Konzept `ToDelete` aufgerufen, dass entfernt werden soll. Zunächst werden alle Spezialisierungen von `ToDelete` bestimmt, damit anschließend der *Delete Specialization* Operator an ihnen aufgerufen werden kann. Danach berechnet der Operator alle Instanzen und entfernt durch den Aufruf des *Delete Instantiation* Operators an Ihnen die Instanziierung. Ähnlich wird mit dem eventuell vorhandenen Powertyp verfahren. Dieser wird zunächst bestimmt, um dann den *Delete Partition* Operator aufzurufen, der die Partitionierung auflöst. Danach werden alle Instanz-Spezialisierungen migriert. Dazu werden diese zunächst ermittelt, damit anschließend die Instanz-Spezialisierung mit Hilfe des *Delete Concrete Use Of* Operators aufgelöst wird.

Beziehungen und Zuweisungen migrieren

Im nächsten Schritt werden alle Beziehungen zu `ToDelete` entfernt. Dazu werden alle Attribute bestimmt, die als Typ `ToDelete` besitzen, damit diese danach mit Hilfe des *Delete Attribute* Operators gelöscht werden. Analog werden anschließend alle Zuweisungen, die als Wert `ToDelete` zuweisen, ermittelt und mit dem *Set Value* Operator um diesen Wert bereinigt.

Zuweisungen und Attribute löschen

Als nächstes werden alle Attribute, die `ToDelete` definiert, bestimmt und dann mittels des *Delete Attribute* Operators entfernt. Ebenso werden alle Zuweisungen von `ToDelete`, nachdem sie bestimmt wurden, durch den *Delete Assignment* Operator gelöscht.

Elementare Änderung

Nachdem alle beinhalteten Modellelemente entfernt wurden, kann das Konzept selbst entfernt werden, bevor der Operator stoppt.

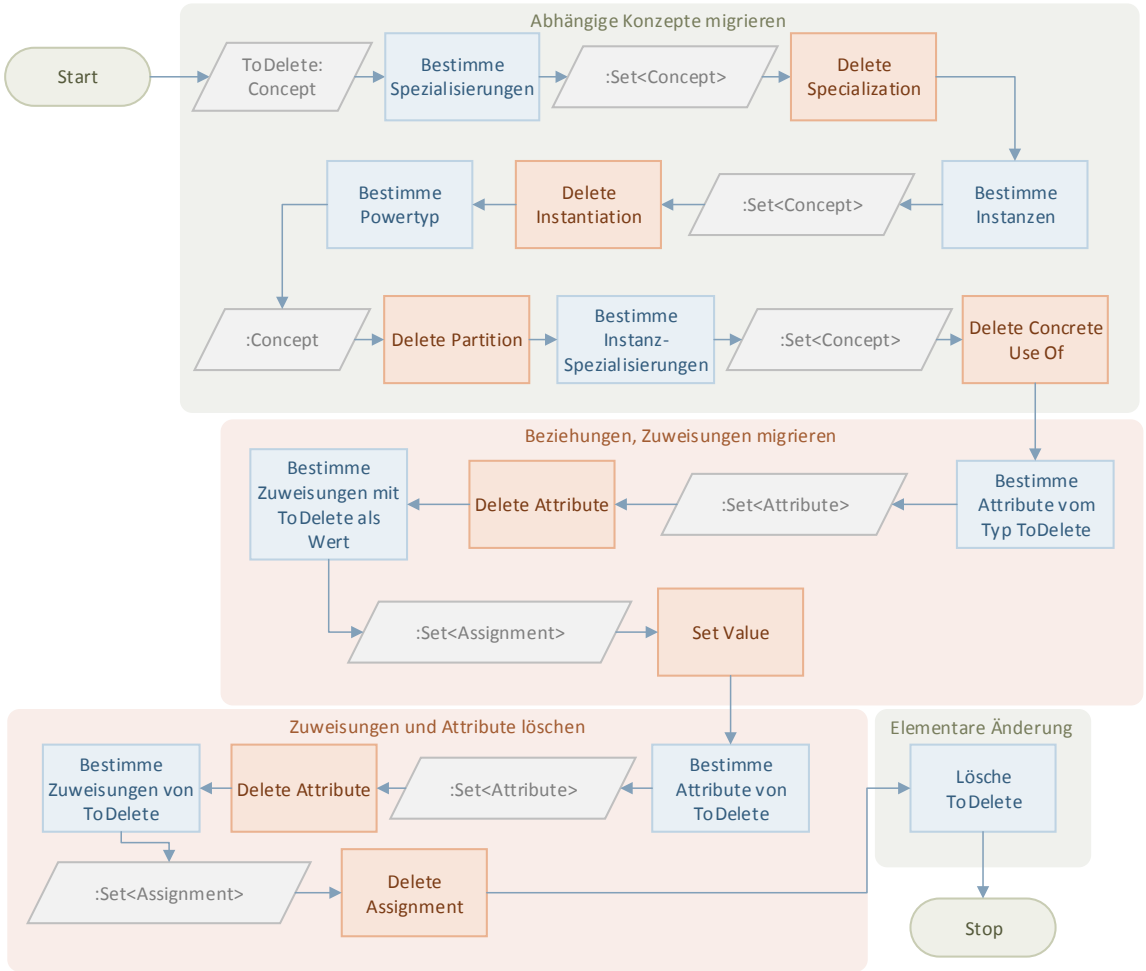

Abbildung 5-32 Ablauf des Delete Concept Operators

5.4 Enumeration (Enum)

Enumerationen dienen der Definition einer festgelegten Menge an Werten, die bei einer Zuweisung gesetzt werden dürfen. Im Folgenden werden die Operatoren zum Ändern der Literale (Menge der Werte) einer Enumeration und zum Löschen einer Enumeration vorgestellt.

5.4.1 Set Enum Literals

Auswirkung

Der Operator setzt Literale einer Enumeration und bereinigt dabei alle Zuweisungen, die ein Literal zugewiesen haben, das durch den Operator entfernt wird.

Ablauf

Zuweisungen bereinigen

Der Operator (Abbildung 5-33) erhält als Ausgangsparameter eine Enumeration `ToChange`. Im ersten Schritt werden alle bisherigen Literale `OldLiterals` bestimmt und anschließend die neue Menge an

Literalen (`Literals`) von `ToChange` gewählt. Danach werden alle Attribute gesucht, deren Typ `ToChange` ist, damit im nächsten Schritt die zu den Attributen gehörigen Zuweisungen bestimmt werden können. Diese werden dann mit Hilfe des *Set Value* Operators um alle Werte aus `OldLiterals` bereinigt.

Elementare Änderung

Danach werden die Literale aus `Literals` bei `ToChange` gesetzt

Leere Enumeration entfernen

Falls `ToChange` keine weiteren Literale mehr besitzt (die Auswahl von `Literals` also leer war), wird auch die Enumeration mit Hilfe des *Delete Enum* Operators entfernt und die Ausführung des Operators ist beendet.

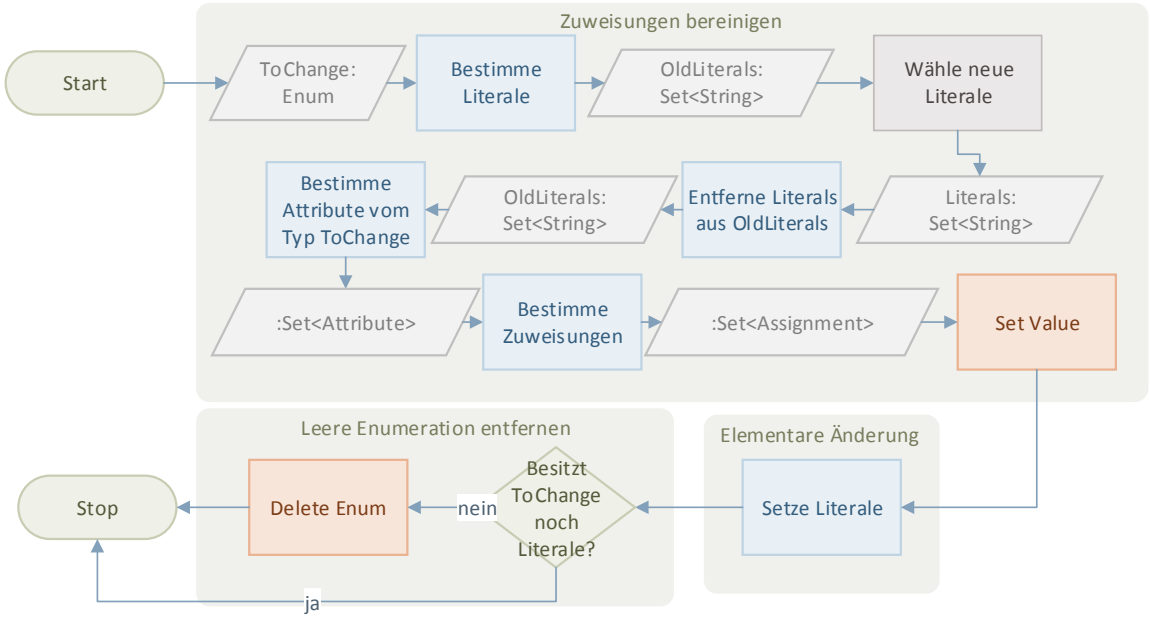

Abbildung 5-33 Ablauf des Set Enum Literal Operators

5.4.2 Delete Enum

Auswirkung

Der Operator entfernt die Enumeration, an der er aufgerufen wurde und löscht alle Attribute, die als Typ die Enumeration besitzen.

Ablauf

Attribute entfernen

Zu Beginn wird der Operator (Abbildung 5-34) an einer Enumeration `ToDelete` aufgerufen. Im ersten Schritt werden alle Attribute vom Typ `ToDelete` ermittelt und anschließend durch den *Delete Attribute* Operator gelöscht.

Elementare Änderung

Im letzten Schritt wird die Enumeration (inklusive der Literale) entfernt und der Operator endet.

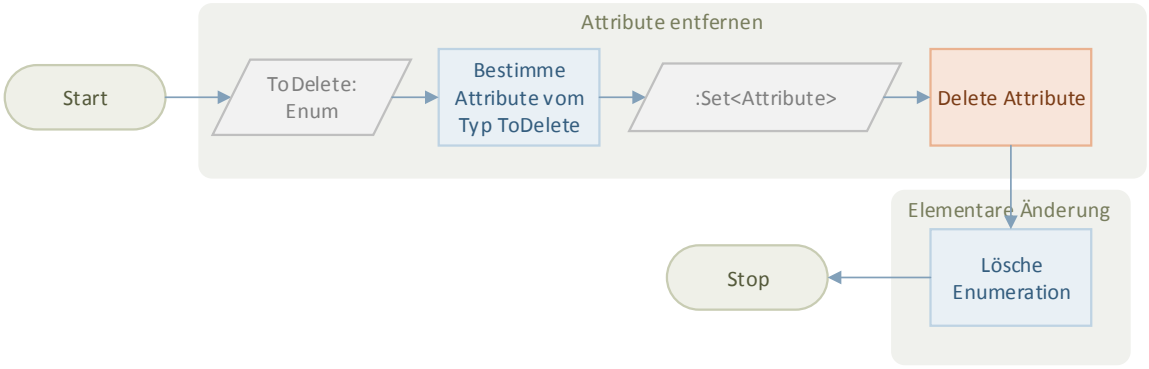

Abbildung 5-34 Ablauf des Delete Enum Operators

5.5 Attribut (Attribute)

Die Operatoren in diesem Abschnitt behandeln elementare Änderungen eines Attributs und migrieren betroffene Modellelemente entsprechend. Attribute definieren Eigenschafen eines Konzepts und besitzen eine Multiplizität, eine Sichtbarkeit, einen Deep Instantiation Zähler, eine Materialisierungserweiterung und einen Attributtyp. Weiterhin können sie komposit oder ein Diskriminatorattribut sein. Für die Materialisierungserweiterung wurde für jeden Bestandteil (`MemberDefinition`, `Visibiliy` und `Multiplicity`) ein separater Operator zum Setzen der Eigenschaft und zusätzlich noch ein weiterer Operator definiert (Abschnitt 5.5.8), der die gesamte Erweiterung löscht und damit die Materialisierung auflöst.

5.5.1 Set Multiplicity

Auswirkung

Die Ausführung des Operators ändert die Multiplizität eines Attributes und passt die entsprechenden Zuweisungen so an, dass sie der neuen Multiplizität genügen.

Ablauf

Fehlende Zuweisung erstellen

Der in Abbildung 5-35 dargestellte Ablauf des Operators beginnt mit dem Aufruf an einem Attribut `ToChange`, von dem im ersten Schritt die Multiplizität (`oldMulti`) bestimmt wird. Danach wird die neue Multiplizität `multi` gewählt. Wenn die Untergrenze von `0` auf `1` angehoben wurde, bedeutet dies, dass nun das Attribut nicht mehr optional sondern obligatorisch ist. Folglich werden im nächsten Schritt zunächst der Deep Instantiation Zähler (`n(ToChange)`) und anschließend alle abgeleiteten Spezialisierungen des umgebenden Konzepts (`Containers`) bestimmt. Damit können alle Instanzen bis zur Ordnung `n(ToChange)` von `Containers` (`Instances`) ermittelt werden, da diese `ToChange` setzten müssen. Für jede Instanz `n(ToChange)`-ter Ordnung[40] wird also überprüft, ob eine virtuelle Zuweisung an `ToChange` existiert. Wenn dies nicht der Fall ist, wird eine neue Zuweisung direkt bei der Instanz `n(ToChange)`-ter Ordnung oder einem abgeleiteten instanziierten Typ davon (entspricht einem Standardwert) erstellt und durch den *Set Value* Operators ein neuer Wert für die Zuweisung gesetzt. Für den Fall, dass das Minimum nicht erhöht wurde, kann direkt mit dem bereinigen der Werte fortgefahren werden.

[40] Diese Instanzen sind eine Teilmenge der Menge `Instances`, die deshalb auch alle Instanzen kleinerer Ordnung enthält, weil deren Zuweisungen als Standardwert fungieren können.

Werte bereinigen

Falls das Maximum der Multiplizität (auch Kardinalität genannt) von `*` auf `1` verringert wurde, müssen alle Zuweisungen, die bisher mehr als einen Wert besitzen, bereinigt werden. Deshalb werden zunächst alle Zuweisungen bestimmt und überprüft, ob mehr als ein Wert zugewiesen wurde. Falls dem so ist, muss ein Wert ausgewählt werden und die Zuweisung mit Hilfe des *Set Value* Operators bereinigt werden.

Elementare Änderung

Im letzten Schritt wird `multi` als neue Multiplizität von `ToChange` gesetzt und die Ausführung des Operators endet.

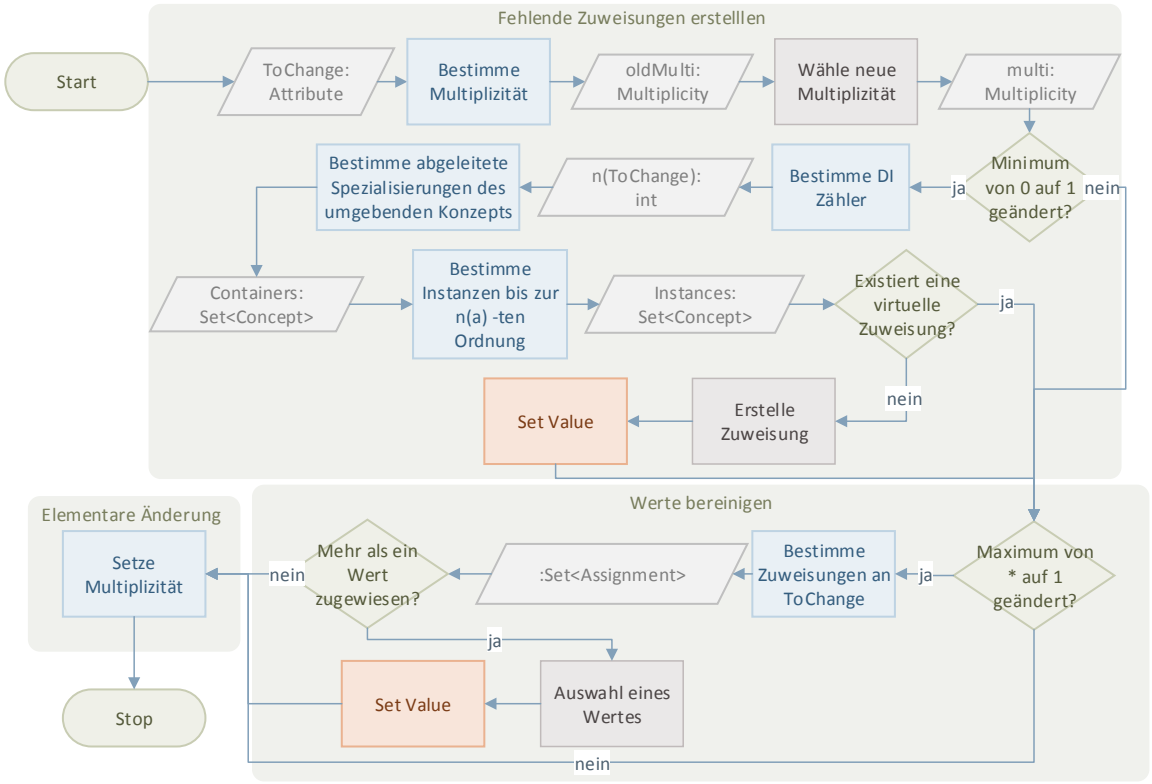

Abbildung 5-35 Ablauf des Set Multiplicity Operators

Beispiel

Abbildung 5-36 zeigt auf der linken Seite ein Modell in dem auf Ebene `M1` ein Mitarbeiter (`Employee`) keiner oder mehrerer Abteilungen (`Department`) zugeordnet (Attribut `worksFor`) ist. Ebene `M0` definiert vier Konzepte: `Alice` und `Bob` als Instanzen von `Employee`, sowie `Sales` und `Office` als Instanzen von `Department`. `Bob` ist dabei zwei Abteilungen (`Sales` und `Office`) zugewiesen, während `Alice` frei von einer Abteilung arbeitet.

Im Folgenden soll nun jeder Mitarbeiter genau einer Abteilung zugeordnet werden. Deshalb wird der *Set Multiplicity* Operator am Attribut `worksFor` aufgerufen und zunächst die bisherigen Multiplizität `0..*` bestimmt. Danach wählen wir als neue Multiplizität `1`. Da deshalb `worksFor` obligatorisch wird, werden danach alle Instanzen bis zur 1-ten (Deep Instantiation Zähler) Ordnung von `Employee` (umgebendes Konzept ohne weitere abgeleitete Spezialisierungen) ermittelt, was in diesem Beispiel `Employee` (Instanz 0-ter Ordnung, Möglichkeit des Setzens eines Standardwertes), `Alice` und `Bob` als Ergebnis liefert. Da `Alice` keine Zuweisung besitzt, wird entschieden für sie `Office` als neuen Wert festzulegen. Dadurch erstellt der Operator die entsprechende Zuweisung und ruft anschließend den

Set Value Operator auf, in dem der neue Wert von uns ausgewählt wird. Danach werden, aufgrund der Tatsache, dass auch die Kardinalität verringert wurde, alle Zuweisungen an `worksFor` ermittelt, was als Ergebnis die Zuweisungen bei `Bob` und `Alice` liefert. Da `Bob` zwei Werte zuweist, wird einer (`Sales`) ausgewählt und die Zuweisung um den Wert `Office` bereinigt. Als letztes erhält `worksFor` die Multiplizität `1` und das in Abbildung 5-36 auf der rechten Seite gezeigte Modell entsteht.

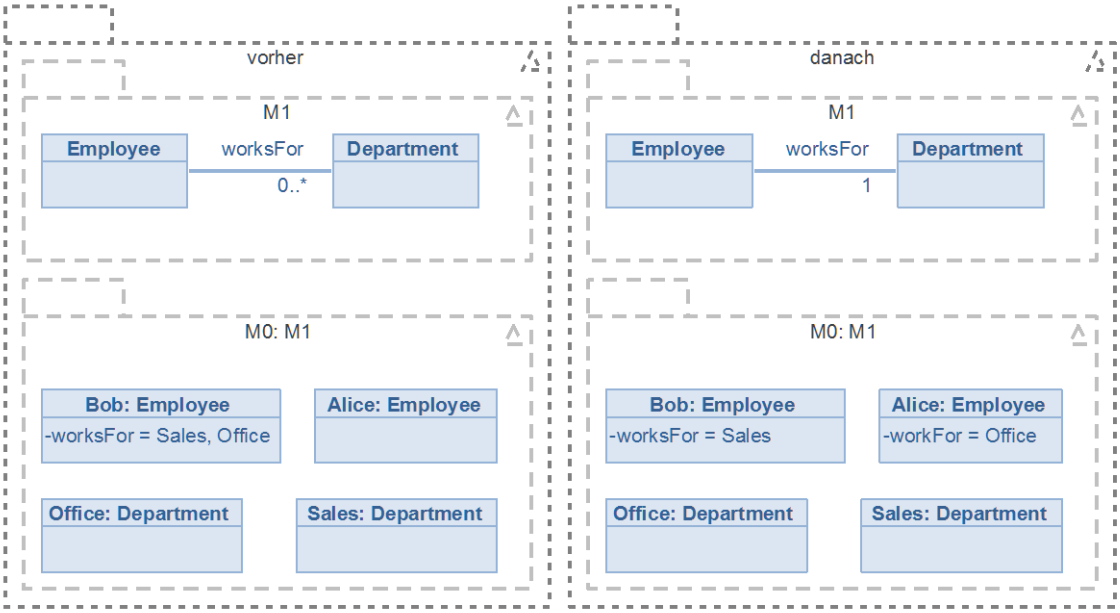

Abbildung 5-36 Beispiel vor (links) und nach (rechts) der Ausführung des Set Multiplicity Operators

5.5.2 Set Visibility

Auswirkung

Der Operator ändert die Sichtbarkeit eines Attributes und löscht ungültige Zuweisungen, die dadurch entstehen, dass die Sichtbarkeit eingeschränkt wurde.

Ablauf

Zuweisungen entfernen

Zu Beginn wird der Operator an einem Attribut `ToChange` aufgerufen, von dem im ersten Schritt die Sichtbarkeit `oldVis` bestimmt wird. Als nächstes wird die neue Sichtbarkeit `vis` gewählt, wobei `private` als Wert nur dann gewählt werden kann, wenn das umgebende Konzept nicht abstrakt ist (siehe Regel C.4). Wenn die Sichtbarkeit von `ToChange` von `public` auf `private` eingeschränkt wurde, sind nur noch Zuweisungen von direkten Instanzen (und deren abgeleiteten Instanzen beliebiger Ordnung) an `ToChange` möglich. Um diese Bedingung zu überprüfen, wird zunächst der Deep Instantiation Zähler von `ToChange` (`n(ToChange)`) bestimmt und anschließend das umgebende Konzept (`Container`) ermittelt. Wenn die Sichtbarkeit eingeschränkt wurde, werden im nächsten Schritt alle Instanzen bis zur `n(ToChange)`-ten Ordnung (`Instances`) von `Container` berechnet. Als nächstes werden alle Zuweisungen an `ToChange` ermittelt und überprüft, ob das umgebende Konzept in `Instances` liegt. Falls dies nicht zutrifft, wird die jeweilige Zuweisung durch den *Delete Assignment* Operator entfernt und es schließt sich die elementare Änderung an.

Obligatorisches Attribut setzen

Wenn die Sichtbarkeit von `private` auf `public` erweitert wurde und `ToChange` ein obligatorisches Attribut ist, müssen alle Instanzen der abgeleiteten Spezialisierungen das Attribut setzen. Folglich

werden in diesem Fall zunächst alle abgeleiteten Spezialisierungen **Subs** des umgebenden Konzeptes (ohne dieses selbst) bestimmt. Von dieser Menge werden anschließend alle Instanzen **n(ToChange)**-ter Ordnung ermittelt und überprüft, ob eine virtuelle Zuweisung (in diesem Fall ein Standardwert für **ToChange** bei **Container**) existiert. Falls nicht, wird eine Zuweisung erstellt, wobei das entsprechende Element aus **Instances** zunächst festgelegt und mit Hilfe des *Set Value* Operators ein Wert für die neue Zuweisung gewählt wird.

Elementare Änderung

Anschließend wird die neue Sichtbarkeit gesetzt und die Ausführung des Operators endet.

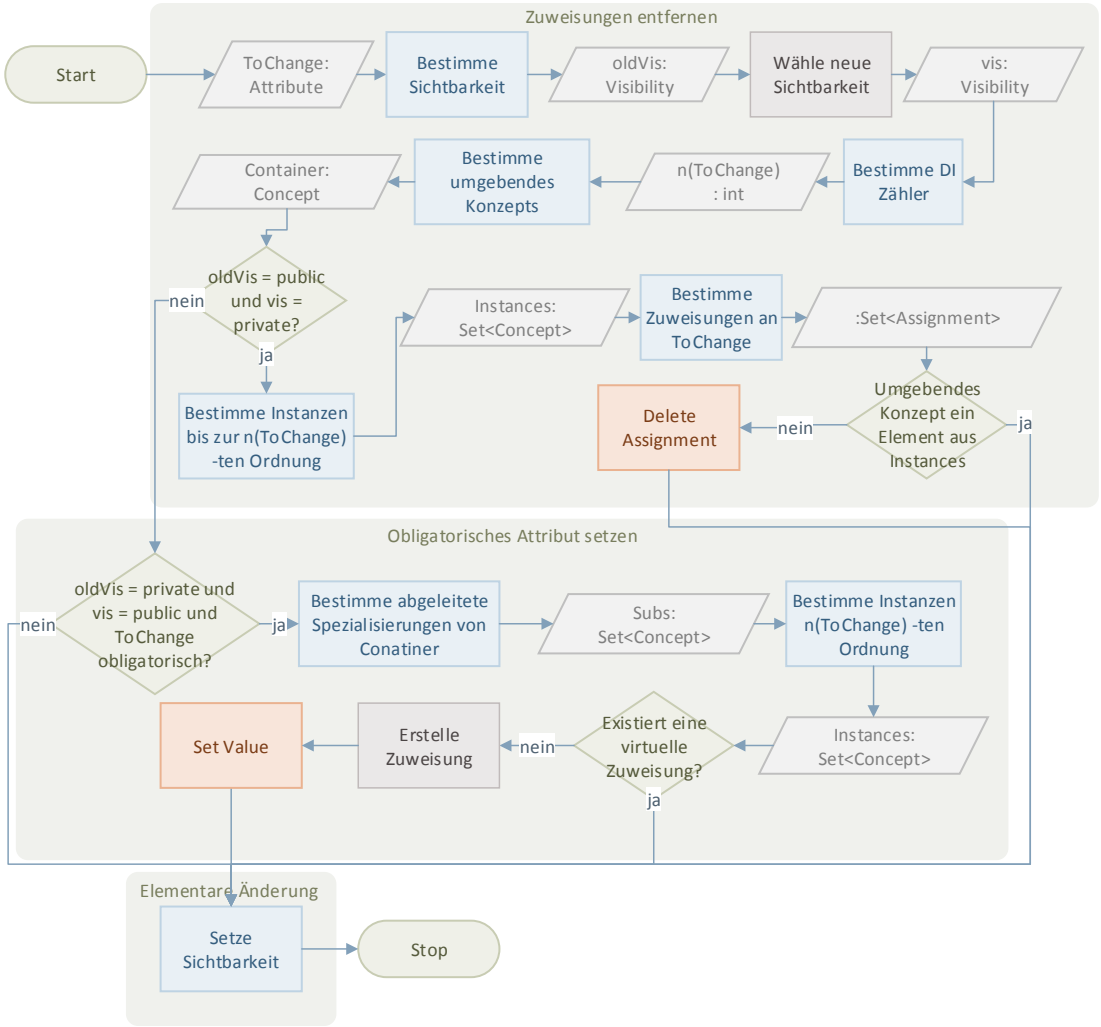

Abbildung 5-37 Ablauf des Set Visibility Operators

Beispiel

In Abbildung 5-38 ist auf der linken Seite ein Beispiel dargestellt, auf das der Operator angewendet werden soll. Auf Ebene **M1** wurde eine Vererbung definiert, in der **Person** die Generalisierung von **Employee** ist. Zudem existiert ein Attribut **height**, das die Größe einer Person angibt und daher bei **Person** definiert wurde. Die Ebene **M0** besitzt zwei Konzepte **Alice** (Instanz von **Person**) und **Bob** (Instanz von **Employee**), die beide das Attribut **height** gesetzt haben.

Nun soll die Sichtbarkeit des Attributs **height** mit Hilfe des *Set Visibility* Operators von **public** (Standardwert) auf **private** gesetzt werden. Dazu wird, nachdem die bisherige Sichtbarkeit **public** ermittelt wurde, die neue Sichtbarkeit **private** gewählt. Da die Sichtbarkeit verengt wurde, wird der

Deep Instantiation Zähler (= `1`, ebenfalls Standardwert) von `height` und `Person` als umgebendes Konzept bestimmt. Folglich müssen alle Instanzen bis zur 1-ten Ordnung berechnet werden, was in diesem Beispiel `Person` (Instanz 0-ter Ordnung) und `Bob` (Instanz 1-ter Ordnung) als Ergebnis liefert. Danach werden alle Zuweisungen an `height` bestimmt. Aufgrund der Tatsache, dass `Alice` keine Instanz von `Person`, sondern von `Employee` ist, wird die Zuweisung von `Alice` an `height` gelöscht. Danach kann die Sichtbarkeit von `height` zu `private` geändert werden und das in Abbildung 5-38 auf der rechten Seite gezeigte Modell entsteht.

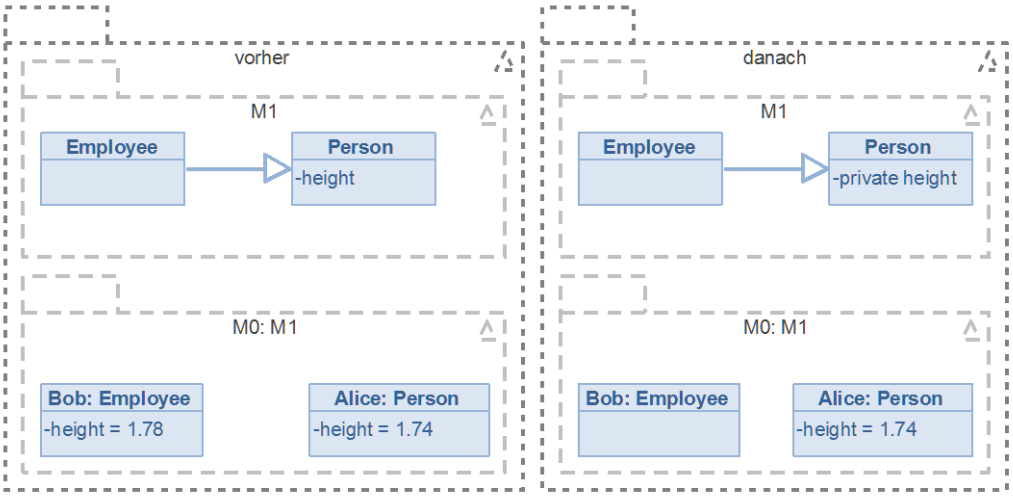

Abbildung 5-38 Beispiel vor (links) und nach (rechts) der Ausführung des Set Visibility Operators

5.5.3 Set Composite

Auswirkung

Der Operator setzt ein Attribut komposit und stellt anschließend sicher, dass jeder Wert bei genau einer Zuweisung gesetzt wurde.

Ablauf

Eindeutige Zuweisung überprüfen

Zu Beginn wird der Operator (Abbildung 5-39) an einem Attribut `ToChange` aufgerufen, dessen Typ ein Konzept ist, da ansonsten das Setzen der komposit Eigenschaft nicht sinnvoll ist. Anschließend werden alle Attribute bestimmt, die den gleichen Attributtyp wie `ToChange` besitzen. Ist unter diesen Attributen bereits ein komposites Attribut, dann wird der Operator beendet, da jedes Konzept nur Teil eines anderen Konzepts sein kann. Wenn noch kein anderes komposites Attribut existiert, werden zunächst alle Zuweisungen an `ToChange` (`Assigns`) ermittelt und dazu verwendet, um festzustellen welche Konzepte bei mehr als einer Zuweisung gesetzt wurden. Für jedes dieser Konzepte muss dann eine Zuweisung gewählt werden, die nachfolgend als einzige den Wert besitzt. Alle anderen Zuweisungen werden durch den *Set Value* Operator vom jeweiligen Wert (dem Konzept) bereinigt. Anschließend werden der Deep Instantiation Zähler von `ToChange` (`n(ToChange)`) und alle abgeleiteten Spezialisierungen des Attributtyps von `ToChange` ermittelt. Danach berechnet der Operator für diese Menge alle Instanzen `n(ToChange)`-ter Ordnung (`Instances`). Jedes Element aus `Instances`, das bei keiner Zuweisung von `Assigns` als Wert zugewiesen wird, muss dann durch den *Delete Concept* Operator entfernt werden.

Elementare Änderung

Zuletzt wird `ToChange` komposit gesetzt und der Operator stoppt.

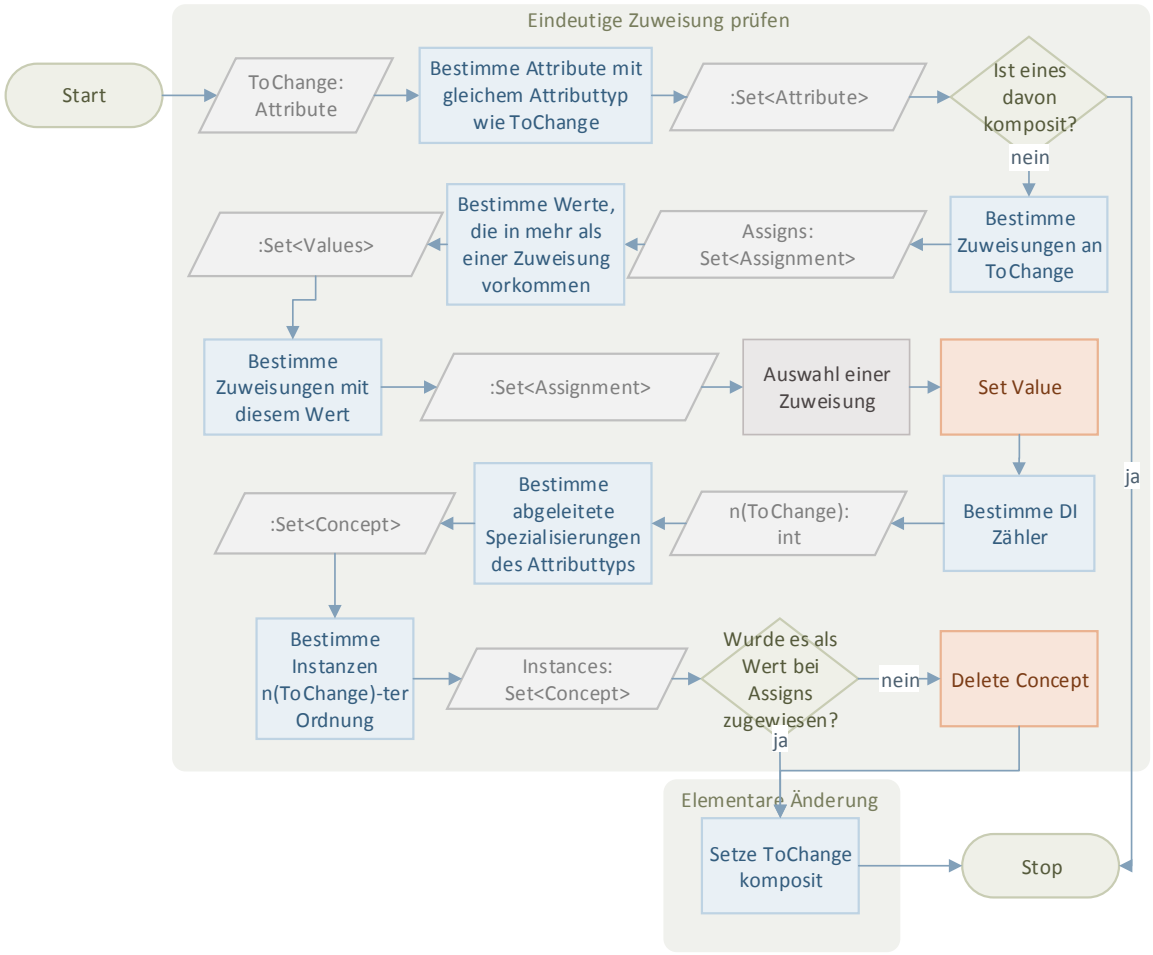

Abbildung 5-39 Ablauf des Set Composite Operators

5.5.4 Set Deep Instantiation Counter of Attribute

Auswirkung

Durch den Operator wird der Deep Instantiation Zähler eines Attributs neu gesetzt.

Ablauf

Zuweisung migrieren

Zu Beginn wird der Operator (Abbildung 5-40) an einem Attribut **`ToChange`** aufgerufen. Dieses Attribut darf kein Diskriminatorattribut sein, da sonst die Änderung des Deep Instantiation Zählers zu Inkonsistenzen führen würde (Regel A.5). Demzufolge wird in diesem Fall der Operator beendet. Ansonsten wird zunächst der Deep Instantiation Zähler **`oldDI`** von **`ToChange`** ermittelt. Anschließend wird der neue Deep Instantiation Zähler **`newDI`** gewählt. Falls **`newDI`** und **`oldDI`** gleich sind, muss keine Änderung erfolgen und der Operator wird beendet. Wenn beide Zähler ungleich sind, wird zunächst untersucht, ob **`ToChange`** ein Referenzattribut ist und als Typ ein Konzept besitzt. Wenn dies der Fall ist, hat sich durch die Änderung des Deep Instantiation Zählers auch der Wertebereich (von Instanzen **`oldDI`**-ter Ordnung zu Instanzen **`newDI`**-ter Ordnung des umgebenden Konzepts) geändert. Deshalb müssen in diesem Fall alle Zuweisungen an **`ToChange`** durch den *Delete Assignment* Operator entfernt werden. Deshalb wird dann festgestellt, ob der neue Deep Instantiation Zähler größer als der alte ist. Falls dies zutrifft, wird mit der elementaren Änderung fortgefahren, weil die Erhöhung des Deep Instantiation Zählers bewirkt, dass alle bisherigen Zuweisungen zu Standardwerten für die jeweiligen

Instanzen des umgebenden Konzepts werden. Deshalb existieren auch im Falle eines obligatorischen Attributs bei allen Instanzen virtuelle Zuweisungen und es treten keine Inkonsistenzen auf.

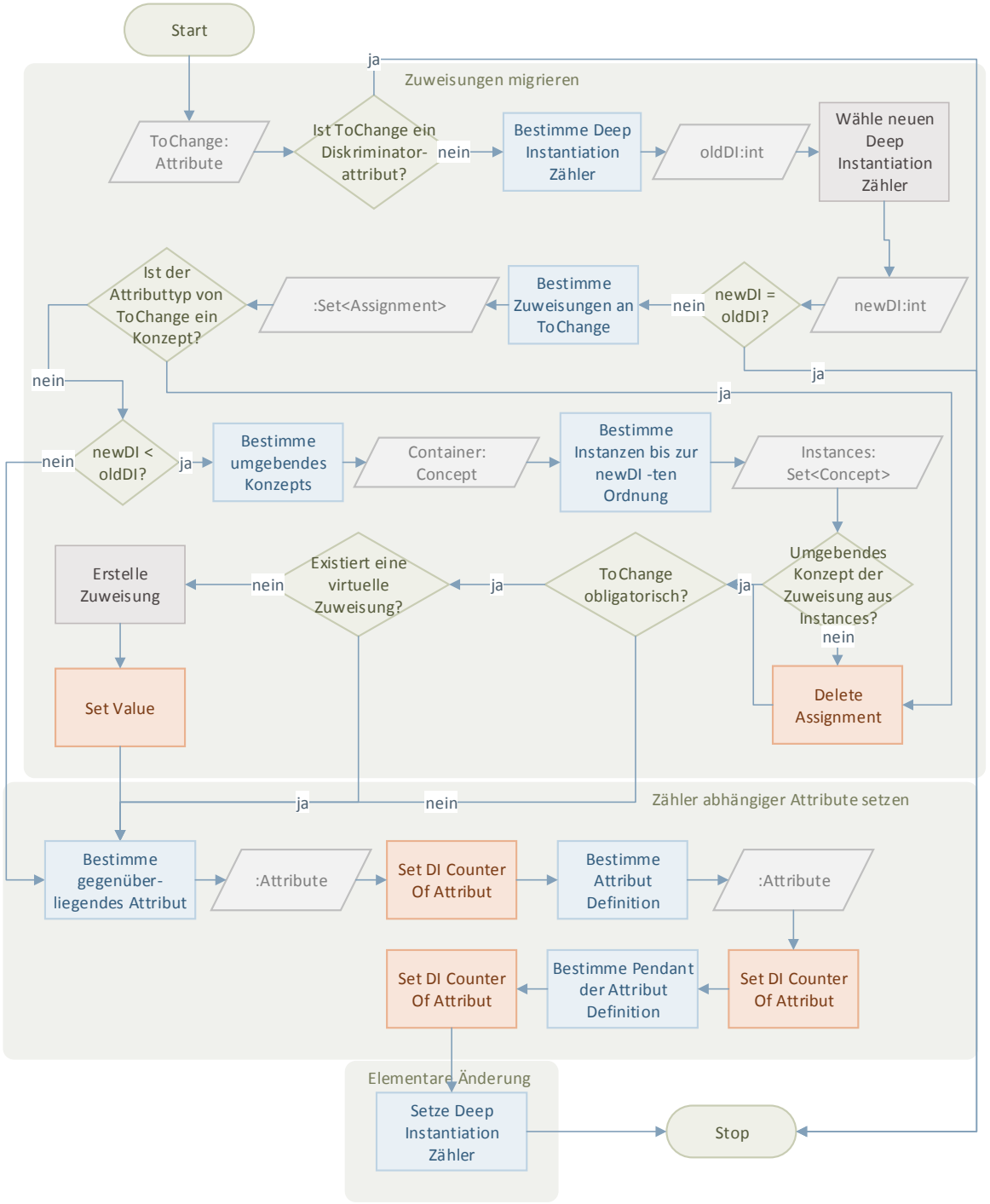

Abbildung 5-40 Ablauf des Set Deep Instantiation Counter of Attribute Operators

Wenn **newDI** kleiner als **oldDI** ist, wird zunächst das umgebende Konzept **Container** von **ToChange** bestimmt Danach werden alle Instanzen bis zu **newDI**-ten Ordnung (**Instances**) von **Container** und alle Zuweisungen von **ToChange** ermittelt. Da Zuweisungen an **ToChange** nur noch von Elementen aus **Instances** definiert werden dürfen, wird dies zunächst überprüft und im Falle eines Verstoßes, wird die entsprechende Zuweisung durch den *Delete Assignment* Operator entfernt. Anschließend wird, falls **ToChange** ein obligatorisches Attribut ist, für jede Instanz **newDI**-ter Ordnung (in **Instances** enthalten) untersucht, ob eine virtuelle Zuweisung vorliegt. Wenn dies nicht der Fall ist, wird eine neue

Zuweisung bei einem entsprechenden Element aus `Instances` (Auswahl von außerhalb des Operators, z.B. Benutzer) erstellt und der Wert mit Hilfe des *Set Value* Operators gesetzt.

Zähler abhängiger Attribute setzen

Danach werden die Deep Instantiation Zähler aller abhängigen Attribute ebenfalls auf den Wert `newDI` gesetzt, falls diese existieren. Diese Attribute sind das gegenüberliegende Attribut, die Attribut-Definition der Materialisierungserweiterung von `ToChange` und das entsprechende Attribut mit der Materialisierungserweiterung, falls `ToChange` eine Attribut-Definition ist. Die Änderung der Wertes wird durch den *Set Deep Instantiation Counter of Attribute* Operator vollzogen, wodurch es erneut zu zyklischen Aufrufen kommen kann, die, wie zu anfangs des Kapitels erwähnt, aufgelöst werden müssen.

Elementare Änderung

Im letzten Schritt des Operators wird der neue Deep Instantiation Zähler `newDI` für `ToChange` gesetzt und der Operator endet.

5.5.5 Set Attribute Type

Auswirkung

Durch den Operator wird der Attributtyp des übergebenen Attributes geändert und alle Zuweisungen die dadurch invalide werden angepasst.

Ablauf

Ein Typ literal, anderer referentiell

Initial wird der Operator (Abbildung 5-41) an einem Attribut `ToChange` aufgerufen, von dem als erstes der bisherige Attributtyp `OldType` bestimmt wird. Danach wird der neue Attributtyp `Type` aus der Menge an möglichen Attributtypen (Regel C.13) gewählt. Anschließend werden alle Zuweisungen an `ToChange` (`Assigns`) zusammen mit den jeweiligen umgebenden Konzepten bestimmt (`AssCon`). Als nächstes wird das umgebende Konzept von `ToChange` (`Container`) ermittelt, bevor unterschieden wird, wie sich der neue Attributtyp zum alten verhält. Wenn einer der beiden literal und der andere referentiell ist, werden alle Zuweisungen aus `Assigns` mit Hilfe des *Delete Assignment* Operators entfernt, bevor die elementare Änderung erfolgt.

Beide referentiell

Für den Fall, dass beide Attributtypen referentiell sind, muss zunächst ein eventuell vorhandenes gegenüberliegendes Attribut `Opposite` bestimmt werden. Danach wird untersucht, ob `Type` eine abgeleitete Generalisierung von `OldType` ist, weil dann keine Inkonsistenzen aufgrund des Substitutionsprinzips entstehen. Deshalb kann in diesem Fall das Attribut `Opposite` mit Hilfe des *Move Attribute to Super Type* Operators zur Generalisierung `Type` verschoben werden. Hierbei wird `OldType` als einzige Spezialisierung und `Opposite` als einziges Attribut im inneren Operator gewählt. Dadurch wird im letzten Schritt des *Move Attribute to Super Type* Operators auch nichts weiter als der neue Typ gesetzt. Folglich ist die elementare Änderung, im Suboperator bereits durchgeführt worden, weshalb sie nicht noch einmal durchgeführt werden muss und als nächstes der Teilschritt Neusetzen der Zuweisungen stattfindet.

Falls `Type` eine abgeleitete Spezialisierung von `OldType` ist, sind i.A. nicht alle Werte der Zuweisungen invalide, da aufgrund des Substitutionsprinzips auch Instanzen von `Type` bzw. von abgeleiteten Spezialisierungen davon bei `ToChange` weiterhin zugewiesen werden dürfen. Deshalb werden in diesem Fall zunächst alle abgeleiteten Spezialisierungen von `Type` (`Subs`) und anschließend der Deep Instantiation Zähler von `ToChange` (`n(ToChange)`) ermittelt. Damit werden alle Instanzen

n(ToChange)-ter Ordnung (**Instances**) der Menge **Subs** berechnet. Jede Zuweisung aus **Assigns** wird dann überprüft, ob ihre Werte in **Instances** liegen. Wenn dies nicht zutrifft, wird mit Hilfe des *Set Value* Operators die Zuweisung bereinigt. Als nächstes muss, wenn es existiert, **Opposite** zur Spezialisierung **Type** durch den *Move Attribute to Sub Type* Operator verschoben werden. Dabei wird **Type** als Spezialisierung und die Strategie, die die Zuweisungen entfernt, gewählt. Ähnlich wie oben beschrieben, führt auch hier der Suboperator bereits die elementare Änderung aus und es kann ebenfalls mit dem Teilschritt Neusetzen der Zuweisungen fortgefahren werden.

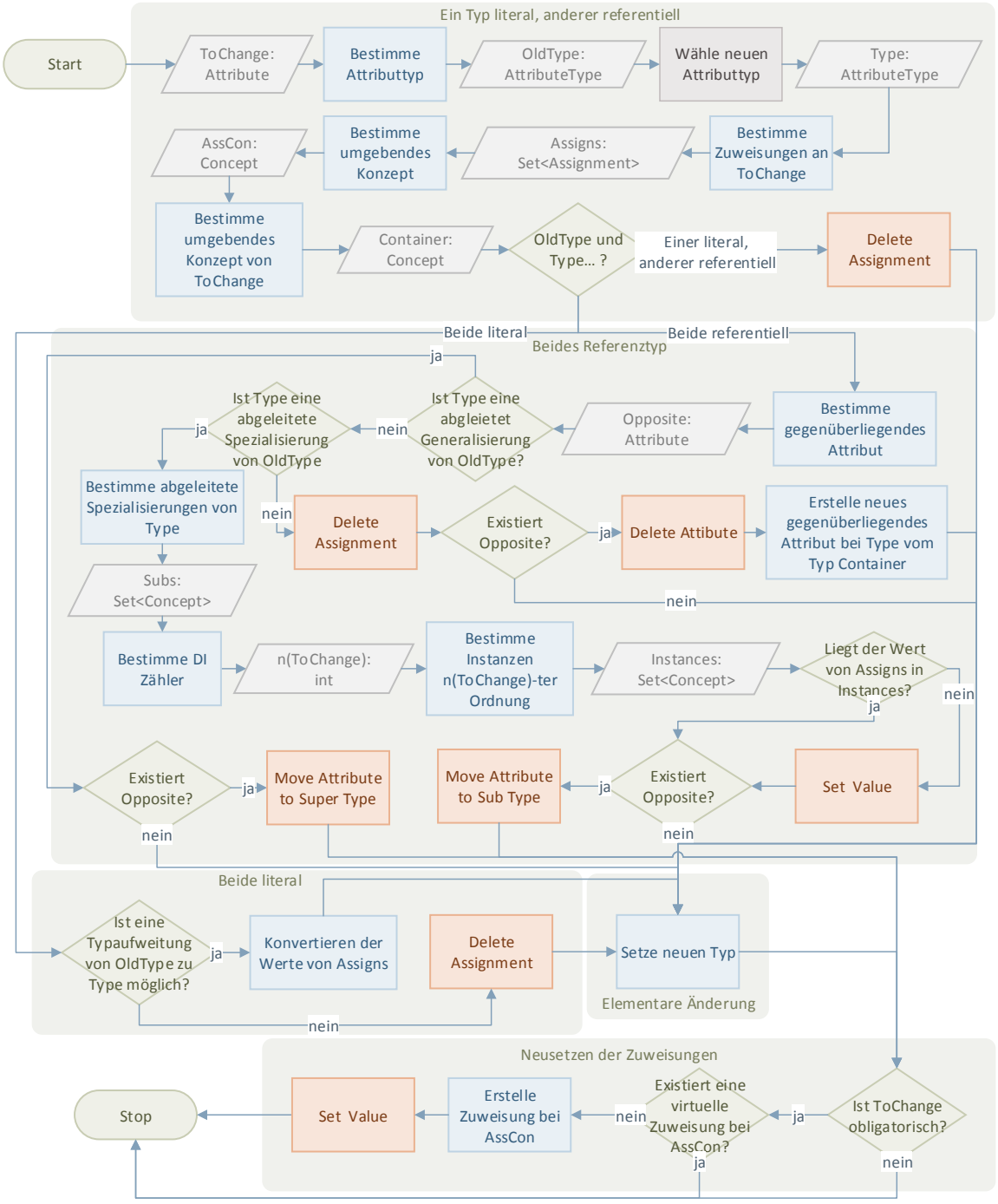

Abbildung 5-41 Ablauf des Set Attribute Type Operators

Für den Fall, dass **Type** keine Spezialisierung von **OldType** ist, sind alle Zuweisungen ungültig und müssen mit Hilfe des *Delete Assignment* Operators entfernt werden. Wenn **Opposite** existiert, wird

dieses durch den *Delete Attribute* Operator entfernt, um auch die entsprechenden Zuweisungen zu entfernen. Anschließend erhält `Type` ein neues Attribut, das `ToChange` gegenüberliegt und ansonsten die gleichen Eigenschaften wie `Opposite` aufweist. Danach kann auch in diesem Fall die elementare Änderung ausgeführt werden.

Beide literal

Wenn `Type` und `OldType` literal sind, wird als erstes untersucht, ob eine Aufweitung des Typs vorliegt. Gegebenenfalls, werden alle Zuweisungen analog wie in bekannten Programmiersprachen (z.B. Java) konvertiert. Einige Beispiele zu Konvertierung der verschiedenen literalen Typen finden sich in Tabelle 5-1. Diese zeigt ebenfalls, wann eine Aufweitung vorliegt (Zelle besitzt einen Beispielwert) und wann eine Konvertierung nicht möglich ist (Zeichen „-"). Einen Spezialfall stellt der Typ `pointer`, der nur sinnvoll in den Typ `string` konvertiert werden kann. Wenn keine Konvertierung möglich ist, muss die Zuweisung unter Verwendung des *Delete Assignment* Operators entfernt werden.

Tabelle 5-1 Beispiele der Konvertierung von literalen Datentypen

von ↓ \ nach→	boolean	integer	real	pointer	string
boolean	true/false	0/1	0.0/1.0	-	'true'/'false'
integer	-	123/5	123.0/5.0	-	'123'/'5'
real	-	-	3.1415	-	'3.1415'
pointer	-	-	-	a.b.c	'a.b.c'
string	-	-			'Hallo Welt!'

Elementare Änderung

Als nächstes wird der neue Attributtyp für `ToChange` gesetzt.

Neusetzen der Zuweisung

Für den Fall, dass `ToChange` ein obligatorisches Attribut ist, muss bei allen Konzepten, die vorher eine Zuweisung besaßen (`AssCon`),[41] ein neuer Wert für die Zuweisung gesetzt werden, falls keine virtuelle Zuweisung vorhanden ist. Deshalb wird zunächst eine neue Zuweisung erstellt und dann durch den *Set Value* Operator ein neuer Wert gesetzt. Danach ist die Ausführung des Operators beendet.

Beispiel

Abbildung 5-42 zeigt auf der linken Seite ein einfaches Modell, in dem die Ebene `M1` drei Konzepte definiert: `Permanent` als Spezialisierung von `Employee` und `Department`, das eine Beziehung zu `Employee` über das Attribut `employees` festlegt. Damit wird beschrieben, dass Mitarbeiter (`Employee`) zum einen festangestellt (`Permanent`) sein können und zum anderen, dass Abteilungen (`Department`) Mitarbeiter besitzen (Attribut `employees`). In der Ebene `M0` wurden ebenfalls drei Konzepte erstellt. Dabei ist `Bob` eine Instanz von `Employee`, während `Alice` eine Instanz von `Permanent` ist. Beide Mitarbeiter sind der Büroabteilung (Konzept `Office` als Instanz von `Department`, inklusive der Zuweisung an `employees` mit den Werten `Bob` und `Alice`) tätig.

[41] Korrekterweise müssten alle Instanzen der Ordnung `n(ToChange)` eine virtuelle Zuweisung besitzen, dies wurde aber vorher ebenfalls verlangt. Da die Multiplizität des Attributes nicht verändert wurde, genügt es an dieser Stelle Zuweisungen bei alle Konzepten zu definieren, die vorher ebenfalls eine besaßen.

Im Nachfolgenden soll nun mit Hilfe des Operators der Attributtyp von `employees` zu `Permanent` geändert werden. Dazu bestimmt der Operator zunächst `Employee` als alten Attributtyp. Danach wird `Permanent` als neuer Typ gewählt und alle Zuweisungen an `employees` ermittelt, was in unserem Fall lediglich die Zuweisung von `Office` liefert. Als nächstes wird `Office` als umgebendes Konzept dieser Zuweisung und `Department` als umgebendes Konzept von `employees` ermittelt. Da beide Attributtypen referentiell sind und kein gegenüberliegendes Attribut existiert, wird festgestellt, dass `Permanent` eine abgeleitete Spezialisierung von `Employee` ist. Deshalb werden alle abgeleiteten Spezialisierungen von `Permanent` ermittelt (= {`Permanent`}) und anschließend alle Instanzen 1-ter Ordnung (der Deep Instantiation Zähler von `employees` ist `1`) davon berechnet. Das Ergebnis dieses Schrittes liefert `Alice` als einzige Instanz von `Permanent`. Nun wird die Zuweisung an `employees`, die durch `Office` erstellt wurde, untersucht, ob alle Werte in der Menge {`Alice`} liegen. Da dies nicht der Fall ist, weil auch `Bob` zugewiesen wurde, wird die Zuweisung angepasst und `Bob` als Wert entfernt. Anschließend wird der neue Typ `Permanent` für `employees` gesetzt und das in Abbildung 5-42 auf der rechten Seite gezeigte Modell entsteht.

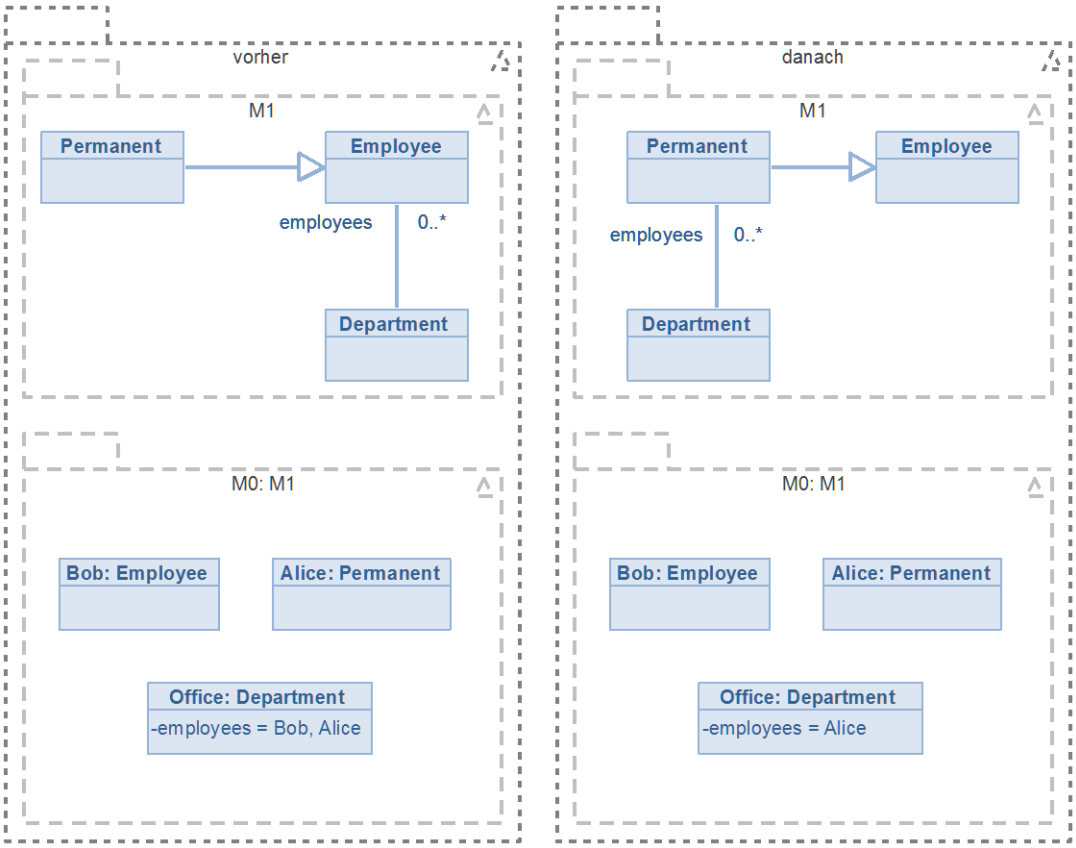

Abbildung 5-42 Beispiel vor (links) und nach (rechts) der Ausführung des Set Attribute Type Operators

5.5.6 Set Opposite

Auswirkung

Der Operator setzt bzw. ändert das gegenüberliegende Attribut einer bidirektionalen Beziehung. Dabei werden die Deep Instantiation Zähler beider Attribute angepasst und Zuweisungen neu gesetzt.

Ablauf

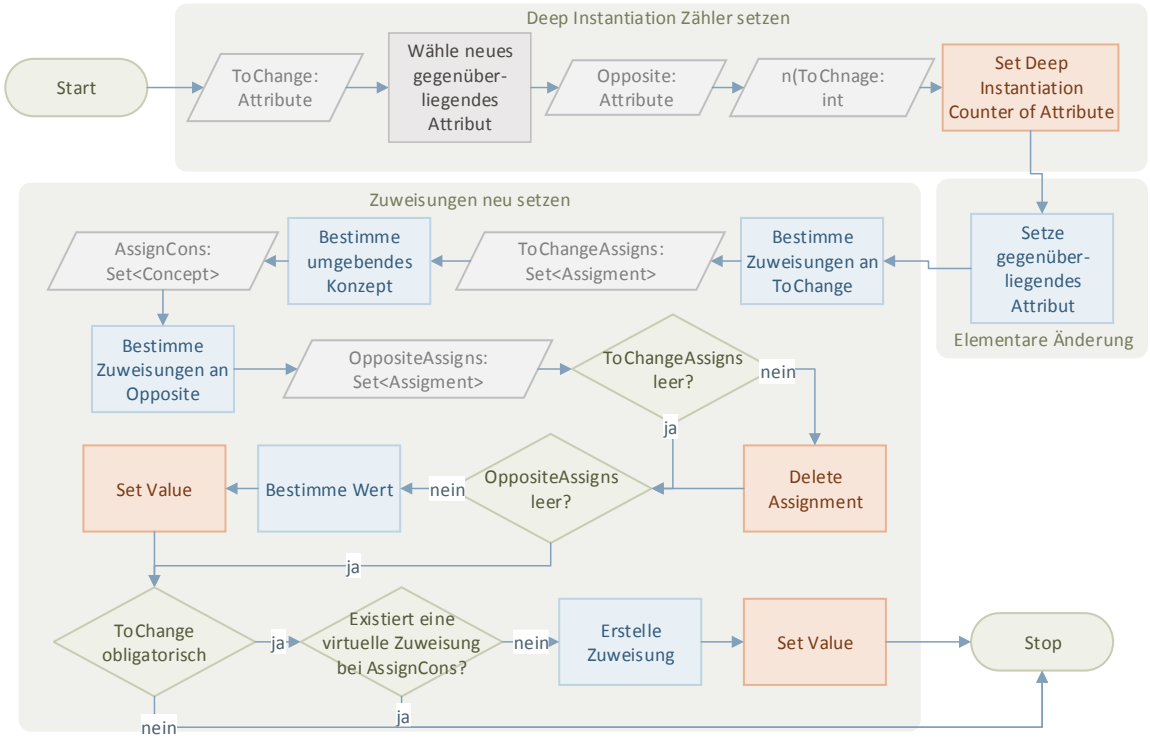

Abbildung 5-43 Ablauf des Set Opposite Operators

Deep Instantiation Zähler setzen

Zu Beginn wird der Operator (Abbildung 5-43) an einem Attribut **`ToChange`** mit einem Konzept als Typ aufgerufen. Im ersten Schritt wird das neue gegenüberliegende Attribut **`Opposite`** aus der Menge der Attribute, die der Attributtyps von **`ToChange`** definiert und die wiederum als Typ das umgebende Konzept von **`ToChange`** haben, gewählt.[42] Da die Attribute einer bidirektionalen Beziehung den gleichen Deep Instantiation Zähler besitzen müssen (Regel A.8), wird als nächstes für **`Opposite`** dieser durch den *Set Deep Instantiation Counter of Attribute* Operator auf den Wert von **`ToChange`** gesetzt.

Elementare Änderung

Danach wird **`Opposite`** als das neue gegenüberliegende Attribut von **`ToChange`** festgelegt.

Zuweisungen neu setzen

In den nachfolgenden Schritten werden alle Zuweisungen von **`ToChange`** (**`ToChangeAssigns`**), deren umgebende Konzepte (**`AssignCons`**) und alle Zuweisungen an **`Opposite`** (**`OppositeAssigns`**) ermittelt. Wenn **`ToChangeAssigns`** nicht leer ist, werden alle Zuweisungen dieser Menge durch den *Delete Assignment* Operator entfernt. Durch das spätere Setzen der Zuweisungen bei **`Opposite`** werden die neuen Werte für **`ToChange`** dann entsprechend festgelegt. Als nächstes wird überprüft, ob **`OppositeAssigns`** leer ist. Wenn dies nicht der Fall ist, werden die Werte der Zuweisungen bestimmt und durch den *Set Value* Operator neu gesetzt, wodurch die Werte bei gegenüberliegenden Konzepten ebenfalls festgelegt werden. Wenn **`ToChange`** obligatorisch ist, kann es durch das obige Entfernen von Zuweisungen (**`ToChangeAssigns`**) dazu kommen, das nicht jedes Element aus **`AssignCons`** eine virtuelle Zuweisung besitzt. Falls dies zutrifft, wird eine Zuweisung beim

[42] Wenn bereits vorher ein anderes gegenüberliegendes Attribut existierte, dann bleiben die Zuweisungen an es vom Operator unverändert. Deshalb wird dieses auch vom Operator nicht bestimmt.

entsprechenden Element von `AssignCons` erstellt und anschließend der *Set Value* Operator aufgerufen, der die Auswahl eines neuen Wertes ermöglicht und ihn anschließend setzt. Danach ist die Ausführung des Operators beendet.

Beispiel

Da das Ändern des gegenüberliegenden Attributes relativ selten vorkommt und meist durch einen vorherigen Fehler in der Modellierung entsteht, soll ein abstraktes Beispiel dazu dienen, die Wirkung des Operators zu verdeutlichen. Abbildung 5-44 zeigt auf der linken Seite eine Meta-Hierarchie mit zwei Ebenen. `M1` definiert dabei zwei Konzepte `A` und `B`, die durch zwei Beziehungen (eine bidirektional, die andere nicht) verbunden sind. Während das Attribut `a1s` von `B` ein gegenüberliegendes Attribut `bs` von `A` besitzt, hat das zweite Attribut von `B` `a2s` kein solches Attribut. Auf der Ebene `M0`, sind je zwei Instanzen von `A` (`Inst1A`, `Inst2A`) und `B` (`Inst1B`, `Inst2B`) definiert worden. `Inst1A` setzt das Attribut `bs` mit dem Wert {`Inst1B`}, `Inst1B` das Attribut `a1s` und das Attribut `a2s` jeweils mit dem Wert {`Inst1A`} und `Inst2B` das Attribut `a2s` mit {`Inst1A, Inst2A`}.

Nun soll das gegenüberliegende Attribut von `bs` von `a1s` auf `a2s` geändert werden. Dazu wird im ersten Schritt `a2s` als neues gegenüberliegendes Attribut gewählt. Da der Deep Instantiation Zähler von `a2s` bereits `1` ist, wird als nächstes die `oppositeOf` Referenz zwischen `bs` und `a2s` gesetzt. Anschließend werden die Zuweisung von `Inst1A` an `bs` sowie die Zuweisungen von `Inst1B` und `Inst2B` an `a2s` ermittelt. Da die Zuweisungen an `bs` und `a2s` existieren, wird zunächst die Zuweisung von `Ins1tA` an `bs` durch den *Delete Assignment* Operator entfernt und der Wert für das Attribut `a2s` von `Inst1B` bzw. `Inst2B` bestimmt. Diese werden anschließend durch den *Set Value* Operator neu gesetzt, wodurch die Zuweisungen von `Inst1A` (Wert `Inst1B`, `Inst2B`) und `Inst2A` (Wert `Inst2B`) von `bs` entsteht. Da `bs` ein optionales Attribut ist, endet der Operator. Das resultierende Modell ist in Abbildung 5-44 auf der rechten Seite dargestellt.

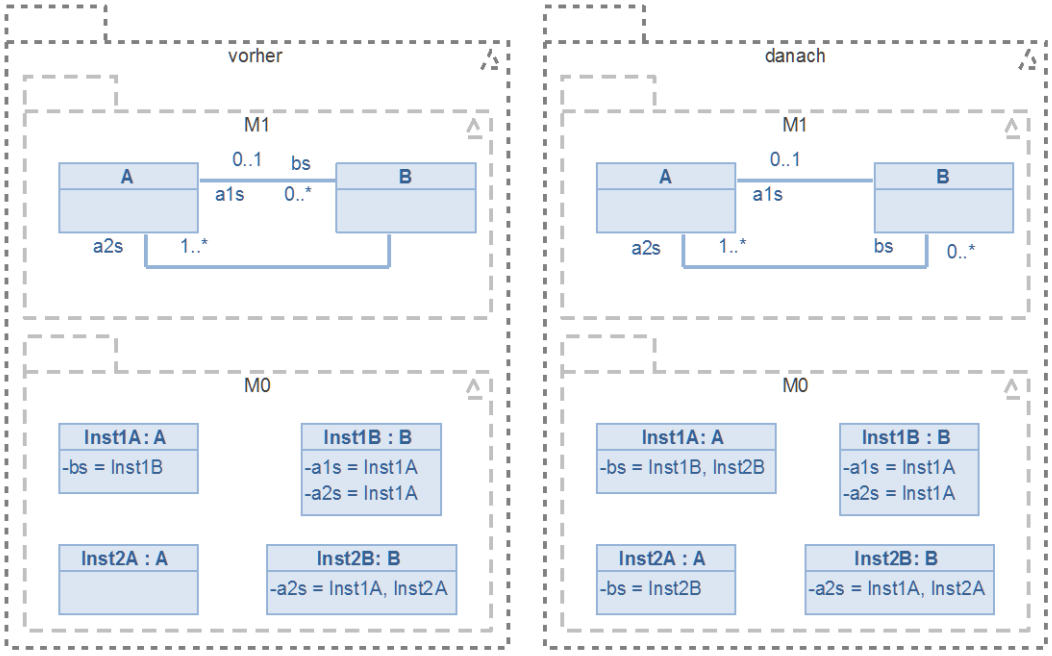

Abbildung 5-44 Beispiel vor (links) und nach (rechts) der Ausführung des Set Opposite Operators

5.5.7 Set Enables

Auswirkung

Der Operator setzt die für ein Diskriminatorattribut notwendige Referenz zu einem Attribut des partitionierten Typs und erzeugt für alle anderen Attribute des partitionierten Typs ebenfalls Diskriminatorattribute.

Ablauf

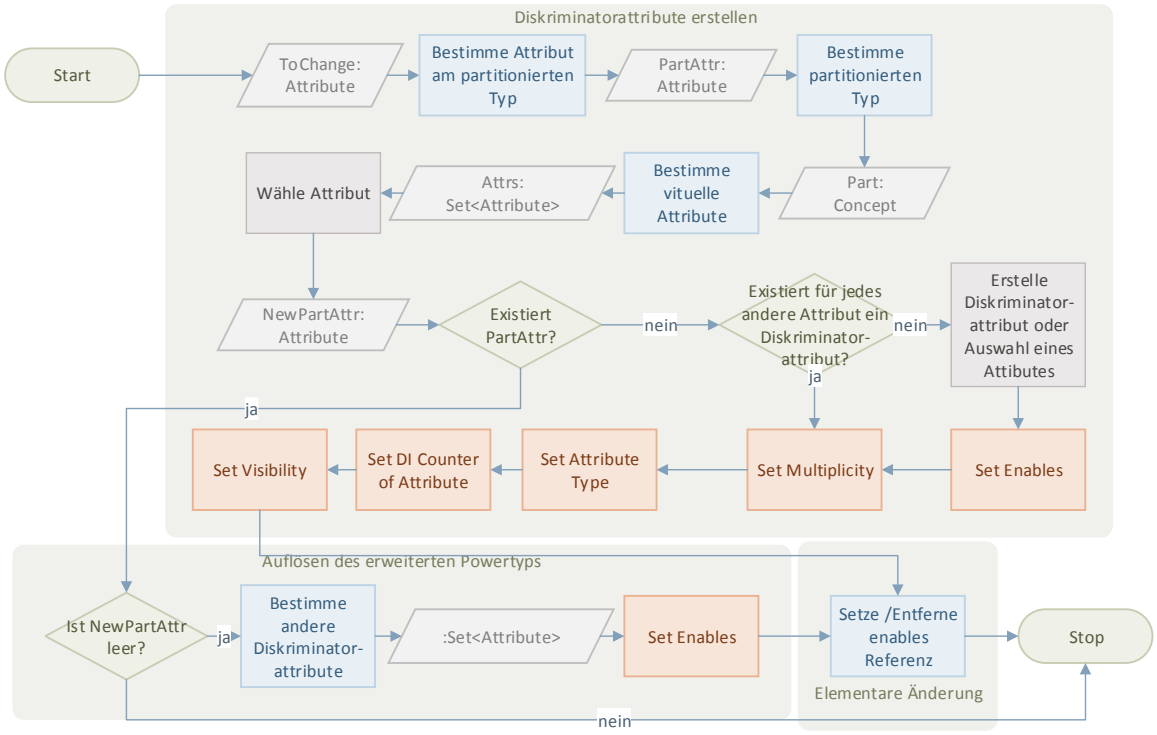

Abbildung 5-45 Ablauf des Set Enables Operators

Diskriminatorattribute erstellen

Initial wird der Operator an einem Attribut **`ToChange`** aufgerufen, von dem im ersten Schritt ein eventuell bereits gesetztes Attribut vom partitionierten Typ (**`PartAttr`**) bestimmt wird. Anschließend wird zunächst der partitionierte Typ **`Part`** und anschließend dessen virtuelle Attribute (**`Attrs`**) bestimmt (Regel A.4). Aus dieser Menge wird dann ein Attribut (**`NewPartAttr`**), das das Ziel der **`enables`** Referenz darstellen soll, gewählt. Wenn **`PartAttr`** nicht existiert, wird für alle anderen virtuellen Attribute, falls noch nicht existent, ebenfalls ein Diskriminatorattribut erzeugt oder ein entsprechendes Attribut vom umgebenden Konzept von **`ToChange`** gewählt. Für alle diese Attribute, wird ebenfalls der *Set Enables* Operator aufgerufen.

Bei Erstellen eines Diskriminatorattributes muss vor allem der Name des Attributes festgelegt werden, da alle anderen Eigenschaften des Attributes nicht frei wählbar sind. Aufgrund der Tatsache, dass Diskriminatorattribute immer den Attributtyp **`boolean`** besitzen, die Sichtbarkeit **`public`**, einen Deep Instantiation Zähler von **`1`** und die Multiplizität **`1`** haben, werden diese Eigenschaften, falls nicht bereits vorhanden, durch die Operatoren *Set Attribute Type*, *Set Deep Instantiation Counter of Attribute*, *Set Multiplicity* und *Set Visibility* gesetzt (Regel A.5).

Auflösen des erweiterten Powertyps

Für den Fall, dass `PartAttr` existiert und der neue Wert `NewPartAttr` nicht leer ist, wird die Ausführung des Operators beendet. Dies ist insofern sinnvoll, da ein erweiterter Powertyp für jedes virtuelle Attribut am partitionierten Typ ein Diskriminatorattribut definieren muss (Regel A.6). Deshalb macht die Änderung der `enables` Referenz in diesem Fall keinen Sinn. Wenn jedoch die Referenz durch den Operator entfernt (`NewPartAttr` auf leer gesetzt) wird, werden alle Diskriminatorattribute des umgebenden Konzepts von `ToChange` bestimmt und für sie ebenfalls der *Set Enables* Operator mit einem leeren Wert aufgerufen, damit die `enables` Referenz entfernt wird. Damit wird aus dem erweiterten Powertyp ein Powertyp.

Elementare Änderung

Anschließend wird die `enables` Referenz von `ToChange` zu `NewPartAttr` erstellt oder entsprechend die alte Referenz zu `PartAttr` entfernt, falls `NewPartAttr` leer gewählt wurde. Danach endet der Operator.

5.5.8 Delete Materialization Extend

Auswirkung

Die Anwendung des Operators löscht die Materialisierungserweiterung eines Attributs und damit das Materialisierungsmuster. Dabei werden materialisierte Attribute durch direkt deklarierte ersetzt.

Ablauf

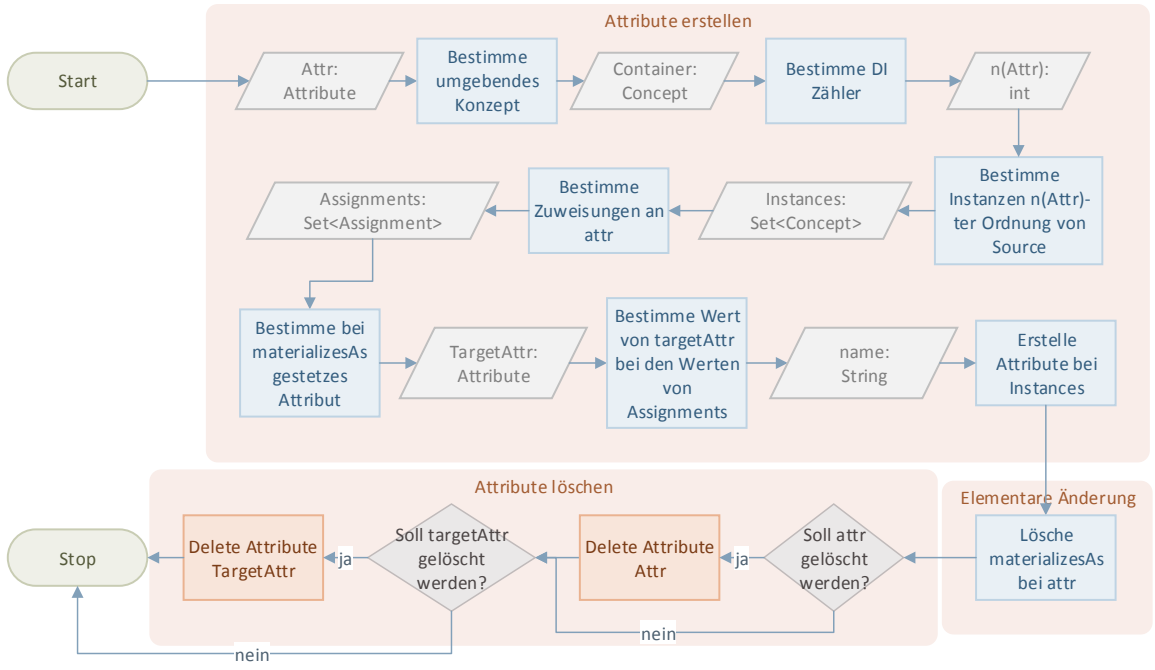

Abbildung 5-46 Ablauf des Delete Materialization Operators

Attribute erstellen

Zu Beginn wird der Operator an einem Attribut `Attr` aufgerufen, welches die Materialisierung kennzeichnet, d.h. die `materializesAs` Eigenschaft gesetzt hat. Danach wird das umgebende Konzept `Container` bestimmt, an dem `Attr` definiert ist. Im nächsten Schritt wird der Deep Instantiation Zähler `n(Attr)` von `Attr` ermittelt, damit anschließend alle Instanzen `n(Attr)`-ter Ordnung (`Instances`) von `Container` berechnet werden können, da sie Zuweisungen an `Attr` besitzen (können). Nachdem diese Zuweisungen im darauffolgenden Schritt bestimmt wurden, kann

das entsprechende materialisierte Attribut durch ein direkt definiertes ersetzt werden. Dazu wird ein neues Attribut bei der Instanz von **Container** erstellt. Um den Namen des Attributes zu erhalten, wird die Zuweisung bei der Instanz verwendet und für jedes zugewiesene Konzept (**TargetInst**) der Wert der Attribut-Definition (**TargetAttr**) ausgelesen. Pro zugewiesenen Konzept entsteht so ein Attribut beim entsprechenden Element aus **Instances**. Die Multiplizität und die Sichtbarkeit aller erzeugten Attribute werden dabei durch die entsprechend gesetzten Werte bei der Materialisierungserweiterung festgelegt.

Elementare Änderung

Im nächsten Schritt wird die **materializesAs** Eigenschaft bei **Attr** gelöscht.

Attribute löschen

Nach der elementaren Änderung kann noch gewählt werden, ob die Attribut-Definition **TargetAttr** und/oder das Attribut **Attr** mit Hilfe des *Delete Attribute* Operators gelöscht werden sollen. Durch den Aufruf des Operators werden auch entsprechende Zuweisungen an **Attr** bzw. an **TargetAttr** gelöscht. Danach endet der Operator.

Beispiel

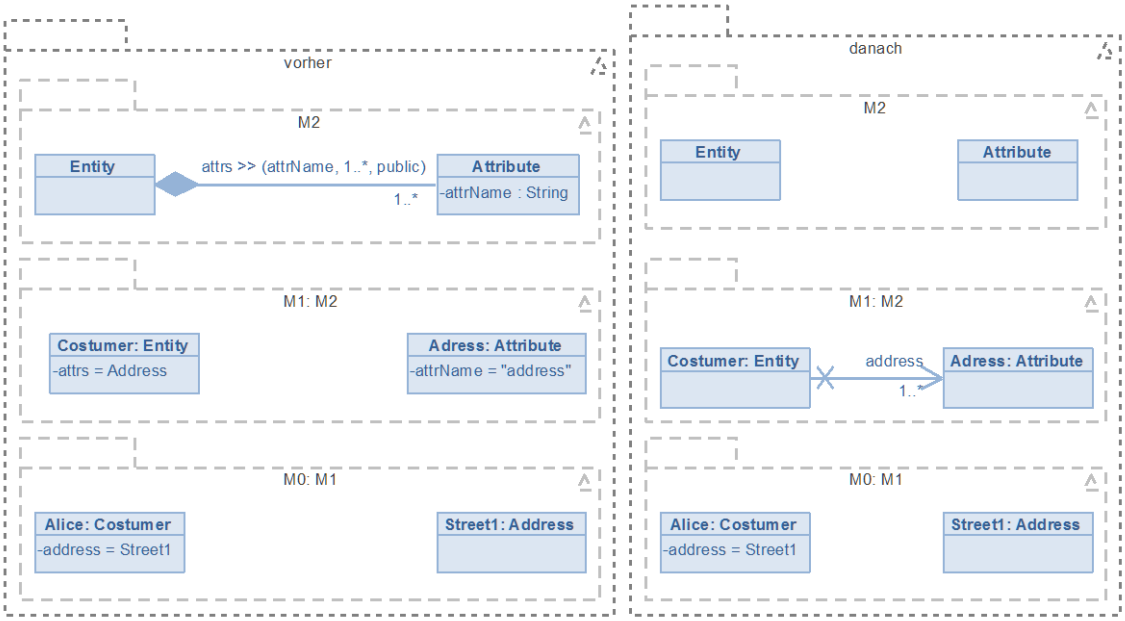

Abbildung 5-47 Beispiel vor (links) und nach (rechts) der Ausführung des Delete Materilaization Operators

Als Beispiel soll die Umkehrung des in Kapitel 6.5.1 gezeigten Beispiels des *Introduce Materialization* Operators dienen. Die Beschreibung der Domäne des Modells findet sich folglich dort. Das Ausgangsmodell ist in Abbildung 5-47 auf der linken Seite visualisiert. Der *Delete Materialization Extend* Operator wird nun am Attribut **attrs** aufgerufen. Danach wird **Entity** als das Konzept, an dem **attrs** definiert ist, bestimmt. Im nächsten Schritt wird der Deep Instantiation Zähler von **attrs** ermittelt (=**1**) und danach alle Instanzen 1-ter Ordnung von **Entity** berechnet, was in diesem Beispiel als Ergebnis nur **Costumer** liefert. Als nächstes wird für **Costumer** die Zuweisung an **attrs** bestimmt, welche den Wert **Address** hat. Da **attrName** die Attribut-Definition vom Typ **string** ist, wird dessen Wert **„address“** bei **Address** ausgelesen. Danach wird ein neues Attribut bei **Costumer** mit Namen **address** erstellt, welches als Multiplizität **1..*** und als Sichtbarkeit **public** besitzt. Anschließend wird die **materializesAs** Eigenschaft bei **attrs** gelöscht und wir entscheiden uns dafür, sowohl **attrs** als auch **attrName** zu löschen. Deshalb werden die Zuweisungen bei **Costumer** an **attrs** und die

Zuweisung bei **Address** an **attrName** ebenfalls entfernt und man erhält das in Abbildung 5-47 auf der rechten Seite gezeigte Ergebnis.

5.5.9 Set Member Definition

Auswirkung

Durch die Ausführung des *Set Member Definition* Operators wird die Attribut-Definition, die den Namen des materialisierten Attributes festgelegt, auf ein anderes Attribut vom Typ **string** gesetzt. Dabei werden eventuell fehlende Zuweisungen gesetzt.

Ablauf

Der Aufruf des Operators erfolgt an einem Attribut **Attr** vom Typ Konzept. Zunächst wird der Typ **Target** von **Attr** bestimmt, welcher ein Konzept sein muss (Regel A.2). Im nächsten Schritt werden alle Attribute vom Typ **string**, die an **Target** definiert wurden, gesucht. Aus der Menge dieser Attribute wird dann eine neue Attribut-Definition (**NewAttr**) gewählt. Falls die Multiplizität ungleich **1** ist, muss sie durch den Operator *Set Multiplicity* auf diesen Wert gesetzt werden, da jedes materialisierte Attribut genau einen Namen haben muss (Regel A.3). Dies hat zur Folge, dass eventuell mehrwertige Zuweisungen auf einen Wert reduziert werden. Zusätzlich erhalten dadurch Konzepte, die Instanzen von **Target** sind und bei denen das Attribut noch nicht virtuell zugewiesen wurde, Zuweisungen. Ähnlich verhält es sich mit dem Deep Instantiation Zähler (Regel A.2). Dieser muss den Wert des Deep Instantiation Zählers von **Attr** haben, da sonst die Zuweisung für den Attributnamen nicht bei entsprechenden Instanzen **n(Attr)**-ter Ordnung gesetzt ist. Folglich wird, falls **NewTarget** gegen diese Bedingung verstößt, der *Set Deep Instantiation Counter of Attribute* Operator aufgerufen und der Zähler damit auf den entsprechenden Wert festgelegt.

Elementare Änderung

Zum Abschluss wird die neue Attributdefinition an der Materialisierungserweiterung von **Attr** gesetzt. Dadurch werden die materialisierten Attribute mit einem neuen Namen versehen, der gleich dem Wert von **NewAttr** bei der entsprechenden Instanz von **Target** ist. Dies bedeutet zwar eine Umbenennung des materialisierten Attributs, die Zuweisungen an die materialisierten Attribute bleiben dadurch aber unberührt. Nachdem die elementare Änderung durchgeführt wurde, endet der Operator.

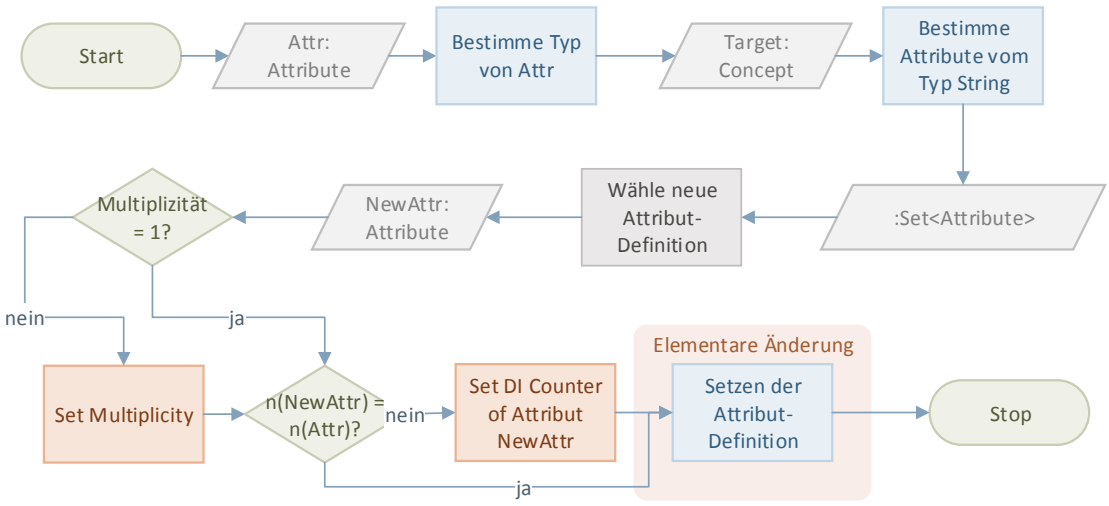

Abbildung 5-48 Ablauf des Set Member Definition Operators

5.5.10 Set Visibility of Materialization Extend

Auswirkung

Durch den Operator wird die Sichtbarkeit der Materialisierungserweiterung eines Attributs geändert. Falls diese verengt wird, werden alle ungültigen Zuweisungen entfernt.

Ablauf

Zuweisungen entfernen

Der Operator (Abbildung 5-49) wird an einem Attribut `ToChange` aufgerufen, das eine Materialisierungserweiterung definiert. Im ersten Schritt wird die bisherige Sichtbarkeit (`oldVis`), die diese Erweiterung besitzt, ermittelt und danach die neue Sichtbarkeit (`vis`) gewählt. In den nächsten Schritten werden dann der Deep Instantiation Zähler `n(ToChange)`, alle abgeleiteten Spezialisierungen des umgebenden Konzepts (`Containers`) und alle Instanzen `n(ToChange)`-ter Ordnung von `Containers` (`InstContainers`) ermittelt. Alle Elemente aus `InstContainers` können

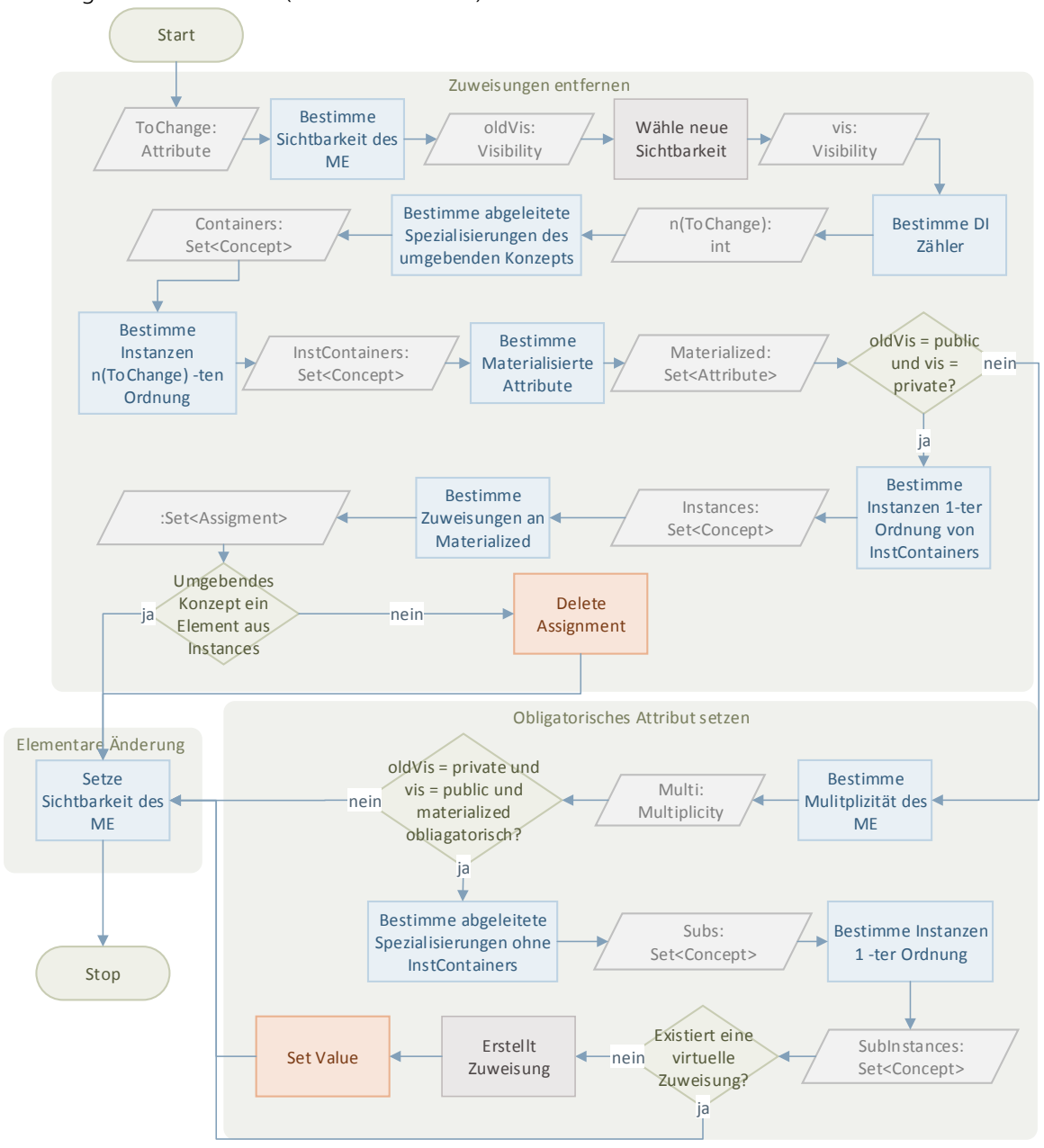

Abbildung 5-49 Ablauf des Set Visibility of Materialization Extend Operators

das Attribut `ToChange` zuweisen und damit materialisierte Attribute (`Materialized`) definieren, die als nächstes bestimmt werden. Falls sich die Sichtbarkeit aller materialisierten Attribute von `public` auf `private` geändert hat, müssen alle Zuweisungen, die von Instanzen von abgeleiteten Spezialisierungen eines Konzeptes aus `InstContainers` gemacht wurden, entfernt werden. Deshalb werden zunächst alle Instanzen 1-ter Ordnung[43] der Elemente aus `InstContainers` und anschließend alle Zuweisungen an ein Attribut aus `Materialized` ermittelt. Wenn eine Zuweisung nicht an einem Konzept aus `Instances` definiert wurde, wird sie durch den *Delete Assignment* Operator entfernt. Danach wird die elementare Änderung ausgeführt.

Obligatorisches Attribut setzen

Für den Fall, dass die Sichtbarkeit nicht eingeschränkt wurde, wird zunächst die Multiplizität der Materialisierungserweiterung bestimmt. Wenn diese `1` oder `1..*` beträgt, sind alle Elemente aus `Materialized` obligatorisch. Falls zusätzlich noch die Sichtbarkeit erweitert wurde (von `private` auf `public`), dann müssen bei allen Instanzen der abgeleiteten Spezialisierungen von `InstContainers` Zuweisungen erstellt werden. Folglich werden in diesem Fall zuerst alle abgeleiteten Spezialisierungen (`Subs`) der Elemente aus `InstContainers` (ohne die Elemente selbst) bestimmt. Anschließend werden von `Subs` alle Instanzen 1-ter Ordnung (`SubInstances`) berechnet und überprüft, ob eine virtuelle Zuweisung (in diesem Fall ein Standardwert) existiert. Wenn dies nicht der Fall ist, wird eine Zuweisung beim jeweiligen Element aus `SubInstances` oder `InstContainers` (Standardwert) erstellt und mit Hilfe des *Set Value* Operators ein Wert festgelegt.

Elementare Änderung

Danach wird die neue Sichtbarkeit bei der Materialisierungserweiterung von `ToChange` gesetzt und der Operator endet.

5.5.11 Set Multiplicity of Materialization Extend

Auswirkung

Der Operator ändert die Multiplizität der Materialisierungserweiterung eines Konzeptes, erstellt neue Zuweisungen, falls nötig, und bereinigt ungültige Werte.

Ablauf

Fehlende Zuweisungen erstellen

Abbildung 5-50 zeigt den Ablauf des *Set Multiplicity of Materialization Extend* Operators, der zu Beginn an einem Attribut `ToChange` mit einer Materialisierungserweiterung aufgerufen wird. Zunächst wird die Multiplizität dieser Erweiterung (`oldMulti`) ermittelt und danach die neue Multiplizität `multi` gewählt. In den folgenden Schritten werden der Deep Instantiation Zähler `n(ToChange)`, alle abgeleiteten Spezialisierungen (`Containers`) des umgebenden Konzepts und deren Instanzen `n(ToChange)`-ter Ordnung (`InstContainers`) bestimmt. Die Elemente aus `InstContainers` können `ToChange` zuweisen und daher materialisierte Attribute (`Materialized`) definieren, die als nächstes ermittelt werden. Wenn durch die Änderung der Multiplizität die optionalen Attribute aus `Materialized` obligatorisch werden, müssen alle Instanzen 1-ter Ordnung von `InstContainers` untersucht werden, ob sie das jeweilige materialisierte Attribut virtuell zugewiesen haben. Wenn dies nicht der Fall ist, wird eine neue Zuweisung bei dem entsprechenden Element aus `Instances` oder `InstContainers` (Standardwert, externe Auswahl) erstellt und mit Hilfe des *Set Value* Operators der Wert für das materialisierte Attribut festgelegt.

[43] Der Deep Instantiation Zähler von materialisierten Attributen ist immer eins.

Werte bereinigen

Wenn das Maximum der Multiplizität von * auf **1** verringert wurde, müssen zunächst alle Zuweisungen an die Attribute **`Materialized`** ermittelt werden, da diese nun nur noch einen Wert besitzen dürfen. Wenn eine Zuweisung mehr als einen Wert enthält, wird zunächst ein verbleibender Wert gewählt und danach die Zuweisung mittels des *Set Value* Operators bereinigt.

Elementare Änderung

Anschließend kann die neue Multiplizität der Materialisierungserweiterung von **`ToChange`** gesetzt werden, womit die Ausführung des Operators endet.

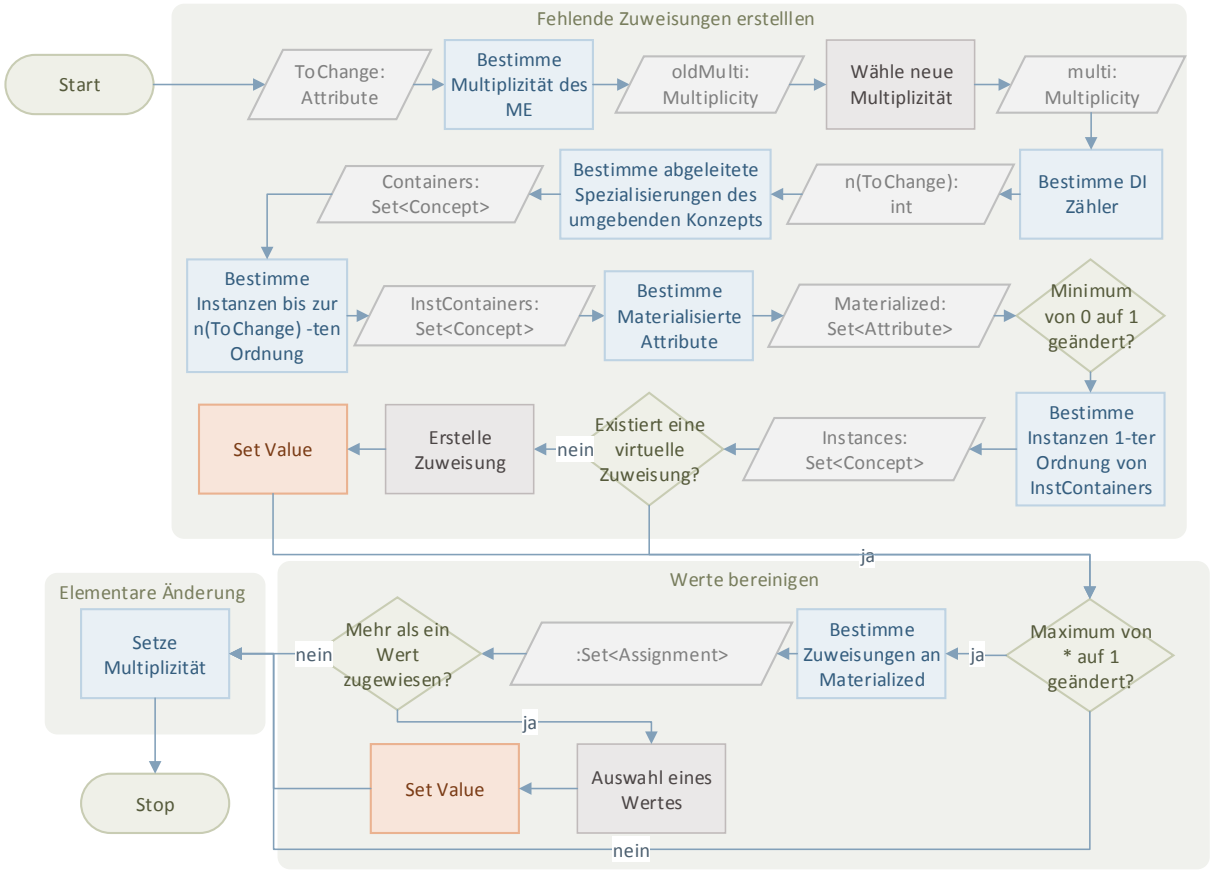

Abbildung 5-50 Ablauf des Set Multiplicity of Materialization Extend Operators

5.5.12 Delete Attribute

Auswirkung

Der Operator löscht ein Attribut und entfernt alle eingehenden und ausgehenden Referenzen des Attributs sowie alle Zuweisungen.

Ablauf

Abhängige Attribute und Zuweisungen migrieren bzw. löschen

Abbildung 5-51 zeigt den Ablauf des Operators, der initial an einem Attribut **`ToDelete`** aufgerufen wird. Zunächst wird von **`ToDelete`** festgestellt, ob es komposit ist. Wenn dies der Fall ist, dann wird diese Eigenschaft entfernt, da sonst durch das spätere Entfernen der Zuweisungen auch die umgebenden Konzepte gelöscht würden. Danach wird ein eventuell vorhandenes gegenüberliegendes Attribut bestimmt und anschließend, falls dieses existiert, die entsprechende Referenz entfernt. Als

nächstes wird ein möglicherweise vorhandenes Diskriminatorattribut ermittelt und durch den Aufruf des *Delete Attribute* Operators ebenfalls gelöscht. Danach wird die Materialisierungserweiterung vom *Delete Materialization Extend* Operator entfernt, wobei die Auswahl im letzten Teilschritt des Operators so gewählt wird, dass keines der beiden Attribute entfernt wird. Falls **`ToDelete`** ein Diskriminatorattribut ist, wird als nächstes mit Hilfe des *Set Enables* Operator die **`enables`** Referenz (setzen eines leeren Wertes) entfernt. Wenn **`ToDelete`** ein obligatorisches Attribut ist, wird das Minimum der Multiplizität durch den *Set Multiplicity* Operator auf 0 gesetzt, damit auch anschließend die Zuweisungen entfernt werden können. Diese werden im nächsten Schritt ermittelt und durch den *Delete Assignment* Operator entfernt.

Elementare Änderung

Im letzten Schritt wird **`ToDelete`** gelöscht und die Ausführung des Operators ist beendet.

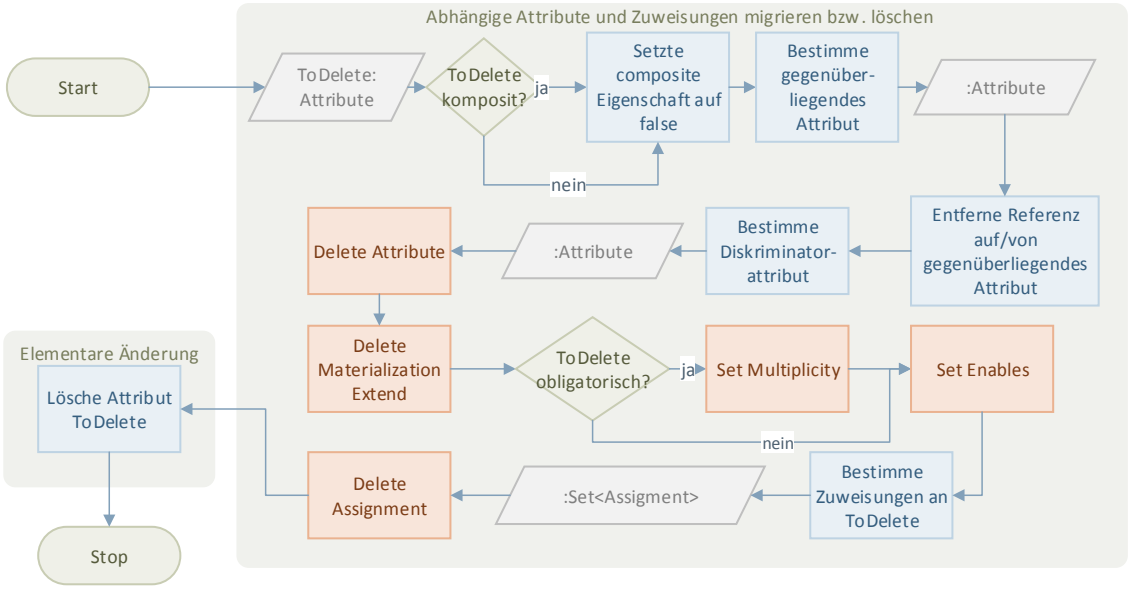

Abbildung 5-51 Ablauf des Delete Attribute Operators

5.6 Zuweisung (Assignment)

Zuweisungen legen für ein bestimmtes Attribut einen Wert bzw. eine Menge von Werten in einem Konzept fest. Weiterhin definieren sie im Kontext der Instanz-Spezialisierung das Überschreibungsverhalten am Prototyp. Die jeweiligen Operatoren für die Änderung dieser Eigenschaften einer Zuweisung werden in diesem Abschnitt vorgestellt.

5.6.1 Set Attribute Of

Auswirkung

Durch den Operator wird ein Attribut zu einer Zuweisung (neu-)gesetzt. Bereits vorhandene Werte werden angepasst, falls dies möglich ist.

Ablauf

Werte der Zuweisung migrieren

Der Operator, dessen Ablauf in Abbildung 5-52 dargestellt ist, wird an einer Zuweisung **`ToChange`** aufgerufen, von der zunächst das neue Attribut **`Attr`** aus der Menge aller zuweisbaren Attribute (Regel Z.4, Definition 4.22) gewählt wird. Danach werden die Werte (**`Values`**) der Zuweisung bestimmt und

überprüft, ob diese im Wertebereich (Instanz des Attributtyps, Literal der Enumeration oder Wertebereich des literalen Typs) von **`Attr`** liegen. Wenn dies nicht der Fall ist, werden die Werte aus den Zuweisungen entfernt oder falls möglich konvertiert. Wie die Konvertierung exemplarisch abläuft ist in Tabelle 5-1 im Abschnitt 5.5.5 dargestellt. Nachdem die Werte der Zuweisung angepasst wurden, wird das umgebende Konzept **`Container`** von **`ToChange`** ermittelt, um danach zu überprüfen, ob es bereits eine Zuweisung (**`Other`**) von **`Container`** für **`Attr`** gibt. Wenn dies der Fall ist, werden die Werte von **`Other`** zu **`Values`** hinzugefügt und die beiden Zuweisungen **`Other`** und **`ToChange`** vereinigt. Danach wird überprüft, ob **`Attr`** als Maximum der Multiplizität **`1`** hat und **`Values`** aus mehreren Werten besteht, ist dies der Fall, muss ein einzelner Wert aus **`Values`** gewählt werden, bevor der *Set Value* Operator **`Values`** als neuen Wert setzen kann.

Elementare Änderung

Als nächstes wird das Attribut bei der Zuweisung gesetzt, wodurch die Ausführung des Operators endet.

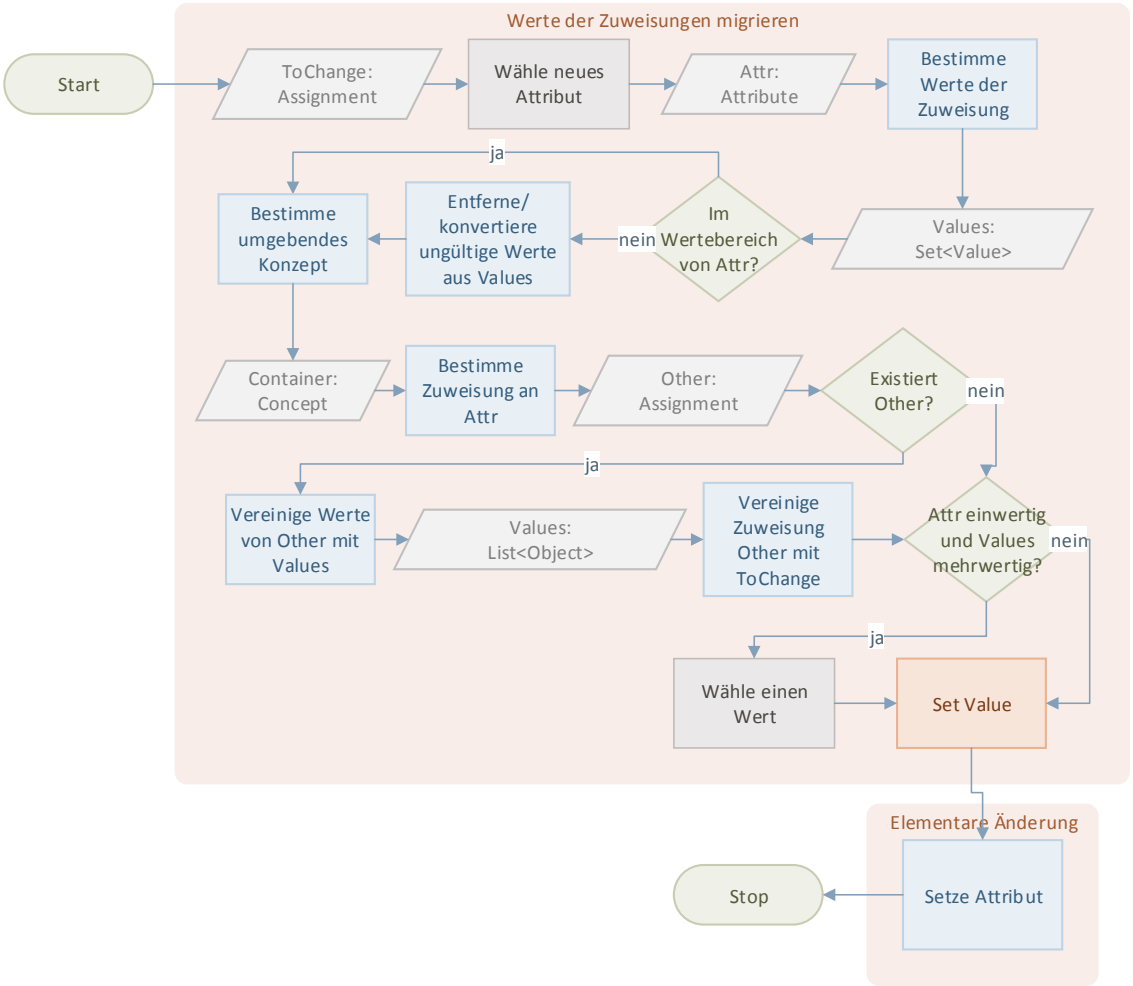

Abbildung 5-52 Ablauf des Set Attribute Of Operators

5.6.2 Set Value

Auswirkung

Der Operator setzt den Wert einer Zuweisung und führt alle Anpassungen durch, die aufgrund der Semantik der Sprachmuster notwendig werden.

Ablauf

Inkonsistenzen beheben

Der Operator wird an einer Zuweisung `ToChange` aufgerufen. Danach werden zunächst alle für den Operator relevanten Informationen ermittelt. Dies sind das zu `ToChange` gehörende Attribut `Attr`, das umgebende Konzept von `ToChange Container`, sowie alle bisherigen Werte (`OldValues`) der Zuweisung. Nachdem diese Informationen bestimmt wurden, werden die neuen Werte (`Values`) der Zuweisung aus dem Wertebereich des Attributes gewählt. Für einwertige Attribute darf in diesem Schritt maximal ein Wert gewählt werden. Ähnlich verhält es sich mit obligatorischen Attributen, für die in diesem Schritt ein Wert gewählt werden muss. Als nächstes werden alle Werte bestimmt, die durch den Operator von der Zuweisung gelöscht werden. Die Menge dieser Werte erhält man, indem man alle Werte, die in `Values` liegen aus `OldValues` entfernt. Anschließend werden die verschiedenen Sprachmuster, die durch den Wert einer Zuweisung beeinflusst werden, migriert. Dazu werden im Folgenden drei Subroutinen vorgestellt. Die Parameter aller Subroutinen entsprechend den gleichnamigen Elementen im Operator. Die Subroutinen dienen in diesem Fall nur der Hierarchisierung innerhalb des Operators und werden außerhalb nicht weiter verwendet.

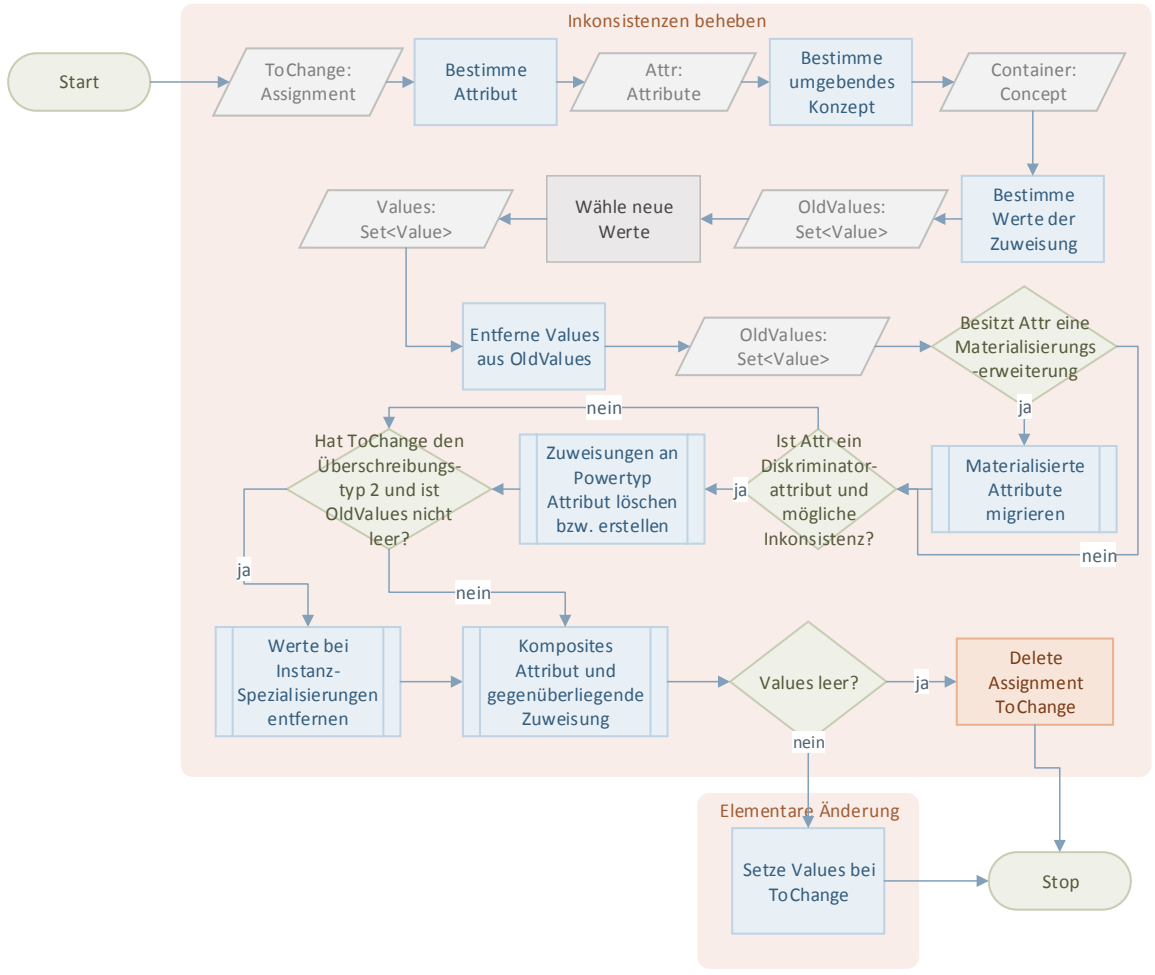

Abbildung 5-53 Ablauf des Set Value Operators

Eine Zuweisung im Kontext der Materialisierung erzeugt ein materialisiertes Attribut. Folglich müssen alle Zuweisungen an dieses Attribut entfernt werden, falls der Wert an der Zuweisung nicht mehr existiert. Außerdem implizieren alle Werte die neu hinzugefügt wurden ein weiteres materialisiertes Attribut, das gesetzt werden muss, wenn es obligatorisch ist. Für diesen Zweck wird die Subroutine *Materialisierte Attribute migrieren* (siehe unten) in dem Fall verwendet, dass `Attr` eine

Materialisierungserweiterung definiert. Wenn **Attr** ein Diskriminatorattribut und damit **Container** ein erweiterter Powertyp ist, führt das Setzen des Wertes auf **false** dazu, dass das entsprechende Attribut des partitionierten Typs nicht mehr von der jeweiligen Powertyp-Instanz geerbt wird und damit Zuweisungen bei Instanzen ungültig werden. Auf der anderen Seite führt das Setzen des Wertes auf **true** und ein obligatorisches Attribut beim partitionierten Typ dazu, dass Zuweisungen erstellt werden müssen. Für den Fall, dass eine dieser beiden Bedingungen zutrifft, wird die Subroutine *Zuweisung an Powertyp Attribut löschen bzw. erstellen* (siehe unten) ausgeführt, um diese Inkonsistenzen zu beheben. Auch für das Muster der Instanz-Spezialisierung ist der Wert einer Zuweisung von Bedeutung: Wenn **ToChange** den Überschreibungstyp 2 (**limited**) definiert und Werte von **ToChange** entfernt werden, dann dürfen diese Werte auch nicht mehr bei den Zuweisungen der Instanz-Spezialisierungen angenommen werden. Folglich wird dann die Subroutine *Werte bei Instanz-Spezialisierungen entfernen* verwendet, um diese Werte zu löschen. Nachdem die Auswirkungen der Änderung des Zuweisungswertes auf die Sprachmuster durch die Subroutinen ausgeführt wurden, wird die Subroutine *Komposites Attribut und gegenüberliegende Zuweisung* aufgerufen, die die Komposition, die durch ein Attribut **Attr** entsteht, und eine gegenüberliegende Zuweisung anpasst. Als nächstes wird durch den Operator überprüft, ob **Values** leer ist und damit die Zuweisung keinen Wert besitzt. In diesem Fall wird **ToChange** mittels des *Delete Assignment* Operators entfernt.

Elementare Änderung

Anderenfalls, wird der neue Wert **Values** gesetzt und die Ausführung des Operators endet.

Subroutine: Materialisierte Attribute migrieren

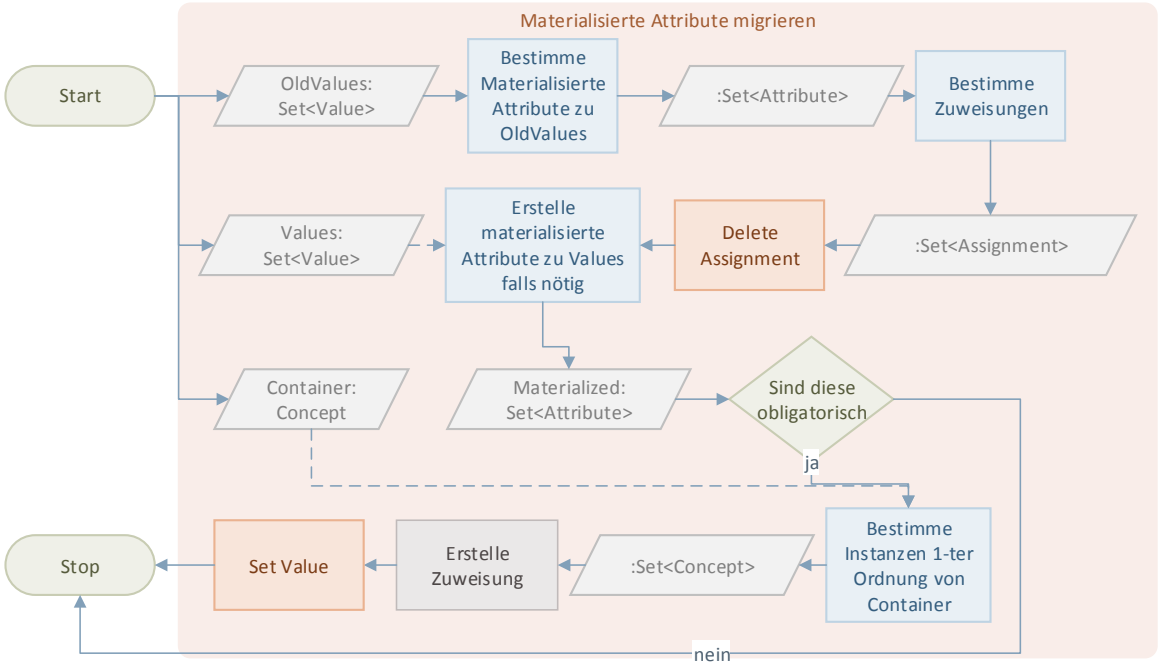

Abbildung 5-54 Ablauf der Subroutine Materialisierte Attribute migrieren

Die Subroutine erhält drei Parameter **OldValues**, **Values** und **Container**. Im ersten Schritt werden zunächst alle materialisierten Attribute von **Container**, die zu **OldValues** gehören, ermittelt. Danach werden alle Zuweisungen an diese Attribute bestimmt und durch den *Delete Assignment* Operator gelöscht. Anschließend erstellt die Subroutine mit Hilfe der Werte von **Values** materialisierte Attribute (**Materialized**) in **Container**, wenn sie nicht bereits existierten. Wenn die Attribute aus **Materialized** obligatorisch sind, werden als nächstes alle Instanzen 1-ter Ordnung von **Container** bestimmt, da diese eine Zuweisung definieren müssen. Nachdem die Zuweisungen erstellt wurden,

wird der *Set Value* Operator aufgerufen, um den Wert festzulegen. Danach endet die Ausführung der Subroutine.

Subroutine: Zuweisung an Powertyp Attribut löschen bzw. erstellen

Initial erhält die Subroutine drei Parameter: Konzept **`Container`**, Diskriminatorattribut **`Attr`** und eine Menge an Werten **`Values`**. Zunächst wird für **`Attr`** das entsprechende Attribut am partitionierten Typ **`PartAttr`** bestimmt. Wenn **`Container`** ein Powertyp ist (und die Zuweisung **`ToChange`** damit als Standardwert für **`Attr`** dient), werden alle Instanzen 1-ter Ordnung (**`Instances`**, Regel A.5) von **`Container`** und anschließend die virtuellen Zuweisung an **`Attr`** bestimmt. Danach werden aus der Menge **`Instances`** alle Konzepte entfernt, die **`PartAttr`** erben und folglich den Wert **`true`** für **`Attr`** zugewiesen haben. Für jedes Element dieser bereinigten Menge (**`Instances`**) oder für **`Container`** (falls dieser kein Powertyp ist), wird anschließend der Deep Instantiation Zähler $\mathtt{n_{rel}(PartAttr)}$ von **`PartAttr`** bezüglich des Elements (**`Container`** oder **`Instances`**) berechnet, um anschließend alle Instanzen bis zur $\mathtt{n_{rel}(PartAttr)}$-ten Ordnung (**`AssignInsts`**) von **`Container`** zu ermitteln. Falls **`Values`** den Wert **`false`** trägt, müssen alle Zuweisungen der Elemente von **`AssignInsts`** an **`PartAttr`** entfernt werden, deshalb werden diese zunächst ermittelt und durch den *Delete Assignment* Operator gelöscht. Wenn jedoch **`Values`** gleich {**`true`**} ist und gleichzeitig noch **`PartAttr`** obligatorisch, dann muss das Attribut bei allen Elementen aus **`AssignInsts`** gesetzt werden, bevor die Ausführung der Subroutine beendet ist. Deshalb wird zunächst eine Zuweisung erstellt und danach der *Set Value* Operator aufgerufen.

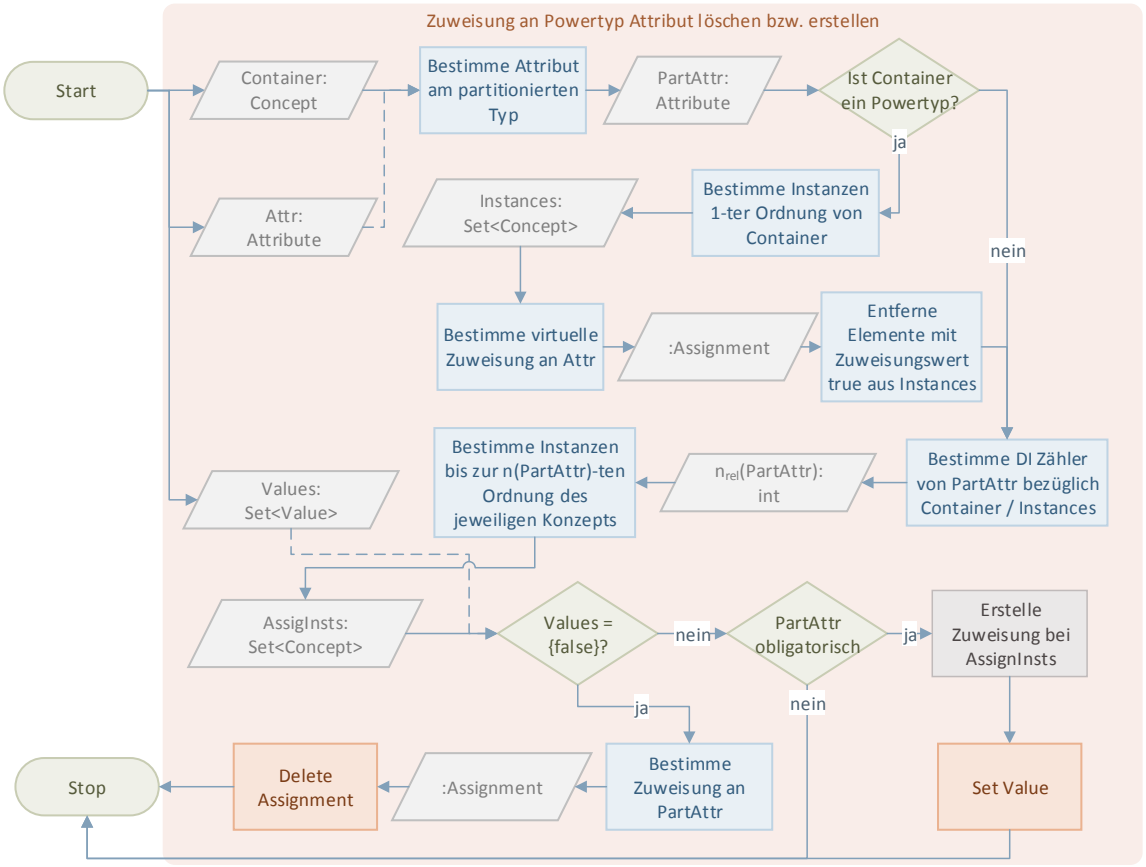

Abbildung 5-55 Ablauf der Subroutine Zuweisung an Powertyp Attribut löschen bzw. erstellen

Subroutine: Werte bei Instanz-Spezialisierungen entfernen

Die Subroutine wird mit drei Parametern (**`Container`**, **`Attr`**, **`OldValues`**) parametriert. Zunächst werden alle Instanz-Spezialisierungen von **`Container`** und danach die jeweilige Zuweisung an **`Attr`**

bestimmt. Diese wird dann um die Werte von `OldValues` bereinigt und der neue Wert wird durch den *Set Value* Operator gesetzt, womit die Subroutine endet.

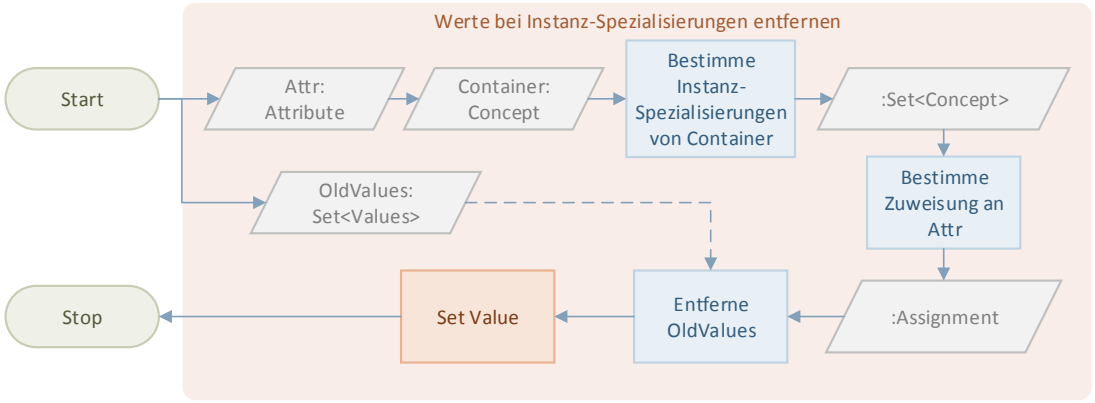

Abbildung 5-56 Ablauf der Subroutine Werte bei Instanz-Spezialisierungen entfernen

Subroutine: Komposites Attribut und gegenüberliegende Zuweisung

Die Subroutine wir mit fünf Parametern aufgerufen: `Attr`, `Values`, `OldValues`, `ToChange` und `Container`.

Zu Beginn wird das gegenüberliegende Attribut `Opposite` von `Attr` ermittelt und falls es existiert die zu `OldValues` gehörenden Konzepte ermittelt. Zu allen Konzepten aus `Oldvalues`, werden anschließend die Zuweisung an `Opposite` ermittelt und `Container` als Wert durch den *Set Value* Operator entfernt, falls dieser vorhanden ist. Analog werden als nächstes alle Konzepte zu `Values` mitsamt ihrer Zuweisungen an `Opposite` bestimmt und `Container` als Wert zur Zuweisung mit Hilfe des *Set Value* Operators hinzugefügt, falls eine Zuweisung existiert und es nicht bereits als Wert vorliegt. Wenn keine Zuweisung vorhanden ist, wird eine neue erstellt und `Container` als Wert zugewiesen (Aufruf des *Set Value* Operators).

Anschließend wird untersucht, ob `Attr` ein komposites Attribut ist. Wenn dem so ist, ist `Attr` ein Referenzattribut. Deshalb werden die zu `OldValues` gehörenden Konzepte (`OldCon`) ermittelt und anschließend für jedes Element aus `OldCon` alle Zuweisungen an `Attr` (`OldValAssigns`), die das Konzept als Wert besitzen, bestimmt. Wenn diese Menge nur aus `ToChange` besteht[44], dann wird durch den Operator die letzte Zuweisung von einem Vaterkonzept gelöscht und deshalb wird das Element aus `OldCon` durch den *Delete Concept* Operator entfernt. Anschließend werden alle neu hinzugefügten Werte dahingehend untersucht, ob sie bereits ein anderes Vaterkonzept haben. Dazu werden die zu `Values` gehörenden Konzepte `ValCon` zunächst ermittelt, falls `Values` nicht leer ist. Als nächstes werden alle Zuweisungen (`ValAssigns`) an `Attr` bestimmt, die `ValCon` als Wert haben. Wenn diese Menge an Zuweisungen leer ist (die Zuweisung `ToChange` besitzt noch nicht zwangsweise alle Werte aus `Values`) oder nur aus `ToChange` besteht, dann müssen alle Zuweisungen ungleich `ToChange` aufgrund der eindeutigen Zuordnung eines Kindkonzepts zu seinem Vaterkonzept mit Hilfe des *Set Value* Operators dahingehend angepasst werden, dass das Element aus `ValCon` von der Zuweisung entfernt wird. Danach ist die Ausführung der Subroutine beendet.

[44] Wenn beispielsweise ein Konzept ein neues Vaterkonzept erhält, dann können an dieser Stelle mehr als eine Zuweisung in `OldValAssigns` liegen.

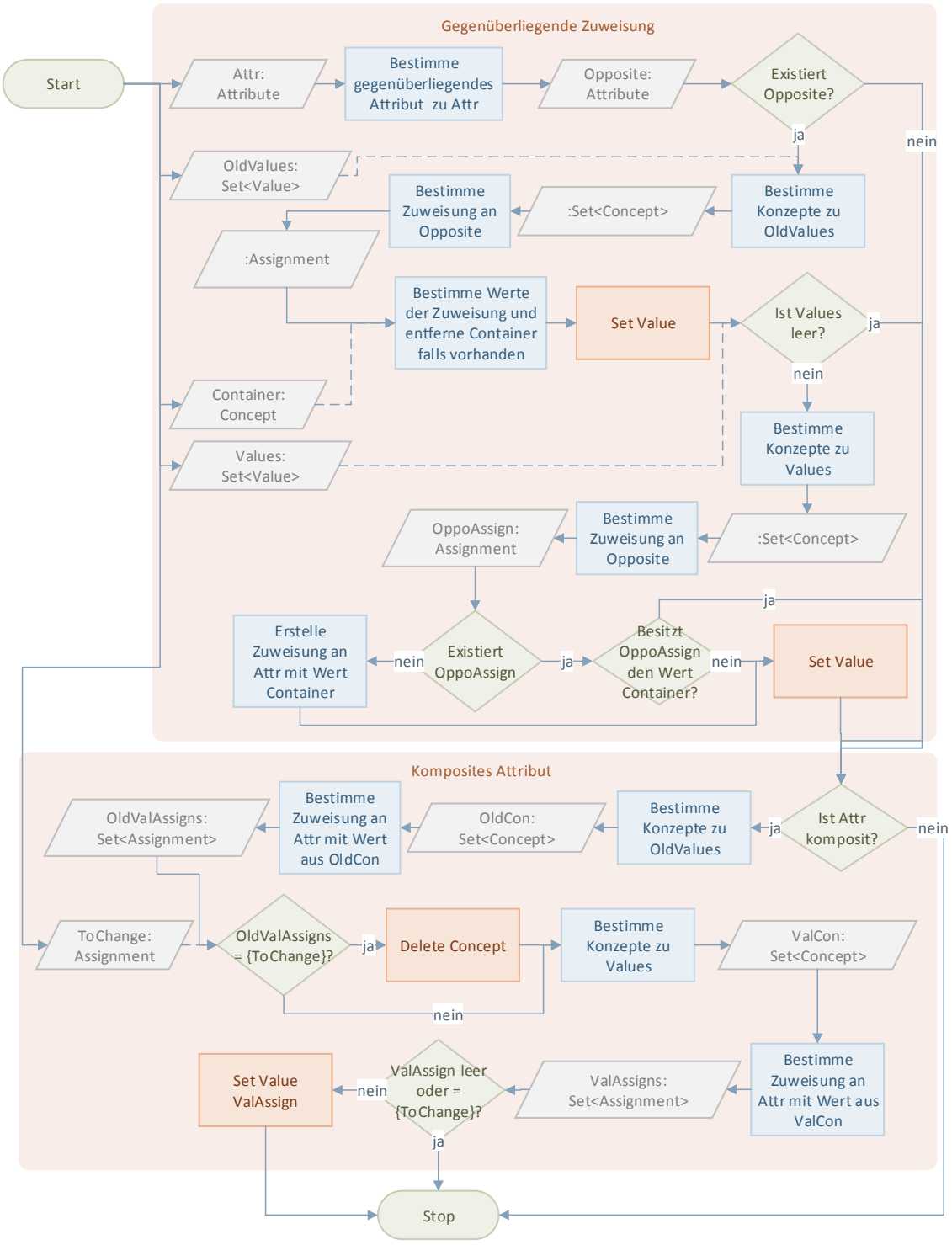

Abbildung 5-57 Ablauf der Subroutine Komposites Attribut und gegenüberliegende Zuweisung

Beispiel

In Abbildung 5-58 ist auf der linken Seite ein kleiner Ausschnitt aus einem Unternehmensmodell dargestellt. Auf Ebene **M1** ist dabei das Konzept **Employee** für einen Mitarbeiter definiert worden. Aus steuerlichen Zwecken (Überweisung des Kindergeldes) können jedem Mitarbeiter Angehörige (**Dependent**) zugeordnet werden. Die dafür modellierte Beziehung ist bidirektional (Attribute **parent** und **children**) und das Attribut **children** vom Konzept **Employee** ist komposit, da die Angehörige, nur von Bedeutung sind, wenn der Mitarbeiter im Unternehmen tätig ist. Neben der Ebene **M1** wurde

auch eine Ebene **M0** mit je zwei Instanzen von **Employee** (**Homer** und **Marge**) und **Dependent** (**Bart** und **Lisa**) definiert. Darin ist **Lisa Homer** zugeordnet, während **Bart Marge** zugewiesen ist.

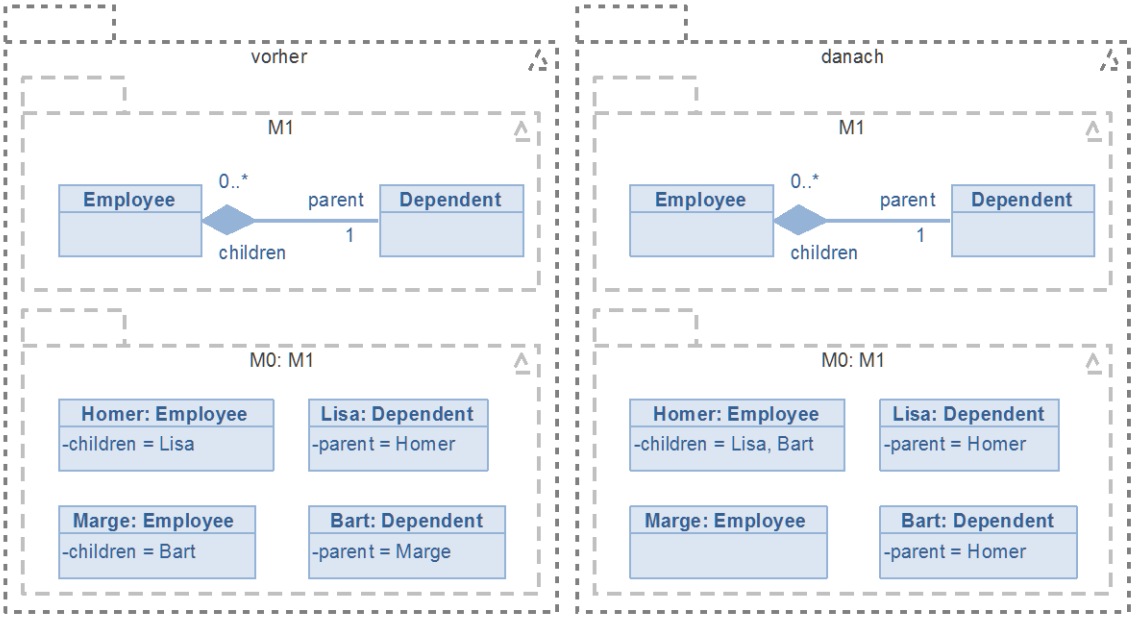

Abbildung 5-58 Beispiel vor (links) und nach (rechts) der Ausführung des Set Value Operators

Nun soll die Zuweisung von **Homer** an **children** geändert werden und beide Angehörigen über **Homer** abgerechnet werden. Dazu wird der *Set Value* Operator an der Zuweisung von **Homer** aufgerufen. Um den Überblick über die Parameterbelegung und den aktuellen Operatorablauf zu gewährleisten, wurde in Abbildung 5-59 ein Sequenzdiagramm zu diesem Beispiel dargestellt. Zu Beginn werden alle relevanten Parameter wie **Attr** (**children**), **Container** (**Homer**) und **OldValues** ({**Lisa**}) ermittelt. Danach wird der neue Wert {**Lisa, Bart**} gewählt und **Lisa** aus **OldValues** entfernt, wodurch die Menge leer ist. Da **children** keine Materialisierungserweiterung besitzt, kein Diskriminatorattribut ist und **OldValues** leer (sowie der Überschreibungstyp der Zuweisung 1 ist), wird als nächstes die Subroutine *Komposites Attribut und gegenüberliegende Zuweisung* aufgerufen. Dabei wird zunächst das gegenüberliegende Attribut **parent**, das bei **Dependent** definiert wurde, ermittelt. Weil **OldValues** leer ist, werden als nächstes die Konzepte zu **Values**, also **Lisa** und **Bart** bestimmt. Da die Zuweisung an **parent** bei **Lisa** bereits **Homer** enthält, wird nur für **Bart** der *Set Value* Operator aufgerufen.

Dieser innere Operator (Aufruf an der Zuweisung von **Bart** für **parent**) bestimmt zunächst wieder alle relevanten Informationen (**Attr** = **parent**, **Container** = **Bart**, **OldValues** = {**Marge**}) und erhält als **Values** die Menge {**Homer**} (ausgewählt durch den äußeren Operator). Mit der gleichen Begründung wie oben wird als nächstes ebenfalls die Subroutine *Komposites Attribut und gegenüberliegende Zuweisung* ausgeführt. Dazu wird zunächst **children** als gegenüberliegendes Attribut bestimmt und dann die Zuweisung von **Marge** an **children** ermittelt und dafür wiederrum der *Set Value* Operator benutzt, um **Bart** als Wert zu entfernen.

Die 3.Stufe des Operators beginnt analog wie die anderen beiden ((**Attr** = **children**, **Container** = **Marge**, **OldValues** = {**Bart**}) und erhält als **Values** eine leere Menge (ausgewählt durch den inneren Operator). Danach kommt es erneut zur Ausführung der Subroutine, die als erstes wieder **Bart** und die Zuweisung an **parent** bestimmt. Nun kommt es zu einem Zyklus, da für **Bart** erneut der *Set Value* Operator aufgerufen werden soll. Dieser muss, wie zu Anfang des Kapitels erläutert, erkannt werden und infolgedessen darf der *Set Value* Operator nicht erneut an **Bart** ausgeführt werden. Da **Values**

eine leere Menge darstellt, wird danach untersucht, ob **children** ein komposites Attribut ist. Weil dem so ist, werden alle Zuweisungen ermittelt, die **Bart** als Wert haben, also die Zuweisung von **Marge** und **Homer** an **children**. Aufgrund dessen und der Tatsache, dass **Values** und damit auch **ValAssigns** leer sind, wird die Subroutine danach beendet und die Ausführung kehrt zum *Set Value* Operator der 3. Stufe zurück. Dieser erkennt, dass **Values** leer ist und entfernt die Zuweisung von **Marge** an **children** mit Hilfe des *Delete Assignment* Operators, wodurch der Operator 3. Stufe endet und der innere Operator mit der Bestimmung der Konzepte für **Values** (= **Homer**) innerhalb der Subroutine *Komposites Attribut und gegenüberliegende Zuweisung* fortfährt. Danach wird die Zuweisung von **Homer** an **children** ermittelt und da **Homer** den Wert **Bart** noch nicht besitzt, wird versucht, erneut den *Set Value* Operator für diese Zuweisung aufzurufen. Auch an dieser Stelle kommt es zu einem Zyklus, weshalb der Operator nicht ausgeführt wird. Weil **parent** nicht komposit ist, endet die Subroutine und anschließend wird **Values** (=**Homer**) für das Attribut **parent** bei **Bart** gesetzt, wodurch der innere Operator endet.

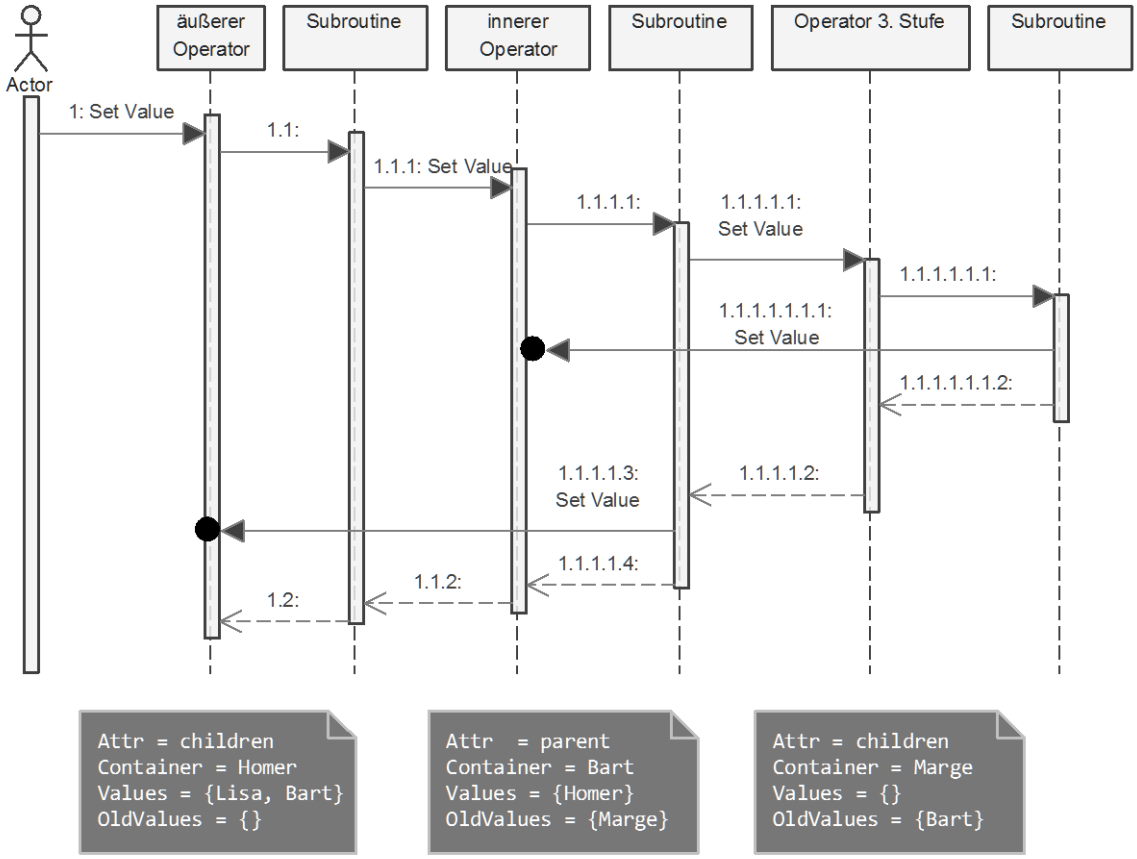

Abbildung 5-59 Sequenzdiagramm zum Beispiel des Set Value Operators

Danach schließt sich die Überprüfung, ob **Attr** (=**children**) komposit ist, des äußeren Operators innerhalb der Subroutine *Komposites Attribut und gegenüberliegende Zuweisung* an. Da dies zutrifft, werden dann, weil **OldValues** leer ist, alle zu **Values** zählenden Konzepte ermittelt, was als Resultat **Lisa** und **Bart** liefert. Als nächstes wird für jedes dieser Konzepte die Menge an Zuweisungen bestimmt, die **children** als Attribut und das jeweilige Konzept als Wert haben. Für beide besteht die Menge in diesem Fall nur aus der Zuweisung von **Homer**, da die Zuweisung von **Marge** bereits entfernt wurde. Folglich endet die Subroutine und der äußere Operator setzt die Zuweisung von **Homer** auf

den neuen Wert {`Lisa, Bart`}. Dadurch endet auch der äußere Operator mit dem in Abbildung 5-58 auf der rechten Seite gezeigten Modell.

5.6.3 Set Update Behaviour

Auswirkung

Durch den Operator wird der Überschreibungstyp für die Zuweisung gesetzt bzw. geändert. Dabei werden Zuweisungen der Instanz-Spezialisierung angepasst oder entfernt, falls dies nötig ist.

Ablauf

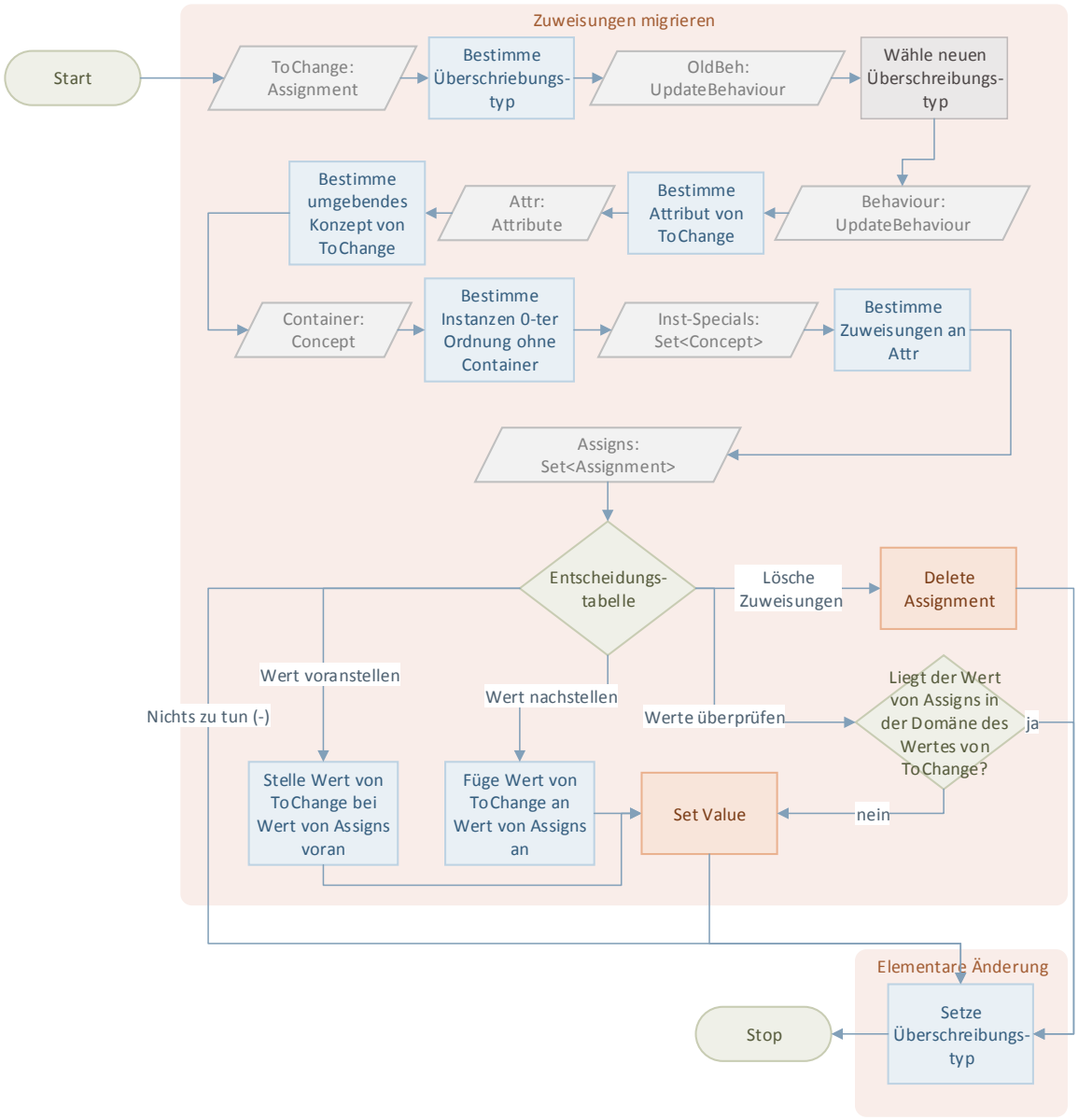

Abbildung 5-60 Ablauf des Set Update Behaviour Operators

Zuweisungen migrieren

Der Ablauf des Operators (Abbildung 5-60) beginnt mit dem Aufruf an einer Zuweisung `ToChange`, von der im ersten Schritt der Überschreibungstyp `OldBeh` ermittelt wird. Danach wird der neue Überschreibungstyp `Behaviour` gewählt, bevor im Anschluss das zu `ToChange` gehörige Attribut `Attr` bestimmt wird. Als nächstes wird das umgebende Konzept (`Container`) von `ToChange` und

anschließend alle Instanz-Spezialisierungen von `Container` (`Inst-Specials`) ermittelt. Als nächstes werden alle Zuweisungen an `Attr` (`Assigns`) von `Inst-Specials` bestimmt, da deren Wert durch `ToChange` als Zuweisung am Prototyp (`Container`) beeinflusst wird. Je nachdem, wie sich der alte (`OldBeh`) zum neuen Überschreibungstyp `Behaviour` verhält, werden verschiedene Schritte ausgeführt. Die möglichen Kombinationen und ihre Folgen sind in Tabelle 5-2 gezeigt. Wenn `ToChange` als neuen Überschreibungstyp `forbidden` (Typ 0) erhält, er sich von Typ 3 (`append`) oder 4 (`prepend`) auf Typ 2 (`limited`) oder von Typ 1 oder 2 auf Typ 3 oder 4 ändert, werden alle Zuweisungen durch den *Delete Assignment* Operator entfernt. Für den Fall, dass `OldBeh append` (Typ 3) oder `prepend` (Typ 4) als Wert besaß und nun Typ 1 (`normal`) gewählt wurde, wird die vorher implizite Berechnung aller Zuweisungswerte durchgeführt, indem der Wert von `ToChange` voran- (Typ 3) oder nachgestellt (Typ 4) wird, und die Zuweisungen von `Assigns` durch den *Set Value* Operator entsprechend angepasst. Falls der Überschreibungswert von `normal` (Typ 1) auf `limited` (Typ 2) verändert wird, werden alle Zuweisungen überprüft, ob die einzelnen Werte in der Domäne, die durch `ToChange` aufgespannt wird, liegen. Falls dies nicht der Fall ist, werden die invaliden Werte durch den *Set Value* Operator gelöscht. In allen anderen Kombinationsfällen müssen die Zuweisungen nicht angepasst werden.

Elementare Änderung

Anschließend wird der neue Überschreibungstyp für `ToChange` gesetzt und der Operator endet.

Tabelle 5-2 Entscheidungstabelle des Set Update Behaviour Operators

OldBeh → Behaviour ↓	Typ 0 (forbidden)	Typ 1 (normal)	Typ 2 (limited)	Typ 3 (append)	Typ 4 (prepend)
Typ 0 (forbidden)	-	Lösche Zuweisungen	Lösche Zuweisungen	Lösche Zuweisungen	Lösche Zuweisungen
Typ 1 (normal)	-	-	-	Wert voranstellen	Wert nachstellen
Typ 2 (limited)	-	Werte überprüfen	-	Lösche Zuweisungen	Lösche Zuweisungen
Typ 3 (append)	-	Lösche Zuweisungen	Lösche Zuweisungen	-	-
Typ 4 (prepend)	-	Lösche Zuweisungen	Lösche Zuweisungen	-	-

Beispiel

Abbildung 5-61 zeigt auf der linken Seite ein sehr einfaches Modell. Ebene `M1` definiert nur ein Konzept `Car`, das ein Attribut `typeName` besitzt. Die darunterliegende Ebene `M0` definiert eine Instanz von `Car` (`Fiesta`) und eine Instanz-Spezialisierung von `Fiesta` (`FiestaTitanium`). Der Prototyp setzt das Attribut `typeName` auf den Wert **„Fiesta“** und definiert den Überschreibungstyp 3 (`append`) für diese Zuweisung. `FiestaTitanium` wiederum, setzt das Attribut mit dem Wert **„ Titanium“**, wodurch virtuell der Wert **„Fiesta Titanium“** entsteht. Im Nachfolgenden wird durch den *Set Update Behaviour* Operator der Überschreibungstyp der Zuweisung bei `Fiesta` an `typeName` von `3` auf `1` (normal, Standardwert) geändert. Dazu wird nachdem der Überschreibungstyp ermittelt und der neue Typ gewählt wurde, `typeName` als Attribut der Zuweisung zusammen mit `Fiesta` als umgebendes Konzept bestimmt. Danach werden alle Instanz-Spezialisierungen (in diesem Fall `FiestaTitanium`) und die entsprechenden Zuweisungen ermittelt. Da der Überschreibungstyp von `append` auf `normal` gesetzt werden soll, wird zunächst der virtuelle Zuweisungswert durch voranstellen des Wertes von `Fiesta` bestimmt und danach der neue Wert **„Fiesta Titanium“** für `typeName` bei

`FiestaTitanium` durch den *Set Value* Operator gesetzt. Als letztes wird noch der neue Überschreibungstyp 1 (in der Abbildung nicht dargestellt, da dies der Standardwert ist) bei der Zuweisung gesetzt und das in Abbildung 5-61 auf der rechten Seite gezeigte Modell entsteht.

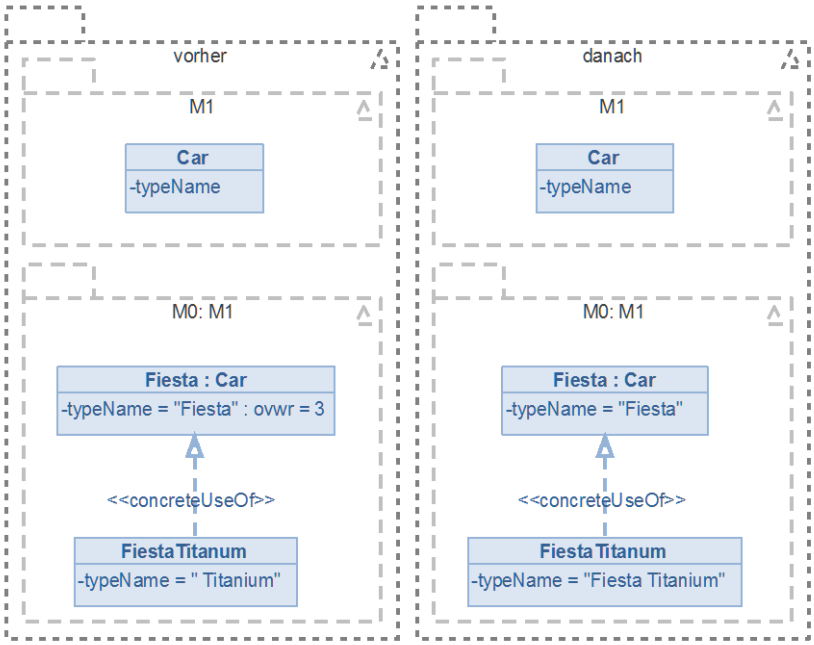

Abbildung 5-61 Beispiel vor (links) und nach (rechts) der Ausführung des Set Update Behaviour Operators

5.6.4 Delete Assignment

Auswirkung

Die Ausführung des Operators löscht eine Zuweisung, falls das entsprechende Attribut nicht obligatorisch ist oder noch eine andere virtuelle Zuweisung an das Attribut existiert.

Ablauf

Abhängigkeiten auflösen

Der Operator wird an einer Zuweisung `ToDelete` aufgerufen, die gelöscht werden soll. Im ersten Schritt wird das zu `ToDelete` gehörende Attribut `Attr` bestimmt. Wenn dieses Attribut obligatorisch ist, wird zunächst das umgebende Konzept der Zuweisung (`Container`) bestimmt und anschließend werden alle virtuellen Zuweisungen von `Container` an `Attr` ermittelt. Für den Fall, dass keine weitere virtuelle Zuweisung außer `ToDelete` existiert, kann die Zuweisung nicht entfernt werden (da `Attr` in diesem Fall ja obligatorisch ist), deshalb wird der Operator beendet. Wenn allerdings eine weitere virtuelle Zuweisung vorhanden ist oder `Attr` optional, dann wird überprüft, ob die Zuweisung einen Überschreibungstyp von 3 (`append`) oder 4 (`prepend`) besitzt. Ist dies der Fall, werden alle Zuweisungen der Instanz-Spezialisierungen ermittelt und der Wert von `ToDelete` angehängt (Typ 4) oder vorangestellt (Typ 3). Danach wird überprüft, ob der Wert der Zuweisung bereits leer ist. Wenn er dies nicht ist, wird unterschieden, ob `Attr` ein Diskriminatorattribut ist. In diesem Fall wird zunächst die nächste virtuelle Zuweisung (Definition 4.24) für `Attr` bestimmt. Falls diese den Wert `false` hat, müssen alle Zuweisungen bei allen Instanzen von `Container` an das entsprechende Attribut des partitionierten Typs entfernt werden. Dies geschieht dadurch, dass der *Set Value* Operator für `ToDelete` aufgerufen und `false` als Wert ausgewählt wird. Wenn kein Diskriminatorattribut vorliegt, wird der Wert von `ToDelete` mit Hilfe des *Set Value* Operators entfernt (Auswahl der leeren Menge), wodurch alle Sprachmuster migriert werden.

Elementare Änderung

Zum Abschluss wird die Zuweisung gelöscht und der Operator ist beendet.

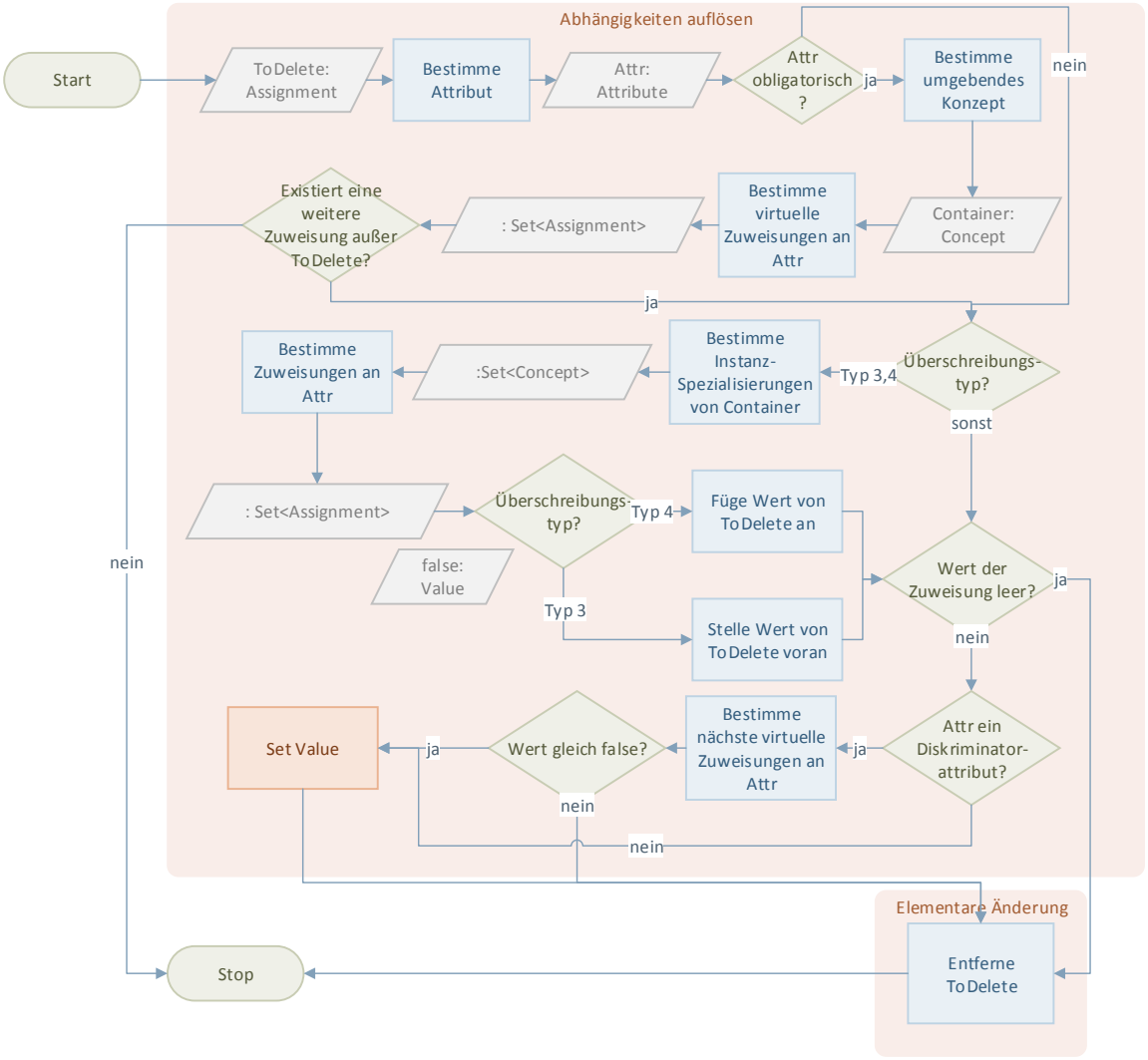

Abbildung 5-62 Ablauf des Delete Assignment Operators

5.7 Zusammenfassung

In diesem Abschnitt wurden Operatoren für alle Änderungen (inklusive dem Entfernen) eines Modellelements des LMMs vorgestellt. Diese führen neben der elementaren Änderung auch eine entsprechende Koevolution durch, die einen im Sinne der Regeln aus Abschnitt 4.2 konsistenten Zustand erzeugen. Dabei wurde auch die Einhaltung der Konformität der Semantik aller Sprachmuster gewährleistet. Dadurch entsteht eine Bibliothek an Operatoren, die in dieser Vollständigkeit bei keiner anderen Forschungsarbeit vorliegt und gleichzeitig die Herausforderung H2 umsetzt. Die vorgestellten elementaren Operatoren dienen als Grundlage für die komplexen Operatoren des folgenden Kapitels 6.

6 Komplexe Operatoren zur Unterstützung sprachbasierter Muster

Dieses Kapitel stellt verschiedene Operatoren vor, die den Umgang mit sprachbasierten Mustern erlauben. Die Operatoren ermöglichen es, die in Kapitel 2 gezeigten Muster in ein vorhandenes Modell einzuführen, die verschiedenen Eigenschaften eines Musters zu ändern oder es aus einem Modell zu entfernen. In diesem Sinne ist die Bibliothek an Operatoren auch vollständig. Damit stellt sie eine Neuerung dar im Vergleich zu den Ansätzen der aktuellen Forschung und löst folglich Herausforderung H1.

Im Gegensatz zu den elementaren Operatoren aus Kapitel 5 verwenden die hier vorgestellten Operatoren zumeist mehrere elementare Änderungen, um ein Modell anzupassen. Für jedes Sprachmuster wurde ein eigenes Unterkapitel erstellt, in dem alle Operatoren präsentiert werden, die sich auf das Muster beziehen. Die Operatoren für das Muster Deep Instantiation sind bereits in den betroffenen Modellelementen Attribut (Attribute) und Konzept (Concept) im Abschnitt 5.5 bzw. 5.3 vorgestellt und werden daher an dieser Stelle nicht erneut aufgeführt.

Die Vorbemerkungen zum Aufbau der Abschnitte, sowie zu den Operatoren selbst, können aus der Einleitung zu Kapitel 5 übernommen werden. Die Abschnitte sind erneut in Auswirkung, Ablauf und (optionalen) Beispielen unterteilt. Auch bei den hier vorgestellten komplexen Operatoren können Zyklen auftreten, die erkannt und aufgelöst werden müssen. Wie dies konkret umgesetzt werden kann, ist ebenfalls zu Beginn des Kapitels 5 beschrieben.

6.1 Meta-Modellierung mit mehreren Ebenen

Die in diesem Abschnitt vorgestellten Operatoren dienen dazu, mit Meta-Modellhierarchien umzugehen, die aus mehr als zwei Ebenen bestehen. Sie erlauben es, Konzepte auf andere Ebenen zu verschieben, Ebenen zu verschmelzen oder eine neue Ebene zu erstellen, die über der Ausgangsebene liegt.

6.1.1 Move Type to Upper Level

Auswirkung

Der *Move Type to Upper Level*[45] Operator verschiebt ein ausgewähltes Konzept oder eine ausgewählte Enumeration eine Meta-Ebene nach oben. Alle aufwärts ebenen-abhängigen Referenztypen werden dabei ebenfalls verschoben.

Ablauf

Neue Ebene festlegen

In Abbildung 6-1 findet sich der Ablauf des Operators. Der Operator wird mit einem Referenztypen (ein Konzept oder einer Enumeration) `ToMove` initialisiert. Als erstes wird die Ebene, in die `ToMove` verschoben werden soll, festgelegt. Dazu sucht der Operator zunächst die nächst höhere Ebene. Falls

[45] Ein vereinfachte Form des Operator wurde bereits in [61] veröffentlicht.

diese nicht existiert, wird eine neue Meta-Ebene `NewLevel` erstellt und die Instanziierung zwischen beiden Ebenen hergestellt.

Instanziierung migrieren

Wenn eine darüber liegende Ebene existiert, muss zunächst überprüft werden, ob `ToMove` Instanz eines anderen Konzepts `Type` ist. Wenn dem so ist, ist die Instanziierung nach der Verschiebung ungültig und muss folglich mittels des Operators *Delete Instantiation* gelöscht werden.

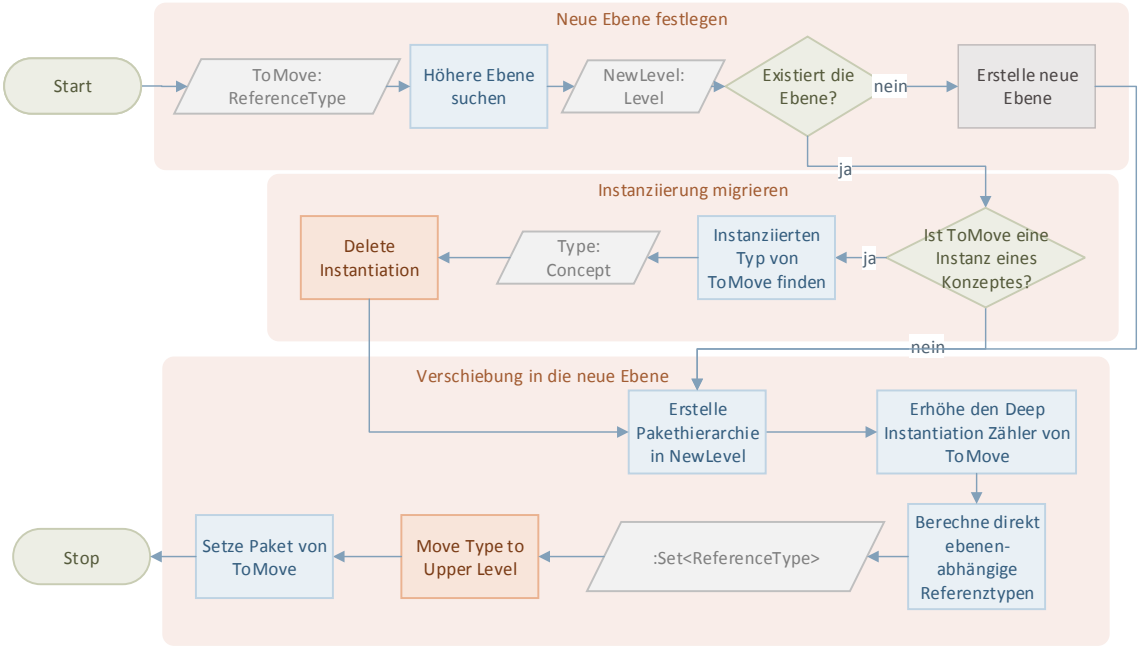

Abbildung 6-1 Ablauf des Move Type to Upper Level Operators

Verschiebung in die neue Ebene

Da Referenztypen immer in einem Paket liegen, muss der Operator nun die Pakethierarchie, in der `ToMove` definiert wurde, in der neuen Ebene nachbauen, soweit diese noch nicht existiert. Danach wird der Deep Instantiation Zähler von `ToMove` inkrementiert [46], damit Instanziierungen sowie Verbindungen zu abwärts ebenen-abhängigen Konzepten (Definition 4.11) weiterhin bestehen können.

Um die Migration korrekt abzuschließen, berechnet der Operator im nächsten Schritt alle von `ToMove` aufwärts ebenen-abhängigen (Definition 4.11) Referenztypen. Konkret sind dies alle Typen der an `ToMove` deklarierten Referenzattribute. Falls `ToMove` ein Konzept ist, kommen noch die Generalisierung, der Prototyp (entspricht `extends` und `concreteUseOf` Referenz) sowie ein möglicher partitionierter Typ (entspricht `partitions` Beziehung) hinzu.

Danach wird für alle oben berechneten Referenztypen der *Move Type to Upper Level* Operator aufgerufen. Da zwischen Referenztypen zyklische direkte Ebenen-Abhängigkeiten (z.B. eine bidirektionale Beziehung) bestehen können, müssen diese aufgebrochen werden, um eine Endlosschleife zu vermeiden. Dies erfolgt, wie bereits in Kapitel 5 erwähnt, außerhalb des Operators.

[46] Hierbei wird nicht der entsprechende Operator verwendet, da die entstehende Inkonsistenz durch die Verschiebung von `ToMove` wieder aufgehoben wird.

Zum Abschluss des Operators wird das Paket in der oben neu erstellten Pakethierarchie, das dem alten Paket entspricht, beim Referenztyp `ToMove` gesetzt und damit `ToMove` in die neue Ebene verschoben.

Beispiel

Eine beispielhafte Anwendung des Operators findet sich im Beispiel des *Create Powertype For* Operators in 6.3.4.

6.1.2 Extract Level

Auswirkung

Der Operator erstellt aus den ausgewählten Konzepten eine neue Meta-Ebene, die über der bisherigen Ebene liegt.

Ablauf

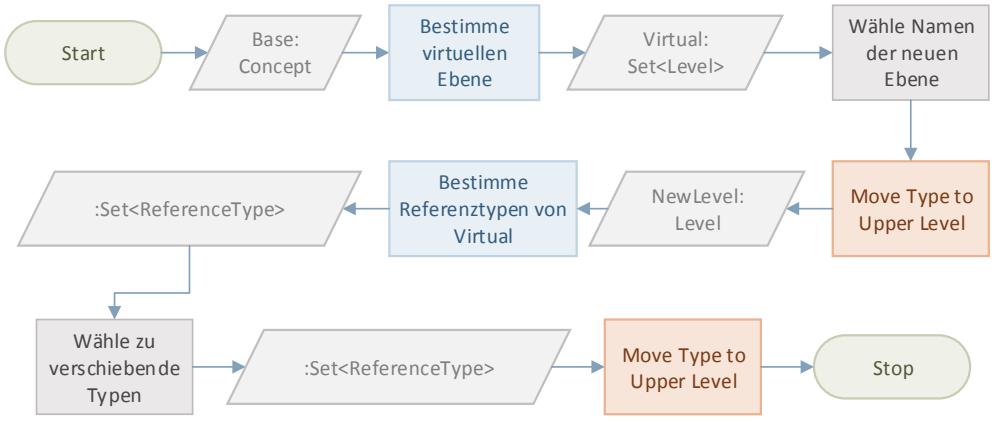

Abbildung 6-2 Ablauf des Extract Level Operators

Zunächst erhält der Operator ein Konzept `Base`, für das im nächsten Schritt die virtuelle Ebene `Virtual` berechnet wird. Anschließend wird der Name der neuen Ebene festgelegt und an den *Move Type to Upper Level* Operator mitsamt `Base` übergeben. Dadurch entsteht eine neue Ebene (`NewLevel`), in der `Base` liegt. Im nächsten Schritt werden alle Referenztypen von `Virtual` ermittelt und die zu verschiebenden Elemente daraus ausgewählt. Für diese Menge von Konzepten und Enumerationen wird dann jeweils der *Move Type to Upper Level* Operator aufgerufen, wobei `NewLevel` als Ziel der Verschiebung gesetzt wird. Auch hier kann es wieder zu Zyklen kommen, die aufgebrochen werden (siehe Kapitel 5).

6.1.3 Move Type to Lower Level

Auswirkung

Die Anwendung des Operators verschiebt einen Referenztyp um eine Ebene nach unten. Dabei werden auch abwärts ebenen-abhängige Referenztypen verschoben und eine eventuell vorhandene Instanziierung migriert.

Ablauf

Neue Ebene festlegen

Initial wird der Operator (Abbildung 6-3) mit einem Referenztyp `ToMove` aufgerufen, der eine Meta-Ebene nach unten verschoben werden soll. Im ersten Schritt werden alle Meta-Ebenen bestimmt, die die aktuelle Ebene von `ToMove` instanziieren und damit unter ihr in der Meta-Hierarchie liegen. Anschließend wird eine Ebene `NewLevel` gewählt oder erstellt, in die `ToMove` verschoben werden soll.

Falls keine Ebene existiert, die die Ebene von `ToMove` instanziiert, ist die Erstellung einer Ebene obligatorisch. Als nächstes muss überprüft werden, ob alle Instanz-Ebenen mit `NewLevel` logisch verschmolzen sind, da sonst z.B. keine Beziehungen oder Spezialisierungen zu `ToMove` von den beinhalteten Referenztypen möglich wäre. Falls die Verschmelzung noch nicht existiert, wird sie durch den Aufruf des *Set Aligned Level* Operators beim jeweiligen Instanz-Level (Auswahl des Wertes `NewLevel`) hergestellt. Daran anschießend baut der Operator die Pakethierarchie, in der `ToMove` liegt (soweit nötig) in der neuen Meta-Ebene nach, um später im letzten Schritt das entsprechende Paket bei `ToMove` setzen zu können.

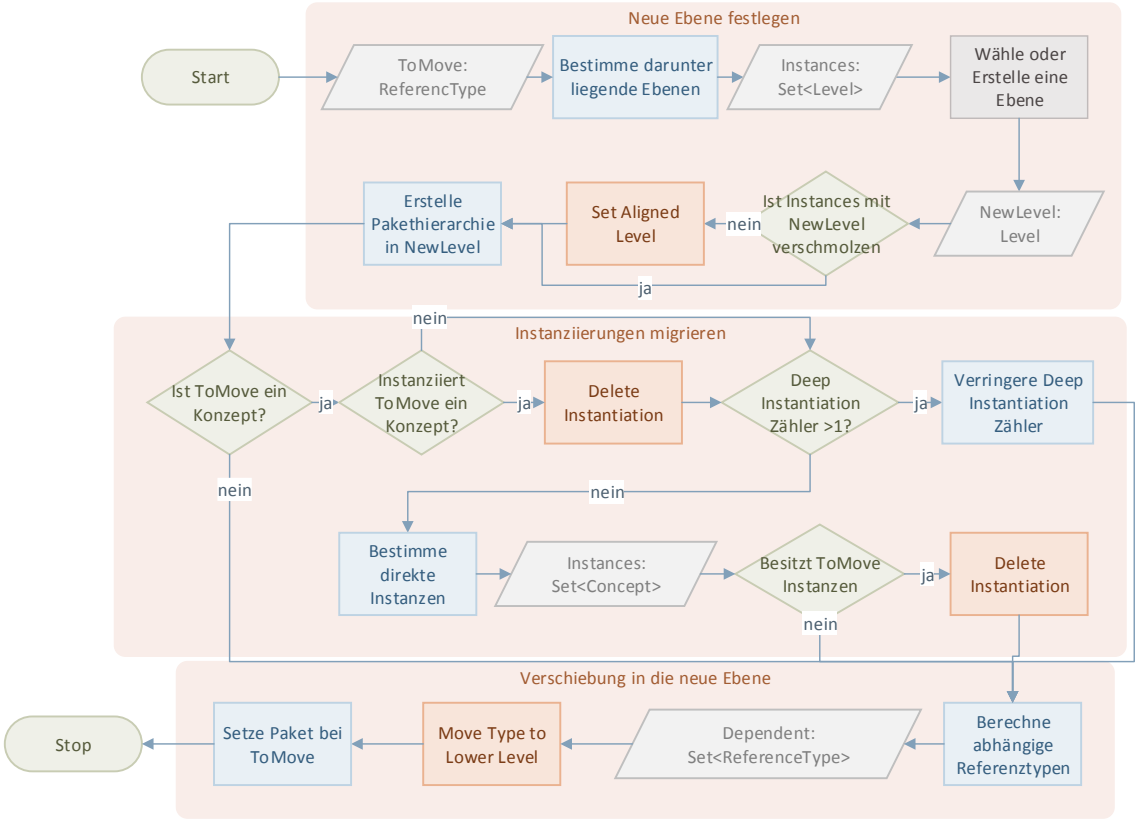

Abbildung 6-3 Ablauf des Move Type to Lower Level Operator

Instanziierungen migrieren

Wenn der Referenztyp eine Enumeration ist, kann der Operator nun sofort mit der Berechnung der abhängigen Referenztypen beginnen. Falls der Referenztyp aber ein Konzept ist, muss zunächst festgestellt werden, ob `ToMove` ein anderes Konzept instanziiert, da dies nach der Verschiebung nicht mehr möglich ist. Folglich wird in diesem Fall die Instanziierung durch den *Delete Instantiation* Operator gelöscht. Anschließend stellt der Operator fest, ob der Deep Instantiation Zähler von `ToMove` größer als eins ist. Wenn dies der Fall ist, dann existieren auch keine Instanzen von `ToMove` in der neuen Meta-Ebene (siehe Regel C.12) und es genügt den Zähler zu dekrementieren.

Falls der Deep Instantiation Zähler eins ist, muss festgestellt werden, ob `ToMove` Instanzen besitzt und welche diese sind, da diese Instanziierung durch die Verschiebung invalide wird (Regel C.12). Um dies zu beheben, wird die Instanziierung mit Hilfe des *Delete Instantiation* Operators gelöscht.

Verschiebung in die neue Ebene

Danach wird auch für den Fall, dass `ToMove` ein Konzept ist, mit der Berechnung der abhängigen Referenztypen `Dependent` fortgefahren. Darin sind zunächst einmal alle abwärts ebenen-abhängigen Referenztypen (Definition 4.11) enthalten. Hinzu kommen alle zu `ToMove` aufwärts ebenen-abhängigen

Referenztypen, die in der virtuellen Ebene von `NewLevel` instanziierbar sind, d.h. deren Deep Instantiation Zähler eins ist oder deren Instanziierung entsprechend hinausgezögert wurde (Definition 4.9).

Für alle Referenztypen dieser Menge wird dann der *Move Type to Lower Level* aufgerufen. Auch hier kann es zu zyklischen Abhängigkeiten unter Konzepten kommen, die aufgebrochen werden müssen, um die Rekursion zu beenden. Als Abschluss setzt der Operator das neue Paket von `ToMove` und verschiebt es damit in die neue Ebene.

Beispiel

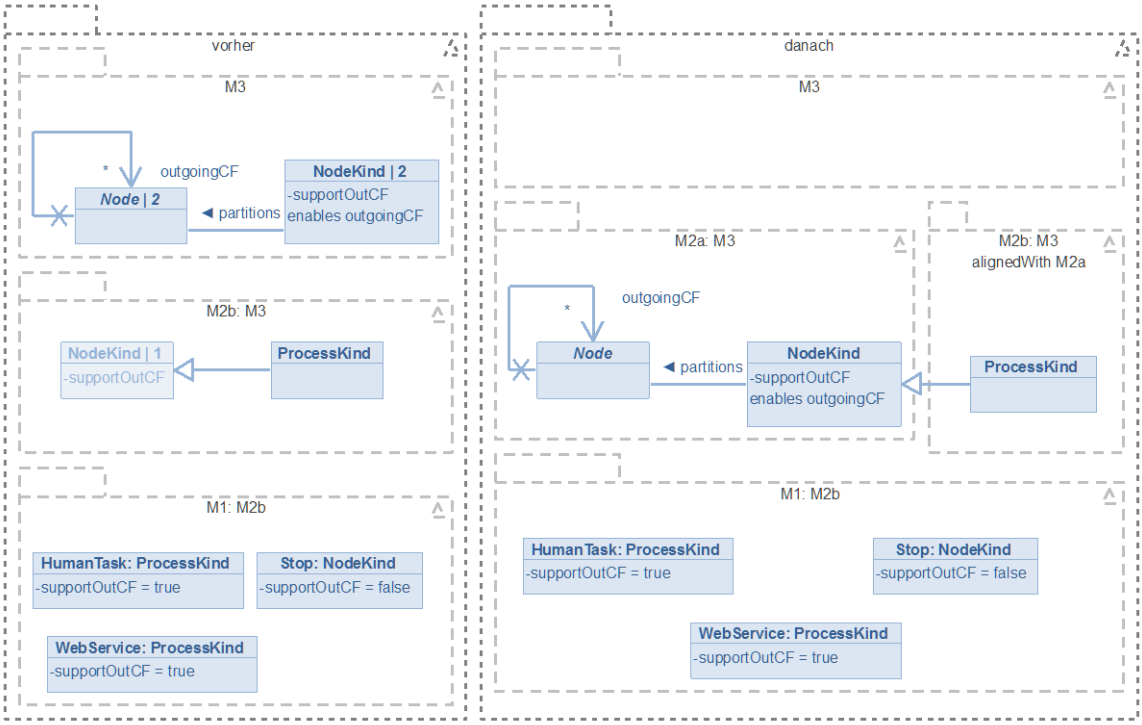

Abbildung 6-4 einfache Prozessmodellierungssprache vor (links) und nach (rechts) der Ausführung des Operators

In Abbildung 6-4 ist eine einfache Prozessmodellierungssprache auf der linken Seite definiert, in der auf `M3` ein erweiterter Powertyp modelliert wurde. Konzept `Node`, das für einen Knoten steht, ist dabei der partitionierte Typ und `NodeKind` der Powertyp, die beide einen Deep Instantiation Zähler von zwei besitzen und folglich erst auf `M1` instanziiert werden können. Weiterhin definiert `Node` eine reflexive Beziehung (durch das Attribut `outgoingCF`), welche der Abbildung eines ausgehenden Kontrollflusses dient und Knoten miteinander verbindet. `NodeKind` auf der anderen Seite besitzt ein Diskriminatorattribut `supportsOutCF`, das steuert, ob `outgoingCF` an die Instanzen von `NodeKind` vererbt wird. Die Verbindung (`enables`, siehe Abschnitt 4.1.4) zwischen den beiden Attributen wurde aus Übersichtsgründen nicht im Diagramm visualisiert.

Auf `M2b` wurde eine Spezialisierung des Powertyp `NodeKind` mit Namen `ProcessKind` definiert. Das Konzept ist deshalb als Erweiterung des Powertyps modelliert, da zum einen nicht nur Prozesse, sondern auch Stopp Knoten modelliert werden sollen und da es innerhalb der Menge an Prozessen erneut unterschiedliche Arten von Prozessen gibt (siehe `M1`). Um die Spezialisierung zu visualisieren, wurde `NodeKind` in der Ebene `M2b` dargestellt, da es durch das Deep Instantiation Muster in der Vererbung sichtbar ist (Regel C.14), auch wenn es dort nicht definiert wurde.

Die Meta-Ebene `M1` zeigt ein exemplarisches Modell, in dem `HumanTask`, `WebService` (Instanzen von `ProcessKind`) und `Stop` (Instanz von `NodeKind)` modelliert wurden. `HumanTask` steht im Modell für einen Prozess, der durch eine Person ausgeführt werden muss, da er zu komplex ist, um ihn automatisch ablaufen zu lassen. `WebService` dagegen steht für einen Prozess, der sich durch die Orchestrierung von verschiedenen Webservices definiert, während `Stop` für ein Stopp-Interface im Prozess steht und damit festlegt, wann der Prozess endet. Durch die Zuweisungen der drei Konzepte an das Diskriminatorattribut erbt `HumanTask` und `WebService` das Attribut `outgoingCF`, während `Stop` keine ausgehende Kontrollflüsse unterstützt und folglich auch den Wert `false` zuweist.

Nun soll der *Move Type to Lower Level* Operator an `NodeKind` aufgerufen und damit `NodeKind` eine Ebene nach unten verschoben werden. Dazu bestimmt der Operator zunächst alle Ebenen, die die aktuelle Ebene von `NodeKind` instanziieren (in diesem Fall `M2b`). Nun könnte man `M2b` als neue Ebene für `NodeKind` wählen, wir entschließen uns aber eine neue Ebene mit Name `M2a` zu erstellen. Weil `M2b` mit `M2a` noch nicht verschmolzen ist, wird nun der *Set Aligned Level* Operator aufgerufen, welcher die `alignedWith` Eigenschaft bei `M2b` setzt. Da in diesem Beispiel keine Pakethierarchie existiert, ist im nächsten Schritt nichts zu tun.

Aufgrund der Tatsache, dass `NodeKind` ein Konzept ist und zudem noch einen Deep Instantiation Zähler größer als eins (`2`) besitzt, wird als nächstes dieser dekrementiert. Danach werden alle abhängigen Konzepte berechnet, was als Ergebnis `Node` liefert, da dieses abwärts ebenen-abhängig ist. Das führt dazu, dass bei `Node` ebenfalls der *Move Type to Lower Level* Operator mit `M2a` als Zielebene aufgerufen wird. Aufgrund der Tatsache, dass `Node` einen Deep Instantiation Zähler von zwei hat, wird dieser dekrementiert. Da `Node` keine Instanzen auf einer darunterliegenden Ebene besitzt und auch neben `NodeKind` keine weiteren abhängigen Konzepte liefert, wird lediglich `Node` in die neue Ebene verschoben (ins Standardpaket). Nachdem der innere Operator (Aufruf an `Node`) endet, wird auch `NodeKind` nach `M2a` verschoben und der äußere Operator stoppt. An dieser Stelle zeigt sich, warum es nötig war `M2b` mit `M2a` zu verschmelzen, da ansonsten die Spezialisierung von `ProcessKind` zu `NodeKind` nicht mehr gültig gewesen wäre. Der Operator endet mit dem auf der rechten Seite von Abbildung 6-4 gezeigten Modell.

6.1.4 Inline Level

Auswirkung

Der Aufruf des Operators *Inline Level* fügt alle Referenztypen einer übergebenen Meta-Ebene einer Ebene hinzu, die diese zuvor instanziiert hatte und löscht die übergebene Meta-Ebene.

Ablauf

Der Operator wird an einer Ebene `ToInline` aufgerufen. Anschließend werden alle Ebenen berechnet, die `ToInline` instanziieren. Aus dieser Menge muss eine Instanz-Ebene `InlineTarget` gewählt werden, in die die Referenztypen von `ToInline` später verschoben werden. Danach wird für jede weitere Instanz-Ebene entschieden, ob sie logisch durch den *Set Aligned Level* Operator mit `InlineTarget` verschmolzen werden soll. Nachdem nun die virtuelle Ebene von `InlineTarget` gegebenenfalls erweitert wurde, werden alle Referenztypen von `ToInline` verschoben. Dazu wird für jeden Referenztyp der *Move Type to Lower Level* aufgerufen, wobei `InlineTarget` als Zielebene für alle Referenztypen gesetzt wird. Im letzten Schritt wird `ToInline` durch den Aufruf des *Delete Level* Operators gelöscht und die Ausführung des Operators endet.

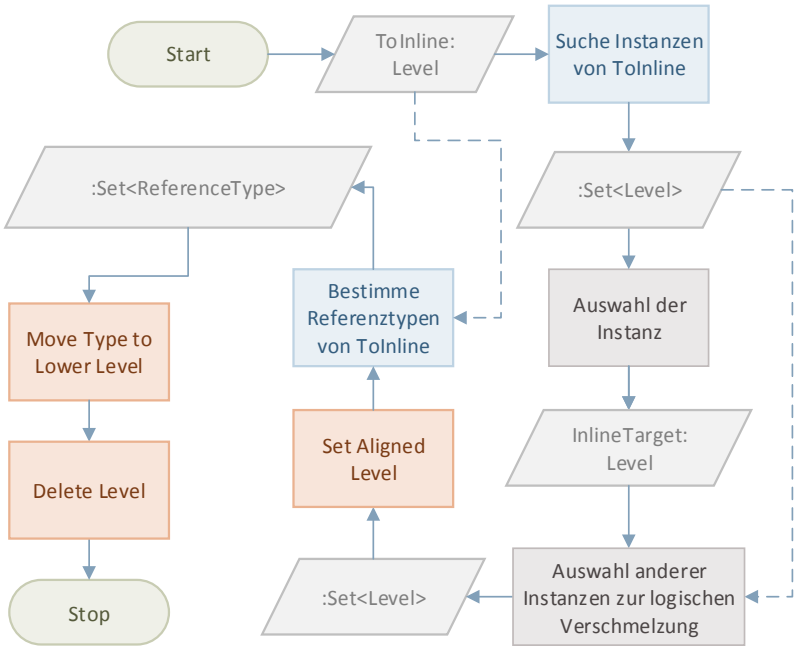

Abbildung 6-5 Ablauf des Inline Level Operators

6.2 Vererbung

Die in diesem Abschnitt vorgestellten Operatoren beziehen sich auf das Sprachmuster der Vererbung. Sie sind in ähnlicher Form in anderen Arbeiten (z.B. [56, 119]) vorhanden. Allerdings berücksichtigen die hier vorgestellten Operatoren auch die Auswirkungen anderer Sprachmuster, wodurch sie sich von existierenden Ansätze unterscheiden. Neben dem Verschieben von Attributen innerhalb einer Vererbungshierarchie werden auch Operatoren zum Generalisieren oder Spezialisieren von Konzepten und deren Umkehroperatoren vorgestellt.

6.2.1 Move Attribute to Super Type

Auswirkung

Der Operator verschiebt ähnliche Attribute von eventuell mehreren Spezialisierung zu einer Generalisierung. Bidirektionale Beziehungen werden bei der Verschiebung entsprechend angepasst.

Ablauf

Attribute zum Verschieben festlegen

Der Operator wird an einem Konzept **Sub** aufgerufen, von dem im ersten Schritt die Generalisierung **Super** bestimmt wird. Anschließend werden alle abgeleiteten Spezialisierungen von **Super** (**Subs**) ermittelt und diejenigen Konzepte aus dieser Menge festgelegt, die in die Berechnung der ähnlichen Attribute eingehen sollen. Die Menge der ausgewählten Spezialisierungen soll im Folgenden **SubTypes** heißen. Als nächstes bestimmt der Operator von allen Elementen aus **SubTyps** die Attribute, die zu **Super** verschoben werden können. Dies sind alle Attribute, die sich bei jedem Konzept aus

`SubTyps` nur in Multiplizität und Sichtbarkeit grundlegend unterscheiden. Konkret heißt dies, dass ein Attribut genau dann zu einem anderen ähnlich[47] ist, wenn:

- Name und Deep Instantiation Zähler übereinstimmen,
- der Attributtyp gleich oder eine abgeleitete Spezialisierung bzw. abgeleitete Generalisierung des anderen ist,
- die gegenüberliegenden Attribute (`oppositeOf`) bis auf Name, Typ, Multiplizität, Sichtbarkeit und `oppositeOf` Referenz übereinstimmen und am gleichem Konzept definiert sind,
- und bei der `materializesAs` Eigenschaft die Attribut-Definition auf dasselbe Attribut zeigt.

Aus der Menge dieser Kandidaten wird anschließend die Menge an Attributen gewählt, die `Super` erhalten soll.

Attribute verschieben

Im letzten Abschnitt wird zunächst für jedes gewählte Attribut ein entsprechendes bei `Super` erzeugt, wobei die Kardinalität den größten und das Minimum der Multiplizität folglich den kleinsten aller angenommenen Werte bei den gewählten Attributen darstellt. Wenn also beispielsweise ein Konzept `A` ein Attribut `similar` mit Multiplizität `0..1` und ein Konzept `B` ein Attribut `similar` mit Multiplizität `1..*` definiert, wobei beide den gleichen Typ im Sinne der obigen Definition haben, wird für das Attribut der Generalisierung `Super` die Multiplizität `0..*` gewählt. Bei der Sichtbarkeit verhält es sich ähnlich. Auch hier wird die größte aller angenommenen Sichtbarkeiten gewählt. Falls also ein Attribut der Spezialisierungen die Sichtbarkeit `public` besitzt, bekommt auch die Generalisierung ein entsprechendes Attribut. Ein ähnliches Vorgehen wird bei der `materializesAs` Eigenschaft angewandt. Da die Attribut-Definitionen (siehe oben) übereinstimmen, wird die Multiplizität und Sichtbarkeit analog festgelegt. Für den Fall, dass bei zwei ähnlichen Attributen der jeweilige Typ eine abgeleitete Spezialisierung oder Generalisierung des anderen ist, stellt die abgeleitete Generalisierung den neuen Typ des Attributs bei `Super` dar.

Anschließend werden alle nun nicht mehr benötigten Attribute bei den Elementen von `SubTypes` entfernt und alle Referenzen (`oppositeOf`, `enables`, `attributeOf`) entsprechend auf das vorher erstellte Attribut (`newAttr`) von `Super` umgebogen. Wenn die Multiplizität von `newAttr` `1` oder `1..*` beträgt und damit das Attribut obligatorisch ist, muss bei allen Instanzen von `Subs` überprüft werden, ob `newAttr` gesetzt wurde. Dies kann beispielsweise dann auftreten, wenn eine Spezialisierung von `Super` nicht zur Berechnung der gemeinsamen Attribute verwendet wurde und damit bei dessen Instanzen `newAttr` vorher auch nicht gesetzt wurde. Deshalb werden zunächst der Deep Instantiation Zähler `n(newAttr)` ermittelt und anschließend alle Instanzen bis zur `n(newAttr)`-ten Ordnung von `Subs` bestimmt. Wenn eine Instanz ohne Zuweisung ist, wird eine neue Zuweisung bei der Instanz oder ein Standardwert erstellt und der Wert für das Attribut durch den *Set Value* Operator festgelegt.

Für den Fall, dass ein Attribut Bestandteil einer bidirektionalen Beziehung ist müssen die gegenüberliegenden Attribute zu einem (`oppoAttr`) verschmolzen werden. Dazu werden diese zunächst bestimmt. Danach muss ein neuer Name für `oppoAttr` gewählt werden, während die

[47] Die Ähnlichkeit zwischen zwei Attributen ist eine Äquivalenzrelation, da sie reflexiv (ein Attribut ist zu sich selbst ähnlich), symmetrisch (wenn Attribut `A` zu Attribut `B` ähnlich, dann ist `B` auch zu `A` ähnlich) und transitiv ist (Attribut `A` ähnlich zu Attribut `B` und `B` ähnlich zu Attribut `C`, dann ist `A` ähnlich zu `C`). Damit ist die Menge aller ähnlichen Attribute eine Äquivalenzklasse.

Festlegung der Multiplizität und Sichtbarkeit den oben erklärten Prinzipien folgt. Der Typ des Attributes wird auf `Super` gesetzt und die `oppositeOf` Eigenschaft bei beiden Attributen der bidirektionalen Beziehung entsprechend auf die Werte `newAttr` bzw. `oppoAttr` festgelegt. Die Referenzen (inklusive der Zuweisungen) auf die alten Attribute, die nun zu `oppoAttr` verschmolzen wurden, werden dann auf `oppoAttr` umgebogen und der Operator stoppt.

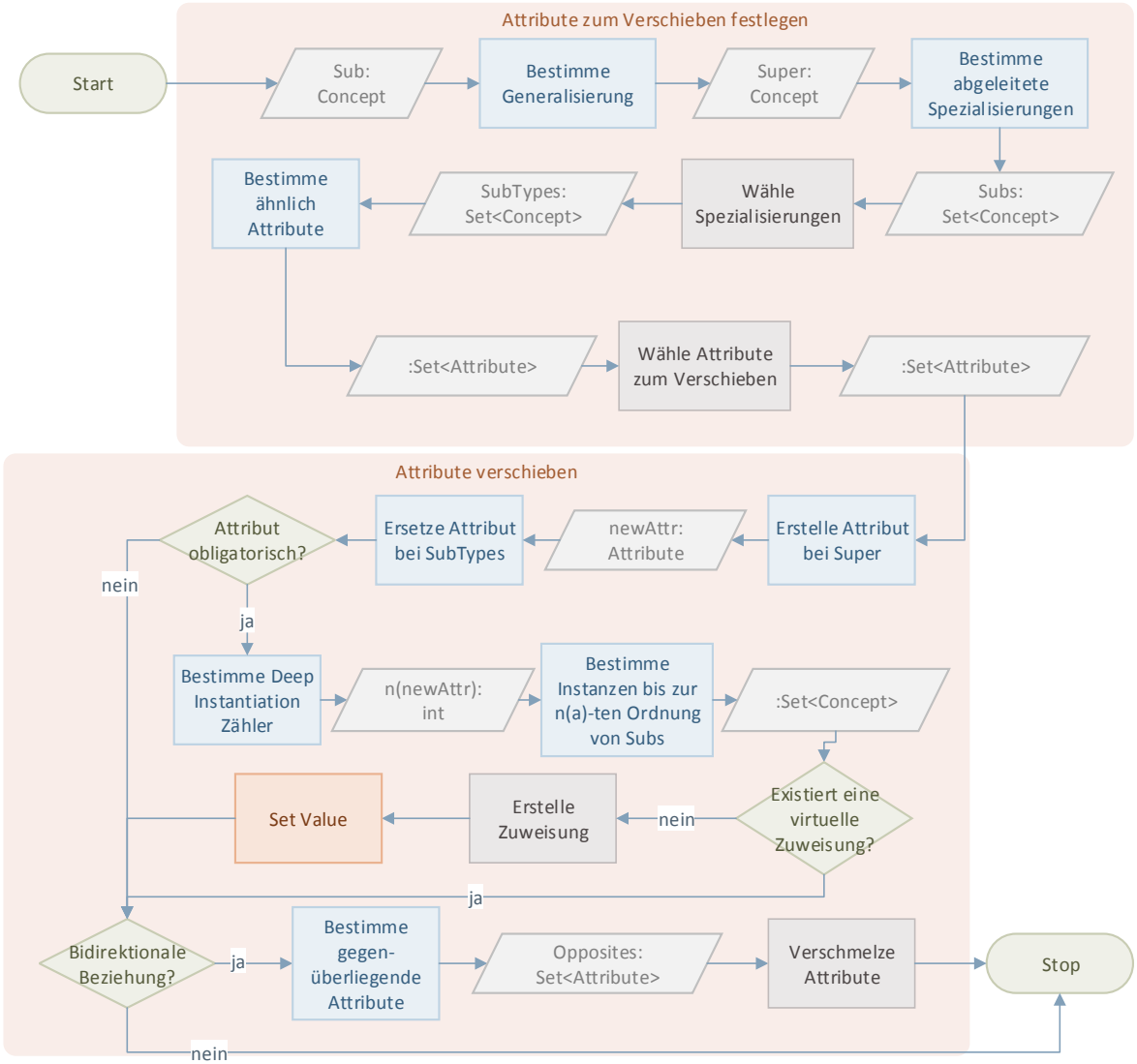

Abbildung 6-6 Ablauf des Move Attribute to Super Type Operators

Beispiel

Ein Beispiel zur Anwendung des Operators findet sich im Kapitel 6.2.3 im Beispiel des *Extract Super Type* Operators.

6.2.2 Move Attribute to Sub Type

Auswirkung

Der Operator verschiebt ein Attribut einer Generalisierung hin zu eventuellen mehreren Spezialisierungen und passt dabei Instanziierungen an oder löscht invalide Zuweisungen an das verschobene Attribut. Daneben wird im Falle einer bidirektionalen Beziehung, die durch das Attribut

entstehen kann, am gegenüberliegenden Konzept ein entsprechendes Attribut für jede Spezialisierung erzeugt.

Ablauf

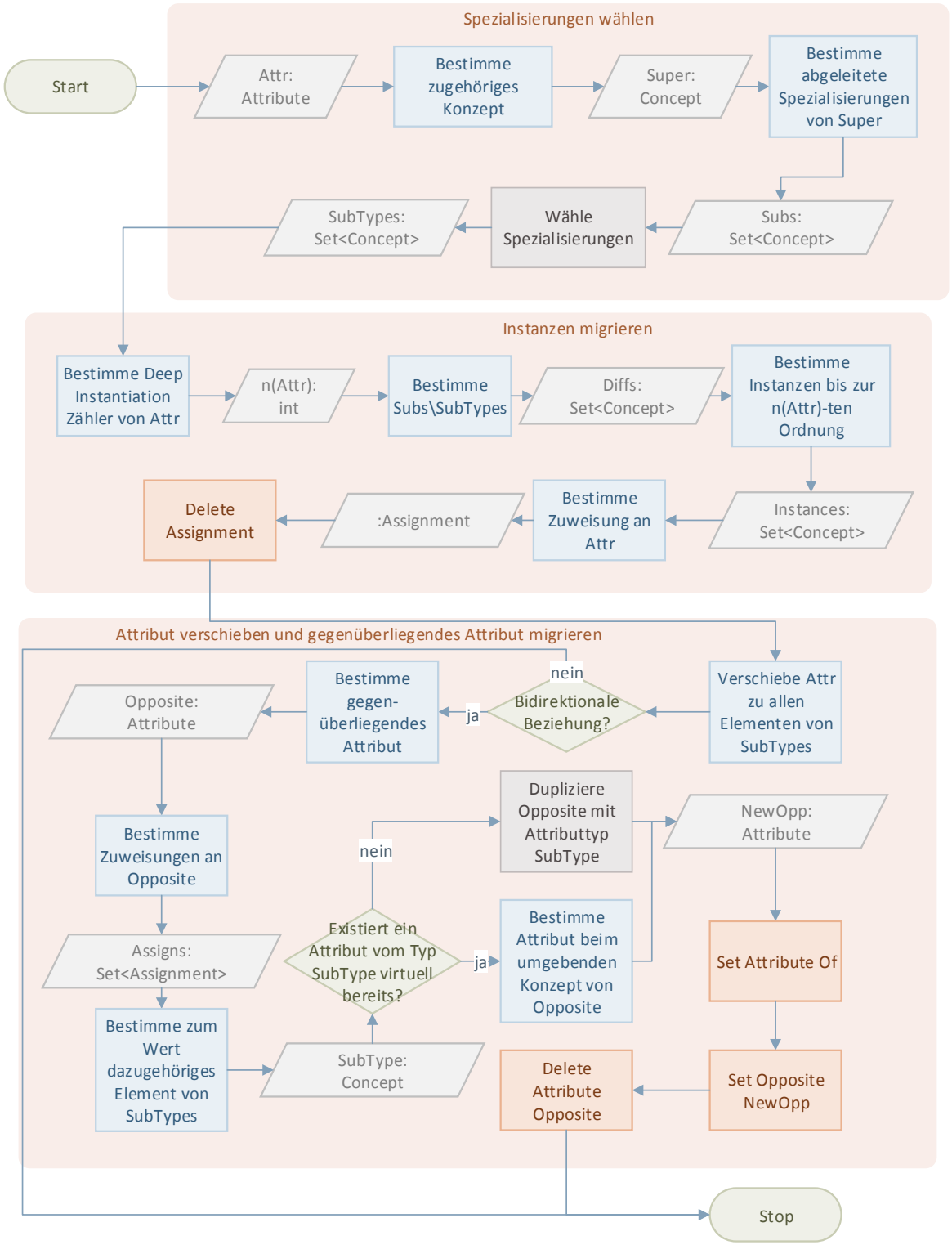

Abbildung 6-7 Ablauf des Move Attribute to Sub Type Operators

Spezialisierungen wählen

Abbildung 6-7 zeigt den Ablauf des *Move Attribute to Sub Type* Operators. Der Operator wird zunächst an einem Attribut `Attr` aufgerufen, das von dem generalisierten Konzept `Super`, welches im ersten Schritt bestimmt wird, zu einer oder mehrerer Spezialisierung verschoben werden soll. Nachdem `Super` ermittelt wurde, werden alle abgeleiteten Spezialisierungen von `Super` berechnet. Im nächsten Schritt wird die gewünschte Teilmenge dieser Konzepte ausgewählt, zu der das Attribut verschoben werden soll.

Instanzen migrieren

Als nächstes wird dann der Deep Instantiation Zähler `n(Attr)` von `Attr` zusammen mit denjenigen abgeleiteten Spezialisierungen (`Diffs`) bestimmt, deren Instanzen `Attr` nicht mehr setzen dürfen. Dies sind alle Elemente aus `Subs`, die nicht in `SubTypes` liegen. Von diesen Konzepten werden alle Instanzen bis zur `n(Attr)`-ten Ordnung berechnet. Danach ermittelt der Operator alle Zuweisungen an `Attr` von diesen Instanzen und entfernt diese mit Hilfe des *Delete Assignment* Operators.

Attribut verschieben und Gegenüberliegendes Attribut migrieren

Anschließend wird `Attr` zu den gewählten Spezialisierungen `SubTypes` verschoben. Dazu muss `Attr` dupliziert werden, falls `SubTypes` mehr als ein Element enthält. Wenn `Attr` Teil einer bidirektionalen Beziehung ist, muss auch das gegenüberliegende Attribut entsprechend angepasst werden. Dazu wird zunächst dieses Attribut (`Opposite`) und anschließend alle Zuweisungen an es bestimmt. Danach wird zu jedem Wert[48] der Zuweisung die entsprechende Instanz von einem Element (`SubType`) aus `SubTypes` ermittelt. Als nächstes wird `Opposite` bei seinem umgebenden Konzept dupliziert, falls ein solches Attribut nicht bereits existiert. Dabei muss ein neuer Name festgelegt werden. Der Typ des Attributes wird bei der Erstellung aufgrund der neuen bidirektionalen Beziehung auf `SubType` gesetzt. Nachdem das neue gegenüberliegende Attribut (`NewOpp`) erstellt bzw. ermittelt wurde, wird für die entsprechende Zuweisung das neue Attribut `NewOpp` unter Verwendung des *Set Attribute Of* Operators gesetzt. Nachdem dies für alle Zuweisungen an `Opposite` geschehen ist, wird für jedes Attribut `NewOpp` das neue gegenüberliegende Attribut `Attr` am Attributtyp von `NewOpp` durch den *Set Opposite* Operator gesetzt. Anschließend kann `Opposite` im nächsten Schritt unter Verwendung des *Delete Attribute* Operators gelöscht werden und der Operator ist beendet.

Beispiel

Die Anwendung des Operators wird im Beispiel des *Extract Sub Type* Operators gezeigt.

6.2.3 Extract Super Type

Auswirkung

Der Operator erstellt für eine Menge an Konzepten eine neue Generalisierung und verschiebt ausgewählte Attribute. Falls das übergebene Konzept bereits eine Generalisierung besitzt, wird der neue Obertyp zwischen beiden Konzepten in der Vererbungshierarchie erstellt.

[48] Durch die Aktionen im Operator die vorher stattgefunden haben, muss an dieser Stelle jede Zuweisung einen Wert haben, der eine abgeleitete Instanz eines Elementes von `SubTypes` ist. Alle anderen Zuweisungen wurden angepasst oder gelöscht.

Ablauf

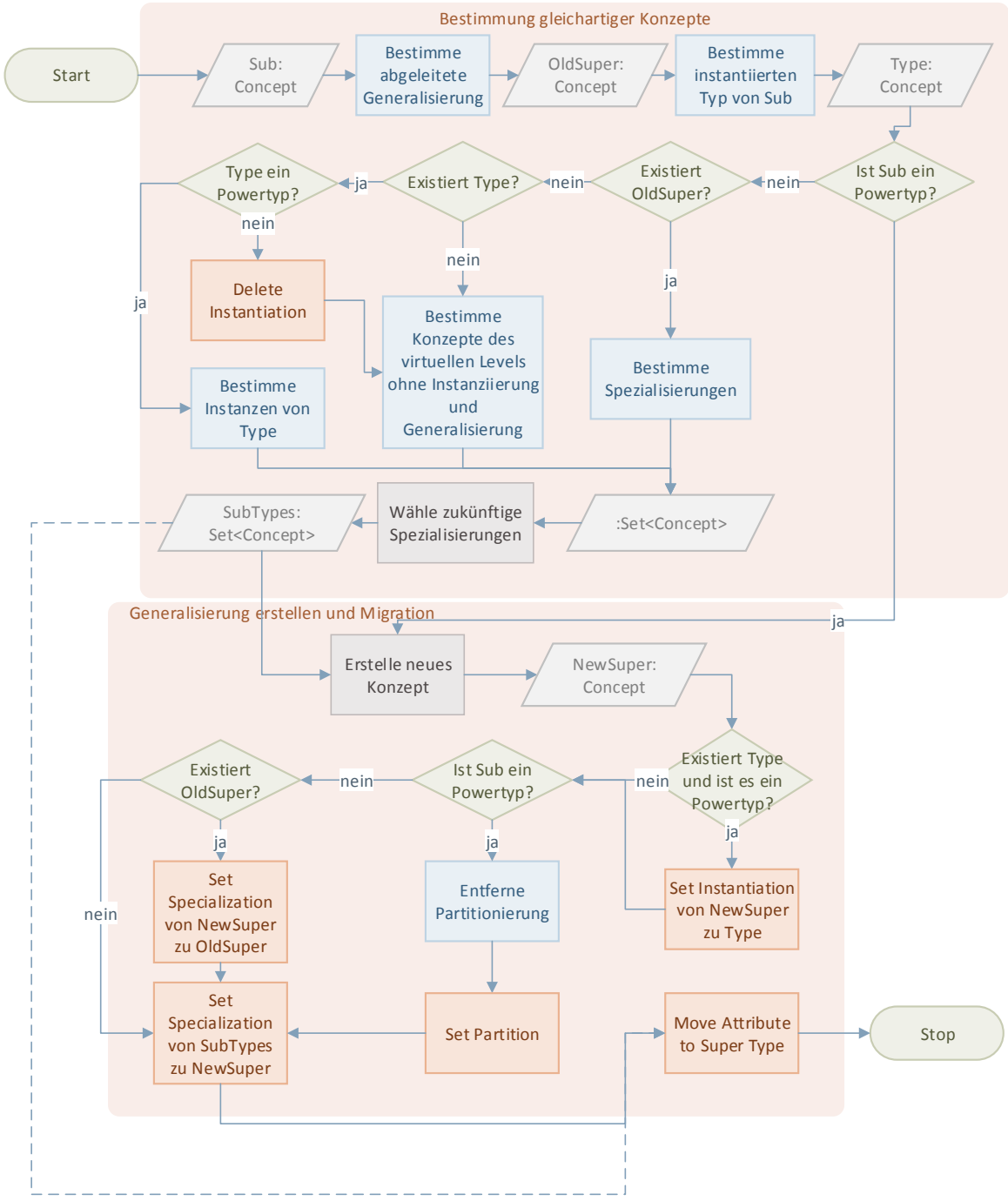

Abbildung 6-8 Ablauf des Extract Super Type Operators

Bestimmung gleichartiger Konzepte

Initial wird dem Operator ein Konzept **`Sub`** übergeben. Danach werden sowohl die bisherige Generalisierung **`OldSuper`** als auch der instanziierte Typ **`Type`** bestimmt[49]. Als nächstes muss unterschieden werden, ob **`Sub`** ein Powertyp ist. In diesem Fall kann die Auswahl anderer Konzept, die ebenfalls eine Spezialisierung zum neuen Konzept erhalten sollen, übersprungen werden, da jeder

[49] Wegen Regel C.8 bzw. Regel C.9 existiert entweder **`Type`** oder **`OldSuper`**.

partitionierte Typ maximal einen Powertyp hat (Regel C.3) und damit kein anderes Konzept außer `Sub` als Spezialisierung in Frage kommen kann. Falls `Sub` eine Generalisierung besitzt, bilden alle Spezialisierungen von `OldSuper` die Menge an möglichen Spezialisierungen der neuen Generalisierung. Falls nicht, muss untersucht werden, ob `Sub` ein Konzept `Type` instanziiert. Falls `Type` ein Powertyp ist, dann berechnet sich die Menge an potentiellen Spezialisierungen aus den Instanzen von `Type`. Wenn `Type` kein Powertyp ist, wird die Instanziierung mit Hilfe des *Delete Instantiation* Operators entfernt. Danach und wenn `Sub` keine Instanziierung besitzt, wird die Menge an Kandidaten aus allen Konzepten der virtuellen Ebenen, die ebenfalls weder eine Instanziierung noch eine Generalisierung besitzen (und auch kein Powertyp sind), gebildet. Aus allen potentiellen Kandidaten wird im nächsten Schritt eine Teilmenge `SubTypes` gewählt, die die späteren Spezialisierungen des neuen Typs (`NewSuper`) enthält.

Generalisierung erstellen und Migration

Im darauffolgenden Schritt wird die neue Generalisierung `NewSuper` erstellt, wobei an dieser Stelle der Name und, ob `NewSuper` abstrakt sein soll, festgelegt werden muss. Für den Fall, dass `Sub` eine Powertyp-Instanz ist (und damit `Type` existiert), wird der instanziierte Typ `Type` durch den *Set Instantiation* Operator bei `NewSuper` gesetzt. Daran anschließend wird untersucht, ob `Sub` ein Powertyp ist. Wenn dem so ist, wird die Partitionierung elementar entfernt, damit im nächsten Schritt durch den *Set Partition* Operator die Partitionierung von `NewSuper` zum partitionierten Typ von `Sub` (vorher ermittelt) gesetzt werden kann. Falls `Sub` kein Powertyp ist, dafür aber `OldSuper` existiert, wird der *Set Specialization* Operator dazu verwendet, um die eine Spezialisierung von `NewSuper` zu `OldSuper` zu erzeugen.

Danach wird die Spezialisierung von allen Elementen aus `SubTypes` und `Sub` zu `NewSuper` durch den *Set Specialization* Operator gesetzt bzw. ersetzt (falls `OldSuper` existiert). Im letzten Schritt wird der *Move Attribute to Super Type* Operator an einer Spezialisierung aus `SubTypes` aufgerufen, wobei alle anderen Elemente der Menge als Spezialisierungen, die in der Berechnung der abhängigen Attribute verwendet werden sollen, im Operator ausgewählt werden. Danach endet die Ausführung des Operators.

Beispiel

In Abbildung 6-9 ist auf der linken Seite ein kleines Modell gezeigt, auf das hier beispielhaft der *Extract Super Type* Operator angewendet werden soll. Darin wurde ein Konzept `Employee` erstellt, das eine Generalisierung der Konzepte `Office`, `Technician` und `Cleaning` darstellt und einen Mitarbeiter in einem fiktiven Unternehmen modelliert. Die Spezialisierungen bilden Büromitarbeiter, technische Angestellte und Reinigungspersonal ab. Büromitarbeiter und technische Angestellte haben ein Gehalt (jeweils das Attribut `salary`) und arbeiten (Beziehung mit dem Attribut `worksFor`) für eine Abteilung (`Department`). Bei der Abteilung wiederrum werden die Büromitarbeiter (Attribut `officeStaff`) und die technischen Angestellten (Attribut `technicianStaff`) abgespeichert. Beide dafür deklarierten Attribute sind die gegenüberliegenden Attribute von `worksFor`, das sowohl bei `Office` als auch `Technician` definiert wurde. Zudem gibt es noch eine weitere bidirektionale Beziehung von `Office` zu `Department` (Attribute `managedDep` bzw. `manager`), die ausdrückt, dass ein Büromitarbeiter eine Abteilung leiten kann.

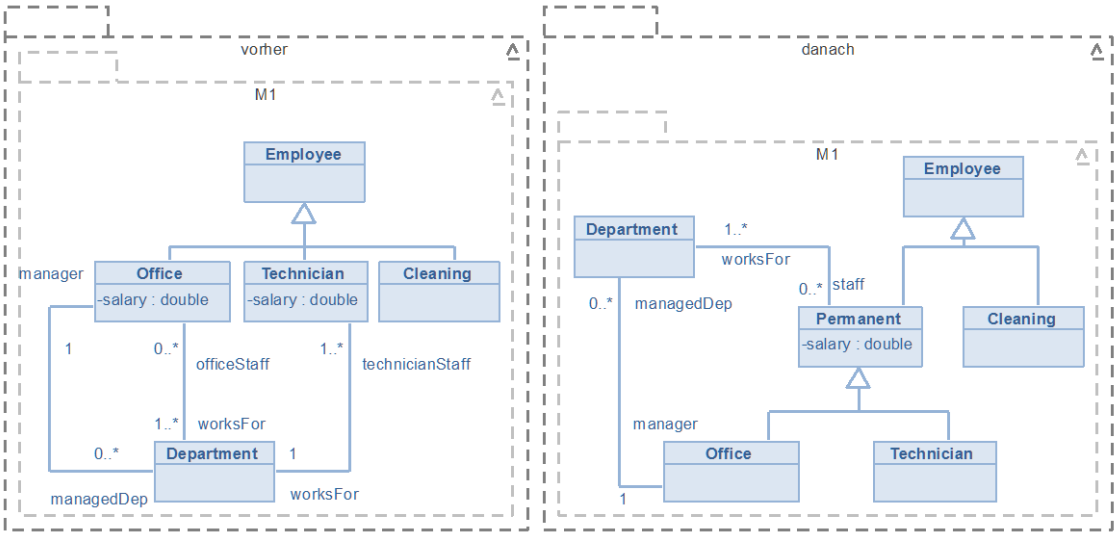

Abbildung 6-9 Beispiel vor (links) und nach (rechts) der Ausführung des Extract Super Type Operators

Nun soll der *Extract Super Type* Operator auf das Konzept `Office` angewendet werden, um ein neues Konzept `Permanent`, welches alle Festangestellten darstellt, zwischen `Office` und `Employee` in die Vererbungshierarchie einzufügen. Deshalb wird zunächst die Generalisierung `Employee` bestimmt. Da `Office` keinen instanziierten Typ besitzt und auch kein Powertyp ist, werden als nächstes alle Spezialisierungen von `Employee` ermittelt. Dies sind in diesem Fall neben `Office` auch noch `Technician` und `Cleaning`. Da das Reinigungspersonal keine festangestellten Mitarbeiter in unserem Beispiel darstellen, werden lediglich `Office` und `Technician` im nächsten Schritt gewählt. Nun wird das neue Konzept erstellt und der Name auf `Permanent` gesetzt. Als nächstes wird, da es keine Instanziierung von `Office` gibt, die Generalisierung von `Permanent` zu `Employee` durch den *Set Specialization* Operator gesetzt. Anschließend wird dieser Operator ebenfalls für `Office` und `Technician` aufgerufen, wobei `Permanent` als neue Generalisierung gewählt wird. Da `Permanent` eine abgeleitete Spezialisierung der bisherigen Generalisierung `Employee` ist und vorher alle obligatorischen Attribute gesetzt waren, führt der Operator hauptsächlich die elementare Änderung aus. Danach wird der *Move Attribute to Super Type* Operator an `Permanent` aufgerufen und dabei `Office` und `Technician` als Spezialisierungen, die in die Berechnung der ähnlichen Attribute eingehen, gewählt. Die Berechnung liefert in diesem Fall `worksFor` und `salary`, da sie sich bei `Office` und `Technician` entweder gar nicht oder nur in der Multiplizität bzw. den gegenüberliegenden Attributen (und da auch nicht außerhalb der Bedingungen) unterscheiden. Das Attribut `managedDep` ist dagegen schon alleine wegen des anderen Namens (zu `worksFor`) nicht in den möglichen Kandidaten enthalten.

Folglich werden beide Attribut zur Deklaration bei `Permanent` gewählt. Das Attribut `salary` kann, da es bei beiden Spezialisierungen gleich ist, entsprechend bei `Permanent` definiert und die jeweiligen Referenzen umgebogen werden. Bei `worksFor` müssen jedoch Multiplizität und die gegenüberliegenden Attribute angepasst werden. Da ein Büromitarbeiter mehreren Abteilungen zugeordnet werden kann, wird die Multiplizität für `worksFor` bei `Permanent` auf `1..*` gesetzt. Aufgrund der bidirektionalen Beziehung werden die Attribute `officeStaff` und `technicianStaff` zu einem neuen Attribut (festgelegter Name soll `staff` sein) verschmolzen. Der Typ von `staff` ist folglich `Permanent`, während die Multiplizität aufgrund von `officeStaff` `0..*` ist. Damit erhält man nach der Ausführung des Operators das in Abbildung 6-9 auf der rechten Seite gezeigte Modell.

6.2.4 Extract Sub Type

Auswirkung

Der Operator erstellt eine neue Spezialisierung für ein vorgegebenes Konzept und verschiebt eine Auswahl an Attributen von der Generalisierung hin zur neuen Spezialisierung.

Ablauf

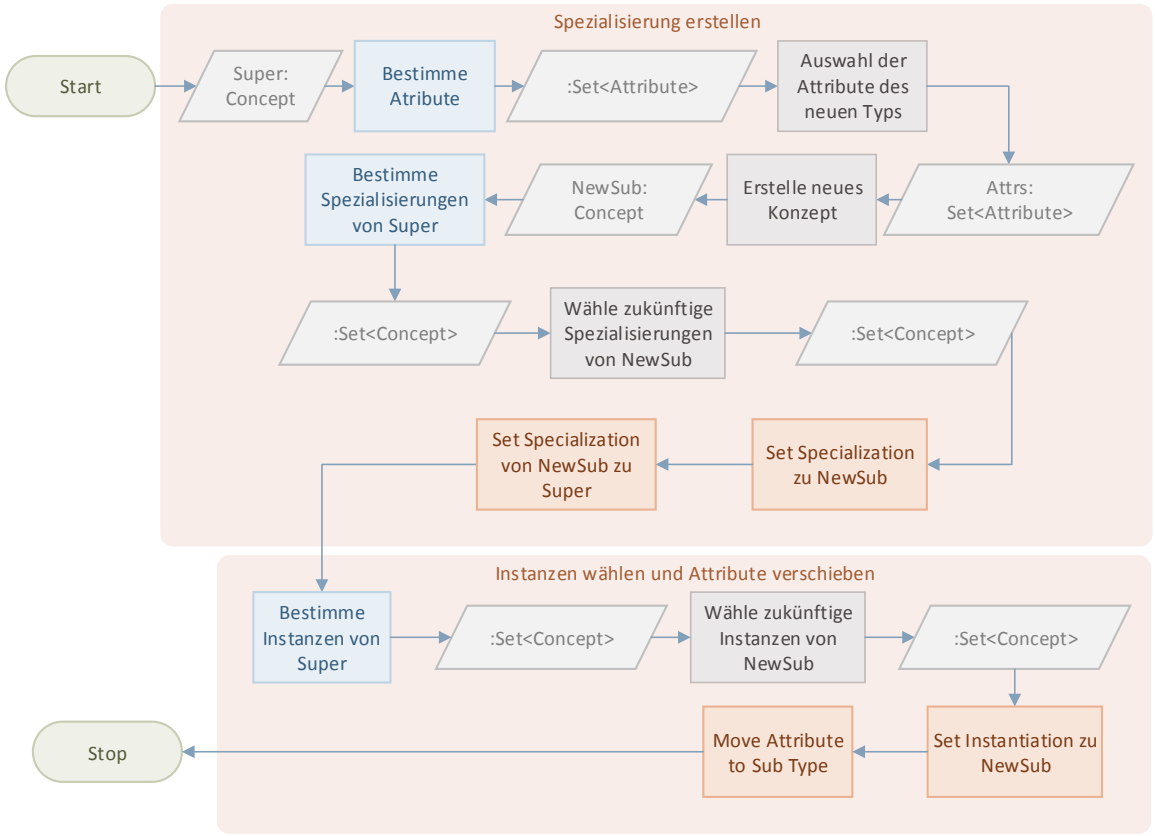

Abbildung 6-10 Ablauf des Extract Sub Type Operators

Spezialisierung erstellen

Der Ablauf des Operators ist in Abbildung 6-10 dargestellt. Zunächst wird zu Beginn ein Konzept **`Super`** übergeben, von dem eine Spezialisierung erstellt werden soll. Dazu werden im ersten Schritt alle Attribute von **`Super`** ermittelt. Aus diesen werden dann vom Benutzer die zu extrahierenden Attribute gewählt. Anschließend wird das neue Konzept **`NewSub`** erstellt und dabei Eigenschafen wie der Name, ob es abstrakt oder final ist, usw. festgelegt. Anschließend bestimmt der Operator alle Spezialisierungen von **`Super`**. Aus diesen wird dann ausgewählt, welche von ihnen **`NewSub`** als Generalisierung besitzen sollen. Wenn **`Sub`** als final gewählt wurde, können an dieser Stelle keine Spezialisierungen gewählt werden. Die Spezialisierung zu **`NewSub`** wird im nächsten Schritt durch den *Set Specialization* Operator für alle gewählten Spezialisierungen hergestellt. Danach wird der Operator erneut aufgerufen, um **`Super`** als neue Generalisierung von **`NewSub`** festzulegen.

Instanzen wählen und Attribute verschieben

Als nächstes werden alle Instanzen von **`Super`** ermittelt und diejenigen gewählt, die eine Instanz von **`NewSub`** werden sollen. Anschließend wird die Instanziierung von diesen Konzepten zu **`Super`** durch eine Instanziierung zu **`NewSub`** ersetzt. Dies wird durch den Aufruf des *Set Instantiation* Operators an der jeweiligen Instanz erreicht, wobei **`NewSuper`** als neuer Instanziierter Typ im Operator gewählt wird.

Im letzten Schritt werden noch alle anfangs ausgewählten Attribute mittels des *Move Attribute to Sub Type* Operators zur neuen Spezialisierung verschoben. Dabei wird die neu erstellte Spezialisierung `NewSub` als Ziel der Verschiebung gewählt und der Operator endet.

Beispiel

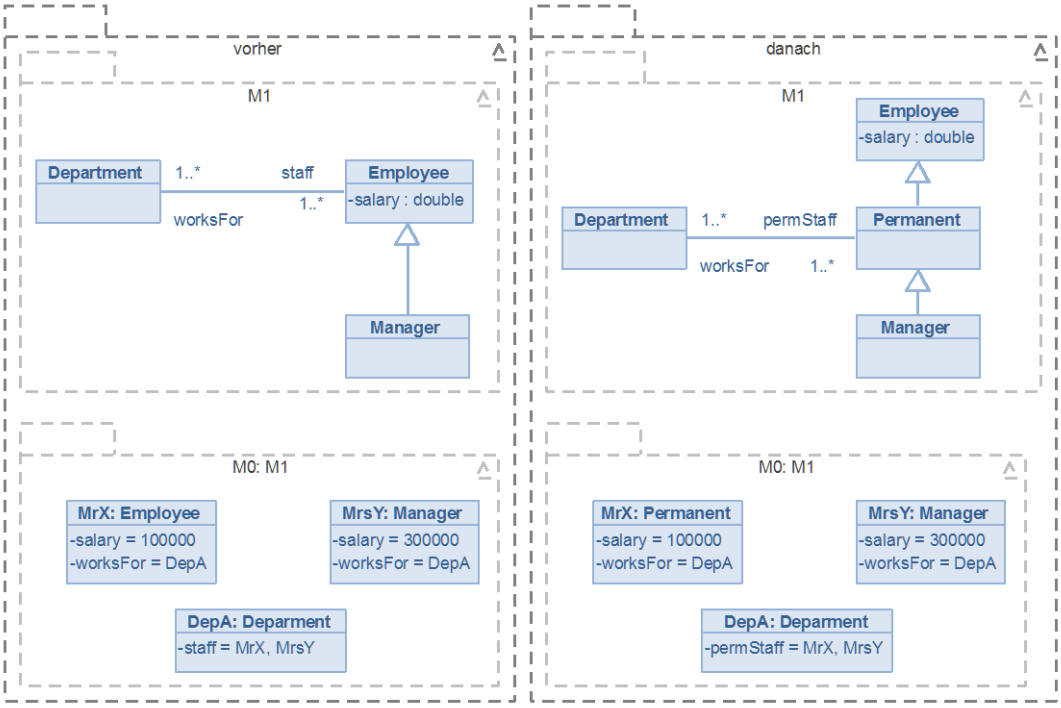

Abbildung 6-11 Beispiel vor (links) und nach (rechts) der Anwendung des Extract Sub Type Operators

Abbildung 6-11 zeigt auf der linken Seite ein einfaches Meta-Modell, auf das der Operator angewendet werden soll. Auf der Meta-Ebene `M1` sind drei Konzepte definiert worden: `Department`, das eine Abteilung in einem Unternehmen abbildet, `Employee`, das einen Mitarbeiter im Unternehmen modelliert und `Manager`, das eine Spezialisierung von `Employee` ist und einen Manager darstellt. Jeder Mitarbeiter besitzt ein Gehalt, was durch das Attribut `salary` am Konzept `Employee` ausgedrückt wird. Weiterhin ist jeder Abteilung eine Gruppe von Mitarbeitern zugeordnet und jeder Mitarbeiter arbeitet für eine oder mehrere Abteilungen. Die bidirektionale Beziehung, die durch die beiden Attribute `worksFor` bei `Employee` und `staff` bei `Department` gebildet wird, spiegelt diesen Sachverhalt wieder. Die Ebene `M0` beinhaltet ein Modell, das beschreibt, dass Manager `MrsY` und Mitarbeiter `MrX` für die Abteilung `DepA` arbeiten. Zudem wurde bei beiden Mitarbeitern, jeweils durch die Zuweisung an `salary`, festgehalten, wie hoch ihr Gehalt ist.

In diesem Bespiel soll nun eine Spezialisierung eingeführt werden, die in der Vererbungshierarchie zwischen `Employee` und `Manager` liegen soll. Die neue Spezialisierung soll dazu dienen, Festangestellte im Unternehmen zu modellieren, da lediglich diese einer Abteilung (Instanz von `Department`) zugeordnet werden sollen. Um die gewünschte Evolution durchzuführen, wird daher der *Extract Sub Type* Operator am Konzept `Employee` aufgerufen. Als nächstes werden alle Attribute von `Employee` bestimmt (`salary`, `worksFor`). Aus diesen wählen wir `worksFor` aus, damit es in die neu erstellte Spezialisierung verschoben wird. Danach legen wir den Namen für das neue Konzept fest (`Permanent`) und der Operator erstellt es daraufhin. Im nächsten Schritt werden alle Spezialisierungen von `Employee`, in diesem Fall also `Manager`, ermittelt. Da jeder Manager ein festangestellter Mitarbeiter sein soll, wird dieser auch als zukünftige Spezialisierung von `Permanent` gewählt. Daran

anschließend wird `Permanent` als Generalisierung von `Manager` und danach `Employee` als Generalisierung von `Permanent` gesetzt und damit die Vererbungshierarchie hergestellt.

Im nächsten Schritt ermittelt der Operator alle Instanzen von `Employee` (`MrX`) und wir entscheiden uns dafür, `MrX` festanzustellen und ihn als Instanz von `Permanent` zu wählen. Anschließend wird diese Instanziierung hergestellt, bevor der *Move Attribute to Sub Type* Operator an `worksFor` aufgerufen wird. Dies führt dazu, dass zunächst `Employee` ermittelt wird. Danach wird durch den *Extract Sub Type* Operator `Permanent` als abgeleitete Spezialisierung, die `worksFor` erhalten soll, gesetzt. Da der Deep Instantiation Zähler von `worksFor` `1` ist und `Employee` keine weiteren abgeleiteten Instanzen besitzt, werden alle Instanzen bis zur 1-ten Ordnung von `Employee` ermittelt. Aufgrund der Tatsache, dass `Employee` keine Instanzen mehr besitzt und auch selbst keine Zuweisung an `worksFor` definiert, wird als nächstes `worksFor` zu `Permanent` verschoben. Nun stellt der Operator fest, dass `worksFor` Teil einer bidirektionalen Beziehung ist. Deshalb wird zunächst das gegenüberliegende Attribut `staff` ermittelt und danach die Zuweisung an `staff` bei `DepA` untersucht. Weil (nun) `MrX` und `MrsY` Instanzen von `Permanent` sind, wird ein Attribut bei `Department` vom Typ `Permanent` angelegt, welches wir `permStaff` nennen. Als nächstes wird für die Zuweisung an `staff` bei `DepA` `permStaff` als neues Attribut der Zuweisung (*Set Attribute Of*) und anschließend das gegenüberliegende Attribut von `worksFor` auf `permStaff` `(`*Set Opposite*`)` gesetzt. Im letzten Schritt des Operators wird `staff` durch den *Delete Attribute* Operator entfernt und sowohl der innere als auch der äußere Operator enden mit dem Ergebnis, das in der rechten Seite der Abbildung 6-11 gezeigt wird.

6.2.5 Inline Super Type

Auswirkung

Der Operator löst ein generalisiertes Konzept auf und verschiebt dessen Attribute in die entsprechenden Spezialisierungen.

Ablauf

Spezialisierung und Powertyp migrieren

Zu Beginn wird der Operator an einer Generalisierung `Super` aufgerufen. Falls diese nicht abstrakt ist, besteht die Möglichkeit, dass Instanzen existieren, die nach der Anwendung des Operators invalide wären. Deshalb wird in diesem Fall `Super` auf abstrakt gesetzt, wodurch die Instanziierung gelöscht oder zu einer Spezialisierung verschoben wird. Danach werden alle Spezialisierungen (`Subs`) und Attribute von `Super` ermittelt. Diese werden durch entsprechende Parametrierung des *Move Attribute to Sub Type* Operators zu allen Spezialisierungen (`Subs`) geschoben. Danach wird überprüft, ob es einen Powertyp `Power` zu `Super` gibt. Wenn dies der Fall ist, wird dann ein neuer partitionierter Typ aus `Subs` für `Power` gewählt und dieser anstelle von `Super` durch den *Set Partition* Operator gesetzt. Anschließend wird untersucht, ob eine Generalisierung (`SuperSuper`) von `Super` existiert. Falls dies zutrifft, wird `SuperSuper` als Generalisierung bei allen Elementen von `Subs` unter Verwendung des *Set Specialization* Operators festgelegt, womit die Vererbungshierarchie angepasst wird.

Partitionierten Typ migrieren

Falls `Super` ein Powertyp ist, muss eine Spezialisierung `NewPower` gewählt werden, die den neuen Powertyp[50] des partitionierten Typs (`Part`) von `Super` darstellt. Folglich wird der partitionierte Typ zunächst bestimmt und der *Set Partition* Operator für `NewPower` aufgerufen, wobei `Part` als neuer partitionierter Typ gewählt wird. Alle anderen Elemente aus `Subs` besitzen noch die Spezialisierung zu

[50] Nach Regel C.3 darf jedes Konzept maximal einen Powertypen besitzen.

Super, die deshalb als nächstes durch den *Delete Specialization* Operator entfernt wird. Für den Fall, dass **Super** kein Powertyp ist, wird dieser Operator für alle Elemente aus **Subs** aufgerufen. Danach muss das Konzept **Super** mittels des *Delete Concept* Operators gelöscht werden. Dies hat zur Folge, dass noch nicht migrierte Modellelemente wie zum Beispiel Instanz-Spezialisierungen ebenfalls noch entsprechend adaptiert werden. Danach ist die Ausführung des Operators beendet.

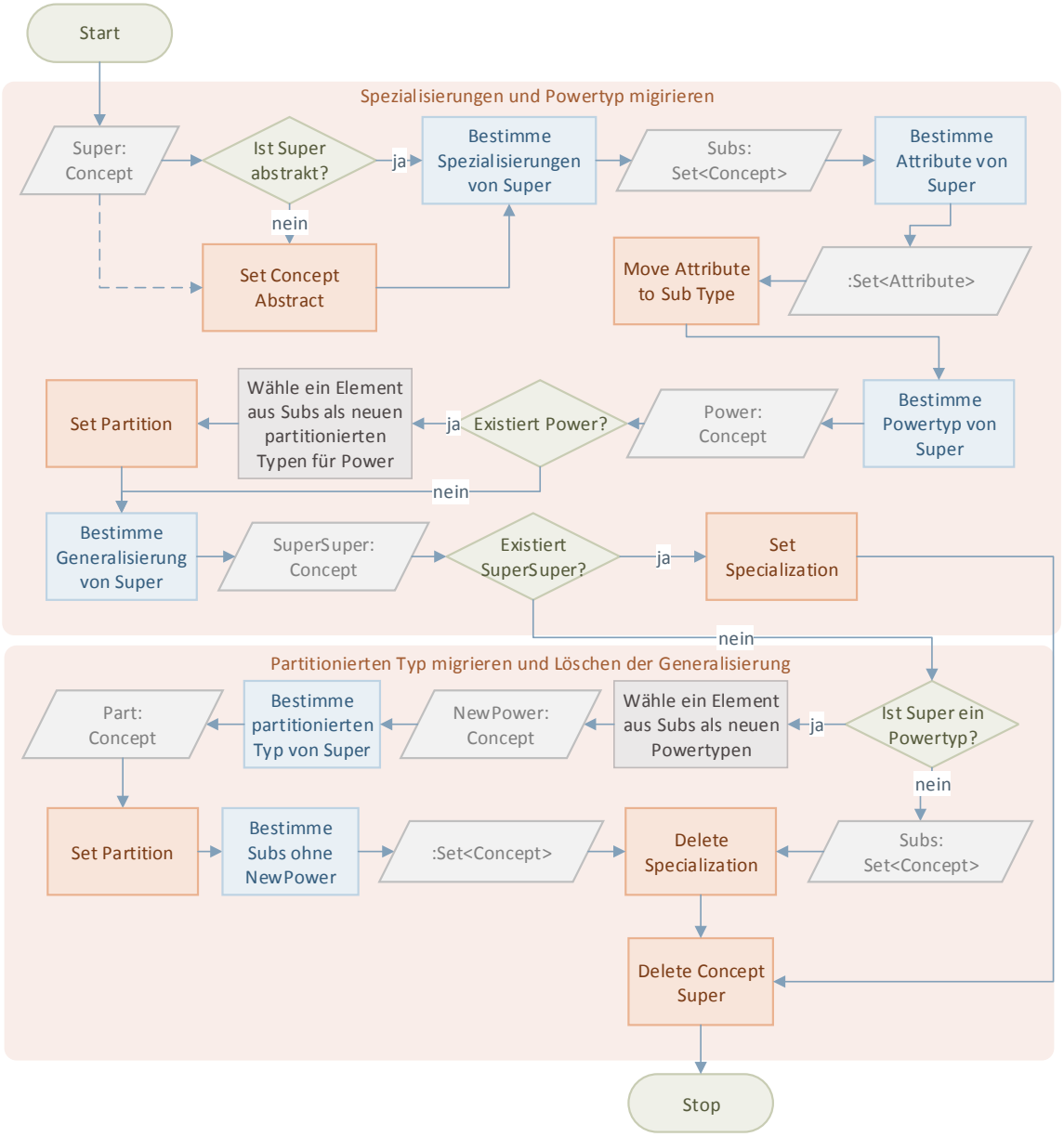

Abbildung 6-12 Ablauf des Inline Super Type Operators

Beispiel

In der Abbildung 6-13 ist auf der linken Seite ein Beispiel dargestellt, in dem verschiedene Produkte modelliert wurden. Auf der Ebene **M2** wurde dazu das Konzept **Product** erstellt, das ein Attribut **taxRate** besitzt und ein Produkt darstellt. **taxRate** dient dabei der Speicherung des jeweiligen Satzes für die Mehrwertsteuer. Auf der Ebene **M1** wurde ein Konzept **Car** erstellt, das ein Auto symbolisiert und eine Beziehung zu **Manufacturer** hat. Durch diese Beziehung und das Konzept **Manufacturer** wird dabei ausgedrückt, dass jedes Auto genau eine Produktionsstätte hat. Jede Produktionsstätte liegt in einer bestimmten Stadt, was durch das Attribut **city** bei **Manufacturer** abgebildet ist. Weiterhin

wurden zwei Typ- und Instanz-Spezialisierungen von `Car` modelliert: `Polo` und `Fiesta`, die sowohl das Attribut `manufacturer` als auch die Zuweisung an `taxRate` von `Car` erben. Auf der Ebene `M0` sind zwei Konzepte erstellt worden: `AliceCar` als Instanz von `Polo` und `VW` als Instanz von `Manufacturer`. Das Auto von Alice wurde dabei bei der VW Produktionsstätte in Wolfsburg gebaut, was durch die jeweiligen Zuweisungen bei `AliceCar` und `VW` modelliert wurde.

Nun soll der *Inline Super Type* Operator auf `Car` angewendet werden, um die Generalisierung aufzulösen. Folglich wir zunächst festgestellt, dass es sich bei `Car` nicht um ein abstraktes Konzept handelt und deshalb wird das Konzept anschließend auf abstrakt gesetzt. Da in diesem Beispiel keine Instanzen von `Car` existieren, tritt dabei keine weitere Änderung auf. Nachdem nun `Car` abstrakt ist, werden alle Spezialisierungen (`Polo` und `Fiesta`) und `manufacturer` als einziges Attribut von `Car` ermittelt Als nächstes wird `manufacturer` durch den *Move Attribute to Sub Type* zu `Fiesta` und `Polo` verschoben, womit die Zuweisung bei `AliceCar` auch valide bleibt. Aufgrund der Tatsache, dass `Car` keinen Powertyp sowie keine Generalisierung besitzt und selbst auch kein Powertyp ist, wird als nächstes die Spezialisierung von `Polo` und `Fiesta` zu `Car` unter Verwendung des *Delete Specialization* Operators entfernt. Im letzten Schritt wird `Car` mittels des *Delete Concept* Operator gelöscht, wodurch bei den beiden Instanz-Spezialisierungen `Fiesta` und `Polo` der *Delete Concrete Use Of* Operator aufgerufen wird. Dieser entfernt die Instanz-Spezialisierung, stellt die Instanziierung zu `Product` her und erstellt bei `Fiesta` und `Polo` die jeweilige Zuweisung an `taxRate`. In Abbildung 6-13 ist auf der rechten Seite das entstandene Modell dargestellt.

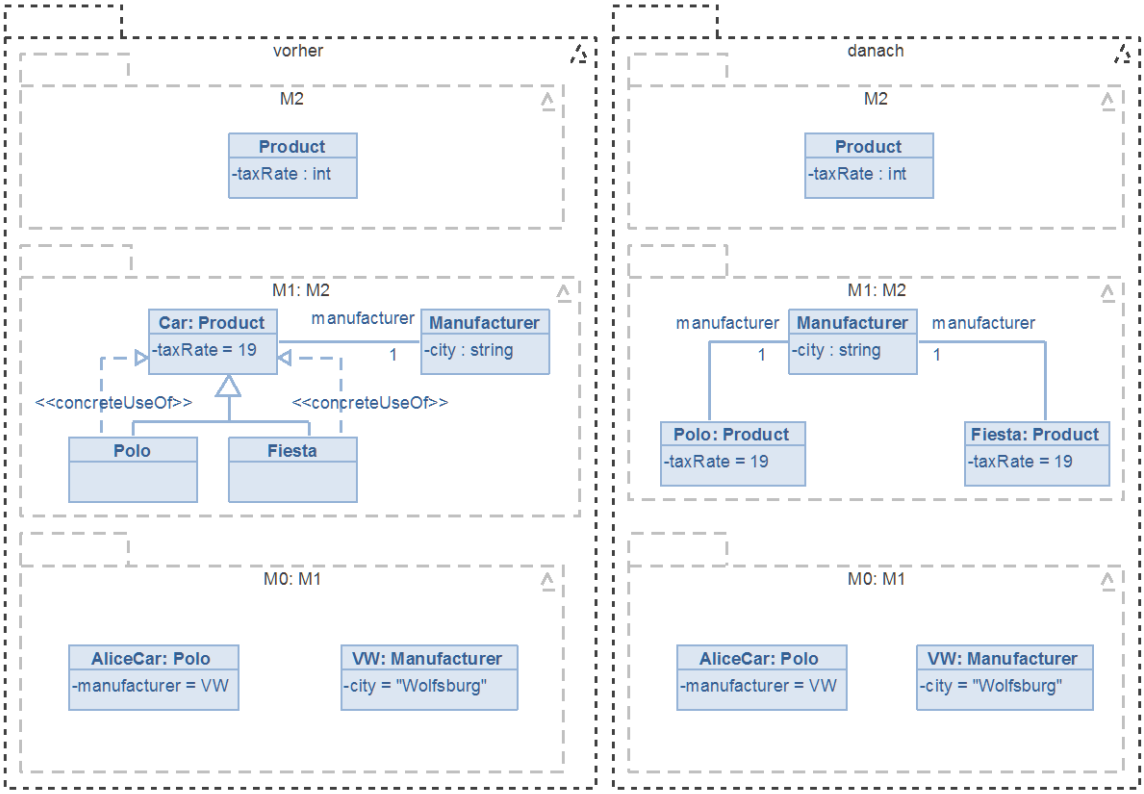

Abbildung 6-13 Beispiel vor (links) und nach (rechts) der Ausführung des Inline Super Type Operators

6.2.6 Inline Sub Type

Auswirkung

Mit Hilfe des Operators wird eine vorhandene Spezialisierung aufgelöst und die Attribute werden zum entsprechenden generalisierten Konzept verschoben.

Ablauf

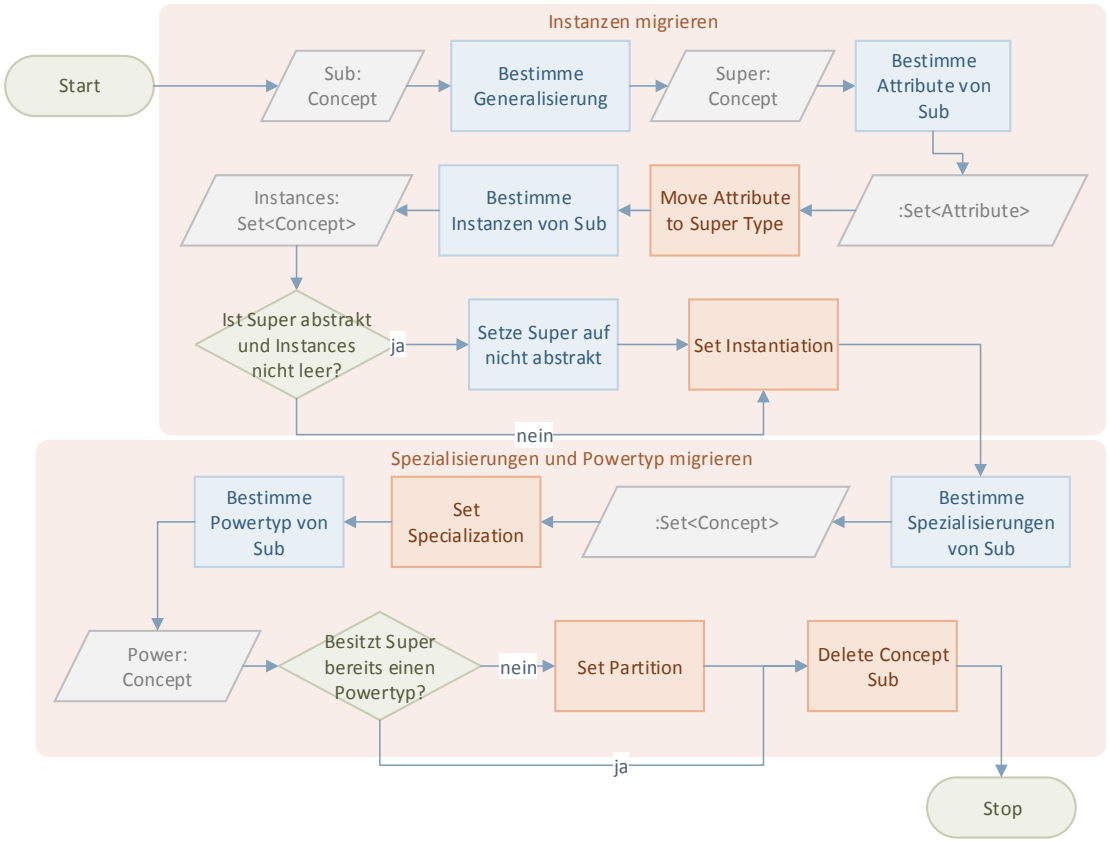

Abbildung 6-14 Ablauf des Inline Sub Type Operators

Instanzen migrieren

Initial wird der Operator an einem Konzept **Sub** aufgerufen, das eine Generalisierung **Super** haben muss. Diese wird im ersten Schritt bestimmt. Nachdem werden alle Attribute von **Sub** ermittelt und mittels des *Move Attribute to Super Type* Operators verschoben. Dabei werden alle Attribute von **Sub** und keine weitere Spezialisierung im aufgerufenen Operator in den entsprechenden Schritten ausgewählt. Danach werden alle Instanzen (**Instances**) von **Sub** ermittelt, damit anschießend die Instanziierung zu **Super** durch den *Set Instantiation* Operator gesetzt werden kann. Dies kann nur geschehen, wenn **Super** nicht abstrakt ist. Wenn dies dennoch der Fall ist und **Instances** nicht leer ist, muss zunächst **Super** auf nicht abstrakt gesetzt werden, bevor die Instanziierung auf **Super** geändert werden kann.

Spezialisierungen und Powertyp migrieren

Im nächsten Schritt werden die Spezialisierungen von **Sub** bestimmt, die danach durch den *Set Specialization* Operator **Super** als neue Generalisierung erhalten. Anschließend wird ein eventueller Powertyp **Power** von **Sub** gesucht. Falls dieser existiert und **Super** nicht bereits einen Powertyp besitzt,

wird **Super** als partitionierter Typ für **Power** gesetzt. Wenn **Super** bereits ein partitionierter Typ war[51], wird die Migration von **Power** im letzten Schritt vollzogen. Als letztes wird **Sub** mit Hilfe des *Delete Concept* Operators gelöscht und der Operator stoppt.

Beispiel

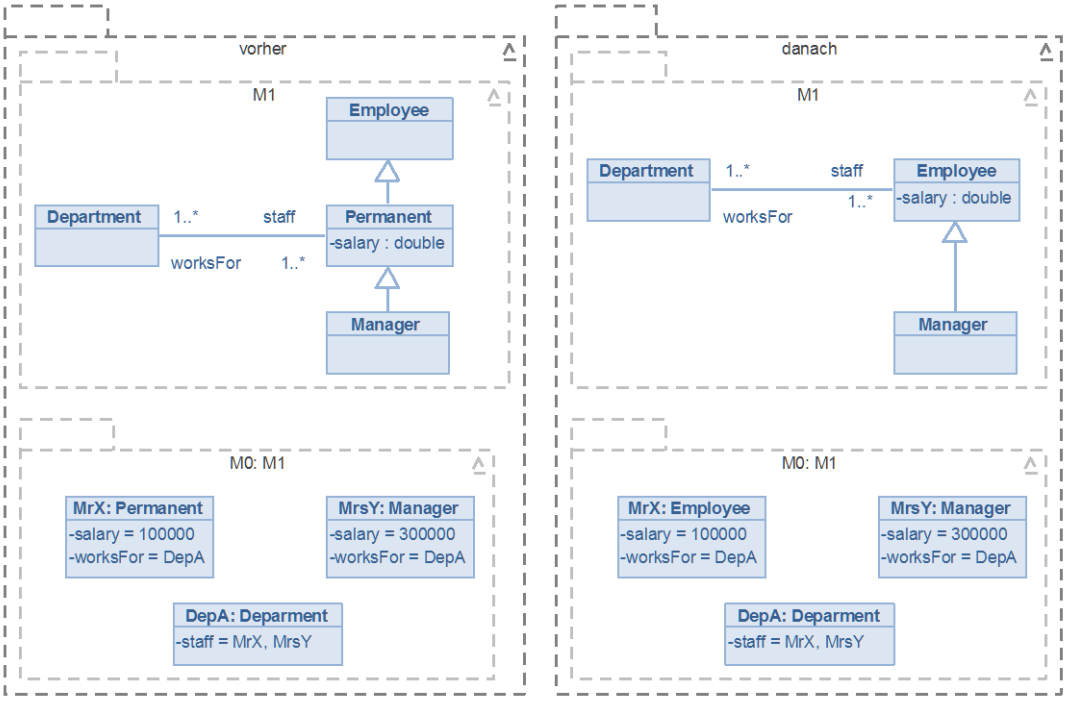

Abbildung 6-15 Beispiel vor (links) und nach (rechts) der Anwendung des Operators

Das in diesem Beispiel verwendete Ausgangsmodell, ist im Wesentlichen gleich mit dem Modell, das im Beispiel des *Extract Sub Type* Operators als Ergebnis entsteht. Die Beschreibung der in Abbildung 6-15 auf der linken Seite gezeigten Modellhierarchie kann also aus Kapitel 6.2.4 entnommen werden. Lediglich einen Unterschied gibt es: Das Attribut salary wurde beim Konzept **Permanent**, das einen Festangestellten darstellt, und nicht wie im anderen Beispiel bei **Employee** definiert.

Nun soll der *Inline Sub Type* Operator auf **Permanent** angewendet werden. Dazu wird als erstes die Generalisierung zu **Permanent**, also **Employee** gesucht. Anschließend werden die Attribute **salary** und **worksFor** von **Permanent** mittels des *Move Attribute to Super Type* Operators zu **Employee** verschoben, wobei der Typ des gegenüberliegenden Attributes (**staff**) von **worksFor** auf **Employee** erweitert wird. Danach werden alle Instanzen von **Permanent** ermittelt, was als Ergebnis **MrX** liefert. Als nächstes wird durch den *Set Instantiation* Operator die Instanziierung von **MrX** zu **Permanent** gelöscht und **Employee** als neuer instanziierter Typ für **MrX** gesetzt, da dieser nicht abstrakt war. Als nächstes wird **Manager** als Spezialisierung von **Permanent** ermittelt und im darauf folgenden Schritt **Employee** als Generalisierung von **Manager** mit Hilfe des *Set Specialization* Operators gesetzt. Da **Permanent** keinen Powertyp besitzt, wird im letzten Schritt **Permanent** gelöscht und man erhält das Ergebnis, das auf der rechten Seite von Abbildung 6-15 zu sehen ist.

[51] Wegen Regel C.3 darf jedes Konzept maximal einen Powertyp besitzen.

6.3 (Erweiterter) Powertyp

Die Operatoren in diesem Abschnitt befassen sich mit den Sprachmustern Powertyp und erweiterter Powertyp. Dabei werden zunächst Operatoren zum Verschieben von Attributen innerhalb des Musters vorgestellt und anschließend ein Operator zum Setzen der Instanziierung einer Powertyp-Instanz präsentiert, der die spezielle Semantik des erweiterten Powertyp Musters abbildet. Zusätzlich werden noch Operatoren zum Einführen und Entfernen des Musters in bzw. aus einem Modell beschrieben.

6.3.1 Move Attribute to Powertype Instance

Auswirkung

Durch die Anwendung des Operators wird ein am partitionierten Typ definiertes Attribut zu einer Instanz des Powertypen verschoben. Dadurch beeinflusste Attribute oder Zuweisungen werden entsprechend gelöscht oder angepasst.

Ablauf

Powertyp Instanz wählen

Der Aufruf des Operators erfolgt an einem Attribut `PartAttr` eines partitionierten Typs, welches zu einer Instanz des dazugehörigen Powertyps verschoben werden soll. Dazu muss zunächst der partitionierte Typ `Part` und anschließend dessen Powertyp `Power` ermittelt werden. Danach können die direkten Instanzen von `Power` bestimmt und im nächsten Schritt eine Instanz als Ziel der Verschiebung gewählt werden. Die gewählte Instanz soll im Folgenden `PowerInstance` heißen.

Attributtyp und gegenüberliegendes Attribut migrieren

Falls das initial übergebene Attribut `PartAttr` als Typ kein Konzept hat (literaler Typ oder Enumeration), kann es direkt zu `PowerInstance` verschoben werden. Falls nicht, muss unterschieden werden, ob es sich um eine reflexive Beziehung handelt, also ob `Part` der Typ (`AttrType`) von `PartAttr` ist. Wenn dem so ist, gibt es zwei alternative Vorgehensweisen. Zum einen kann der Attributtyp auf `PowerInstance` geändert werden. Dies hat zur Folge, dass Zuweisungen an das Attribut, die keine Instanz von `PowerInstance` als Wert besitzen, gelöscht werden (siehe *Set Attribute Type*). Zum anderen kann auch, falls nötig, der Deep Instantiation Zähler von `AttrType` (in diesem Fall ist `AttrType` = `Part`) soweit erhöht werden, dass die Beziehung von `PowerInstance` zu `AttrType` gültig wird. Dies tritt genau dann ein, wenn der Deep Instantiation Zähler von `AttrType` größer als die Instanziierungsdistanz von `PowerInstance` zu `AttrType` ist. Danach wird in diesem Zweig das Attribut zu `PowerInstance` verschoben.

Für den Fall, dass der Attributtyp ein Referenztyp `AttrType` ungleich `Part` ist, wird überprüft, ob es sich um eine bidirektionale Beziehung (entspricht dem Setzen der `oppositeOf` Eigenschaft bei einem Attribut) handelt. Wenn dies der Fall ist, muss das Attribut (`Opposite`), das den gegenüberliegenden Teil der Beziehung darstellt, bestimmt werden. Anschließend wird der Typ von `Opposite` von `Part` auf `PowerInstance` mit Hilfe des *Set Attribute Type* Operators gesetzt. Dies hat zur Folge, dass Zuweisungen an Instanzen, die nicht von `PowerInstance` stammen, gelöscht oder die Werte bei Attributen mit einer Kardinalität größer eins entsprechend angepasst werden. Danach wird die Ebene von `PowerInstance` bestimmt und `AttrType` wird in diese Ebene verschoben. Dieser Schritt ist allerdings nur dann möglich, wenn es keine weitere aufwärts Ebenen-Abhängigkeit (außer der Beziehung) von `AttrType` zu `Part` gibt. Wenn dies der Fall sein sollte, muss diese Abhängigkeit auf- oder der Operator abgebrochen werden.

Für den Fall, dass keine bidirektionale Beziehung vorliegt, kann zwischen der Verschiebung von `AttrType` auf die Ebene von `PowerInstance` und der Anpassung des Deep Instantiation Zählers von `AttrType` (siehe oben) gewählt werden.

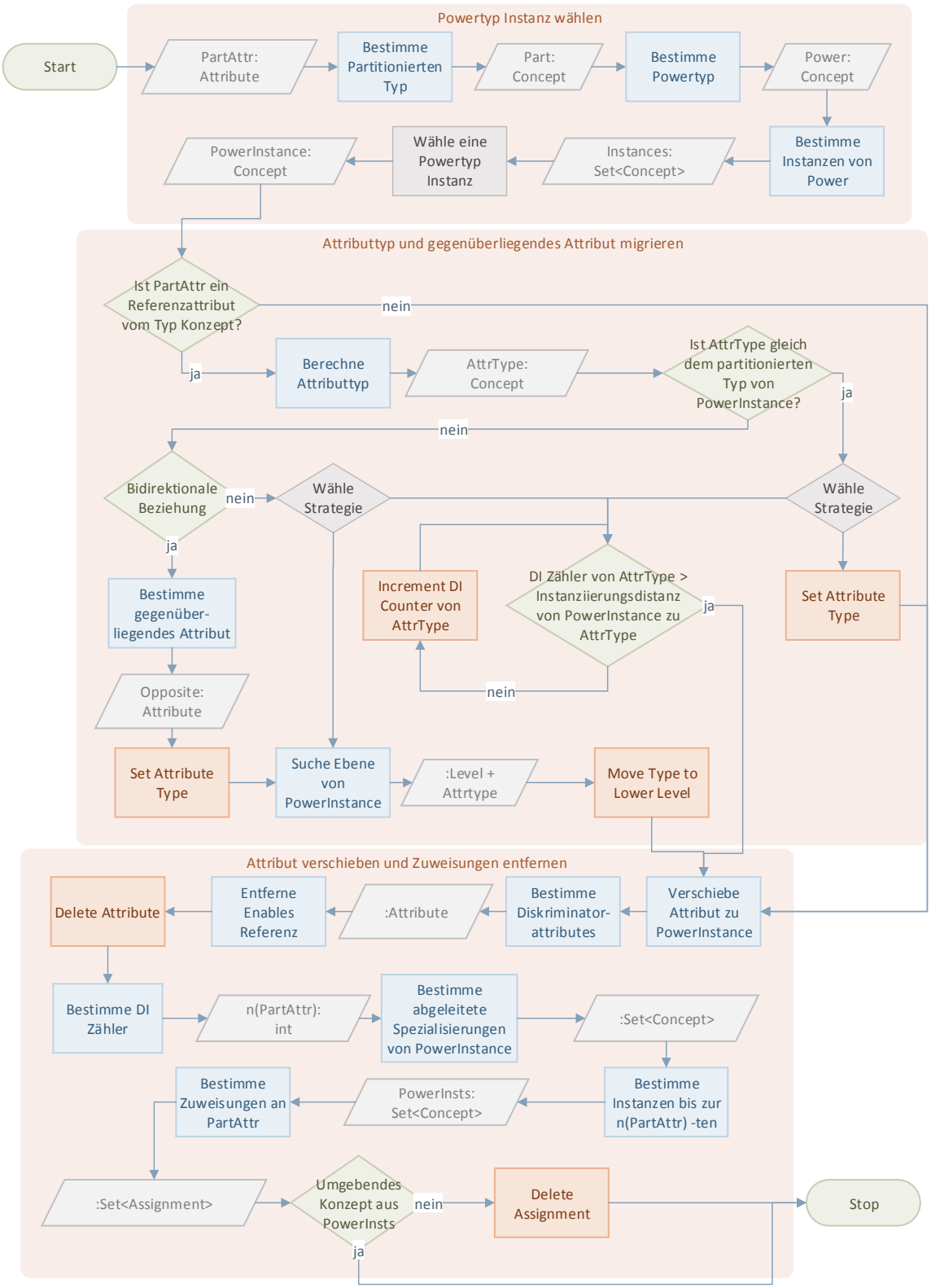

Abbildung 6-16 Ablauf des Move Attribute to Powertype Instance Operators

Attribut verschieben und Zuweisung entfernen

Anschließend kann nun das Attribut zu `PowerInstance` verschoben werden. Nach der Verschiebung des Attributes wird ein eventuell vorhandenes Diskriminatorattribut ermittelt. Falls dieses existiert, wird elementar die `enables` Referenz entfernt, damit beim anschließenden Aufruf des *Delete Attribute* Operators nicht der gesamte erweiterte Powertyp aufgelöst wird. Anschließend wird der Deep Instantiation Zähler von `ParAttr` bestimmt. Danach werden alle abgeleiteten Spezialisierungen von `PowerInstance` ermittelt und als nächstes alle Instanzen bis zur `n(PartAttr)`-ten Ordnung (`Instances`) dieser Menge berechnet, da diese Elemente `PartAttr` weiterhin zuweisen dürfen. Im nächsten Schritt werden alle Zuweisungen an `PartAttr` bestimmt und alle diejenigen mit Hilfe des *Delete Assignment* Operators entfernt, die nicht von einem Element aus `Instances` stammen. Wenn alle ungültigen Zuweisungen gelöscht wurden, stoppt der Operator.

Beispiel

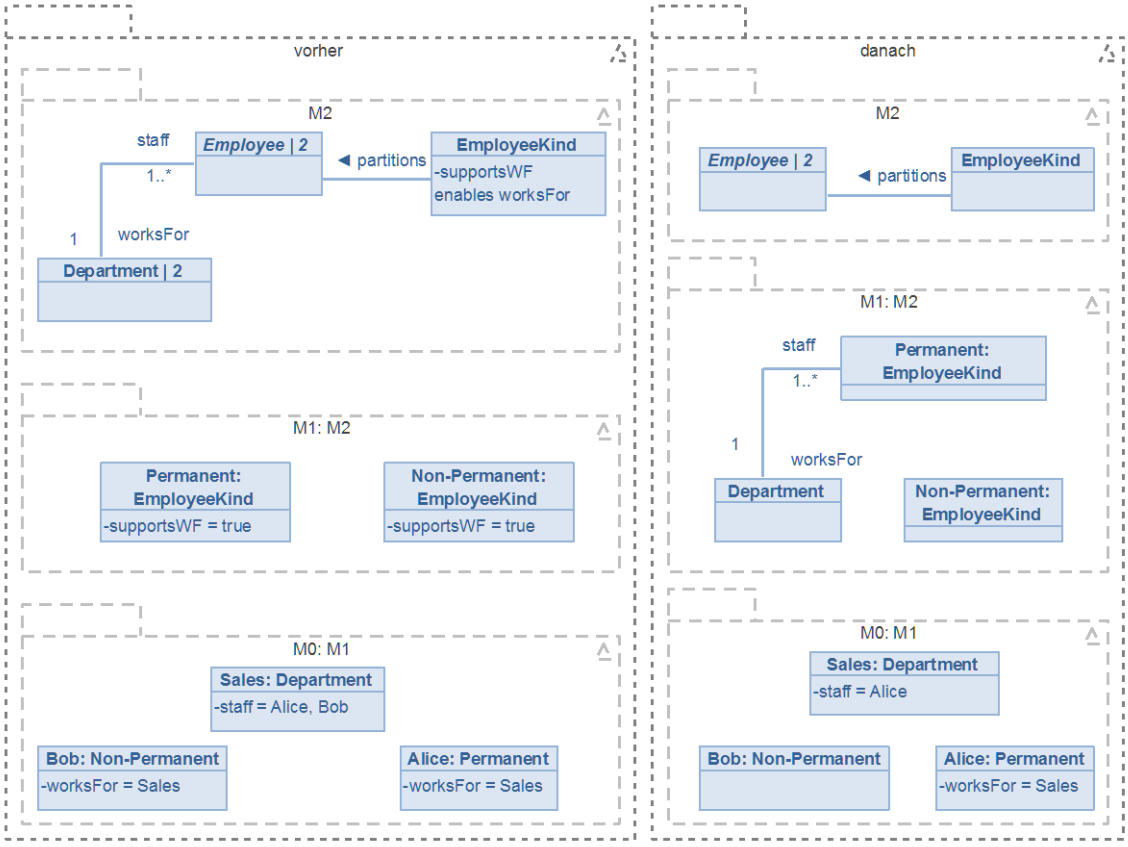

Abbildung 6-17 Beispiel vor (links) und nach (rechts) der Anwendung des Operators

Auf der linken Seite der Abbildung 6-17 ist ein Beispielmodell gegeben, auf das im Folgenden der *Move Attribute to Powertype Instance* Operator angewendet wird. Zunächst soll aber kurz erläutert werden, was im Beispiel dargestellt wurde. Auf `M2` existiert ein Konzept `Employee`, welches einen Mitarbeiter modelliert. Dieser Mitarbeiter arbeitet für genau eine (Multiplizität von `worksFor` ist `1`) Abteilung, während eine Abteilung mindestens einen oder mehrere Mitarbeiter (Multiplizität von `staff` ist `1..*`) hat. Dieser Sachverhalt wird durch das Konzept `Department` und die Beziehung zu `Employee` dargestellt. Die Beziehung ist bidirektional, d.h. bei `Employee` ist ein Attribut `worksFor` mit Typ `Department` und bei `Department` ein Attribut `staff` mit Typ `Employee` definiert. Beide Attribute sind durch die `oppositeOf` Eigenschaft der Attribute miteinander verbunden. `Department` und `Employee` haben außerdem noch einen Deep Instantiation Zähler von zwei, damit sie auf `M0`

instanziiert werden können. Weiterhin ist ein erweiterter Powertyp `EmployeeKind` definiert, der `Employee` partitioniert und ein Diskriminatorattribut zu `worksFor` (`supportsWF`) besitzt.

Auf `M1` gibt es zwei Instanzen von `EmployeeKind`: `Permanent` und `Non-Permanent`. Die beiden Konzepte sollen den Sachverhalt darstellen, dass es festangestellte und nicht festangestellte Mitarbeiter gibt. Beide haben `supportsWF` auf `true` gesetzt, erben daher das Attribut `worksFor` von `Employee` und sind folglich einer Abteilung zugeordnet. `M0` definiert wiederum drei Konzepte: `Alice` und `Bob`, welche Instanzen von `Permanent` bzw. `Non-Permanent` sind, sowie `Sales`, das eine Instanz von `Department` darstellt. `Alice` und `Bob` arbeiten beide für die Abteilung `Sales`, was durch die entsprechenden Zuweisungen bei den Konzepten ausgedrückt wird.

Nach einiger Zeit wird (zum Beispiel von der Unternehmensführung) angeordnet, dass nur noch Festangestellte einer Abteilung zugewiesen werden. Um diese neue Anforderung umzusetzen, wird der *Move Attribute to Powertype Instance* Operator am Attribut `worksFor` des partitionierten Typs `Employee` aufgerufen. Als nächstes wird `Permanent` als die Instanz gewählt, die später das Attribut erhalten soll. Da der Attributtyp `Department` ist und folglich ein Referenztyp ist, der nicht gleich `Employee` ist, wird festgestellt, dass es sich um eine bidirektionale Beziehung mit dem gegenüberliegenden Attribut `staff` handelt. Deshalb muss nun `staff` den Typ von `Employee` zu `Permanent` verengen. Diese Änderungen des Attributtyps zieht nach sich, dass die Zuweisung an `staff` beim Konzept `Sales` um den Wert `Bob` bereinigt wird, da dieser keine Instanz von `Permanent` ist. Dadurch wird auch die Zuweisung von `Bob` an `worksFor` angepasst. Da diese nun nicht mehr den Wert `Sales` besitzt, wird die Zuweisung entfernt.

Danach wird die Ebene von `Permanent` bestimmt (in diesem Fall `M1`). Anschließend wird `Department`, da es außer `worksFor` keine andere Ebenen-Abhängigkeit zu `Employee` besitzt, nach `M1` mittels des Operators *Move Type to Lower Level* und auch `worksFor` von `Employee` zu `Permanent` verschoben. Nach diesem Schritt wird `supportsWF` gelöscht, nachdem die `enables` Referenz entfernt wurde. Dadurch werden die beiden Zuweisungen von `Permanent` und `Non-Permanen`t an `supportsWF` ebenfalls gelöscht. Danach werden alle Instanzen 1-ter Ordnung von `Permanent` (= {`Alice`}) ermittelt, da keine weiteren abgeleiteten Spezialisierungen bestehen. Für alle Zuweisungen an `worksFor` wird dann überprüft, ob sie an einem Konzept dieser Menge definiert wurden. Weil nur noch die Zuweisung von `Alice` besteht, wird also keine Zuweisung entfernt. Als Ergebnis erhält man das in Abbildung 6-17 auf der rechten Seite dargestellte Modell.

6.3.2 Move Attribute to Partitioned Type

Auswirkung

Durch Anwenden des Operators wird ein Attribut von einer Powertyp-Instanz zum partitionierten Typ verschoben. Zudem werden, falls ein erweiterter Powertyp vorliegt, die Diskriminatorattribute erstellt und entsprechende Werte zu diesen Attributen gesetzt.

Ablauf

Attributtyp migrieren

Der Operator wird zu Beginn an einem Attribut `Attr` aufgerufen. Im ersten Schritt wird zunächst der instanziierte Typ des umgebenden Konzepts (der Powertyp) `Power` und danach der partitionierte Typ `PartType` bestimmt. Dann wird je nachdem was `Attr` für einen Attributtyp besitzt unterschieden. Wenn dieser literal ist, kann das Attribut direkt verschoben werden. Wenn dieser eine Enumeration ist, wird diese auf die Ebene von `PartType` verschoben. Ansonsten (`AttrType` ist ein Konzept) muss zunächst, falls der Typ des Attributes (`AttrType`) eine Instanz des Powertyps von `PartType` ist, `AttrType` geändert werden. Er wird auf den partitionierten Typ `PartType` durch den *Set Attribute*

Type Operator gesetzt, da sonst eine Beziehung über mehrere Meta-Ebenen und zwei Konzepten, von denen eines das andere instanziiert, bestehen würde.

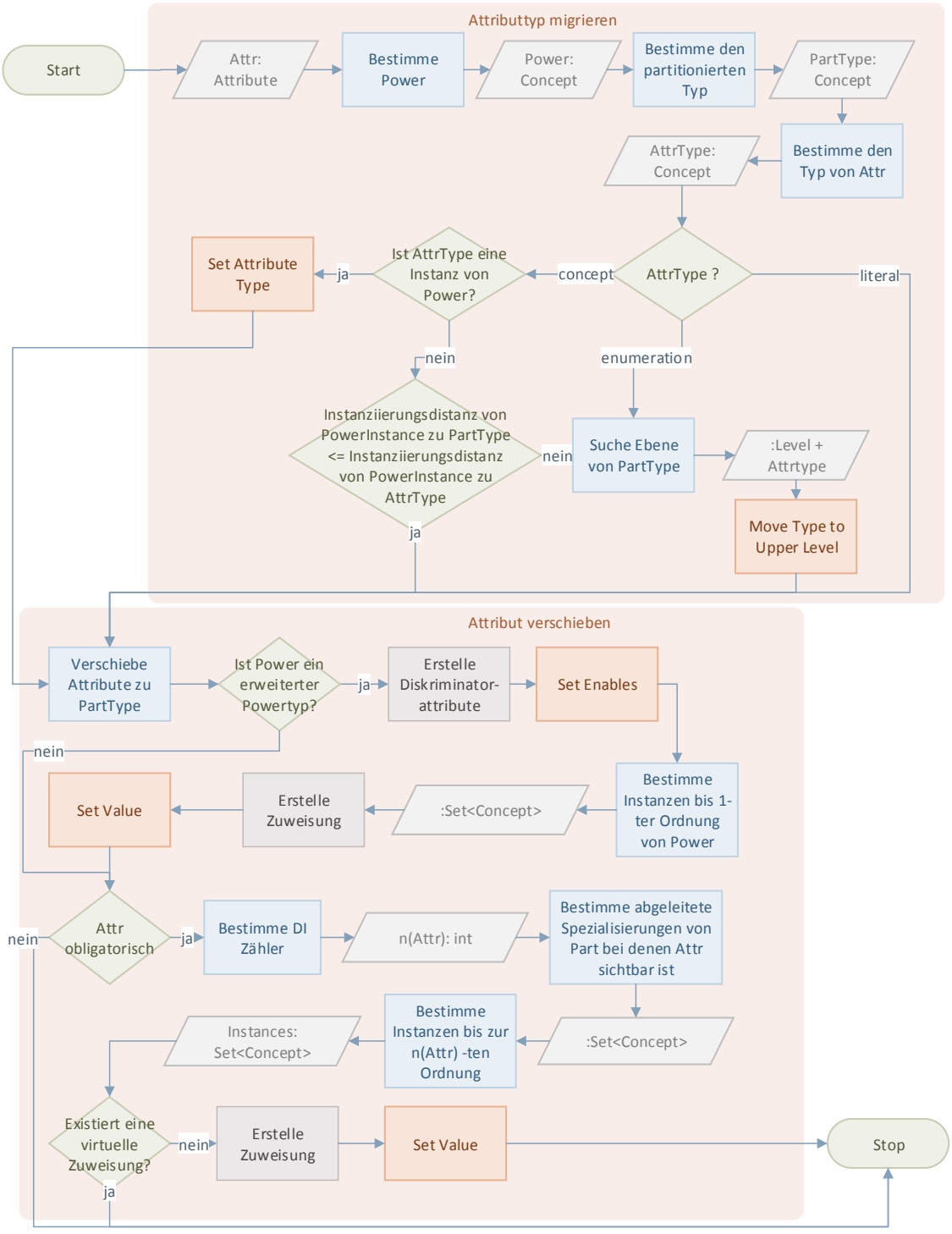

Abbildung 6-18 Ablauf des Move Attribute to Partitioned Type Operators

Im anderen Fall muss **`AttrType`** verschoben werden, falls es nicht auf der gleichen virtuellen Ebene wie **`PartType`** oder auf einer darüber liegenden Ebene liegt. Dies ist dann der Fall wenn die Instanziierungsdistanz Definition 4.3) von **`PowerInstance`** zu **`PartType`** kleiner oder gleich der Instanziierungsdistanz von **`PowerInstance`** zu **`AttrType`** ist. Falls die Verschiebung notwendig ist, wird zunächst die Ebene von **`PartType`** bestimmt und mit Hilfe des *Move Type To Upper Level* Operators **`AttrType`** verschoben. Dies kann nur vollzogen werden, wenn **`AttrType`** keine abwärts

Ebenen-Abhängigkeit zu einer Instanz beliebiger Ordnung von `PartType` hat. Alternativ zum Abbruch könnte auch der Zyklus, der zur Abhängigkeit führt, aufgebrochen werden, in dem eine Kante (Beziehung, `extends`, `partitions`) gelöscht würde.

Attribut verschieben

Im nächsten Schritt wird dann `Attr` zu `PartType` verschoben. Für den Fall, dass ein erweiterter Powertyp vorliegt, erstellt der Operator anschließend ein Diskriminatorattribut für das verschobene Attribut und bestimmt anschließend alle Instanzen bis zur 1-ten Ordnung von `Power`, um danach Zuweisungen für das Diskriminatorattribut zu erstellen. Dabei wird standartmäßig der Wert für die Instanz, an der das Attribut ursprünglich definiert war, auf `true` gesetzt. Das Setzen des Wertes der jeweiligen Zuweisung geschieht durch den *Set Value* Operator. Falls `Attr` ein obligatorisches Attribut ist, kann es durch die Verschiebung dazu kommen, dass nicht alle Instanzen das Attribut bereits gesetzt haben. Deshalb wird zunächst der Deep Instantiation Zähler von `Attr` (`n(Attr)`) und alle abgeleiteten Spezialisierungen von `Part`, bei denen `Attr` sichtbar ist (für eine Powertyp-Instanz bedeutet dies, dass das Diskriminatorattribut `true` gesetzt wurde), ermittelt. Von dieser Menge an Konzepten werden alle Instanzen bis zu `n(Attr)`-ten Ordnung (`Instances`) ermittelt und für diejenigen Instanzen mit voller Ordnung (=`n(Attr)`) wird untersucht, ob eine virtuelle Zuweisung existiert. Wenn keine existiert, muss eine Zuweisung bei einer entsprechenden Instanz aus `Instances` erstellt werden und anschließend der Wert durch den *Set Value* Operator festgelegt werden. Danach ist der Ablauf des Operators beendet.

Beispiel

Die Anwendung des Operators findet sich im Beispiel zum *Set Instantiation to Powertype* Operator.

6.3.3 Set Instantiation to Powertype

Auswirkung

Durch das Ausführen des *Set Instantiation to Powertype*[52] Operators wird eine Instanziierung zu einem Powertypen erstellt. Dabei werden, falls ein erweiterter Powertyp vorliegt, am Konzept definierte Attribute zum partitionierten Typ verschoben. Zusätzlich werden entsprechende Diskriminatorattribute am Powertypen erzeugt, die steuern, ob ein Attribut vom partitionierten Typ geerbt werden soll oder nicht.

Ablauf

Abbildung 6-19 zeigt den Ablauf des Operators: Zu Beginn wird der Operator an einem Konzept `FutureInstance` aufgerufen. Danach wird ein Konzept (`Powertype`) aus den möglichen Kandidaten für die Instanz-Beziehungen (Regel C.12), das außerdem noch ein Powertyp ist, gewählt. Falls `Powertype` kein erweiterter Powertyp ist, wird nun lediglich die Instanziierung von `FutureInstance` zu `Powertype` unter Verwendung des *Set Instantiation* Operators gesetzt. Dabei werden bisherige Instanziierungen oder Generalisierungen von `FutureInstance` migriert. Im Falle eines erweiterten Powertyps besteht die Möglichkeit die deklarierten Attribute von `FutureInstance` zum partitionierten Typ zu verschieben. Für die ausgewählten Attribute wird dann der *Move Attribute to Partitioned Type* Operator aufgerufen, der auch dafür sorgt, dass fehlende Zuweisungen erstellt werden. Nachdem der innere Operator endet, stoppt auch der äußere Operator.

[52] Ein ähnlicher Operator wurde bereits in [61] veröffentlicht.

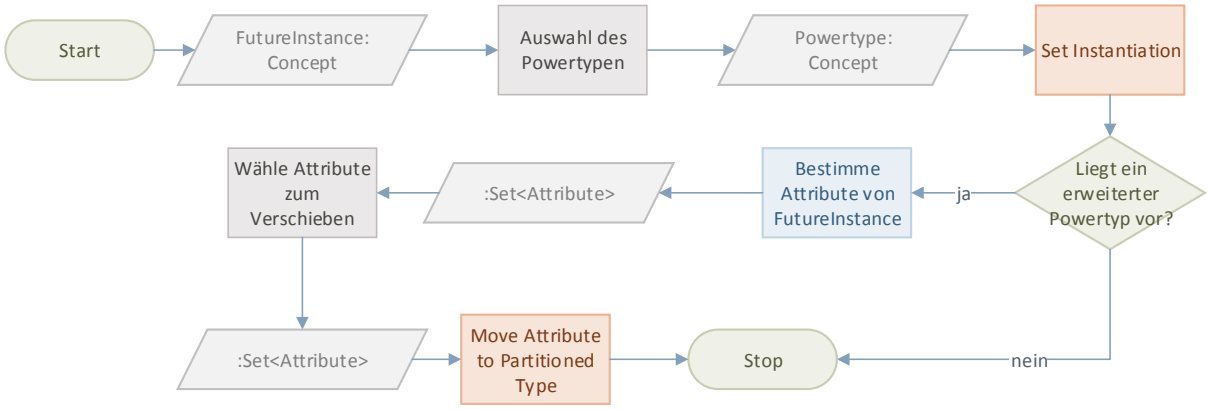

Abbildung 6-19 Ablauf des Set Instantiation to Powertype Operators

Beispiel

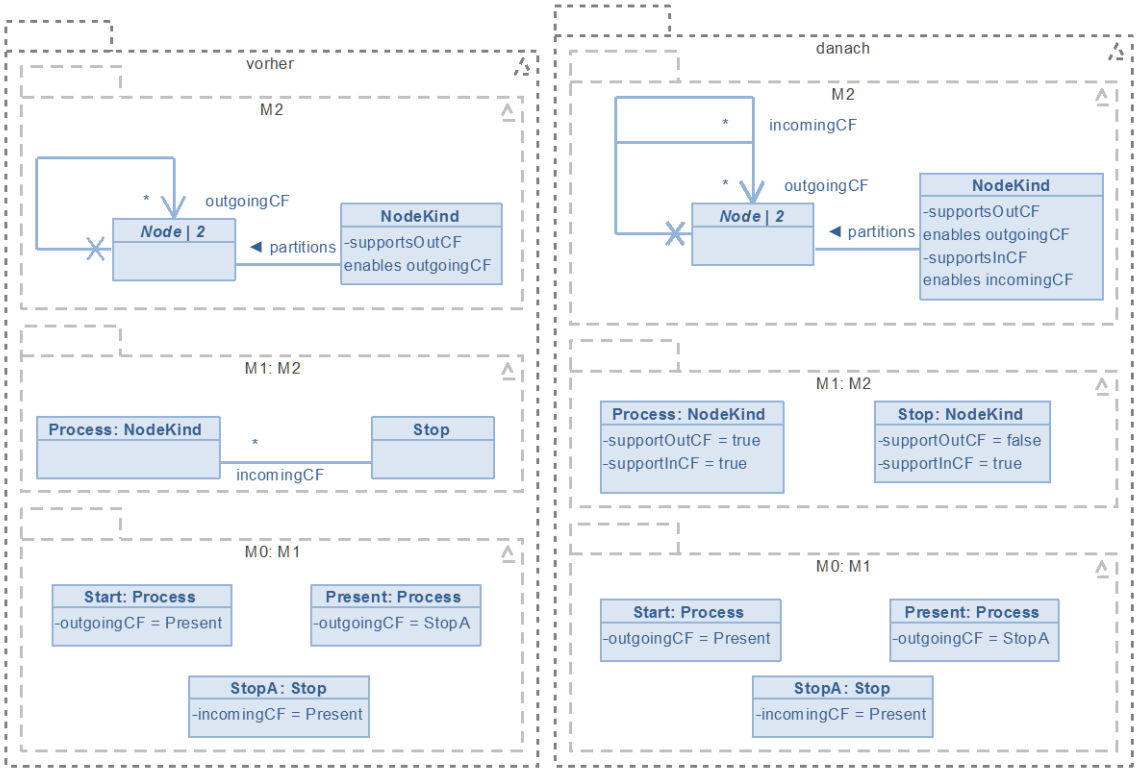

Abbildung 6-20 Einfache Prozessmodellierungssprache vor (links) und nach (rechts) der Ausführung des Set Instantiation to Powertype Operators

In der Abbildung 6-20 ist links eine einfache Prozessmodellierungssprache modelliert. Dazu wurde auf `M2` ein Konzept `Node` (Deep Instantiation Zähler von `2`) mit Powertype `NodeKind` und ein Attribut an `Node outgoingCF` mitsamt Diskriminatorattribut `supportsOutCF` definiert. Folglich soll in dieser Ebene beschrieben werden, dass es in der Sprache Knoten gibt, die durch den Powertyp partitioniert werden und durch einen ausgehenden Kontrollfluss miteinander verbunden sind. In der Ebene `M1` liegt das Konzept `Process`, welches bereits eine Instanz von `NodeKind` ist und das Konzept `Stop`, das eine Beziehung zu mehreren Prozessen haben kann (`incomingCF`). Das Modell auf `M0` zeigt einen Prozess der zunächst mit `Start` (Instanz von `Process`) beginnt, dann die Präsentation einer wissenschaftlichen Veröffentlichung durchführt (`Present`) und mit einer Instanz von `Stop` endet.

Offensichtlich handelt es sich auch bei den Instanzen von `Stop` um Knoten und sie bilden eine Untermenge dieser Knoten. Deshalb macht es Sinn `Stop` ebenfalls als Instanz des Powertyps zu modellieren. Deshalb wird der *Set Instantiation to Powertype* Operator aufgerufen und das Konzept `Stop` übergeben. Danach wird `NodeKind` als instanziierter Typ für `Stop` gewählt und anschließend die Instanziierung erstellt. Da `NodeKind` ein erweiterter Powertyp ist und Knoten im Allgemeinen auch mit eingehenden Kontrollflüssen versehen werden können, ist es sinnvoll das Attribut `incomingCF` zu `Node` zu verschieben. Nach der Auswahl des Attributes wird der *Move Attribute to Partitioned Type* Operator aufgerufen. Dieser erkennt, nachdem er den Powertyp `NodeKind` und den partitionierten Typ `Node` ermittelt hat, dass der Typ des Attributs ein Konzept (`Process`) und eine Instanz von `NodeKind` ist. Deshalb wird der Attributtyp auf `Node` geändert und das Attribut zu `Node` verschoben. Danach wird, weil `NodeKind` ein erweiterter Powertyp ist, ein Diskriminatorattribut bei `NodeKind` mit dem von uns gewählten Namen `supportsInCF` erstellt. Als nächstes entscheiden wir uns, das Diskriminatorattribut bei `Process` zu setzen. Da ein Prozess auch eingehende Kontrollflüsse haben kann, wird der Wert `true` für `supportsInCF` festgelegt. Aufgrund der Tatsache, dass `Stop` `incomingCF` vor der Ausführung des Operators definiert hatte, wird auch bei `Stop` automatisch der Wert `supportsInCF` auf `true` gesetzt. Weil ein `Stop` Interface keine ausgehenden Kontrollflüsse definieren soll und der Standardwert für `supportsInCF` nicht gesetzt wurde, wird eine weitere Zuweisung für `Stop` erstellt, die `supportsInCF` auf `false` setzt, womit der Operator endet. Das Ergebnis ist in Abbildung 6-20 auf der rechten Seite gezeigt.

6.3.4 Create Powertype For

Auswirkung

Der *Create Powertype For*[53] Operator erstellt für eine vorgegebenes Konzept einen Powertyp. Dabei werden frühere Spezialisierungen des Konzeptes zu Instanzen des Powertyps (wodurch sie implizit erneut Spezialisierungen des Konzeptes sind). Im Zuge dessen, werden auch ausgewählte Attribute von den Powertyp-Instanzen zum partitionierten Konzept verschoben.

Ablauf

Zu Beginn des Ablaufes (Abbildung 6-21) wird dem Operator ein Konzept (`Part`) übergeben, das den zukünftigen partitionierten Typ darstellt. Im ersten Schritt werden alle direkten Spezialisierungen von `Part` berechnet, die keine abwärts Ebenen-Abhängigkeit bis auf die Spezialisierung zu `Part` besitzen. Diese kommen als mögliche Instanzen des zukünftig erstellten Powertyps in Frage. Falls mindestens eine dieser Spezialisierungen (im Folgenden `Instance` genannt) als zukünftige Poweryp-Instanz bestimmt wird, muss `Part` eine Meta-Ebene nach oben verschoben werden, damit die Instanziierung erfolgen kann (Regel C.12). Davor muss jedoch die Spezialisierung für alle zukünftigen Instanzen (`Instances`) des Powertyps zu `Part` gelöscht werden, damit der *Move Type to Upper Level* Operator nicht diese mitverschiebt.[54]. Danach wird der neue Powertyp inklusive der `partitions` Beziehung zu `Part` erstellt (Aufruf der entsprechenden Operatoren) und dabei festgelegt, ob es sich um einen erweiterten Powertyp handeln soll. Ist dies der Fall, wird für ein Attribut von `Part` ein entsprechendes Diskriminatorattribut erzeugt und der *Set Enables* Operator aufgerufen, wodurch auch alle anderen

[53] Ein ähnlicher Operator wurde bereits in [61] veröffentlicht.

[54] An dieser Stelle zeigt sich auch warum am Anfang keine Konzepte, die eine weitere aufwärts Ebenen-Abhängigkeit zu `Part` haben, als mögliche Instanzen des Powertyps in Frage kommen: Eine Verschiebung von `Part` würde auch zu einer Verschiebung der aufwärts ebenen-abhängigen Konzepte führen und somit eine Instanziierung dieser zu `Part` unmöglich machen.

Diskriminatorattribute erstellt werden.. Für alle ausgewählten zukünftigen Instanzen wird danach der *Set Instantiation to Powertype* Operator aufgerufen und der Operator endet.

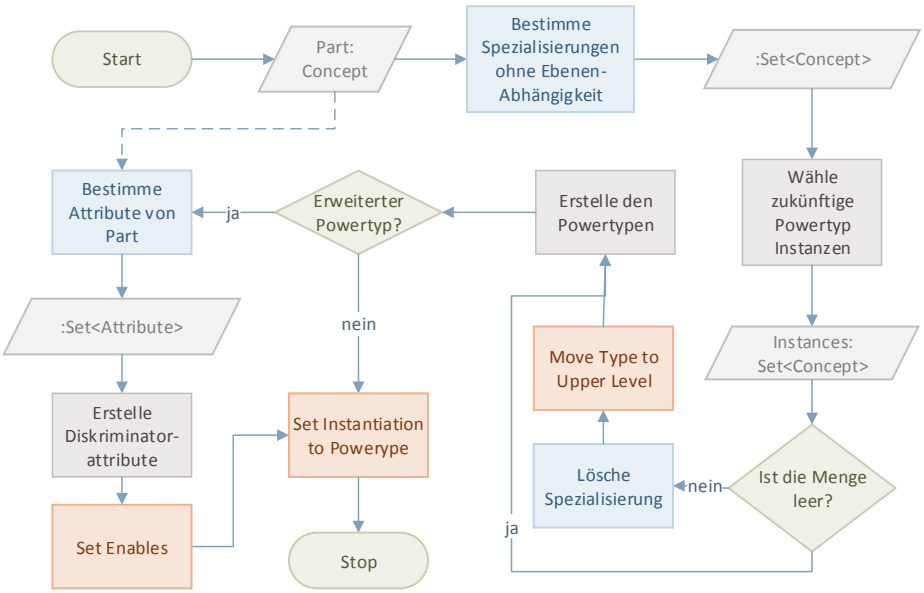

Abbildung 6-21 Ablauf des Create Powertype For Operators

Beispiel 1

Abbildung 6-22 zeigt ein Meta-Modell, das dazu dient eine einfache Sprache für ein Feature-Modell einer Autoproduktionslinie darzustellen. Die Sprache ist angelehnt an das Beispiel aus [78]. Auf `M1` wurden zwei Konzepte `Feature` und `Association` definiert. Jede Assoziation hat ein Feature als Ausgangspunkt (`source`), auf das sie sich bezieht, und kein oder mehrere Features als Ziel (`targets`). Außerdem kann ein Feature mehrere Assoziationen zuweisen (Attribut `association`), wodurch die Beziehung bidirektional wird. Weiterhin gibt es auf `M1` vier Spezialisierungen von `Association`: `Or`, `Xor`, `Mandatory`, und `Optional`. Basierend auf dieser Sprache wurde in `M0` ein Instanz-Modell erstellt, dass vier Features `Car`, `Body`, `Transmission` und `Enigne` und eine Assoziation `CarMandatory`, die Instanz von `Mandatory` ist, enthält. `Car` ist dabei dem `source` und `Engine`, `Body` und `Transmission` sind dem `targets` Attribut der Assoziation `Mandatory` zugewiesen. Es wurde also die Bedingung modelliert, dass jedes Auto (`Car`) eine Karosserie (`Body`), eine Schaltung (`Transmission`) und einen Motor (`Engine`) haben muss (`CarMandatory`).

Nun soll der *Create Powertype For* Operator am Konzept `Association` aufgerufen werden, damit an dieser Stelle das Powertyp Muster erstellt wird. Dazu berechnet der Operator alle Spezialisierungen, die keine aufwärts Ebenen-Abhängigkeit bis auf die Spezialisierung zu `Association` besitzen. Da alle Spezialisierungen dieses Kriterium erfüllen, wählen wir im darauffolgenden Schritt `Or`, `Xor`, `Optional` und `Mandatory` als zukünftige Instanzen des zu erstellenden Powertyps aus. Aufgrund dessen, muss auch `Association` verschoben werden. Bevor dies geschieht, wird die Spezialisierung von `Or`, `Xor`, `Optional` und `Mandatory` gelöscht, damit der auf `Association` aufgerufene *Move Type to Upper Level* Operator die Konzepte nicht ebenfalls verschiebt. Der Aufruf des Operators hat zur Folge, dass neben `Association` auch `Feature` auf die neue Ebene (hier benannt `M2`) verschoben wird. Dadurch wird der Deep Instantiation Zähler von `Association` und `Feature` um eins auf zwei erhöht.

Anschließend wird der Name des neuen Powertyps (`AssociationKind`) und, dass er ein erweiterter Powertyp sein soll, festgelegt. Für alle Attribute von `Association` (`source`, `targets`) werden anschließend Diskriminatorattribute (`supportsSource`, `supportsTargets`) erstellt. Danach wird an allen zukünftigen Instanzen `Or`, `Xor`, `Optional` und `Mandatory` der *Set Instantiation to Powertype*

Operator aufgerufen und dabei jeweils der Powertyp übergeben. Der *Set Instantiation to Powertype* Operator stellt dann die Instanziierung zu **`AssociationKind`** für das jeweilige Konzept (**`Or`**, **`Xor`**, **`Optional`**, **`Mandatory`**) her. Da zwar **`AssociationKind`** ein erweiterter Powertyp ist, keines der übergebene Konzepte aber ein direkt definiertes Attribut besitzt, werden zum Schluss nur noch die Diskriminatorattribute **`supportsSource`** und **`supportsTargets`** bei **`Or`**, **`Xor`**, **`Optional`** und **`Mandatory`** auf **`true`** gesetzt und das Ergebnis auf der rechten Seite der Abbildung 6-22 entsteht.

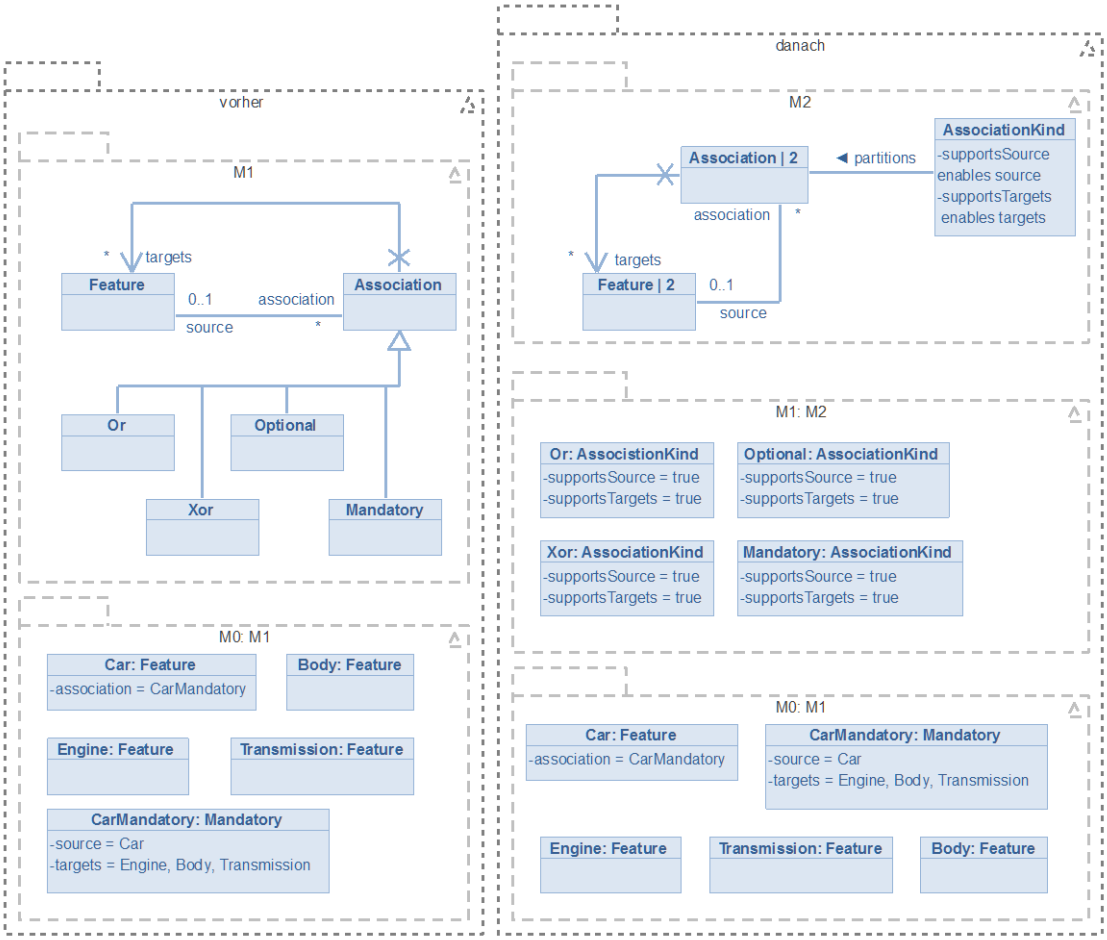

Abbildung 6-22 Car Product Line Beispiel vor (links) und nach (rechts) der Ausführung des Operators

Beispiel 2

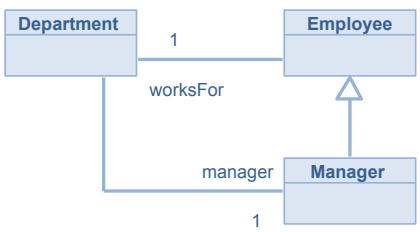

Abbildung 6-23 Beispiel zur Ebenen-Abhängigkeit

Im Folgenden, soll exemplarisch verdeutlicht werden, warum die Bedingung bezüglich der Ebenen-Abhängigkeit von Nöten ist. Dazu ist in Abbildung 6-23 ein Modell gezeigt, dass beschreibt, dass Mitarbeiter in einer Abteilung arbeiten (Konzepte **`Employee`** und **`Department`**, verbunden durch **`worksFor`**). Eine Abteilung wiederum wird von einem Manager geführt (Beziehung **`manager`** von **`Department`** zu **`Manager`**).

Nehmen wir an, dass an **`Employee`** der *Create Powertype For* Operator aufgerufen würde und dabei **`Manager`** als Instanz gewählt werden dürfte. Dies hätte zur Folge, dass **`Employee`** eine Ebene nach oben verschoben werden müsste, damit später die Instanziierung zwischen **`Manager`** und dem neuen Powertypen gesetzt werden dürfte. Diese Verschiebung verursacht aufgrund der Beziehung **`worksFor`**, dass auch

`Department` eine Ebene nach oben geschoben werden müsste. Was wiederum zur Folge hätte, dass auch `Manager`, da dieser durch die Beziehung bzw. das Attribut `manager` auf der gleichen (virtuellen) Ebene wie `Department` liegen muss (Regel C.13), verschoben werden müsste. Da der neue Powertyp auf der gleichen (oder einer darunter liegenden) Ebene wie `Employee` (und `Department` sowie `Manager`) liegen muss, kann die Instanziierung von `Manager` zu `Employee` nicht gesetzt werden. Damit ist die Ausführung des Operators nicht möglich.

6.3.5 Inline Powertype

Auswirkung

Die Anwendung des Operators bewirkt, dass der Powertyp des übergebenen Konzeptes aufgelöst wird. Dabei werden die Instanzen des Powertyps zu Spezialisierungen des partitionierten Typs. Zudem werden die Werte von Diskriminatorattributen verwendet, um eventuell Attribute zu verschieben. Weiterhin wird der Konflikt, dass der partitionierte Typ und die Instanzen des Powertyps auf unterschiedlichen Ebenen definiert sind, aufgelöst.

Ablauf

Attribut Migration

Der Operator wird an einem partitionierten Typ `Part` aufgerufen. Als nächstes wird dessen Powertyp (`Powertype`) und alle abgeleiteten Spezialisierungen davon (`Powertypes`) bestimmt, die später gelöscht werden sollen. Da alle Instanzen 1-ter Ordnung (`Instances`) der Elemente aus `Powertypes` zu Spezialisierungen von `Part` migriert werden sollen, werden diese im nächsten Schritt ermittelt. Falls `Powertype` ein erweiterter Powertyp ist, besteht die Möglichkeit, dass Instanzen des Powertyps existieren, die nicht alle Attribute von `Part` geerbt haben. Deshalb werden in diesem Fall zunächst alle Attribute von `Part` mitsamt deren Diskriminatorattributen beim Powertyp ermittelt. Nun muss für jedes Diskriminatorattribut `feature` untersucht werden, bei welchen Elementen von `Instances` es auf `true` gesetzt wurde. Wenn dies bei allen der Fall ist, kann das entsprechende Attribut bei `Part` verbleiben. Ansonsten wird es zunächst für jedes Element von `SubInstances` dupliziert und mittels *Move Attribute to Powertype Instance* verschoben.

Partitionierten Typ und Powertyp Instanzen migrieren

Danach muss die Auswahl getroffen werden, ob `Part` auf die virtuelle Ebene der Instanzen verschoben werden soll. Wenn dies der Fall sein soll, dann wird an `Part` der *Move Type to Lower Level* Operator so oft aufgerufen, bis `Part` auf der gleichen virtuellen Ebene wie die Elemente aus `Instances` liegt. Konkret muss also der Operator genauso oft aufgerufen werden, wie der Deep Instantiation Zähler von `Part` zu Beginn des Operators groß ist. Alternativ zur Verschiebung von `Part` kann auch der Deep Instantiation Zähler von `Part` dahingehend angepasst werden, dass die Spezialisierung oder eine Beziehung auch auf einer, von `Part` aus gesehen, weiter unten liegenden Ebene erfolgen kann (siehe Regel C.13 bzw. Regel C.14). Für den Fall, dass die Instanziierungsdistanz von `Part` zu einem Element aus `Instances` größer als der Deep Instantiation Zähler von `Part` ist, ändert dies der eventuell mehrmalige Aufruf des *Increment DI Counter of Concept* Operators.[55]

[55] Prinzipiell wäre an dieser Stelle auch eine Verschiebung der Instanzen des Powertyps hin zur Ebene des partitionierten Typs denkbar. Da solch eine Verschiebung jedoch stark in die Struktur der Meta-Hierarchie eingreift, da viele Referenztypen betroffen sind, und sie meist nicht sinnvoll ist, soll diese Möglichkeit hier außer Acht gelassen werden.

Danach wird die Instanziierung aller Elemente von **`Instances`** zu **`Powertype`** (oder einer Spezialisierung davon) durch die Spezialisierung zu **`Part`** mit Hilfe des *Set Specialization* Operators ersetzt. Im letzten Schritt wird noch der Powertyp mitsamt aller Spezialisierungen davon mittels des *Delete Concept* Operators entfernt und der Operator stoppt. Dabei werden auch alle Zuweisungen an Attribute, die bei einem Element von **`Powertypes`** definiert wurden, gelöscht.

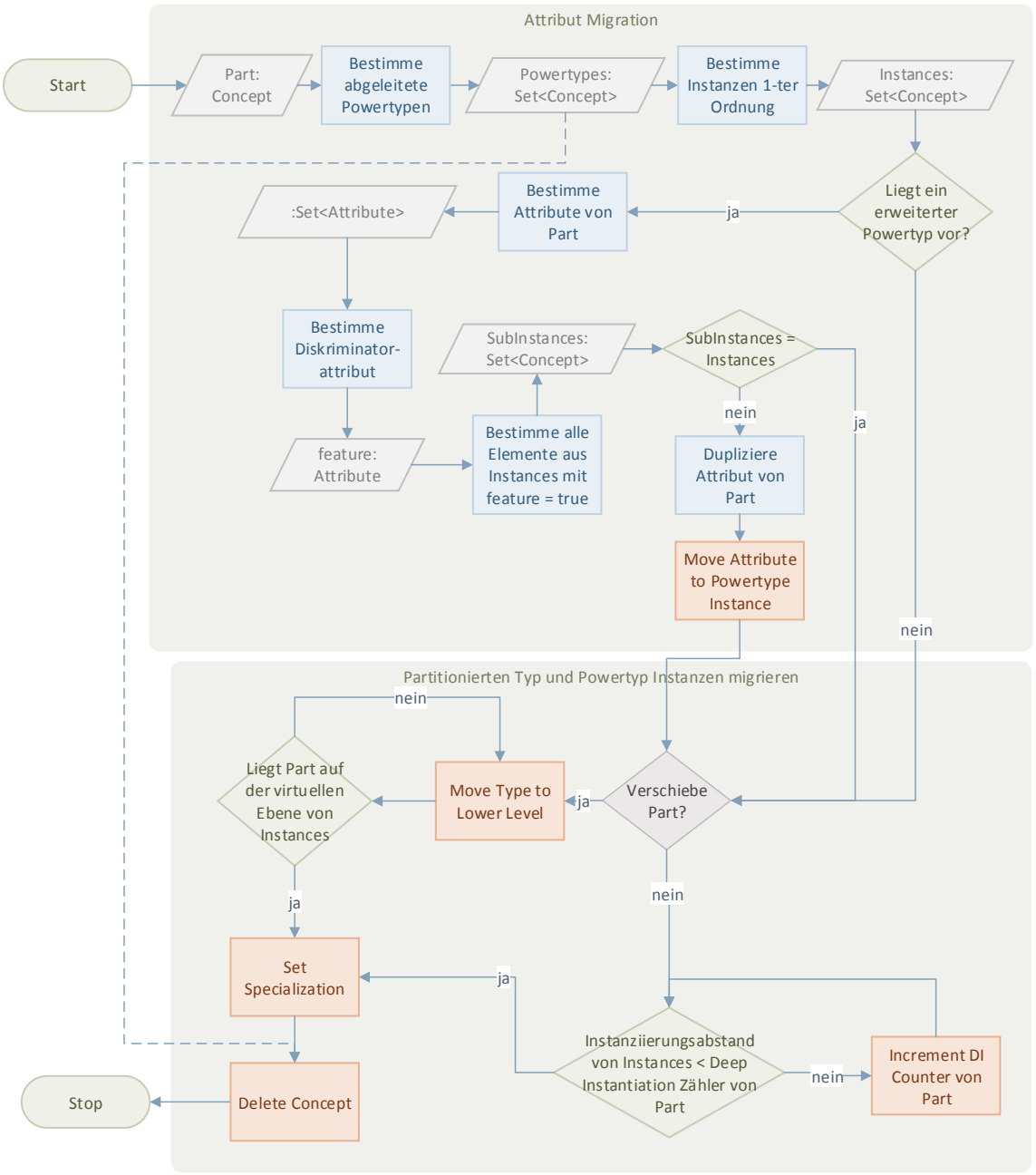

Abbildung 6-24 Ablauf des Inline Powertype Operators

Beispiel

In Abbildung 6-25 ist eine einfache Prozessmodellierungssprache dargestellt. Auf Ebene **`M2`** wurde ein Powertyp (**`NodeKind`**) zum Konzept **`Node`** (Deep Instantiation Zähler von **`2`**) definiert. Dieser partitioniert die durch **`Node`** modellierten Knoten eines Modells. Um Knoten miteinander zu verbinden, wurden zwei Attribute definiert, die einen eingehenden (**`incomingCF`**) und eine ausgehenden (**`outgoingCF`**) Kontrollfluss darstellen sollen. Für beide Attribute definiert **`NodeKind`** jeweils ein

Diskriminatorattribut, wobei `supportsInCF` bzw. `supportsOutCF` steuern, ob ein Konzept `incomingCF` bzw. `outgoingCF` erbt. Auf der unter `M2` liegenden Ebene `M1` wurden ebenfalls zwei Konzepte erstellt, die beide Instanzen von `NodeKind` sind. Das Konzept `Process` steht für einen Teilprozess im modellierten Prozess und unterstützt sowohl eingehende als auch ausgehende Kontrollflüsse. Dies wird durch die Zuweisung des Wertes `true` an die jeweiligen Diskriminatorattribute umgesetzt. Das andere Konzept `Stop` steht für ein Stopp Interface eines Prozesses, das festlegt, wann ein Prozess `beendet` ist. Da nach einer Instanz von `Stop` kein weiterer Kontrollfluss erlaubt ist, hat `Stop supportsOutCF` auf `false` gesetzt. Die Ebene `M0` definiert ein Beispielmodell der Sprache. Dabei wurden drei Konzepte erstellt. `Start` und `Present` sind ein Teilprozess und daher Instanzen von `Process`, während `StopA` eine Instanz von `Stop` ist. Der Ablauf des Prozesses (erst `Start`, dann `Present`, zuletzt `Stop`) ist durch die Zuweisungen bei den Konzepten abgebildet.

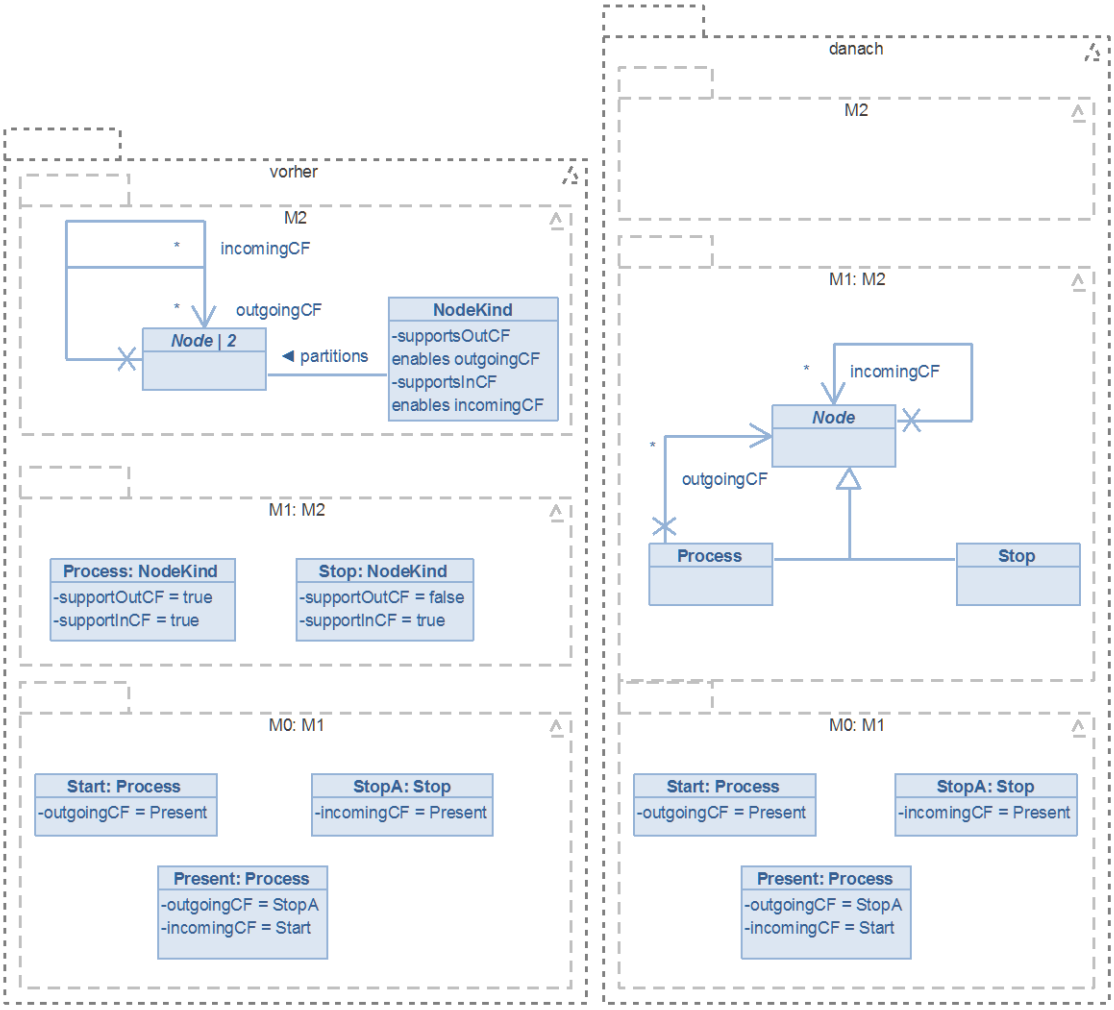

Abbildung 6-25 Einfache Prozessmodellierungssprache vor (links) und nach (rechts) der Ausführung des Inline Powertype Operators

Im Folgenden soll der Powertyp `NodeKind` mit Hilfe des *Inline Powertype* Operators entfernt werden. Dazu wird der Operator am Konzept `Node` aufgerufen. Danach werden alle abgeleiteten Powertypen bestimmt, was in diesem Fall `NodeKind` liefert. Anschließend bestimmt der Operator alle Instanzen 1-ter Ordnung von `NodeKind` (also `Process` und `Stop`). Da `NodeKind` ein erweiterter Powertyp ist, werden danach alle Attribute von `Node` gesucht (`incomingCF` und `outgoingCF`) und die dazugehörigen Diskriminatorattribute `supportsInCF` und `supportsOutCF` bestimmt. Im nächsten Schritt werden alle Elemente ermittelt, die die Diskriminatorattribute auf `true` gesetzt haben. Für

`supportsInCF` ergibt sich als Ergebnis `Process` und `Stop`, während `supportsOutCF` nur bei `Process` auf `true` gesetzt wurde. Weil `supportsInCF` bei allen Instanzen von `NodeKind` im Ergebnis liegt, bleibt `incomingCF` am Konzept `Node` erhalten. Weil dies für das andere Diskriminatorattribut `supportsOutCF` nicht der Fall ist, wird `outgoingCF` mittels des Operators *Move Attribute to Powertype Instance* zu `Process` verschoben. Danach entscheiden wir uns `Node` mit Hilfe des *Move Type to Lower Level* Operators in die Ebene von `Process` und `Stop` zu verschieben. Danach wird die Instanziierung von `Process` und `Stop` zu `NodeKind` gelöscht und die Spezialisierung zu `Node` hergestellt. Da `Node` eine abgeleitete Spezialisierung von sich selbst ist, bleiben alle Zuweisungen der Instanzen von `Process` und `Stop` unberührt. Im letzten Schritt wird `NodeKind` gelöscht, wodurch die Diskriminatorattribute mitsamt der dazugehörigen Zuweisungen bei `Process` und `Stop` entfernt werden. Das resultierende Modell ist in Abbildung 6-25 auf der rechten Seite dargestellt.

6.3.6 Extract Powertype and Partitioned Type

Auswirkung

Der Operator erzeugt einen Powertypen, der dann von aufgerufenen Konzept instanziiert wird, zusammen mit einem entsprechenden partitionierten Typ.

Ablauf

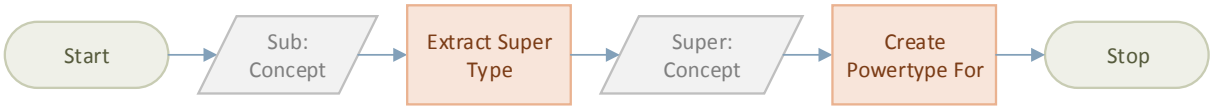

Abbildung 6-26 Ablauf des Extract Powertype and Partitioned Type Operators

Initial wird dem Operator eine Konzept `Sub` übergeben. Danach wir daran der *Extract Super Type* Operator aufgerufen, der eine neue Generalisierung `Super` erstellt, die den zukünftigen partitionierten Typ darstellt. An dieser wird dann der *Create Powertype For* Operator aufgerufen, der den entsprechenden Powertyp zu `Super` erzeugt. `Sub` wird dabei der Menge der gewählten Instanzen hinzugefügt. Danach endet die Ausführung des Operators.

6.4 Instanz-Spezialisierung

Dieser Abschnitt stellt Operatoren für den Umgang mit dem Sprachmuster der Instanz-Spezialisierung bereit, die zur Einführung oder zum Entfernen eines Prototyps dienen.

6.4.1 Extract Prototype

Auswirkung

Der *Extract Prototype*[56] Operator erstellt einen Prototyp aus einem vorgegebenen Konzept und anderen Instanzen des gleichen Typs. Dabei werden die Zuweisungen am Prototyp gemäß dem jeweiligen Überschreibungstyp der gesetzt und die Zuweisungen bei den neuen Instanz-Spezialisierungen entsprechend angepasst.

[56] Der Operator ist in [60] bereits veröffentlicht.

Ablauf

Instanz-Spezialisierung wählen

Der *Extract Prototype* (Abbildung 6-27) wird initial an einem Konzept `Base` aufgerufen, das eine Instanziierung zu einem Konzept `Type` besitzt. Dieser instanziierte Typ wird zunächst bestimmt und alle Instanzen von `Type`, die im gleichen virtuellen Level wie `Base` liegen, im darauffolgenden Schritt berechnet. Aus diesen Instanzen werden dann alle weiteren (neben `Base`) Instanz-Spezialisierungen des zukünftigen Prototyps gewählt. Diese Menge wird im Folgenden `Instances` genannt. Anschließend werden alle instanziierten Attribute von `Base` bestimmt.

Prototyp erstellen

Danach wird aus dieser Menge ausgewählt, welche Attribute beim zukünftigen Prototyp gesetzt werden sollen. In dieser Menge `AttrsToSet` sind alle Attribute von `Type` enthalten, die obligatorisch gesetzt werden müssen, also eine Untergrenze bei der Multiplizität von eins haben. Danach werden für jedes Attribut aus `AttrsToSet` alle Zuweisungen von Konzepten aus `Instances` bestimmt, da diese den Wert des jeweiligen Attributes beim zu erstellenden Prototyp beeinflussen. Das Anlegen des neuen Prototyps `Prototype` erfolgt dann im nächsten Schritt. Dabei wird der Name des neuen Konzeptes festgelegt und anschließend die Instanziierung zu `Type` mittels des *Set Instantiation* Operators erstellt.

Zuweisung beim Prototyp festlegen

Als nächstes wird für jedes Attribut aus `AttrsToSet` eine Zuweisung bei `Prototype` erstellt Dabei muss festgelegt werden, welchen Überschreibungstyp die Zuweisung bei `Prototype` besitzen soll. Je nachdem welcher Überschreibungstyp dann gewählt wurde, wird unterschiedlich weiter verfahren:

- Wenn eine Überschreibung des Attributwertes am Prototyp verboten werden soll (Typ 0), dann muss eine Zuweisung, die bei einem Element aus `Instances` definiert wurde, ausgewählt werden, damit der Wert bei `Prototype` durch den *Set Value* Operator gesetzt wird.

- Wenn der Wert des Prototyps beliebig überschrieben werden darf (Typ 1), dann wird ebenfalls der neue Wert aus der Menge von Zuweisungen der Instanz-Spezialisierungen gewählt. Anschließend muss noch festgelegt werden, bei welchen Instanz-Spezialisierungen die alten Zuweisungen unter Verwendung des *Delete Assignment* Operators gelöscht werden sollen und damit der Wert vom Prototyp geerbt wird.

- Falls die Zuweisung bei `Prototype` die Domäne der jeweiligen Zuweisungswerte der Instanz-Spezialisierungen festlegt (Typ 2), werden alle verschiedenen Werte der Zuweisungen bei `Instances` gesammelt und zu einer Menge[57] zusammengeführt, die dann die Domäne bildet und wieder durch den *Set Value* Operator gesetzt wird. Die Zuweisungen bei den Instanz-Spezialisierungen bleiben dabei unberührt.

- Für die Überschreibungstypen 3 und 4 wird dagegen zunächst ein Basiswert bei `Prototype` festgelegt (*Set Value* Operator), der aus einer Zeichenkette oder einer Teilmenge von allen Zuweisungswerten der Instanz-Spezialisierungen besteht.

Als letztes wird der *Set Concrete Use Of* Operator bei allen Elementen von `Instances` aufgerufen, wodurch die Instanziierung zu `Type` gelöscht und die Instanz-Spezialisierung zu `Prototype` erstellt

[57] Die Eigenschaften einer Menge verhindern, dass mehrmals zugewiesene Werte als Duplikate in der Domäne auftauchen würden.

wird (aufgrund Regel C.10 schließt die Instanz-Spezialisierung eine Instanziierung aus). Dabei werden die Zuweisungen bei den Instanz-Spezialisierungen entsprechend des Überschreibungstyps der Prototyp-Zuweisung gelöscht (Typ 0) oder angepasst (Typ 3 und 4). Nachdem alle Zuweisungen migriert wurden, endet der Operator.

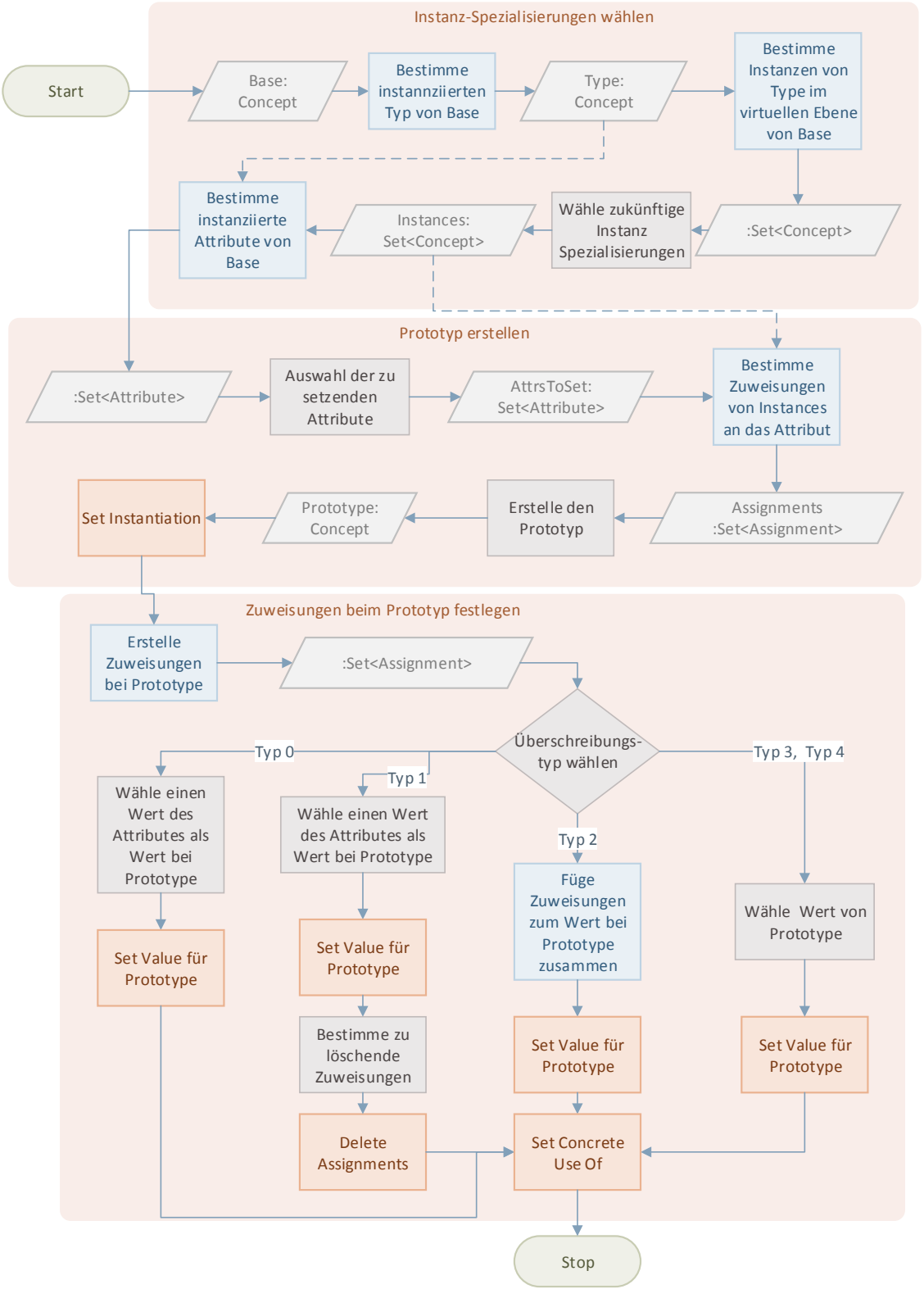

Abbildung 6-27 Ablauf des Extract Prototype Operators

Beispiel

Im Beispiel aus Abbildung 6-28 wurde eine einfache Sprache zur Beschreibung von Autos modelliert. Auf `M1` wurde daher das Konzept `Car` definiert, das für ein Auto steht und einen Hersteller (`manufacturer`), einen Typ (`typeName`) und eine Beziehung (`equipment`) zum Konzept `Equipment`, das als Attribut einen Namen hat (`name`) und die Serienausstattung eines Autos modelliert. Ein Auto kann mehrere Ausstattungsteile besitzen. Auf der Ebene `M0` sind zwei Autos, also Instanzen von `Car`, definiert: `IbizaStyle` und `IbizaReference`. Beide haben für das Attribut `manufacturer` den Wert `„Seat“`, da sie von Seat produziert werden. Der Typ stimmt im Wesentlichen bei beiden mit dem Klassennamen (`“Ibiza Reference“`, `“Ibiza Style“`) überein. Weiterhin wurden noch zwei Ausstattungsbestandteile modelliert, die Konzepte `ABS` und `ESC`, welche für das Antiblockiersystem (ABS) und das Elektronische Stabilitätsprogramm (ESP)[58] stehen und die entsprechenden Namen als Wert des Attributes `name` gesetzt haben. Durch die Zuweisungen an `equipment` bei den beiden Instanzen von `Car` wird ausgedrückt, dass der Ibiza Reference nur ABS als Ausstattung besitzt, während der Ibiza Style sowohl ABS als auch ESP unterstützt[59].

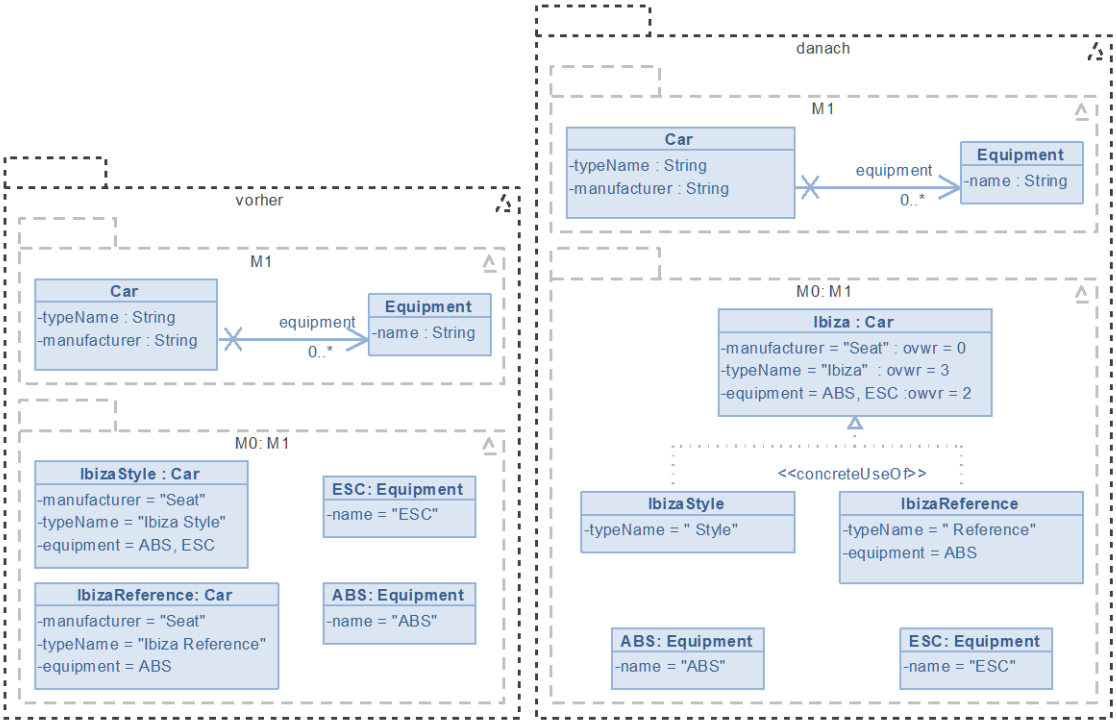

Abbildung 6-28 Beispiel vor (links) und nach (rechts) der Ausführung des Extract Prototype Operators

Da Ibiza Style und Ibiza Reference zwei verschiede Ausprägungen einer Spezialisierung des Ibiza darstellen, soll das entsprechende Konzept als Prototyp mit Hilfe des *Extract Prototype* Operators eingeführt werden. Dazu wird der Operator am Konzept `IbizaReference` aufgerufen. Anschießend werden alle anderen Instanzen von `Car` gesucht, die sich in der gleichen virtuellen Ebene befinden. Dabei erhält man als Ergebnis `IbizaStyle`, was ebenfalls zur Menge der späteren Instanz-Spezialisierungen hinzugenommen wird. Im nächsten Schritt werden alle Attribute von `Car` bestimmt

[58] Im Englischen: electronic stability control (ESC)

[59] Die Ausstattungsmerkmale sind hier nur exemplarisch genannt und müssen nicht mit der wirklichen Ausstattung der beiden Modelle übereinstimmen.

(`manufacturer`, `typeName` und `equipment`), die bei `IbizaReference` instanziierbar sind, und alle ausgewählt, um sie beim späteren Prototyp zu setzen.

Nun werden alle Zuweisungen der beiden gewählten Instanz-Spezialisierungen an eines der drei oben genannten Attribute ermittelt. Danach kann der Prototyp als neues Konzept angelegt werden. Dazu wird in diesem Beispiel der neue Name `Ibiza` gewählt und anschließend die Instanziierung zu `Car` hergestellt. Danach werden für alle oben ausgewählten Attribute bei `Ibiza` Zuweisungen erstellt und die Überschreibungstypen wie folgt festgelegt: Das Attribut `manufacturer` darf von Spezialisierungen der Instanzfacette von `Ibiza` nicht überschreiben werden (Typ 0), während `typeName` einen Zusatz in Form einer anderen Zeichenkette erhalten darf (Typ 3, `ovwr = 3`). Das zur Beziehung gehörende Attribut `equipment` erhält den Überschreibungstyp 2 (`ovwr = 2`) und veranlasst daher, dass alle Werte der Instanz-Spezialisierungen aus der Domäne, die ein Prototyp definiert, entstammen müssen.

Da `manufacturer` keine Überschreibung zulässt, wird die Zuweisung von z.B. `Ibiza Reference` mit dem Wert **„Seat“** als Zuweisung für `Ibiza` gesetzt und später durch den *Set Concrete Use Of* Operator die Zuweisungen der beiden Instanz-Spezialisierungen gelöscht. Da `typeName` eine Konkatenation der Werte bei den Instanz-Spezialisierungen zulässt, wird für `Ibiza` der Wert „`Ibiza`“ gewählt, während `IbizaReference` den Wert **„** `Reference`“ und `IbizaStyle` den Wert **„** `Style`“ zugewiesen bekommen (innerhalb des *Set Concrete Use Of* Operators). Weil der Wert von `Ibiza` als Basiswert für die Zuweisungen bei den Instanz-Spezialisierungen fungiert, haben beide letztlich die gleichen Werte wie zuvor. Das Attribut `equipment` hat bei `Ibiza` den Überschreibungstyp 2 und verlangt daher, dass der Wert bei `Ibiza` die Domäne für alle Zuweisungen bei den Instanz-Spezialisierung darstellt. Daher wird in diesem Fall jeder zugewiesene Wert in einer Menge vereinigt. Es entsteht also der Wert {`ABS, ESC`} für `equipment` bei `Ibiza`. Die Zuweisungen bei den Instanz-Spezialisierungen bleiben in diesem Fall unberührt. Im letzten Schritt löscht der Operator die Instanziierung von `IbizaReference` und `IbizaStyle` und stellt dafür die Instanz-Spezialisierung zu `Ibiza` her. Das resultierende Modell ist in Abbildung 6-28 auf der rechten Seite zu sehen.

6.4.2 Inline Prototype

Auswirkung

Die Anwendung des Operators löscht den Prototypen und fügt die Zuweisungen soweit nötig bei den Instanz-Spezialisierungen ein.

Ablauf

Da der *Delete Concept* Operator bei allen Instanz-Spezialisierungen den *Delete Concrete Use Of* Operator aufruft und dieser wiederum das jeweilige Konzept korrekt migriert, wird der Prototyp durch die alleinige Anwendung des *Delete Concept* Operators auf den Prototyp aufgelöst und korrekt migriert.

6.5 Materialisierung

In diesem Abschnitt wird ein Operator für die Einführung der Materialisierung beschrieben. Andere Operatoren im Umgang mit dem Muster wurden bereits in den Abschnitten 5.5.8-5.5.11 vorgestellt.

6.5.1 Introduce Materialization

Auswirkung

Der *Introduce Materialization* Operator wendet das Materialisierung Muster (siehe Abschnitt 2.5) auf ein vorhandenes Meta-Modell an. Dabei ersetzt es direkt definierte Attribute durch materialisierte, soweit dies möglich bzw. erwünscht ist.

Ablauf

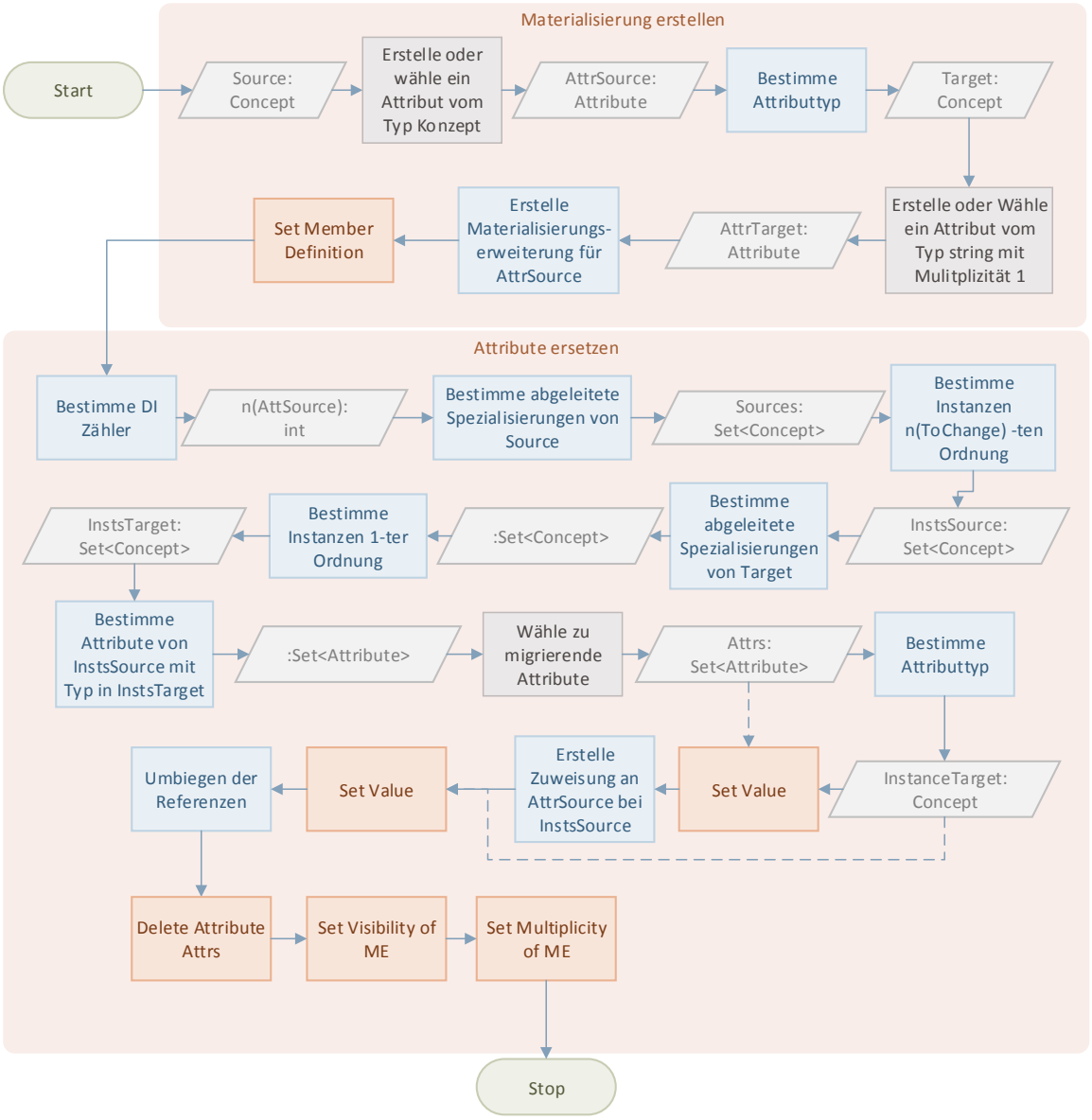

Abbildung 6-29 Ablauf des Introduce Materialization Operators

Materialisierung erstellen

Der Operator wird mit einem Konzept initialisiert, das im Nachfolgenden `Source` heißen soll. Von diesem Konzept wird ein Attribut `AttrSource` gewählt oder neu angelegt, das referentiell ist und ein Konzept als Typ besitzt. Anschließend wird dieser Typ des Attributs vom Operator ermittelt und am entsprechenden Konzept (hier genannt `Target`) wird ein anderes Attribut vom Typ `string` und Multiplizität `1` ausgewählt oder erstellt (Regel A.3). Der Wert dieses Attributs `AttrTarget` bestimmt später den Namen der entsprechend materialisierten Attribute und muss folglich bei jeder Instanz von

`Target` genau einmal gesetzt sein. Als nächstes wird die Materialisierungserweiterung für `AttrSource` erstellt und mit Hilfe des *Set Member Definition* Operators `AttrTarget` als Attribut-Definition gesetzt.

Attribute ersetzen

Nach diesem Schritt wird der Deep Instantiation Zähler von `AttrSource` (`n(AttrSource)`) und alle abgeleiteten Spezialisierungen (`Sources`) von `Source` bestimmt. Mit Hilfe dieser beiden Informationen, werden danach alle Instanzen der `n(AttrSource)`-ten Ordnung (`InstsSource`) von `Sources` berechnet. Im nächsten Schritt werden alle abgeleiteten Spezialisierungen von `Target` und danach alle Instanzen 1-ter Ordnung[60] (`InstsTarget`) von diesen Konzepten bestimmt. Als nächstes werden alle Attribute, die an `InstsSource` definiert sind und als Typ ein Konzept aus `InstsTarget` besitzen, ermittelt, da diese in das Muster eingebunden werden können. Aus dieser Menge werden die zu migrierenden Attribute gewählt. Ein Element aus dieser Menge (`Attrs`) soll im Folgenden `AttrToMigrate` genannt werden.

Für jedes dieser Attribute innerhalb der Auswahl wird der Typ `InstanceTarget` bestimmt, an dem dann die Zuweisung an `AttrTarget`, die existieren muss, da eine Attribut-Definition immer obligatorisch ist, den Namen des bisherigen Attributes (`AttrsToMigrate`) als Wert durch den *Set Value* Operator erhält. Anschließend wird mit Hilfe desselben Operators `InstanceTarget` als Wert beim entsprechenden Element aus `InstsSource` für `AttrSource` hinzugefügt. Danach werden alle Referenzen von `AttrToMigrate` auf das materialisierte Attribut, das im vorherigen Schritt entsteht, umgebogen und anschließend das Attribut `AttrToMigrate` durch den *Delete Attribute* Operator entfernt. Daran anschließend werden die Sichtbarkeit und die Multiplizität der Materialisierungserweiterung mit Hilfe der beiden Operatoren *Set Visibility of Materialization Extend* und *Set Multiplicity of Materialization Extend* gewählt und dadurch ungültige Zuweisungen gelöscht oder fehlende erstellt, womit die Ausführung des Operators endet.

Beispiel

In der Abbildung 6-30 ist auf der linken Seite ein Meta-Modell gegeben, das eine einfache Sprache darstellt, um Entitäten zu modellieren. Auf der obersten Meta-Ebene `M2` gibt es zwei Konzepte: `Entity` und `Attribute`. Diese beiden Konzepte sind zunächst nicht miteinander verknüpft, obwohl sie die Tatsache modellieren sollen, dass eine Entität Attribute besitzt. Eine Beziehung zwischen beiden Konzepten könnte dieses Problem lösen, allerdings würde dies zu Redundanzen führen (Zuweisung der Beziehung mit Wert `Address` und Definition des Attributes `address` bei `Costumer`), die manuell gepflegt werden müssten. Deshalb ist es sinnvoll in diesem Fall die entsprechenden Attribute bei `Costumer` zu materialisieren, da diese dann je nach Zuweisung der Instanzen auf `M1` von `Attribute` dynamisch erzeugt werden, sodass die Zuweisung der Adresse `Street1` bei `Alice` weiterhin erhalten bleibt.

Um das gewünschte Muster in das vorgestellte Modell einzuführen, wird also der *Introduce Materialization* Operator am Konzept `Entity` aufgerufen. Es besteht noch keine Beziehung von `Entity` zu `Attribute`, deshalb wird diese erstellt (`attrs` genannt). Da in diesem Fall eine konkrete Entität (z.B. `Costumer`) mehrere Attribute haben kann und auch mindestens eines haben muss (den Primärschlüssel), wird die Multiplizität auf `1..*` festgelegt. Nun muss beim Konzept `Attribute` ein entsprechendes Attribut gewählt werden, dessen Wert dann den Namen des materialisierten Attributes festlegt und folglich vom Typ `string` sein muss. Weil `Attribute` noch kein solches Attribut

[60] Diese Konzepte können bei `InstsSource` an `AttrSource` zugewiesen werden.

besitzt, wird auch hier ein entsprechendes Attribut **attrName** erzeugt. Damit kann am Attribut **attrs** die Materialisierungserweiterung erstellt und die Attribut-Definition gesetzt werden.

Im nächsten Schritt wird der Deep Instantiation Zähler von **attrs** (=**1**) und anschließend alle abgeleiteten Spezialisierungen von **Entity** (={**Entity**}) ermittelt. Damit werden alle Instanzen der 1-ten Ordnung berechnet, was als Ergebnis **Costumer** liefert. Als nächstes werden alle Instanzen 1-ter Ordnung von **Attribute** (besitzt ebenfalls keine weiteren abgeleiteten Spezialisierungen) berechnet (={**Address**}) und alle Attribute von **Costumer** bestimmt, die **Address** als Typ haben. Resultat ist also nur das Attribut **address** am Konzept **Costumer**, welches in diesem Beispiel zur Migration ausgewählt wird. Nun wird an **Address** das oben erstellte Attribut **attrName** gesetzt und zwar auf den Namen des ehemaligen Attributes (**„address“**). Danach wird Address als Wert für **attrs** bei **Costumer** hinzugefügt und alle Referenzen (inkl. der Zuweisungen) auf das materialisierte Attribut umgebogen. Als nächstes wird **attrs** entfernt und danach die Multiplizität und Sichtbarkeit aller materialisierten Attribute von uns festgelegt. Wegen der Tatsache, dass **address** die Multiplizität **1..*** hatte, entscheiden wir uns ebenfalls dafür diesen Wert und die Sichtbarkeit **public** zu setzen. Das Ergebnis des Operators ist in Abbildung 6-30 auf der rechten Seite abgebildet.

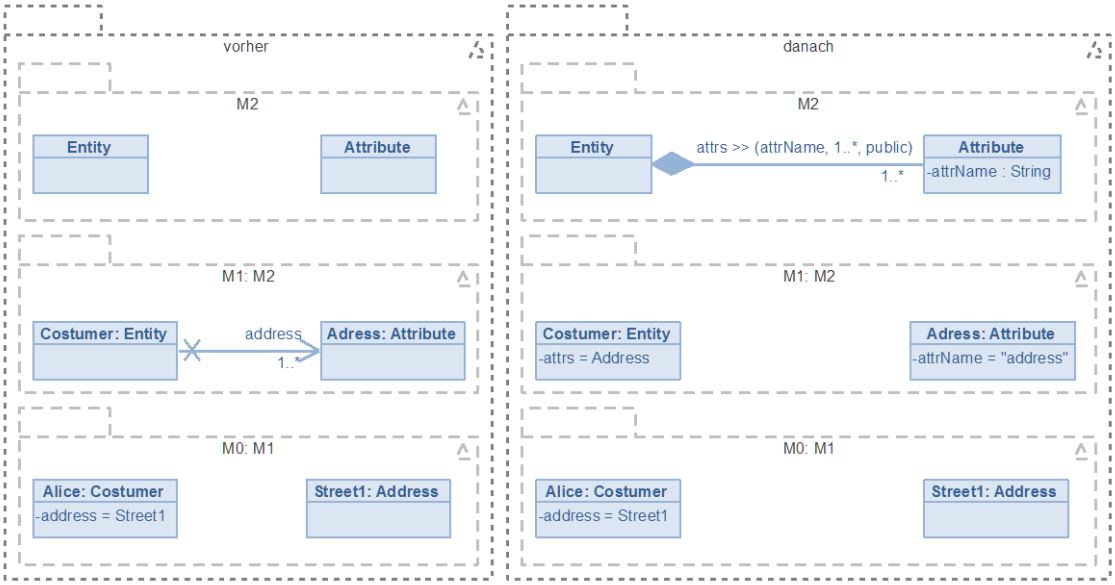

Abbildung 6-30 Beispiel vor (links) und nach (rechts) der Ausführung des Introduce Materialization Operators

6.6 Zusammenfassung

Dieses Kapitel hat Operatoren zur Einführung oder Änderung der sprachbasierten Muster aus Kapitel 2 vorgestellt. Diese ermöglichen die Unterstützung des Benutzers in der Meta-Modellierung mit mehrerer Ebenen, der Verwendung von Vererbung, des (erweiterten) Powertyps, der Instanz-Spezialisierung und der Materialisierung. Zusammen mit den Operatoren aus den Abschnitten 5.3.5 bzw. 5.5.4, die die Deep Instantiation behandeln, existiert somit hinsichtlich der Sprachmuster eine praxis-vollständige Bibliothek, die ein Novum darstellt. Auf diese Weise wurde Herausforderung H1 realisiert, da die vorgestellten Operatoren die gewünschte Strategie inklusive der Evolution und Migration der betroffenen Modellelemente umsetzen.

7 Benutzerunterstützung im Umgang mit Sprachmustern

Das folgende Kapitel stellt die Modellierungsplattform *Model Workbench* vor und zeigt, in welcher Form darin der Umgang mit Sprachmustern im Zusammenhang mit der Evolution von Modellen durch das System unterstützt wird. Damit belegt das System, dass die vorgestellten Konzepte realisierbar sind. Zusätzlich wird noch einmal explizit darauf eingegangen, wie sich die iterative Evolution in einem Modellierungssystem unterstützen lässt. Die *Model Workbench* wurde im Projekt *Kompetenzzentrum für praktisches Prozess- und Qualitätsmanagement* entwickelt und eingesetzt, welches vom Europäischen Fonds für regionale Entwicklung (EFRE) gefördert wird. Weiterhin dient das Modellierungssystem dem Einsatz in entsprechenden Lehrveranstaltung.

7.1 Implementierung

Das *Model Evolution* System, das die Umsetzung der in dieser Arbeit vorgestellten Konzepte beinhaltet, ist als Erweiterung des im Zug dieser Dissertation und der Dissertation von Roth [96] entwickelten Systems *Model Workbench* (Abbildung 7-1) umgesetzt. Diese Erweiterung erfolgte in Form eines projizierenden Text-Editors, der bereits die elementaren Operatoren des LMMs nutzt. Daneben wurde ein generischer Wizard erstellt, der die Ausführung der komplexen Operatoren ermöglicht. Zunächst soll aber die Architektur der *Model Workbench* vorgestellt werden, die bereits den Operatorgedanken in sich trägt.

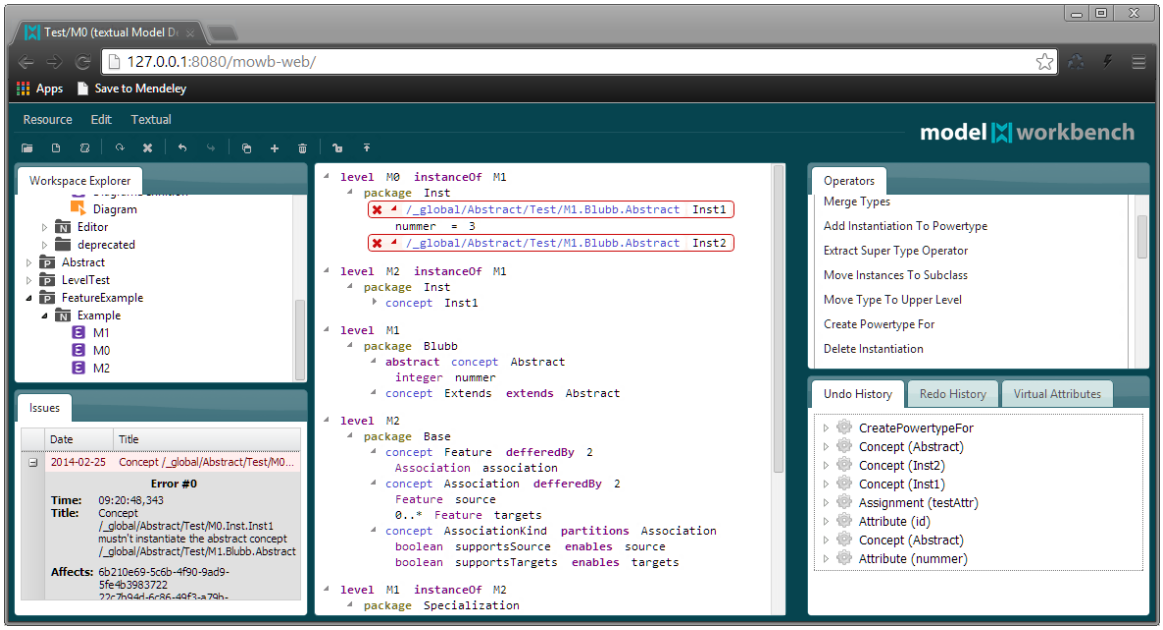

Abbildung 7-1 Screenshot der Model Workbench

7.1.1 Architektur der Model Workbench

Abbildung 7-2 zeigt die grobe Architektur der *Model Workbench*. Sie ist in verschiedenen Schichten organisiert und setzt ein typisches Client-Server Modell um. Der Server wurde dabei mit Java EE

Technologien [62] und Neo4j [77] umgesetzt, während auf Clientseite HTML5 und TypeScript [120] zusammen mit einschlägigen JavaScript Frameworks (jQuery [63], ExtJS [102], Backbone [8], …) zum Einsatz kommen.

Repositorium

Die zentrale Datenhaltungskomponente der *Model Workbench* stellt das Repositorium dar. Wie bereits in Kapitel 4 ersichtlich wird, bietet sich die Umsetzung eines Modells als Graph besonders an, um die Algorithmen für die entsprechenden Berechnungen der Elementmengen umzusetzen. Daher wurde das Repositorium auf Grundlage der Graph-Datenbank Neo4J implementiert, die sowohl das ACID Prinzip unterstützt als auch mehrere reichhaltige APIs zum Auslesen der Modelldaten bereitstellt und damit die Grundlage für die effiziente Umsetzung des Typsystems bildet. Ein Graph in Neo4J besteht wie üblich aus Knoten und Kanten, die beide mit Eigenschaften versehen werden können. Weiterhin ist dieser gerichtet, kann aber dennoch in beide Richtungen durchlaufen werden.

Für die Persistierung des LMMs wurden die Modellelemente auf typisierte Knoten (Level, Package, Konzept,…) mit den entsprechend in 4.1 vorgestellten Eigenschaften abgebildet. Analog werden die Relationen zwischen den Elementen, inklusive den Containerbeziehungen, durch typisierte Kanten (extends, instanceOf, partitions,…) realisiert.

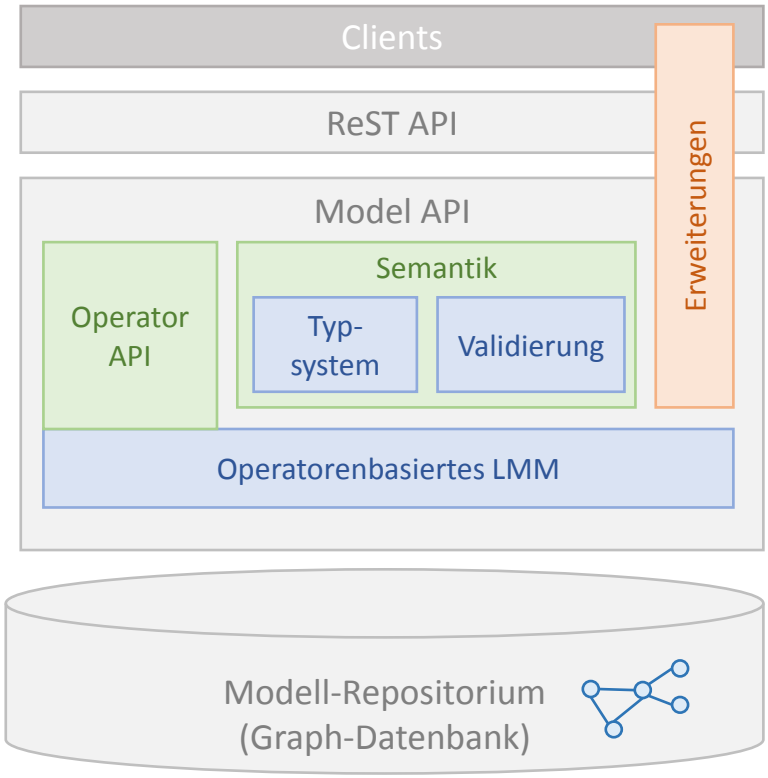

Abbildung 7-2 Architekturbild der Model Workbench

Model API

Um den Zugriff auf die Graph-Datenbank uniform und unabhängig ablaufen zu lassen, agieren die Elemente des LMMs als Wrapper für die jeweiligen Knoten (und Kanten) des Modell-Repositoriums. Die elementaren Änderungen an der Datenbank wurden dabei in separate Kommandos gekapselt. Daneben bilden die Modellelemente aber auch die Grundlage für die Erstellung komplexer Operatoren, die, um ein Element im Repositorium zu ändern, lediglich den entsprechenden Modellelementoperator nutzen müssen. Dadurch können Funktionalitäten wie Undo/Redo oder das

Berechnen der elementaren Änderungen im Falle eines komplexen Operators ebenfalls aus der Wiederverwendung bzw. Aggregation der Werte der Suboperatoren entstehen. Dies führt zu einer erheblichen Reduktion des Aufwandes bei der Erstellung neuer Operatoren.

Die Ausführung eines Operators wird zentral durch das System durchgeführt, wobei jeder Operator in einer eigenen Transaktion abläuft und Suboperatoren innerhalb einer geschachtelten Transaktion ausgeführt werden. Aufgrund dessen wird die Einhaltung des ACID Prinzips durch die verwendete Datenhaltung gewährleistet. Dabei werden Suboperatoren automatisch beim Aufruf des äußeren Operators durchgeführt, falls sie nicht bereits vorher ausgeführt wurden. Somit ist die Entstehung eines Zyklus innerhalb eines Aufrufes eines Operators unproblematisch, da dieser direkt aufgebrochen wird (siehe Kapitel 5).

Um die Historie an Operatoren zu erhalten, sind zusätzlich zwei globale Operatorenstacks vorhanden, die zum einen die ausgeführten und zum anderen die rückgängig gemachten Operatoren vorhalten. Diese werden für die Umsetzung der Undo/Redo Funktionalität und für die Berechnung der Bewertung der Evolution benötigt.

Ein weiterer wichtiger Block innerhalb dieser Schicht ist die Modellierungssemantik (siehe Abschnitt 4.2). Sie beinhaltet auf der einen Seite das Typsystem, das für die Berechnung der Wertebereiche der Eigenschaften der LMM Elemente zuständig ist. Auf der anderen Seite ist die Validierung ein integraler Bestandteil der Modellierungssemantik, da sie an Modellelemente geknüpfte Bedingungen überprüft. Die Validierung wird nach jeder Änderung am Modell durchgeführt und erkennt Brüche in der Konsistenz bzw. Konformität. Diese können mit Hilfe von definierten Vorgehensweisen (Operatoren) in einen validen Zustand aufgelöst werden.

ReST API und Clients

Die beiden obersten Schichten stehen für Clients, die zumeist Editoren darstellen, und für die Anbindung dieser an die Model API in Form der ReST API. Die Oberfläche der *Model Workbench* ist ähnlich zu bekannten IDEs wie Eclipse, Visual Studio und IntelliJ IDEA aufgebaut und kann in Form von Editoren, Sichten und Aktionen erweitert werden.

Erweiterungen

Alle Schichten des Systems sind so ausgelegt, dass sie erweiterbar sind. Besonders die Definition von neuen Operatoren wurde in den Fokus gestellt. So können bei der Erstellung eines Operators Annotationen verwendet werden, die den generischen Wizard steuern und damit den Prozess des Operators festlegen. Ebenso kann das Modellierungsparadigma durch Änderung des Typsystems und der Validierung beeinflusst, sprich verändert, werden. Weiterhin ist, wie oben bereits erwähnt, die Clientseite ebenfalls erweiterbar gehalten, um neue Editoren oder Sichten bereitzustellen. Dabei macht zumeist auch eine Anpassung bzw. Erweiterung des entsprechenden ReST Services Sinn, so dass der Informationsaustausch so gering wie möglich gehalten wird.

7.1.2 Unterstützung im Umgang mit Sprachmustern

Die in Kapitel 2 vorgestellten Sprachmuster haben Einfluss auf viele Modellelemente. Um die Auswirkungen der Sprachmuster dem Modellierer bewusst zu machen, ist es sinnvoll, verschiedene Mengen von Modellelementen aus Abschnitt 4.2 auch dem Benutzer zu visualisieren. Dazu zählen die virtuellen (Definition 4.20) und instanziierten Attribute (Definition 4.21) eines Konzeptes, da mit ihnen die Sprachmuster auf Attribut-Ebene aufgelöst werden. Weiterhin ist eine Darstellung der virtuellen Zuweisungen (Definition 4.24) sinnvoll, um den realen Wert eines Attributes an einem Konzept zu erhalten. Durch diese Unterstützung lässt sich ebenfalls ein Lernprozess hinsichtlich des Einsatzes eines

Musters unterstützen. Die Umsetzung innerhalb eines Modellierungssystems kann unterschiedlich erfolgen. So könnten die verschiedenen Mengen direkt innerhalb eines Editors integriert oder als Sichten realisiert werden (siehe Abbildung 7-3).

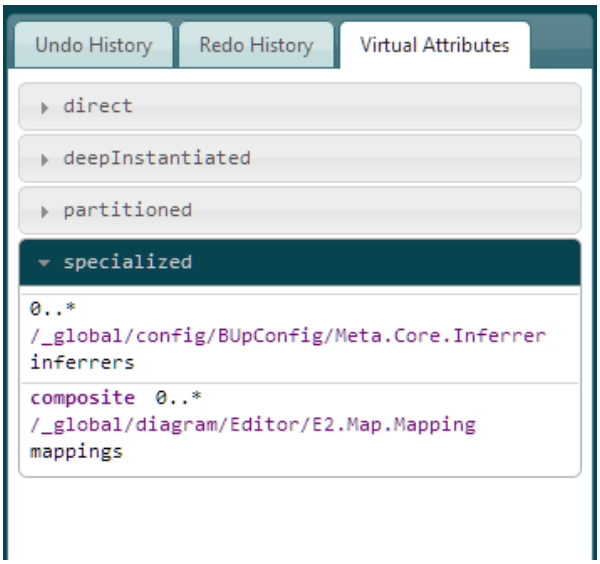

Abbildung 7-3 Sicht der virtuellen Attribute innerhalb der Model Workbench

7.1.3 Rapid Design von Modellierungssprachen

Neben der Unterstützung der Evolution von Meta-Modellen erfüllt die *Model Workbench* auch alle anderen in [23] genannten Patterns eines flexiblen Modellierungssystems (bis auf *Augmented Models)*. Darunter fällt auch die Unterstützung des Rapid Design von domänenspezifischen Modellierungssprachen. Die dabei zugrunde liegenden Konzepte werden in der Dissertation von Bastian Roth [96] im Detail vorgestellt.

Der Ansatz widmet sich im Generellen dem Problem, dass Modelle in den herkömmlichen Ansätzen erst dann erstellt werden können, wenn zuvor die Sprache definiert wurde. Dadurch muss durch den Modellierer eine doppelte Typabstraktion durchgeführt werden, da zumeist initial die Instanz Ebene (z.B. Interview mit einem Domänenexperten) untersucht wird. Folglich muss der Modellierer das Modell vorausdenken und dafür eine Sprache definieren ohne das Modell vorher materialisiert zu haben. Der Ansatz von Roth sieht eine Abkehr von gewohnten Top-down Prinzip vor, dahingehend, dass die verwendete Modellierungssprache aus den vorhandenen Instanz-Modellen generiert und später eventuell mit Hilfe einer Evolution (*Model Evolution*) weiter verfeinert wird. Dadurch können direkt Instanz-Modelle erstellt werden, die später noch valide sind und nicht erst nach der Sprachdefinition ins System eingepflegt werden müssen. Die Forschungsgruppe um Roth wenden diesen Rapid Design Ansatz sowohl auf grafische als auch auf textuelle [99] Modellierungssprachen [97] an.

7.2 Unterstützung iterativer Evolution

Die Evolution einer Sprache und damit auch eines Meta-Modells ist nicht trivial und die Auswirkungen können deshalb zumeist nicht vorher in Gänze überblickt werden. Weiterhin zeigt die Vielfalt an Sprachen, die für eine bestimmte Domäne existieren und deren Anpassung über die Zeit, dass viele Möglichkeiten existieren, um Sprachen auszudrücken oder zu verbessern. Dementsprechend ist die Unterstützung einer iterativen Evolution ein wichtiger Punkt. Deshalb ist in der *Model Workbench* bereits innerhalb der Model API die Undo/Redo Funktionalität verankert. Weiterhin bietet das System

zwei Historien an Operatoren, wobei eine die bereits ausgeführten und die andere die rückgängig gemachten Operatoren beinhaltet. Für jeden Evolutionsschritt in einer Historie können die elementaren Änderungen an jedem Modellelement visualisiert werden, damit die Evolution transparent für den Modellierer abläuft. Dieses Evolutions-Review (Abbildung 7-4) teilt die Änderungen in drei verschiedene Kategorien ein: Änderungen an Eigenschaften (hier sind auch die Referenzen zu anderen Elementen enthalten) und das Hinzufügen (Erstellen) oder Löschen von Kindelementen in der Containerhierarchie. Die Review stellt also das Pendant von Refactoring Reviews, die in modernen IDEs angeboten werden, in der Meta-Modellierung dar.

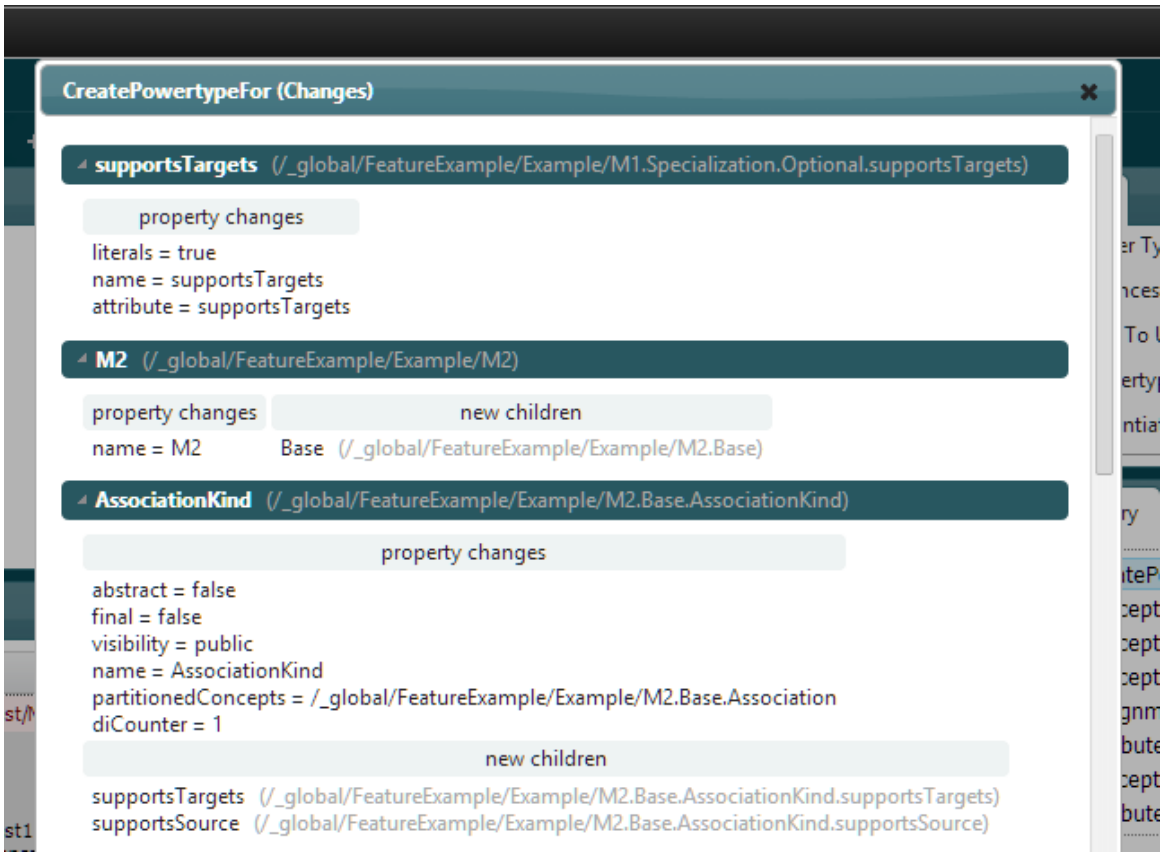

Abbildung 7-4 Review eines Operators

Einen weiteren Punkt stellt das oben bereits erwähnte Validierungskonzept dar. Dabei folgt nach jeder elementaren Änderungen an Eigenschaften des Modells eine Validierungsphase. Wenn die Validierung eine bestimmte Bedingung vorfindet und damit ein Konformitäts- oder Konsistenzbruch vorliegt, dann wird ein Fehler, eine Warnung oder ein Hinweis generiert. Diese können neben einer Beschreibung auch eine Möglichkeit (einen Operator) enthalten, die den Bruch auflöst. Auf diese Weise lassen sich die Operatoren aus Kapitel 5 zweistufig umsetzen. So wird nach einer elementaren Änderung einer Modelleigenschaft ein Fehler erzeugt, der den entsprechenden Operator (und eventuell auch andere) als Quick-Fix anbietet. Damit kann der Modellierer frei wählen, ob er das Standardverhalten wünscht oder seine eigene Migrationsstrategie verfolgt.

Operatoren im Allgemeinen kapseln immer eine Änderungssemantik. Bei einer großen Bibliothek an Operatoren ist es daher essentiell, dass diese Semantik dem Benutzer bekannt ist. Diese Herausforderung hat Lerner ([62], S.84) ebenfalls erkannt. Sie tritt in ähnlicher Form auch bei großen APIs auf. Um ihr zu begegnen, können Beschreibungen (ähnlich JavaDoc) von Operatoren (siehe Abschnitt *Auswirkung* bei den entsprechenden Operatoren) und Strukturierungen der Bibliotheken verwendet werden.

8 Fazit & Ausblick

Dieses Kapitel soll zusammenfassen, wie die eingangs formulierten Herausforderungen aus Abschnitt 1.4 durch diese Arbeit erfüllt sind. Weiterhin soll es aufzeigen inwieweit weiterführende Forschungsansätze aufbauend auf den hier vorgestellten Konzepten anknüpfen können.

8.1 Bezug zu den Herausforderungen

Der **Herausforderung H1** wird hauptsächlich im Kapitel 6 und teilweise im Kapitel 5 begegnet, in denen Operatoren definiert werden, die die Änderung und Einführung von den in Kapitel 2 vorgestellten Sprachmustern unterstützen. Dabei werden die in Herausforderung H1 geforderten Strategien in Form von Operatoren umgesetzt, die auch die Auswirkungen auf andere Sprachmuster berücksichtigen und die Änderungen der abhängigen Modellelemente vollziehen. Dies stellt ein Novum dieser Arbeit im Vergleich zu bisher existierenden Arbeiten dar.

Die Umsetzung der **Herausforderung H2** wird in Kapitel 5 vorgestellt. Darin werden für alle Änderungen der Eigenschaften eines Modellelements, die zu Brüchen hinsichtlich der Konformität der Sprachmustersemantik oder der Inkonsistenz bezüglich der Regeln aus Abschnitt 4.2 führen, elementare Operatoren bereitgestellt. Diese führen den Migrationsprozess durch und bieten damit die Möglichkeit diese Brüche aufzulösen. Die vollständige Lösung dieser Herausforderung im Sinne der sprachbasierten Muster ist eine weitere Innovation dieser Arbeit.

Um der **Herausforderung H3** zu begegnen, wurde das Linguistische Meta-Modell hin zum operatorenbasierten LMM erweitert (siehe Abschnitt 4.1), das bereits die Grundlage für die Unterstützung der iterativen Evolution bereitstellt. Darauf aufbauend wurde die *Model Workbench* und das *Model Evolution* System entwickelt (siehe Kapitel 7), die das LMM umsetzen und zusätzlich verschiedene Methoden liefern, den Modellierer beim Umgang mit sprachbasierten Mustern zu unterstützen und die Evolution eines Meta-Modells zu bewerten.

8.2 Ausblick

Die in dieser Arbeit vorgestellten Konzepte zusammen mit dem Prototyp der *Model Workbench* können als Grundlage für weiterführende Forschungsarbeiten dienen.

Durch den operatorenbasierten transaktionalen Ansatz und dem Modellrepositorium, sind bereits wichtige Bausteine für die Unterstützung der Kollaboration innerhalb der Meta-Modellierung gelegt. Um diese Funktionalität bereitzustellen, könnten ähnlich wie bei Ellis und Gibbs [32] konfligierende elementare Änderungen (Eigenschaften der Modellelemente) festgelegt und für jedes Paar ein Lösungsstrategie definiert werden. Weiterhin wäre für die Unterstützung der Kollaboration eine separate Sicht der Historie sinnvoll, die lediglich die Operatoren des Benutzers vorhält. Eine mögliche Umsetzung wird in [88] vorgestellt.

Neben Instanz-Modellen sind auch typischerweise andere Software-Artefakte von einem Meta-Modell abhängig und können daher von einer Evolution beeinflusst werden [100]. Dabei resultiert eine Änderung im Meta-Modell oftmals in einer händischen Migration oder in einer Neugenerierung der Artefakte (z.B. generierter Code). Dadurch gehen eventuell Änderungen, die manuell an den Artefakten vorgenommen wurden, verloren. Ebenso ist die umgekehrte Synchronisation in manchen Fällen

wünschenswert in der Form, als dass sich die Evolution der Artefakte auch auf die Meta-Modelle auswirkt.

Um eine noch bessere Bewertungsgrundlage für die Evolution zu schaffen, könnte das Review der Evolution ebenfalls erweitert werden. So könnten beispielsweise Änderungen aggregiert werden und anschließend in Form eines Sankey Diagramms visualisiert werden, in dem Abschnitte, die durch die Evolution beeinflusst wurden, in einer Signalfarbe dargestellt werden. Dadurch kann eine grobe Abschätzung der Evolution durch den Modellierer erfolgen. Ein weiterer Ausbaupunkt bieten die Berechnung und Visualisierung von Metriken, die die Modellqualität messen. So gilt zum Beispiel eine Vererbungshierarchie mit mehr als sieben Stufen als sehr tief und sollte eventuell ersetzt oder umgebaut werden [121]. Andere Qualitätskriterien für Meta-Modelle finden sich in [15], die ebenfalls mit in die Unterstützung der Bewertung einfließen könnten.

Durch jeden Operator wird ein Prozess definiert. Innerhalb der *Model Workbench* wird dieser Prozess durch Annotationen festgelegt, die dann den generischen Wizard steuern. Um die Erstellung eines Operators zu erleichtern, könnte in folgenden Forschungsarbeiten die Prozessdefinition mit üblichen Techniken der Prozessmodellierung (z.B. grafische Modellierung eines Operators) besser unterstützt werden. Dabei könnten zum Beispiel die Einführung von verschiedenen Perspektiven (ähnlich zu [58]) dazu führen, eine strukturierte Vorgehensweise bei der Definition zu bieten.

Ein weiterer Ansatzpunkt liefert die Erkennung von komplexen elementaren Evolutionsänderungen, die nicht durch Operatoren ausgeführt wurden und einer vollständigen Migration bedürfen. Diese können zum Beispiel bei der manuellen Änderung eines Meta-Modells außerhalb des Repositoriums auftreten. Dazu müssen Überdeckungen und Auslöschungen erkannt werden, damit die Reihenfolge der Evolutionsoperatoren bestimmt werden kann. Vermolen et al. zeigen einen solchen Ansatz in [114], liefern dabei aber keine Unterstützung für Sprachmuster.

Mit Hilfe der virtuellen und instanziierten Attribute lässt sich nicht nur die Modellierung an sich, sondern auch die Code Generierung unterstützen. Je nachdem welche Sprachmuster in der entsprechenden Zielplattform vorhanden sind, können angepasste Domänenklassen generiert werden, die einen einfachen Umgang erlauben. Damit können sie als Grundlage für die Code Generierung oder den Zugang anderer Systeme auf das Modell-Repositorium dienen. Wenn diese generierten Klassen in der Modellierungsplattform (z.B. die *Model Workbench*) laufen und die entsprechenden Methoden auf die Methoden der Konzepte delegiert werden, bleibt sogar die Verbindung zum Repositorium bestehen und man erhält eine Art interne DSL [38].

Zur Verdeutlichung soll die Umsetzung in Java skizziert werden. Die konkrete Umsetzung bietet allerdings noch weiteren Forschungsbedarf. Da Java im Wesentlichen zwei Sprachebenen unterstützt, muss eine Ebene festgelegt werden, die als Objekt-Ebene fungiert. Als erstes wird jedes darin liegende Konzept, ermittelt, das einen abgeleiteten instanziierten Typ besitzt. Danach muss der abgeleitete instanziierte Typ als Klasse generiert werden. Dabei wird zunächst die Vererbungshierarchie in Klassen aufgebaut und beim Wurzelkonzept begonnen. Jede entstehende Klasse in dieser Hierarchie erhält alle virtuellen Attribute seines entsprechenden Konzepts, die nicht bei der Generalisierung (Klasse) definiert sind. Allerdings müssen in diesem Fall auch nichtsichtbare Attribute einer Instanz eines erweiterten Powertyps beachtet werden. Dies könnte zum Beispiel durch das Werfen einer Exception beim Aufruf des Getters erfolgen.

Nachdem alle Klassen generiert wurden, werden alle oben ermittelten Konzepte der „unteren" Ebene als Objekte angelegt, wobei die Instanz-Spezialisierungen als Objekte der entsprechenden Klasse des abgeleiteten instanziierten Typs erstellt werden. Das spezielle Verhalten der Instanz-Spezialisierung

hinsichtlich des Erbens der Zuweisungswerte des Prototyps, muss zum Beispiel durch Delegation innerhalb der entsprechenden Getter Methoden abgebildet werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

Quellenverzeichnis

[1] Aboulsamh, M.A. and Davies, J. 2010. A Metamodel-Based Approach to Information Systems Evolution and Data Migration. *Proceedings of the 2010 Fifth International Conference on Software Engineering Advances (ICSEA)* (Nice, 2010), 155–161.

[2] Atkinson, C. 1997. Meta-modelling for distributed object environments. *Proceedings of the 1st International Conference on Enterprise Distributed Object Computing (EDOC '97)* (1997), 90–101.

[3] Atkinson, C. and Kühne, T. 2005. Concepts for comparing modeling tool architectures. *Model Driven Engineering Languages and Systems (MoDELS 2005), Lecture Notes in Computer Science*. 3713, (2005), 398–413.

[4] Atkinson, C. and Kühne, T. 2000. Meta-level independent modelling. *International Workshop on Model Engineering at the 14th European Conference on Object-Oriented Programming*. 12, (2000), 16.

[5] Atkinson, C. and Kühne, T. 2002. Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*. 12, 4 (Oct. 2002), 290–321.

[6] Atkinson, C. and Kühne, T. 2001. The essence of multilevel metamodeling. *«UML» 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Lecture Notes in Computer Science*. 2185, (2001), 19–33.

[7] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D. and Zdonik, S. 1992. The Object-Oriented Database System Manifesto. *Building an object-oriented database system*. Morgan Kaufmann Publishers Inc. 1–20.

[8] Backbone.js: *http://backbonejs.org/*. Accessed: 2014-05-12.

[9] Banerjee, J., Chou, H. and Garza, J. 1987. Data model issues for object-oriented applications. *ACM Transactions on Information Systems (TOIS)*. 5, 1 (1987), 3–26.

[10] Becker, S. and Gruschko, B. 2007. A process model and classification scheme for semi-automatic meta-model evolution. *1st Workshop "MDD, SOA und IT-Management" (MSI'07)* (2007), 35–46.

[11] Behm, A., Geppert, A. and Dittrich, K. 1997. On the migration of relational schemas and data to object-oriented database systems. *Proceedings of the 5th International Conference on Re-Technologies in Information System* (Klagenfurt, Austria, 1997), 1333.

[12] Bernstein, P. a., Halevy, A.Y. and Pottinger, R. a. 2000. A vision for management of complex models. *ACM SIGMOD Record*. 29, 4 (Dec. 2000), 55–63.

[13] Bernstein, P.A. 2003. Applying Model Management to Classical Meta Data Problems. *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR)* (2003), 209–220.

[14] Bernstein, P.A. and Melnik, S. 2007. Model Management 2.0: Manipulating Richer Mappings. *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD 07)* (2007), 1 – 12.

[15] Bertoa, M. and Vallecillo, A. 2010. Quality attributes for software metamodels. *Proceedings of the In 13th TOOLS Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2010) (2010)* (2010).

[16] Burger, E. 2008. Metamodel Evolution in the Context of a MOF-Based Metamodeling Infrastructure. Diplomarbeit, Institut für Theoretische Informatik, Universität Karlsruhe (TH).

[17] Burger, E. and Gruschko, B. 2010. A Change Metamodel for the Evolution of MOF-based Metamodels. *Proceedings of Modellierung 2010* (Klagenfurt, Austria, 2010), 285–300.

[18] Cicchetti, A., Di Ruscio, D., Eramo, R. and Pierantonio, A. 2008. Automating co-evolution in model-driven engineering. *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC '08)* (2008), 222–231.

[19] Cicchetti, A., Ruscio, D. Di, Eramo, R., Pierantonio, A. and Informatica, D. 2008. Meta-model Differences for Supporting Model. *2nd Workshop on Model-Driven Software Evolution* (2008).

[20] Cicchetti, A., Ruscio, D. Di and Pierantonio, A. 2009. Managing dependent changes in coupled evolution. *Theory and Practice of Model Transformations, Lecture Notes in Computer Science*. 5563, (2009), 35–51.

[21] Clark, T., Sammut, P. and Willans, J. 2008. *Applied metamodelling: a foundation for language driven development*. CETEVA.

[22] Cook, S., Jones, G., Kent, S. and Wills, A.C. 2007. *Domain-Specific Development with Visual Studio DSL Tools (Microsoft .Net Development)*. Addison-Wesley Longman.

[23] CORREIA, F. and AGUIAR, A. 2013. Patterns of Flexible Modeling Tools. *20th Conference on Pattern Languages of Programs (PLoP 2013)* (Monticello, 2013).

[24] Curino, C.A., Moon, H.J., Deutsch, A. and Zaniolo, C. 2010. Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. *Proceedings of the VLDB Endowment*. 4, 2 (2010), 117–128.

[25] Curino, C.A., Moon, H.J. and Zaniolo, C. 2008. Graceful database schema evolution: the prism workbench. *Proceedings of the VLDB Endowment*. 1, 1 (2008), 761–772.

[26] Dahchour, M., Pirotte, A. and Zimányi, E. 2005. Generic relationships in information modeling. *Journal on Data Semantics IV*. 3730, (2005), 1–34.

[27] Dahchour, M., Pirotte, A. and Zimányi, E. 2002. Materialization and its metaclass implementation. *IEEE Transactions on Knowledge and Data Engineering*. 14, 5 (2002), 1078–1094.

[28] Demuth, A. 2011. Cross-layer modeler: a tool for flexible multilevel modeling with consistency checking. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. (2011), 452–455.

[29] Demuth, A., Lopez-Herrejon, R. and Egyed, A. 2013. Co-evolution of Metamodels and Models through Consistent Change Propagation. *16th International Conference on Model Driven Engineering Languages and Systems* (Miami, Florida (USA), 2013), 14–21.

[30] Deridder, D., Pierantonio, A., Schätz, B. and Tamzalit, D. 2011. Models and Evolution - ME2010. *Models in Software Engineering*. 6627, (2011), 180–183.

[31] Edapt - Framework for Ecore model adaptation and instance migration: *http://www.eclipse.org/edapt/*. Accessed: 2014-03-31.

[32] Ellis, C. a. and Gibbs, S.J. 1989. Concurrency control in groupware systems. *Proceedings of the 1989 ACM SIGMOD international conference on Management of data (SIGMOD '89)*. 18, 2 (Jun. 1989), 399–407.

[33] Elmasri, R. and Navathe, S. 2010. *Fundamentals of Database Systems*. Prentice Hall International.

[34] Favre, J.M. 2003. Meta-model and model co-evolution within the 3D software space. *Proceedings of the international Workshop ELISA at ICSM* (2003), 98.

[35] Ferrandina, F., Meyer, T., Zicari, R., Ferran, G. and Madec, J. 1995. Schema and Database Evolution in the O2 Object Database System. *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB)* (Zürich, Switzerland, 1995), 170–181.

[36] Flanagan, D. 2011. *JavaScript: The Definitive Guide (Definitive Guides)*. O'Reilly Media, Inc.

[37] Florez, H., Sánchez, M., Villalobos, J. and Vega, G. 2012. Coevolution assistance for enterprise architecture models. *6th International Workshop on Models and Evolution*. (2012), 27–32.

[38] Fowler, M. and Parsons, R. 2010. *Domain-Specific Languages*. Addison-Wesley Longman.

[39] France, R. and Rumpe, B. 2007. Model-driven development of complex software: A research roadmap. *Proceedings of the 2007 Future of Software Engineering (FOSE '07)* (Washington, DC, USA, 2007), 37–54.

[40] Frank, U. 2013. Domain-Specific Modeling Languages – Requirements Analysis and Design Guidelines. *Domain Engineering: Product Lines, Conceptual Models, and Languages*. I. Reinhartz-Berger, A. Sturm, T. Clark, Y. Wand, S. Cohen, and J. Bettin, eds. Springer. 133–157.

[41] Frank, U. 2012. Thoughts on classification/instantiation and generalisation/specialisation. ICB-Research, Report No. 53, Universität Duisburg Essen.

[42] Gamma, E. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional.

[43] Greenfield, J. 2001. *UML profile for EJB*.

[44] Gruschko, B. 2006. Towards structured revisions of metamodels and semi-automatic model migration. *Eclipse Modeling Symposium at Eclipse Summit Europe* (2006).

[45] Gruschko, B., Kolovos, D.S. and Paige, R.F. 2007. Towards Synchronizing Models with Evolving Metamodels. *International Workshop on Model-Driven Software Evolution held with the ECSMR* (2007).

[46] Guerrini, G. and Mesiti, M. 2008. X-Evolution: A Comprehensive Approach for XML Schema Evolution. *Proceedings of the 19th International Conference on Database and Expert Systems Applications (DEXA '08)* (Sep. 2008), 251–255.

[47] Guerrini, G., Mesiti, M. and Sorrenti, M. 2007. XML schema evolution: Incremental validation and efficient document adaptation. *Database and XMLTechnologies, Lecture Notes in Computer Science*. 4704, (2007), 92–106.

[48] Hartung, M. 2011. *Evolution von Ontologien in den Lebenswissenschaften*. PhD Thesis, Fakultät für Mathematik und Informatik, Universität Leipzig.

[49] Hartung, M., Terwilliger, J. and Rahm, E. 2011. Recent advances in schema and ontology evolution. *Schema Matching and Mapping, Data-Centric Systems and Applications*. 149–190.

[50] Heinisch, C., Müller-Hofmann, F. and Goll, J. 2011. Vererbung und Polymorphie. *Java als erste Programmiersprache*. Teubner. 364–440.

[51] Henderson-Sellers, B. and Gonzalez-Perez, C. 2005. The rationale of powertype-based metamodelling to underpin software development methodologies. *Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling*. 43, (2005), 7–16.

[52] Herrmannsdoerfer, M. 2011. *Evolutionary Metamodeling*. PhD Thesis, Fakultät für Informatik, Technische Universität München.

[53] Herrmannsdoerfer, M., Benz, S. and Juergens, E. 2009. COPE - Automating Coupled Evolution of Metamodels and Models. *Proceedings of the 23rd European Conference on (ECOOP)* (2009), 52–76.

[54] Herrmannsdoerfer, M., Benz, S. and Juergens, E. 2008. COPE: a language for the coupled evolution of metamodels and models. *Proceedings of the 1st International Workshop on Model Co-Evolution and Consistency Management.* (2008).

[55] Herrmannsdoerfer, M. and Koegel, M. 2010. Towards a generic operation recorder for model evolution. *Proceedings of the 1st International Workshop on Model Comparison in Practice*. (2010), 76–81.

[56] Herrmannsdoerfer, M., Vermolen, S. and Wachsmuth, G. 2011. An extensive catalog of operators for the coupled evolution of metamodels and models. *Software Language Engineering, Lecture Notes in Computer Science*. 6563, (2011), 163–182.

[57] Hoisl, B., Hu, Z. and Hidaka, S. 2010. Towards Co-Evolution in Model-driven Development via Bidirectional Higher-Order Transformation. *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development – MODELSWARD 2014* (Lissabon, Portugal, 2010).

[58] Jablonski, S. and Bussler, C. 1996. *Workflow management: modeling concepts, architecture and implementation*. International Thomson Computer Press.

[59] Jablonski, S., Volz, B. and Dornstauder, S. 2009. On the Implementation of Tools for Domain Specific Process Modelling. *Proceedings of the 4th International Conference on the Evaluation of Novel Approaches to Software Engineering (ENASE)* (Milan, Italy, 2009), 109–120.

[60] Jahn, M., Roth, B. and Jablonski, S. 2014. Instance Specialization – a Pattern for Multi-level Meta Modelling. *1st International Workshop on Multi-Level Modelling (MULTI 2014), in conjunction with MODELS 2014* (Valencia, Spain, 2014), 23–32.

[61] Jahn, M., Roth, B. and Jablonski, S. 2013. Remodeling to Powertype Pattern. *Proceedings of the Fifth International Conferences on Pervasive Patterns and Applications (PATTERNS 2013)* (2013), 59– 65.

[62] Java EE at a Glance: *http://www.oracle.com/technetwork/java/javaee/overview/index.html*. Accessed: 2014-05-12.

[63] jQuery- with less, do more: *http://jquery.com/*. Accessed: 2014-05-12.

[64] Karagiannis, D. and Kühn, H. 2002. Metamodelling platforms. *Proceedings of the 3rd International Conference onEC-Web* (Aix-en Provence, France, 2002), 182–196.

[65] Kelly, S. and Tolvanen, J.-P. 2008. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons.

[66] Kleppe, A.G., Warmer, J. and Bast, W. 2003. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc.

[67] Kuehne, T. and Schreiber, D. 2007. Can programming be liberated from the two-level style: multi-level programming with deepjava. *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA '07)*. 42, 10 (2007), 229 – 244.

[68] Kühne, T. and Steimann, F. 2004. Tiefe charakterisierung. *Proceedings of Modellierung 2004* (2004), 109–119.

[69] Kurtev, I., Bézivin, J., Jouault, F. and Valduriez, P. 2006. Model-based DSL frameworks. *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (OOPSLA '06)* (2006), 602–616.

[70] Lämmel, R. and Lohmann, W. 2001. Format evolution. *Proceedings of the 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)* (2001), 113–134.

[71] Lieberman, H. 1986. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. *Conference proceedings on Object-oriented programming systems, languages and applications (OOPLSA '86)* (1986), 214–223.

[72] Melnik, S., Rahm, E. and Bernstein, P. 2003. Rondo: A programming platform for generic model management. *International conference on Management of data ACM SIGMOD* (2003), 193–204.

[73] Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R. and Jazayeri, M. 2005. Challenges in Software Evolution. *Proceedings of the 8th International Workshop on Principles of Software Evolution (IWPSE'05)* (2005), 13–22.

[74] Mesiti, M., Celle, R., Sorrenti, M. and Guerrini, G. 2006. X-Evolution: A system for XML schema evolution and document adaptation. *Advances in Database Technology - EDBT 2006, Lecture Notes in Computer Science*. 3896, (2006), 1143–1146.

[75] Moreira, A. and Rossi, G. 2004. An introduction to UML profiles. *UP-GRADE European Jornal for the Informatics Professional*. 5, 2 (2004), 5–13.

[76] Moro, M.M., Malaika, S. and Lim, L. 2007. Preserving XML queries during schema evolution. *Proceedings of the 16th international conference on World Wide Web - WWW '07* (New York, New York, USA, 2007), 1341–1342.

[77] Neo4j - The World's Leading Graph Database: *http://www.neo4j.org/*. Accessed: 2014-05-12.

[78] Neumayr, B., Schrefl, M. and Thalheim, B. 2011. Modeling techniques for multi-level abstraction. *The Evolution of Conceptual Modeling, Lecture Notes in Computer Science*. 6520, (2011), 68–92.

[79] Nguyen, G. and Rieu, D. 1989. Schema evolution in object-oriented database systems. *Data & Knowledge Engineering*. 4, 1 (1989), 43–67.

[80] Noy, N.F., Chugh, A., Liu, W. and Musen, M.A. 2006. A Framework for Ontology Evolution in Collaborative Environments. *Proceedings of the 5th International Semantic Web Conference, ISWC 2006, Lecture Notes in Computer Science* (2006), 544–558.

[81] Noy, N.F. and Klein, M. 2004. Ontology Evolution: Not the Same as Schema Evolution. *Knowledge and Information Systems*. 6, 4 (2004), 428–440.

[82] Odell, J. 1994. Power types. *Journal of Object-Oriented Programming*. 7, 2 (1994), 8–12.

[83] Odell, J.J. 1998. *Advanced object-oriented analysis and design using UML*. SIGS.

[84] OMG 2011. Business Process Model and Notation (BPMN). *Available on: http://www.omg.org/spec/BPMN/2.0-Abstract-BPMN.pdf*. January (2011).

[85] OMG 2014. OMG Meta Object Facility (MOF) Core Specification. *Available on: http://www.omg.org/spec/MOF/2.4.2/*. 2, April (2014).

[86] OMG 2011. Unified Modeling Language (OMG UML)-Infrastructure. *Available on: http://www. omg. org/spec/UML/2.4.1*. August (2011).

[87] Pirotte, A., Zimanyi, E., Massart, D. and Yakusheva, T. 1994. Materialization : a powerful and ubiquitous pattern abstraction. *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. (1994), 630–641.

[88] Prakash, A. and Knister, M. 1994. A framework for undoing actions in collaborative systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*. 1, 4 (1994), 273–280.

[89] Rahm, E. and Bernstein, P.A. 2006. An Online Bibliography on Schema Evolution. *ACM SIGMOD Record*. 35, 4 (2006), 30–31.

[90] Rocco, J. Di, Iovino, L. and Pierantonio, A. 2012. Bridging state-based differencing and co-evolution. *6th International Workshop on Models and Evolution*. (2012), 15–20.

[91] Ronström, M. 2000. On-line schema update for a telecom database. *Proceedings of the 16th International Conference on Data Engineering* (San Diego, CA, 2000), 329–338.

[92] Rose, L.M. 2011. *Structures and Processes for Managing Model-Metamodel Co-evolution*. PhD Thesis, Department of Computer Science, University of York.

[93] Rose, L.M., Kolovos, D.S., Paige, R.F. and Polack, F.A.C. 2010. Model migration with epsilon flock. *Theory and Practice of Model Transformations Lecture Notes in Computer Science*. 6142, (2010), 184–198.

[94] Rose, L.M., Paige, R.F., Kolovos, D.S. and Polack, F.A. 2009. An analysis of approaches to model migration. *In Proceedings of Models and Evolution (MoDSE-MCCM) Workshop*. (2009), 6–15.

[95] Rose, L.M., Paige, R.F., Kolovos, D.S. and Polack, F.A.C. 2008. Constructing models with the human-usable textual notation. *Model Driven Engineering Languages and Systems Lecture Notes in Computer Science*. 5301, (2008), 249–263.

[96] Roth, B. *Beispielgetriebene Entwicklung Domänenspezifischer Modellierungssprachen*. PhD Thesis, Fakultät für Mathematik, Physik und Informatik, Universität Bayreuth.

[97] Roth, B., Jahn, M. and Jablonski, S. 2013. A Method for Directly Deriving a Concise Meta Model from Example Models. *Proceedings of the Fifth International Conferences on Pervasive Patterns and Applications (PATTERNS 2013)* (2013), 52–58.

[98] Roth, B., Jahn, M. and Jablonski, S. 2012. IT-as-a-Service for Building Virtual Research Environments. *Proceedings of the 2rd International Conference on Cloud Computing and Service Science* (Porto, Portugal, 2012).

[99] Roth, B., Jahn, M. and Jablonski, S. 2013. On the way of bottom-up designing textual domain-specific modelling languages. *Proceedings of the 13th Workshop on Domain-Specific Modeling (SPLASH 2013, ACM)* (2013), 51–56.

[100] Ruscio, D. Di 2013. Managing the Coupled Evolution of Metamodels and Textual Concrete Syntax Specifications. *Proceedings of the 39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)* (Santander, Spain, 2013), 114–121.

[101] Ruscio, D. Di 2011. What is needed for managing co-evolution in MDE? *Proceedings of the 2nd International Workshop on Model Comparison in Practice*. (2011), 30–38.

[102] Sencha Ext JS: *http://www.sencha.com/products/extjs/*. Accessed: 2014-05-12.

[103] Shneiderman, B. and Thomas, G. 1982. An Architecture for Automatic Database System Conversion. *ACM Transactions on Database Systems (TODS)*. 7, 2 (1982), 235–257.

[104] Singh, G.B. 1995. Single versus multiple inheritance in object oriented programming. *ACM SIGPLAN OOPS Messenger*. 6, 1 (Jan. 1995), 30–39.

[105] Sprinkle, J.M. 2003. *Metamodel driven model migration*. PhD Thesis, Faculty of the Graduate School, Vanderbilt University.

[106] Sprinkle, J.M. and Karsai, G. 2004. A domain-specific visual language for domain model evolution. *Journal of Visual Languages & Computing*. 15, 3-4 (2004), 291–307.

[107] Stachowiak, H. 1974. *Allgemeine Modelltheorie*. Springer.

[108] Stahl, T., Völter, M., Efftinge, S. and Haase, A. 2007. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Dpunkt Verlag.

[109] Steinberg, D., Budinsky, F., Merks, E. and Paternostro, M. 2008. *EMF: eclipse modeling framework*. Addison-Wesley Longman.

[110] Stojanovic, L. and Maedche, A. 2002. User-driven ontology evolution management. *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web Lecture Notes in Computer Science*. 2473, (2002), 285–300.

[111] Su, H., Kramer, D., Chen, L., Claypool, K. and Rundensteiner, E. a. 2001. XEM: managing the evolution of XML documents. *Proceedings of the 11th International Workshop on Research Issues in Data Engineering (RIDE 2001)* (Heidelberg, Germany, 2001), 103–110.

[112] Terwilliger, J.F., Bernstein, P.A. and Unnithan, A. 2010. Worry-Free Database Upgrades: Automated Model-Driven Evolution of Schemas and Complex Mappings. *Proceedings of the International Conference on Management of Data (ACM SIGMOD 2010)* (2010), 1191–1194.

[113] Tresch, M. and Scholl, M.H. 1992. Meta Object Management and its Application to Database Evolution. *Proceedings of the 11th International Conference on the Entity-Relationship Approach, Lecture Notes in Computer Science Volume 645* (Karlsruhe, Germany, 1992), 299–321.

[114] Vermolen, S.D., Wachsmuth, G. and Visser, E. 2012. Reconstructing complex metamodel evolution. *Software Language Engineering, Lecture Notes in Computer Science*. 6940, (2012), 201–221.

[115] Voelter, M. 2013. Language and IDE modularization, extension and composition with MPS. *Generative and Transformational Techniques in Software Engineering IV, Lecture Notes in Computer Science*. 7680, (2013), 383–430.

[116] Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L., Visser, E. and Wachsmuth, G. 213AD. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. Marcus Voelter.

[117] Volz, B. 2011. *Werkzeugunterstützung für methodenneutrale Metamodellierung*. PhD Thesis, Fakultät für Mathematik, Physik und Informatik, Universität Bayreuth.

[118] Volz, B. and Jablonski, S. 2010. Towards an open meta modeling environment. *Proceedings of the 10th Workshop on Domain-Specific Modeling - DSM '10* (New York, New York, USA, Oct. 2010), 1.

[119] Wachsmuth, G. 2007. Metamodel adaptation and model co-adaptation. *ECOOP 2007 – Object-Oriented Programming, Lecture Notes in Computer Science*. 4609, (2007), 600–624.

[120] Welcome to TypeScript: *http://www.typescriptlang.org/*. Accessed: 2014-05-12.

[121] When to Use Inheritance: 2012. *http://msdn.microsoft.com/en-us/library/27db6csx(v=vs.90).aspx*. Accessed: 2012-09-26.

[122] Xtext Language Development made easy!: 2014. *https://www.eclipse.org/Xtext/*. Accessed: 2014-05-02.

[123] Zengler, B., Hahn, J., Rupp, C., Jeckle, M. and Queins, S. 2007. *UML 2 glasklar: Praxiswissen für die UML-Modellierung und -Zertifizierung*. Hanser Fachbuchverlag.