Ein Rahmenwerk für operationale Spezifikationssprachen

Sprachrealisierung und Ausführung mittels Graphtransformationen

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

von
Diplom-Informatiker
Erhard Herbert Weinell, geborener Schultchen

aus Essen

Berichter: Universitätsprofessor em. Dr.-Ing. M. Nagl
Universitätsprofessor Dr. M. Jarke

Tag der mündlichen Prüfung: 10. März 2011

D 82 (diss. RWTH Aachen University, 2011)

Kurzfassung

Die modellgetriebene Softwareentwicklung lebt von der Grundidee, komplexe softwaretechnische Sachverhalte kompakt aber dennoch präzise zu beschreiben. Dabei erstellt der Entwickler zunächst eine abstrakte Spezifikation eines Softwaresystems die sich an fachlichen Konzepten der jeweiligen Anwendungsdomäne orientiert. Details zur Umsetzung dieser Spezifikation in Quellcode, wie etwa die zu verwendende Ausführungsplattform, werden erst in späteren Entwicklungsphasen festgelegt. Auf dieser Basis kann die spätere Umsetzung in ein lauffähiges System in wesentlichen Teilen automatisiert erfolgen. Für den Entwickler ergibt sich aus dem hohen Abstraktionsgrad der Spezifikation ein echter Mehrwert, der sich insbesondere in einer kürzeren Entwicklungszeit sowie einer leichteren Wartbarkeit des spezifizierten Systems niederschlägt.

Zur Durchsetzung der modellgetriebenen Softwareentwicklung sind zwei Faktoren entscheidend, die jeweils durch geeignete Entwicklungswerkzeuge und Infrastrukturlösungen abgedeckt werden müssen: Eine geeignete *Beschreibungsform* wird benötigt, die sowohl dem Kompaktheits- als auch dem Präzisionsanspruch gerecht wird. Nur auf diese Weise kann ein Entwickler eine verdichtete und für weitere Projektbeteiligte leicht verständliche Spezifikation erstellen. Ferner ist die weitere *Verarbeitung* der erstellten Spezifikationen zu berücksichtigen, die entweder generativ durch Ableitung von Quellcode, oder interpretativ als Konfiguration einer Programmbibliothek realisiert werden kann. Auf diese Weise lassen sich sowohl statische Anteile einer Spezifikation wie etwa dem Datenschema, als auch dynamische Anteile wie die Analyse und Veränderung entsprechender Instanzdaten verarbeiten.

Abseits spezieller Anwendungsgebiete und Forschungslösungen konnte sich bislang allerdings nur die Modellierung statischer Anteile in der Praxis etablieren. Dies liegt zum Teil an der Notwendigkeit fallspezifische Ausdrucksmittel zur Verhaltensspezifikation definieren zu müssen um den erwünschten Abstraktionsgrads zu wahren. Die Abbildung dieser speziellen Ausdrucksmittel, entweder auf Quellcode einer Programmiersprache oder einen Interpreter, gestaltet sich jedoch häufig komplexer als im statischen Fall. Der dadurch entstehende Realisierungsaufwand lässt die Spezifikation von Systemverhalten als zu kostspielig und somit nicht rentabel erscheinen, weshalb hierzu häufig auf herkömmliche Programmierung zurückgegriffen wird.

Im Rahmen der vorliegenden Arbeit wird ein Ansatz vorgestellt um die Realisie-

rung verhaltensorientierter Spezifikationssprachen zu unterstützen. Zentraler Aspekt ist hierbei eine sogen. *Kernsprache*, die als kompakte Beschreibungsform über dem Abstraktionsniveau klassischer Programmiersprachen bereitgestellt wird. Konzeptionell basiert diese Sprache auf den im softwaretechnischen Umfeld langjährig erprobten Graphgrammatiken. Da jedoch unmöglich Anforderungen aller denkbaren Anwendungsszenarien bei der Ausgestaltung dieser Kernsprache berücksichtigt werden können, wird insbesondere eine leichte Anpassbarkeit und Erweiterbarkeit sichergestellt. So sind die angebotenen Sprachkonstrukte strikt im Hinblick auf ihre Zuständigkeiten getrennt, so dass Erweiterungen nach Möglichkeit lokal, d.h. mit möglichst wenigen Ergänzungen der ursprünglichen Definition umgesetzt werden können. Zudem wird ein breites Spektrum an möglichen Datenstrukturen unterstützt, die Graphgrammatiken auf komplexen Graphmodellen nutzbar machen.

Die der Kernsprache zugeordnete *Ausführungsmaschine* lässt sich leicht an fallspezifische Gegebenheiten, wie etwa der zugrundeliegenden Datenzugriffsschicht anpassen. Als Abstraktionsebene kommt hierbei ein nicht-standard Datenbanksystem für graphbasierte Daten zum Einsatz das sich zum Adaptieren bestehender Plattformtechnologien nutzen lässt. Auch wird hierbei die technische Erweiterbarkeit der Kernsprache direkt berücksichtigt: So lassen sich Spracherweiterungen auf die Kernsprache zurückführen, um die bestehende technische Infrastruktur nutzen zu können. Alternativ kann auch die bestehende Implementierung der Ausführungsmaschine leicht erweitert werden.

Um die Werkzeugunterstützung von Spezifikationssprachen schneller entwickeln zu können ist eine leichte Nutzbarkeit der Kernsprache zu gewährleisten. Hierzu wird eine Möglichkeit angeboten, durch automatisierte Transformationen eine verhaltensorientierte Spezifikationssprache in die bereitgestellte Kernsprache zu übersetzen. Dieser Übersetzungsformalismus berücksichtigt Spezifika der Kernsprache um seine kompakte und intuitive Nutzung zu fördern. Dabei folgt die formulierte Übersetzung zwar einem bestimmten Standardverhalten, das jedoch fallweise durch den Entwickler mit eigenen Festlegungen überladen werden kann.

Die vorliegende Arbeit begegnet somit der Notwendigkeit, verhaltensorientierte Spezifikationen durch spezifische Ausdrucksmittel rasch – und sei es zunächst für eine prototypische Erprobung – nutzbar zu machen. Hierfür bietet sich die vorgeschlagene Kernsprache an, da sie auf vergleichbarem Abstraktionsniveau angesiedelt ist wie die hiermit zu realisierenden Spezifikationssprachen. Prototypische Entwicklungs- und Ausführungswerkzeuge erleichtern zusätzlich ihre Nutzung. Als Ergebnis entsteht somit ein Werkzeugsatz, der die Realisierung und Erprobung komplexer operationaler Spezifikationssprachen wirksam unterstützt.

Danksagung

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Die fortschreitende Durchdringung moderner Softwareentwicklungsprozesse mit modellgetriebenen und generativen Ansätzen führt zu einer stetigen Verlagerung des programmiertechnischen Fokus. Dabei ermöglichen *Spezifikationssprachen* eine im Entwicklungsprozess früh gelagerte Systembeschreibung auf *hohem Abstraktionsniveau* [Sel03] und aus *unterschiedlichen Perspektiven* [GA02], die eine klassische Implementierungsphase zum Teil ersetzen kann. In modellgetriebenen Prozessen werden die spezifizierten Modelle stattdessen möglichst *automatisiert* in Softwareprodukte überführt, wodurch Redundanz in der Entwicklung vermieden und Konsistenz mit Entwurfsdokumenten sichergestellt wird.

Üblicherweise zielt die modellgetriebene Softwareentwicklung auf *statische Sachverhalte* ab, wie beispielsweise Architektur und Datenstrukturen eines Systems. Diese Aspekte lassen sich mit mittlerweile verbreiteten Technologien teilweise vollautomatisch bis hin zu Softwarefragmenten überführen. Beispielsweise kann so eine Persistenzschicht bereitgestellt werden, die zum Abgleich von Instanzdaten mit relationalen Datenbanktabellen [Amb03, Kap. 14] lediglich eine abstrakte, technikunabhängige Beschreibung des Datenmodells verwendet.

Die modellhafte Darstellung *dynamischer Anteile*, also das Verhalten eines Systems, stellt dagegen einen bislang weniger und vornehmlich zu Analyse- und Dokumentationszwecken berücksichtigten Aspekt dar. Grund für diese Einschränkung ist primär die wenig verbreitete Werkzeugunterstützung, die eine automatische Überführung einer Verhaltensspezifikation in ein ausführbares System ermöglicht. Zumdem erfordert die Verhaltensmodellierung ein hohes Maß an individueller Anpassbarkeit: So müssen Spezifikationssprachen zum Erreichen des erwünschten Abstraktionsgrads an die jeweilige *Problemdomäne* angepasst werden können, ferner müssen Verhaltensspezifikationen leicht auf spezifische *technische Plattformen* abgebildet werden können. Insbesondere im Forschungsbereich der *Graphgrammatiken* existieren zahlreiche Werkzeuge um Verhaltensbeschreibungen in lauffähige Software zu übersetzen. Diese Werkzeuge bieten jedoch gegenwärtig nicht die erforderliche Flexibilität um die häufig erforderlichen Anpassungen an spezielle Anwendungsfälle zu ermöglichen. Die Neuentwicklung fallspezifisch ausgerichteter Spezifikationssprachen einschließlich der benötigten Ausführungswerkzeuge ist aufgrund der hohen Realisierungskomplexität jedoch nicht wirtschaftlich möglich.

Im Rahmen der vorliegenden Arbeit wird ein Ansatz entwickelt um die *Realisierung verhaltensorientierter Spezifikationssprachen* zur modellgetriebenen Softwareentwicklung zu unterstützen. Aufbauend auf dem hier vorgestellten Ansatz wird der Aufwand zur Neuentwicklung und Adaption von Ausführungswerkzeugen dadurch reduziert, dass eine grundlegende *Ausführungsplattform* geschaffen und mittels *Werkzeugunterstützung* für Entwickler leicht nutzbar gemacht wird. Zentraler Aspekt ist hierbei eine sogen. *Kernsprache* die eine Grundlage für neu entwickelte Ausführungswerkzeuge darstellt. Als Ergebnis können anwendungsspezifische Spezifikationssprachen zur Modellierung dynamischer Systemeigenschaften schneller in die Praxis umgesetzt werden, wodurch insbesondere die Zeitspanne zu einer ersten prototypbasierten Erprobung reduziert wird.

Im Verlauf dieses ersten Kapitels wird zunächst der aktuelle Stand modellgetriebener Softwareentwicklung skizziert. Daran wird das Problemfeld sichtbar, anwendungsfallspezifische Lösungen zur Ausführung einer Spezifikationssprache bereitstellen zu müssen. Die dadurch entstehenden Anforderungen werden mit existierenden Technologien in Relation gesetzt, woraus sich der Lösungsansatz der vorliegenden Arbeit entwickelt.

1.1 Modellgetriebene Softwareentwicklung

Die heutzutage zunehmend praktizierte *modellgetriebene Softwareentwicklung* (model-driven software engineering, MDSE) ist die konsequente Fortentwicklung verschiedener historischer Modellierungsansätze. Hierzu zählen neben der Strukturierten Analyse (SA) [Mar78] zur Darstellung von Prozessabläufen insbesondere Konzepte der Ära objektorientierter Systemgestaltung, bspw. der *Object Modeling Technology* (OMT) [RBP+90]. Die MDSE verknüpft dabei im Wesentlichen die *Abstraktion* von implementierungsnahen Sachverhalten mit der *formalen Spezifikation* von Softwaresystemen und häufig der *Visualisierung* in Form von Diagrammen. Ziel der MDSE ist eine zumindest teilweise Automatisierung der Softwareentwicklung durch Modelle und entsprechende Transformationswerkzeuge, mit denen bspw. Teile eines Softwaresystems automatisch generiert werden können.

1.1.1 Hintergrund der modellgetriebenen Softwareentwicklung

Verschiedene historische Modellierungsansätze als Vorläufer der MDSE wie o.g. SA dienen zur grobgranularen Beschreibung eines Sachverhalts. Auch mittels SA werden zwar *Modelle* eines Systems konstruiert, diese dienen jedoch allenfalls zur Dokumentation auf hohem Abstraktionsniveau und sind aufgrund ihres reinen Daten-

flusscharakters kaum bis hin zu einer Implementierung übertragbar. Dem gegenüber stellen Entity-Relationship-Diagramme [Che76] bereits ein durchweg präzises Mittel zur Datenstrukturbeschreibung bereit, das hauptsächlich Anwendung im Bereich der Datenbankmodellierung gefunden hat. Die ER ist jedoch auf den Aspekt Datenmodellierung beschränkt und bietet kaum Unterstützung für weitergehende Fragestellungen, insbesondere auf dem Umfeld der objektorientierten Systemgestaltung. Neben den genannten visuellen Spezifikationssprachen existieren zudem textuelle Ansätze wie Z [ASM80] sowie mathematische Algebren wie etwa Communicating-Sequential-Processes [Hoa80]. Aufgrund ihres teils sehr hohen Abstraktionsgrads sind solch mathematischen Spezifikationen jedoch kaum hinreichend automatisiert in lauffähige Softwaresysteme überführbar, was den Nutzen auf die Analyse einer Systembeschreibung reduziert.

Obwohl also bereits in frühen Jahren der Softwaretechnik Methoden zur modellbasierten Beschreibung von Softwaresystemen verfügbar waren, sind diese jedoch allenfalls als Entwurfsmittel oder zur Dokumentation verwendet worden. Hieraus ergeben sich gravierende Nachteile wenn im weiteren Entwicklungsverlauf größere Änderungen an der bestehenden Software vorgenommen werden, da diese faktisch nicht zurück in das zuvor ausgearbeitete Modell übernommen werden. Modell und Produkt beginnen daher rasch zu divergieren wodurch das Modell zunehmend an Wert verliert.

Aufbauend auf abstrakten aber dennoch technisch präzisen Beschreibungsformen strebt die MDSE eine gesteigerte Automatisierung in allen Phasen eines Softwareentwicklungsprozesses an. Ziel ist es, Modelle schrittweise und strukturiert bis zu einem lauffähigen Softwareprodukt zu verfeinern und letztlich in ein ausführbares System zu transformieren. Bestandteil dieser Transformationen ist schablonenartig festgehaltenes Wissen, wie Konstrukte aus einer höheren Beschreibungsebene auf die nächst tiefere abzubilden sind. Der finale Verarbeitungsschritt erfolgt häufig durch *Codegenerierung* für eine herkömmliche und auf der gegebenen Plattform ausführbaren Programmiersprache. Das hierbei teilweise angestrebte Idealbild besteht darin, den generierten Code nicht weiter durch Entwickler verändern zu lassen, sondern allein auf Modellebene zu arbeiten. Von den Modellierungsspektren nach Brown steht somit das Prinzip „The model *is* the code“ [Bro04] im Vordergrund einiger MDSE-Ausprägungen. Zwei in der Praxis verbreitete und in dieser Hinsicht konträre Ansätze werden im Folgenden kurz vorgestellt.

1.1.2 Ausprägungen der MDSE

Das soweit dargestellte Paradigma der MDSE ist schwer realisierbar, da es in seiner direkten Form entweder sehr *detaillierte Spezifikationen* mit in etwa dem gleichen Detail-

grad wie herkömmlicher Quellcode erfordern würde. Hierdurch wird der gewünschte Abstraktionsgrad jedoch nicht erreicht, sondern allenfalls eine alternative Darstellung des Quellcodes eingeführt. Auch könnte die benötigte Detailfunktionalität durch hochsprachliche Konstrukte geeignet verkapselt werden, was allerdings fallspezifisch erfolgen müsste. Die beiden im Folgenden behandelten MDSE-Ausprägungen umgehen diese Problematik insofern, als dass in praktischen Anwendungen hauptsächlich Strukturen, nicht jedoch Verhalten eines Systems spezifiziert und bis zu fertigen Softwarefragmenten verfeinert werden. Systemverhalten wird als „Anwendungslogik“ stattdessen händisch in Form von Quellcode ergänzt. Somit bleibt ein hohes Maß an herkömmlicher Programmierung abseits der modellgetrieben entwickelten Anteile erhalten.

Model-Driven Architecture

Die *Model-Driven Architecture* (MDA) [KWB03] bezeichnet die seitens der Object Management Group (OMG) erarbeitete und derzeit wohl bekannteste MDSE Ausprägung. Das in Abbildung 1.1 gezeigte stufenweise Vorgehen sieht zunächst Modelle auf hoher und technikfreier Abstraktionsebene vor, die sogen. *plattformunabhängige* Modelle (PIM)[1]. Diese werden durch sukzessive Transformation, nach Möglichkeit automatisiert, in *plattformabhängige* Modelle (PSM) überführt. Bei dieser Transformation werden Plattforminformationen (*Plattformmodell*, PM) eingebracht, die bspw. die Darstellung von Datenstrukturen in einer konkreten technischen Plattform wie der JavaEE [Mon05] beschreiben. Die plattformabhängigen Modelle dienen schließlich als Grundlage zur Erzeugung von Quellcode der von Entwicklern üblicherweise händisch komplettiert wird.

Den technischen Kernbestandteil der MDA bildet die Unified Modeling Language (UML), eine Beschreibungssprache vornehmlich für objektorientierte Softwaresysteme. Die UML definiert eine überwiegend visuelle formale Sprache [OMG10], die heutzutage insbesondere zum Entwurf von Softwaresystemen angewendet wird. In der vorliegenden Arbeit wird an verschiedenen Stellen auf die UML Bezug genommen ohne dass eine detaillierte Einführung dieser Sprache erfolgt. Als einführende Literatur kann bspw. [Rum04] herangezogen werden.

Durch die Fokussierung der schrittweisen Transition vom PIM bis hin zum Code erzielt die MDA einen vergleichsweise hohen Anteil an Automatisierung. Auch wenn auf jeder der drei Stufen Ergänzungen und Ausformulierungen durch Modellierer und Entwickler nötig sind, so steht doch eine möglichst breite Abdeckung des dar-

[1] Zusätzlich wird eine noch gröbere Ebene, die sogen. berechnungsunabhängigen Modelle (CIM) definiert. Ein solches Modell ist aufgrund der vorgeschlagenen *umgangssprachlichen* Beschreibungsform jedoch nicht für automatisierte Transformationen geeignet.

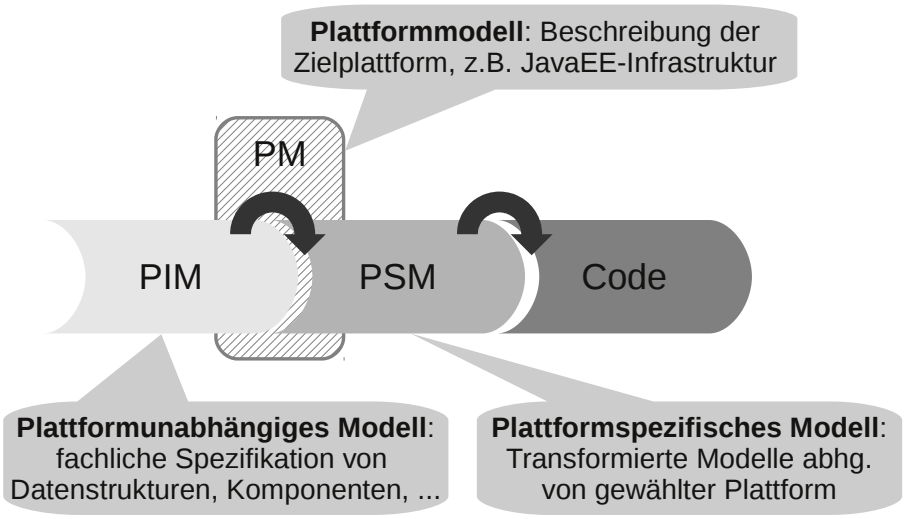


Abbildung 1.1: Schematischer Entwicklungsprozess mittels MDA

zustellenden Systems im Vordergrund. Dies wird durch umfassende Modellierungssprachen wie die UML unterstrichen, die neben statischen Strukturen auch die formale Spezifikation von Verhalten, etwa in Form von Zustandsmaschinen ermöglicht. Der in realen Einsatzszenarien automatisiert in Quelltext überführbare Systemanteil ist jedoch, wie bereits diskutiert, geringer als bspw. für statische Datenstrukturen.

Domänenspezifische Modellierung

Im Unterschied zur oben behandelten MDA, die auf allgemeine Modellierungssprachen wie insbesondere die UML aufbauen, wird im Bereich domänenspezifischer Modellierungssprachen (DSLs) keine derart standardisierte Notation eingesetzt. Durch die spezifische Ausrichtung einer Modellierungssprache auf ein gegebenes Anwendungsszenario – das durchaus unternehmens- oder auch projektspezifisch sein kann – soll stattdessen der Funktionsumfang bewusst klein gehalten werden. Hierdurch kann eine mögliche Überspezifikation in Form von zu detaillierten Beschreibungen vermieden werden, etwa in dem keine Sprachkonstrukte zur feingranularen Ausgestaltung von Datentypen angeboten werden. Auf diese Weise lässt sich mit Hilfe einer DSL der beabsichtigte Abstraktionsgrad einer Spezifikation sichern. Zugleich ist oftmals eine kompaktere Formulierung möglich, da wiederkehrend eingesetzte Modellierungselemente in vorgegebenen Sprachkonstrukten verkapselt werden können. Hierdurch entsteht eine leichtere Anwendbarkeit einer DSL im Vergleich zu allgemeinen, nicht fallspezifisch ausgerichteten Ansätzen wie bei der MDA.

In der *architekturzentrierten MDSE* [SVEH07] werden DSLs zur Beschreibung wie-

derholt auftretender Sachverhalte, dem sogen. Boiler-Plate-Code genutzt. Im wohl einfachsten Fall können dies generativ erzeugte Zugriffsmethoden für Attribute einer Implementierungsklasse sein. Aber auch Anbindungen an komplexe Infrastrukturen wie einem JavaEE-Rahmenwerk können hierdurch stark vereinfacht dargestellt und eine entsprechende Implementierung durch Generierung ergänzt werden. Anstatt wie bei der MDA einen fachlich breit aufgestellten Modellierungsansatz zu propagieren liegt der Fokus der architekturzentrierten MDSE wie auch generell bei DSLs auf repetitiven Anteilen. Diese werden durch spezifisch definierte Sprachkonstrukte ausgedrückt und generativ oder interpretativ in das realisierte System eingebunden.

Neben statischen Aspekten eines modellierten Softwaresystems werden DSLs auch teilweise zur Darstellung verhaltensorientierter, dynamischer Anteile entwickelt und eingesetzt. Beispielsweise stellt [CFP+04] eine solche Sprache zur Analyse großer Datenmengen vor, wobei die eingesetzten Traversierungsstrategien fallspezifisch modelliert werden können. Ein anderes Beispiel ist [BA99], in dem eine domänenspezifische Sprache zur Modellierung von Objektinteraktion realisiert wird. In beiden Fällen werden die vom Benutzer erstellten Dokumente zur Anwendung des spezifizierten Verhaltens in C-Quelltext übersetzt.

Domänenspezifische Sprachen ermöglichen eine leichtere Handhabung durch den Nutzer, da die Anwendungsdomäne eingeschränkt und technische Überspezifikation vermieden wird. Gleichzeitig entsteht allerdings das Risiko mangelnder Flexibilität, falls ein Sachverhalt nicht mit den verfügbaren Mitteln ausgedrückt werden kann. Da diese Situation aufgrund geänderter Anforderungen und Rahmenbedingungen häufig auftritt, unterliegen domänenspezifische Sprachen einer ständigen Evolution. DSL-spezifische Entwicklungswerkzeuge müssen diese Eigenschaft berücksichtigen, insbesondere also leicht anpassbar sein. Ferner sind Ressourcen zur Entwicklung solcher Werkzeuge üblicherweise limitiert, insbesondere wenn eine DSL als firmeninterne Infrastruktur entwickelt wird. Wie der folgende Abschnitt zeigt ist jedoch insbesondere die kosteneffiziente Realisierung von Entwicklungswerkzeugen für *verhaltensorientierte* DSLs mit erheblichen Problemen verbunden.

1.2 Problematik dynamischer Systemaspekte

Eine besondere Herausforderung im Kontext modellgetriebener Softwareentwicklung stellt das Verhalten eines Systems zur Laufzeit dar. Die Formulierung dieses Aspekts wird im Weiteren auch als *dynamische* bzw. *operationale Modellierung* bezeichnet. Zwar existieren zahlreiche – teils auch durchgehend formalisierte – Notationen verschiedenster Generationen zu diesem Zweck, allerdings sind diese kaum automatisiert in ausführbare Produkte überführbar.

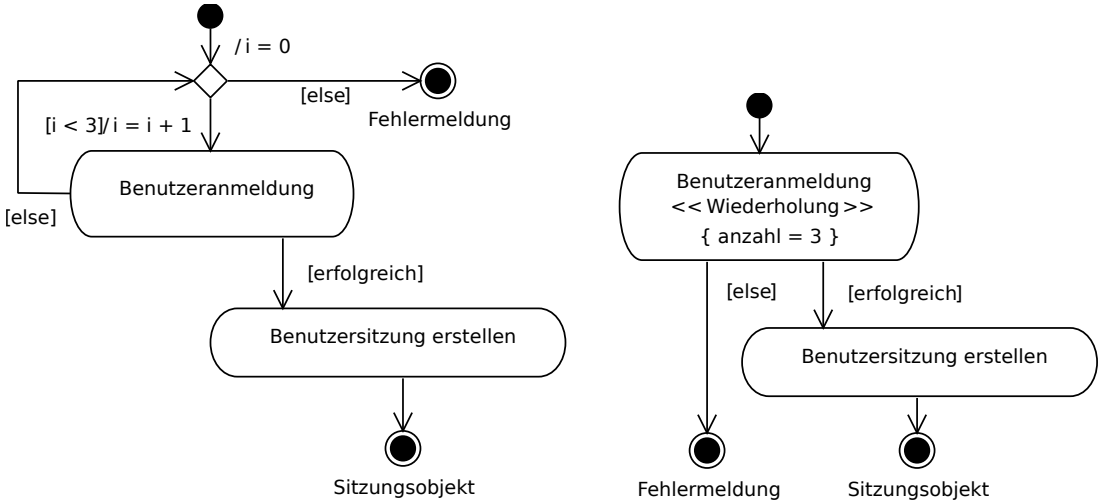
In o.g. MDSE-Ausprägungen wird häufig bewusst auf eine detaillierte und ausführbare Formalisierung eines Systems verzichtet. Allgemeinen Ansätzen wie der UML fehlen oftmals Ausdrucksmittel der gewünschten Domäne auf dem erforderlichen Abstraktionsniveau. Spezialisierte Spezifikationssprachen können dagegen zwar fallweise an Problemstellungen ausgerichtet werden. Hier stellt sich allerdings das Problem, dass diese Verkapselung in einer technisch komplexen Realisierung der Spezifikationssprache und dabei insbesondere ihrer operationalen Anteile resultiert. Ferner ist in beiden Varianten fraglich in wie weit bestehende oder fallweise entwickelte Lösungen zukünftigen Anforderungen, die bspw. eine Erweiterung der Spezifikationssprache bewirken, entsprechen. Zu prüfen ist in diesem Zusammenhang auch die *Anpassbarkeit* an veränderte Ausführungsplattformen.

1.2.1 Unerwünschtes Modellierungsniveau

Als Beispiel des häufig auftretenden unerwünschten Modellierungsniveaus kann die Darstellung eines Algorithmus mittels UML-Aktivitätsdiagrammen gesehen werden: Da die UML keine gesonderten Sprachkonstrukte, etwa zur Wiederholung einzelner Aktivitäten bietet, muss dies vergleichsweise umständlich mittels zyklischen Kontrollflüssen und Verzweigungsbedingungen dargestellt werden. Das in Abbildung 1.2a dargestellte Aktivitätsdiagramm beschreibt hierzu eine Benutzeranmeldung die Benutzerdaten entgegen nimmt und nach dreimaliger Fehleingabe eine Fehlermeldung anzeigt. Das Abstraktionsniveau und somit die Verständlichkeit einer solchen Darstellung wird hierbei reduziert, da niedersprachliche Konstrukte – hier Zuweisung und Abfrage von Variablen – explizit eingesetzt werden. Die Bedeutung der modellierten Schleife kann nur unter Betrachtung der Zusammenhänge der Bedingungen und Aktionen erfasst werden, etwa bezüglich der verwendeten Variablenbezeichner.

Ein Lösungsansatz um das Modellierungsniveau operationaler Spezifikationen zu sichern, besteht in der *Erweiterung* allgemeiner bzw. in der *Neuentwicklung* domänenspezifischer Spezifikationssprachen. Im vorliegenden Fall könnte ein Sprachkonstrukt zur Wiederholung einzelner Anweisungen eingeführt werden, um Wiederholungen explizit modellieren zu können. Analog zu obigem Beispiel zeigt Abbildung 1.2b die Anwendung eines speziell zu diesem Zweck erstellten UML Stereotyps[2] «Wiederholung». Zu klären bleibt in diesem Ansatz allerdings, wie das erweiterte Sprachkonstrukt bzw. die neu entwickelte DSL in eine ausführbare Form übertragen werden kann. Im Idealfall sollte dies auf möglichst einfache Weise, etwa durch versierte Benutzer eines allgemeinen UML Werkzeugs möglich sein. Wie der folgende Abschnitt zeigt, ist dies jedoch für operationale Sprachanteile oftmals nicht einfach

[2]Stereotypen sind ein leichtgewichtiger Erweiterungsansatz der UML.



(a) mittels niedersprachlicher Konstrukte (b) mittels Spracherweiterung

Abbildung 1.2: Verhaltensmodellierung: Wiederholung einer Aktivität

umsetzbar.

1.2.2 Komplexität der Realisierung

Zwischen Spezifikationssprachen auf hohem Abstraktionsniveau einerseits sowie den Programmierschnittstellen der eingesetzten Basisbibliotheken andererseits – bspw. das zur Laufzeit zu nutzende Betriebssystem – besteht üblicherweise ein starker konzeptioneller Unterschied. Die Überbrückung dieser „Semantischen Lücke“ [Sel06] ist Bestandteil der Ausführungswerkzeuge einer Spezifikationssprache.

Dynamische Anteile sind hierbei besonders aufwändig zu realisieren, wie Abbildung 1.3 am Beispiel einer hochsprachlich modellierten *Suchanfrage* zeigt. Für das als UML Objektdiagramm dargestellte Suchmuster, bestehend aus Objekten und Verbindungsstrukturen, sollen alle Vorkommen in einem Objektmodell ermittelt werden. Im Zuge einer generativen Softwareentwicklung könnte daraus das dargestellte Quelltextfragment erzeugt werden. Dieses traversiert Verbindungsstrukturen mittels geschachtelter Schleifen und Prüfoperationen.

Obwohl das dargestellte Textfragment selbst keine übermäßig hohe Komplexität aufweist, ist seine Erzeugung nicht trivial in allgemeiner Form zu beschreiben. Zum einen weist der generierte Code geschachtelte Blöcke mit veränderlicher Schachtelungstiefe auf die von der Gestalt des Suchmusters abhängt und somit nur umständlich in Form von Quelltextvorlagen dargestellt werden kann. Zum anderen beeinflusst die Reihenfolge der verschachtelten Operationen wesentlich die Effizienz des gene-

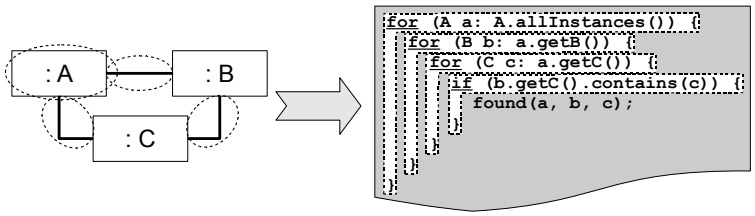


Abbildung 1.3: Codegenerierung für komplexes dynamisches Verhalten

rierten Codes. Aus diesem Grund ist eine möglichst optimale Lösung zu ermitteln. Eine Codegenerierung für dieserart Suchanfragen, wie im statischen Fall allein etwa durch Vorlagen zur Quelltextvorlagen zu bewältigen, ist insgesamt also schwer möglich. Auch gängige Entwurfspraxen für Interpreter, bspw. in Form von sogen. *Visitoren* [GHJV94] auf den Syntaxelementen der gestellten Anfrage, sind hier nicht zufriedenstellend anwendbar.

1.2.3 Anpassbarkeit und Erweiterbarkeit

Im Kontext modellgetriebener Softwareentwicklung ist es häufig nötig, den generierten Code auf bestehende Altsysteme oder Rahmenwerke aufzusetzen. Falls zu diesem Zweck allgemeine Modellierungswerkzeuge eingesetzt werden sollen, müssen diese flexibel und leicht an die jeweilige technische Plattform angepasst werden können. Für statische Modellierungsaspekte kann dies häufig leicht bewerkstelligt werden, da diese häufig eine vergleichsweise einfache Laufzeitsemantik besitzen. Hierbei können Quelltextvorlagen zur Generierung von Quelltext eingesetzt werden, die sich je nach Zielplattform leicht anpassen lassen. Im dynamischen Fall ist fraglich ob der gewählte Realisierungsansatz effizient auf eine Zielplattform wie etwa ein Datenbanksystem übertragen werden kann. Aufgrund der deutlich komplexeren Realisierung ist eine leichte Anpassbarkeit etwa durch Quelltextvorlagen hier oftmals nicht möglich.

Als ähnlich gelagert stellt sich die Erweiterbarkeit einer Spezifikationssprache bzw. deren Implementierung dar. Statische Aspekte lassen sich auch hier häufig nahezu direkt auf entsprechenden Quelltext abbilden, weshalb Quelltextvorlagen auch zur Erweiterung solcher Sprachen verwendet werden können. Erweiterungen verhaltensorientierter Sprachen können dagegen komplexe Querbezüge zu anderen Sprachkonstrukten herstellen und sind daher bei ihrer Auswertung entsprechend zu beachten.

1.2.4 Auswirkung auf die MDSE

Als Resultat der aufgezeigten Problematik wird dynamisches Verhalten bis dato nicht durchgängig modellgetrieben bis hin zum Softwareprodukt überführt. Stattdessen

werden operationale Teile eines Systems zu Dokumentations- und Analysezwecken modelliert, die jedoch häufig nicht formal mit dem tatsächlich erstellten Produkt verknüpft werden können. Neuere Ansätze gehen dazu über, Verhalten durch Vor- und Nachbedingungen sowie Invarianten nach Maßgabe des *Design-by-Contract* [Mey00] zu formulieren. Zwar unterstützt dies die Überprüfbarkeit des resultierenden Softwaresystems, dennoch lässt sich daraus für gängige Plattformen kein ausführbares Programm ableiten. Die Implementierung der eigentlichen Funktionalität findet daher auch hier nach wie vor manuell statt.

Im akademischen Bereich existieren verschiedene Werkzeuge mit entsprechender Funktionalität sowie erste Ansätze im industriellen Umfeld im Kontext des Standards Query/View/Transformation (QVT) [OMG08]. Allerdings entsteht oftmals die Notwendigkeit, eine spezialisierte Sprache zur Verhaltensbeschreibung für eine Problemdomäne entwerfen und realisieren zu müssen. In diesen Fällen entsteht ein unerwünscht hoher Realisierungsaufwand, dem bislang nicht geeignet durch Werkzeuge oder Plattformen entgegengewirkt wird.

1.3 Problemstellung & Umfeld der Arbeit

Wie im bisherigen Verlauf dieses Kapitels festgestellt wurde führen insbesondere Komplexitätsgründe zu den dargelegten Schwierigkeiten, Verhaltensmodellierung bis hin zu ausführbaren Softwaresystemen werkzeugunterstützt zu begleiten. Dieser Abschnitt erarbeitet zunächst eine Formulierung der Problemstellung, um anschließend existierende Lösungsansätze dahingehend zu untersuchen.

1.3.1 Problemstellung

Aus der bisher ermittelten Problemstellung lassen sich eine Reihe von Anforderungen ermitteln, die eine mögliche Werkzeuglösung für ausführbare Spezifikationssprachen mindestens bieten sollte. Diese ergeben sich aus dem grundsätzlichen Bedürfnis, das Verhalten eines Systems in einer Form spezifizieren zu können die direkt in eine lauffähige Implementierung umsetzbar ist. Diese Umsetzung soll das Abstraktionsniveau der Spezifikationssprache sicherstellen und zudem im Nachhinein anpassbar und erweiterbar bleiben. Ferner müssen technische Plattformen wie etwa Datenzugriffsschichten für zukünftige Anpassungen austauschbar bleiben.

Anforderung 1 – *Unterstützung zur Verhaltensmodellierung*

Die Modellierung von Verhalten muss angemessen unterstützt und durch eine Überführungsmöglichkeit in ausführbare Systeme begleitet werden. Konkret ergeben sich

dabei mehrere Problematiken, die aufgrund ihres Umfangs im Kontext dieser Arbeit selektiv fokussiert werden:

1. Unterstützung des Entwurfs einer verhaltensorientierten Modellierungssprache – bspw. in Form von Konzepten oder Entwicklungswerkzeugen die den Sprachentwurf im Anwendungsfeld verhaltensorientierter Sprachen vereinfachen.

2. Eine Ausführungsmaschine mit Anschlussmöglichkeit an die endgültige technische Plattform – hierdurch wird die leichte Umsetzbarkeit der konzeptionellen Lösung gesichert.

3. Werkzeugbasierte Prozessunterstützung obiger Hilfsmittel – insbesondere zur Reduktion der Einarbeitungs- und Entwicklungszeit bei Rückgriff auf o.g. Basistechnologien.

Da insbesondere die erste Fragestellung bezüglich des Entwurfs domänenspezifischer Sprachen ein umfassendes Forschungsgebiet darstellt, fokussiert die vorliegende Arbeit hauptsächlich den zweiten Punkt. Der dritte Aspekt wird zunächst in prototypischer Form behandelt.

Anforderung 2 – *Erhalt des Abstraktionsniveaus*

Verhalten soll auf Basis einer durchgängig hohen Abstraktionsstufe beschrieben werden können. Dies betrifft alle bei der Verhaltensmodellierung wesentlichen Bestandteile einschließlich obiger Aufzählung. Andernfalls können sprachspezifische Realisierungen mit hoher Komplexität entstehen, die im Weiteren schlecht gewartet werden können. Dieser Entwicklungsaufwand geht für andere Anwendungsfälle insoweitverloren, als dass sich fallspezifische Ergebnisse i.d.R. allenfalls schwierig auf weitere Sprachrealisierungen übertragen lassen. Die Wahrung eines hohen Abstraktionsgrads reduziert die fallweisen Entwicklungskosten und generalisiert gemeinsam genutzte Funktionalität vorab.

Anforderung 3 – *Anpassbarkeit der Lösung*

Da insbesondere domänenspezifische Spezifikationssprachen einer stetigen Evolution unterliegen, sollte eine bereitgestellte Realisierung weiterhin anpassbar bleiben. Gefordert ist also eine Anpassbarkeit der *Sprachrealisierung*, die sich einerseits auf mögliche Erweiterungen, andererseits auch auf eine Redefinition oder Parametrisierung bestehender Sprachkonstrukte beziehen kann. Auf diese Weise wird die Realisierung einer Spezifikationssprache direkt auf zukünftige Veränderungen vorbereitet, was etwa durch eine geeignete Basistechnologie gemäß Anforderung 1 zugesichert werden kann.

Eine weitere Interpretation stellt die flexible Anpassbarkeit des *realisierten Systems* dar, die sich bspw. in der unmittelbaren Ausführbarkeit einer geänderten Spezifikation zeigt. Auch das Hinzufügen weiterer Spezifikationsteile zur Laufzeit des modellierten Systems kann hierunter aufgefasst werden.

Anforderung 4 – *Adaption von technischen Plattformen*

Technische Plattformen sollten nicht explizit von einem Lösungsansatz vorgegeben werden, sondern bei Bedarf adaptiert werden können. Hierunter ist zu verstehen das z.B. die realisierten verhaltensorientierten Spezifikationssprachen nicht ausschließlich auf einer einzelnen Datenquelle arbeiten können sollen. Stattdessen sollten Datenquellen fallweise, falls möglich erst zur Laufzeit konfiguriert werden können. Gleichermaßen könnte auch Unabhängigkeit von einer konkreten Programmiersprache gefordert werden, falls beispielweise Quelltextgenerierung zum Einsatz kommt.

1.3.2 Existierende Ansätze

Basierend auf den soeben eingeführten Lösungsanforderungen untersucht dieser Abschnitt, in wie weit existierende Ansätze diesen genügen. Zunächst wird in diesem Kontext das IPSEN-Projekt behandelt, das am gleichen Lehrstuhl wie die vorliegende Arbeit entstanden ist. Ferner kommen CASE- bzw. ihre erweiterte Abstraktionsstufe, die sogen. Meta-CASE-Werkzeuge, als Lösungsansätze in Betracht.

IPSEN

Die Zielsetzung des am Lehrstuhl für Informatik 3 durchgeführten Projekts IPSEN (Integrated Project Support ENvironment) war, Softwareentwicklungsprozesse durch eine phasenübergreifende Werkzeugarchitektur zu unterstützen [EJS88]. Eine mögliche Anwendung des IPSEN-Systems auf Softwareentwicklungsprozesse stellt daher bspw. einen Editor für Anforderungsdefinitionen und für Softwarearchitekturen zur Verfügung[KLN96, GNS97]. Durch eine Verknüpfung dieser Werkzeuge kann sichergestellt werden, dass die hiermit spezifizierten Softwarearchitekturen die gestellten Anforderungen geeignet abdecken, beide Dokumente also konsistent zueinander sind. Dieser Ansatz lässt sich zudem bis hin zum Programmieren-im-Kleinen verfolgen, so dass der erstellte Quelltext wiederum mit der Architekturbeschreibung in Konsistenz gehalten wird. Auf diese Weise kann eine durchgängige Nachverfolgbarkeit von Entwurfsentscheidungen erreicht und eine Umsetzung abstrakter Anforderungen überprüft werden.

Technisch basiert IPSEN auf einer vereinheitlichten, graph-orientierten Speicherlösung, der Graphdatenbank GRAS. Hiermit können die in verschiedenen Prozesspha-

sen bearbeiteten Dokumente feingranular miteinander in Bezug gesetzt werden, etwa in dem eine spezifizierte Anforderung ein Teilsystem der Architekturbeschreibung referenziert. Zur Beschreibung von Modellen stellt IPSEN eine Standardumgebung für Benutzeroberflächen bereit, mittels derer graphbasierte Dokumente in visueller oder textueller Form dargestellt und bearbeitet werden können. Syntaxgesteuerte Editoren für textuelle Sprachen können unter Angabe einer EBNF-Grammatik automatisiert abgeleitet werden. Für kontextsensitive Sachverhalte wie bspw. Gültigkeitsbereiche von Variablen und Typdeklarationen bietet IPSEN ein allgemeines Metamodell, auf das sich konkrete Sprachkonstrukte beziehen können. Visuelle Sprachen werden in IPSEN mit einem entsprechenden *Unparser* implementiert, welcher die graphische Ausgabe unter Verwendung der Standardarchitektur übernimmt. Diese werden in IPSEN allerdings herkömmlich in Modula-2 programmiert.

Die mit IPSEN realisierten Editoren dienen hauptsächlich zur Konsistenzhaltung der damit beschriebenen Modelle. Eine Operationalisierung dieser Modelle, also etwa das Generieren und Ausführen entsprechenden Quelltextes, wird von IPSEN jedoch kaum unterstützt. Zwar kann als Ergebnis einer Modellauswertung Quelltext ausgegeben werden der wiederum als Eingabedokument für externe Werkzeuge wie Compiler oder Interpreter dient. Nicht weiter unterstützt wird von IPSEN jedoch die hochsprachliche Definition von spezifischen Ausführungsmaschinen, so dass Anforderung 1 im Wesentlichen nicht erfüllt wird. Dennoch bietet IPSEN hierbei ein gewisses Maß an Anpassbarkeit (Anforderung 3), da die erwähnte Codegenerierung hochsprachlich beschrieben werden kann. Da eine zugehörige *Laufzeitumgebung* nicht Teil einer IPSEN Anwendung ist, sondern extern bereitgestellt wird, ist eine Bewertung der weiteren Anforderungen bzgl. des Abstraktionsniveaus (Anforderung 2) sowie der Unabhängigkeit von technischen Plattformen (Anforderung 4) an dieser Stelle nicht bewertbar.

CASE- und Meta-CASE-Werkzeuge

Derzeit sind eine Vielzahl von Werkzeugen zur Softwareentwicklung (Computer Aided Software Engineering, CASE) verfügbar, die das hochsprachliche Spezifizieren eines Softwaresystem und – je nach Ausprägung – dessen Überführung bis hin zu einem ausführbaren Produkt ermöglichen. Zu den CASE-Werkzeugen zählen einerseits verschiedene klassische Systeme auf Basis von SA- und ER-Diagrammen wie bspw. microTool case/4/0 [CAS]. Andererseits sind auch viele heutige UML-basierte Werkzeuge dieser Kategorie zuzuordnen.

Üblicherweise erlauben CASE-Werkzeuge die rudimentäre *Generierung* von Programmteilen, bspw. für Zugriffsschichten zum Anschluss an Datenbanksysteme. Verhaltensanteile eines Systems, insbes. also seine Anwendungslogik, lässt sich häufig

entweder nicht spezifizieren, oder wird nicht durch die bereitgestellten Codegeneratoren verarbeitet. Zudem entsteht aufgrund fest vorgegebener Spezifikationssprachen oben geschilderte Problematik, dass bei der Verhaltensspezifikation ein angemessener Abstraktionsgrad schwierig zu erreichen ist. Von daher erfüllen gängige CASE-Werkzeuge die oben gestellten Anforderungen 1 oder 2 nicht. Auch eine weitreichende Anpassbarkeit, etwa der Spezifikationssprache ist im Allgemeinen nicht gegeben (Anforderung 3). Generativ arbeitende CASE-Werkzeuge bieten dagegen i.d.R. eine Möglichkeit, zwischen verschiedenen Implementierungsstilen und damit der anzusteuernden Plattform zu wählen (Anforderung 4).

Ausgehend von dem Bedarf, Spezifikationssprachen und zugehörige Entwicklungswerkzeuge spezifisch für bestimmte Anwendungsdomänen zu entwerfen, erlauben Meta-CASE-Werkzeuge die modellgetriebene Entwicklung von CASE-Werkzeugen. Somit wird das für die eigentliche Softwareentwicklung einzusetzende Werkzeug fallweise und durch die Problemdomäne getrieben konstruiert [KLR96, ESU97]. Dabei wird häufig ein generativer Ansatz angewendet um ein spezifisches Werkzeug aus einer hochsprachlichen Spezifikation automatisch abzuleiten. Analog zu obigem IPSEN-Ansatz können auf diese Weise visuelle Editoren für spezifische Spezifikationssprachen erzeugt werden. Aufgrund der frei wählbaren Sprachdefinition wird durch den Meta-CASE Ansatz die Modellierung von Verhalten grundsätzlich ermöglicht. Zudem kann durch frei definierbare Sprachkonstrukte ein angemessenes Abstraktionsniveau sichergestellt werden.

Die weitere Verarbeitung der mittels eines so spezifizierten CASE-Werkzeugs erstellten Modelle wird jedoch üblicherweise wenig unterstützt. Angeboten wird häufig eine vorgefertigte Möglichkeit zum standardisierten Datenaustausch etwa in Form von XMI-Dokumenten. Auch die Entwicklung spezifischer Codegeneratoren kann wie bspw. durch MetaEdit+ [KLR96] unterstützt werden. Für verhaltensorientierte Sprachanteile unterliegen diese jedoch wiederum den in Abschnitt 1.2 genannten Schwierigkeiten, insbesondere der zu erwartenden Realisierungskomplexität. Von daher obliegt es dem Benutzer eines solchen Meta-CASE-Werkzeugs, oftmals komplexe Transformationen von einer Spezifikations- zu einer ausführbaren Programmiersprache mittels Quelltextvorlagen zu realisieren.

Modelltransformationswerkzeuge

Modelltransformationen [SK03] dienen zur Übersetzungen von Modellen zwischen üblicherweise verschiedenen Modellklassen. Das hierzu nötige Wissen, wie die jeweiligen Formalismen aufeinander abzubilden sind, wird häufig in hochsprachlicher Form durch Entwickler modelliert. Insbesondere deklarative und regelbasierte Sprachparadigmen bieten hier einen bequem nutzbaren Abstraktionsgrad. Darauf

basierend wird mittels entsprechender Transformationswerkzeuge eine vollautomatische Übersetzung der jeweiligen Instanzmodelle realisiert. Die zahlreichen Anwendungsmöglichkeiten umfassen bspw. die datenbasierte Kopplung von Softwarewerkzeugen, die Abbildung plattformunabhängiger Datenmodelle auf relationale Datenbanktabellen, oder die semantische Abbildung von UML-Aktivitätsdiagrammen auf Prozessalgebren. Neben der häufig unidirektionalen Transformationsausführung existieren zudem Ansätze zur bidirektionalen *Modellsynchronisation*, bei der die Transformationsrichtung unabhängig von ihrer Beschreibung dynamisch gewählt werden kann [KS06b]. Insbesondere im bidirektionalen Fall sind zudem Möglichkeiten zur *Benutzerinteraktion* vorzusehen falls die zu synchronisierenden Modelle entkoppelt voneinander verändert werden können [BHLW07].

Die mittels Modelltransformationen bereitgestellte Technologie zur Übersetzung zwischen Spezifikationssprachen kann prinzipiell auch zu ihrer Ausführung eingesetzt werden. Hierzu wird eine einfache Zieldomäne, bspw. in Form numerischer Werte oder Wahrheitswerte definiert und die auszuführende Spezifikation in diese Zieldomäne übersetzt. Dies wird bspw. als Einsatzmöglichkeit der Transformationssprache ATL zur Modellprüfung verstanden [BJ06], trifft jedoch auf alle verbreiteten Ansätze zu. Anforderung 1 wird durch diese Technologie daher erfüllt. Dagegen ist offen in wie weit bei derart beschriebenen Operationalisierungen ein angemessener Abstraktionsgrad (Anforderung 2) gewahrt bleibt. Dies ist stark von der jeweiligen Transformationssprache bzw. dem zugehörigen Werkzeug abhängig.

Modelltransformationswerkzeuge sind üblicherweise unabhängig von den zu übersetzenden Spezifikationssprachen konstruiert, so dass ein definierter Satz an Funktionalität ausreicht. Hierdurch besteht kein Bedarf an einer fallweisen Anpassbarkeit gemäß Anforderung 3. Gleichzeitig werden aufgrund dieser Unabhängigkeit von den zu verarbeiteten Sprachen auch die jeweiligen technischen Plattformen geeignet abstrahiert. Hierdurch wird Anforderung 4 entsprechend erfüllt.

1.3.3 Zielsetzung der Arbeit

Aus der im Verlauf dieses Abschnitts eingeführten Problemstellung und dem skizzierten Vergleich existierender Ansätze wird die Zielsetzung der vorliegenden Arbeit definiert: Ziel der Arbeit ist eine Unterstützung zur Realisierung *verhaltensorientierter Spezifikationssprachen* anzubieten. Diese soll sowohl eine *vereinfachte Entwicklung* sprachspezifischer *Ausführungsmaschinen*, als auch deren *weitere Anpassung* hinsichtlich einer veränderten Sprachdefinition sowie eines veränderten technischen Umfelds ermöglichen.

Bei der Bearbeitung dieser Fragestellung wird eine bewusste Einschränkung der unterstützten Anwendungsszenarien vorgenommen, um eine handhabbare Komple-

xität erzielen zu können. Konkret fokussiert die vorliegende Arbeit verhaltensorientierte Spezifikationssprachen zur Informationsverarbeitung. Derartige Sprachen beschreiben die Auswertung und ggf. die Manipulation von Datenbeständen auf hohem Abstraktionsniveau. Das Ergebnis der Verarbeitung ist entweder ein verändertes Eingabobjekt oder ein verdichtendes Resultat einer Berechnung. Dagegen erfolgt keine Definition des Datenschemas, das stattdessen bspw. von bestehenden Datenquellen bzw. Metamodellen bezogen wird.

Als Beispiele für informationsverarbeitende Sprachen können die Standards der Object Management Group, OCL und QVT verstanden werden: Beide Sprachen verarbeiten extern bereitgestellte Eingabemodelle, im ersten Fall durch Ableitung von Informationen, im zweiten Fall durch Veränderung der Eingabe oder der Erzeugung abgeleiteter Modelle. In beiden Fällen sind auch die zu verarbeitenden Informationsquellen von der verarbeitenden Spezifikation entkoppelt, bspw. kann eine OCL-Einschränkung auf eine Vielzahl von Instanzmodellen angewendet werden.

Begleitet werden die o.g. Sprachen OCL und QVT durch weitere Teilsprachen der UML, wie etwa Klassendiagramme zur Formulierung der statischen Datenstrukturen. In der vorliegenden Arbeit wird ebenfalls auf eine formalisierte Variante von Klassendiagrammen zurückgegriffen um Metamodelle zu beschreiben. Weitere UML-Teilsprachen werden durch den Ansatz der Arbeit nicht explizit abgedeckt, etwa Zustands- und Sequenzdiagramme. Diese Vertreter beschreiben mögliche Reaktionen eines Systems auf Ereignisse, nicht jedoch die aktive Informationsverarbeitung in obigem Sinne. Diese Unterscheidung wird insbesondere daran ersichtlich, dass bspw. bei Zustandsdiagrammen Verhalten mit dem Erreichen eines Zustands aus einer vorab festgelegten Zustandsmenge sowie dem Auslösen von Ereignissen beschrieben wird. Dagegen entstammt das Ergebnis eines informationsverarbeitenden Systems i.d.R. einer unbeschränkten Wertedomäne, etwa als Instanz eines Metamodells oder in Form einer natürlichen Zahl.

1.4 Lösungsansatz

Dieser Abschnitt gibt eine kurze Einführung in den im Verlauf dieser Arbeit präsentierten Lösungsansatz, der auch in [Wei08b] skizziert wird. Die Beschreibung verwendet als Beispielszenario eine Anfragesprache für graphbasierte Daten, die in Kapitel 3 detailliert vorgestellt wird. Anhand dieses Beispiels wird die erreichte Unterstützung zur Realisierung verhaltensorientierter Spezifikationssprachen vorgestellt. Abschließend erfolgt ein Abgleich mit den Anforderungen aus Abschnitt 1.3.1.

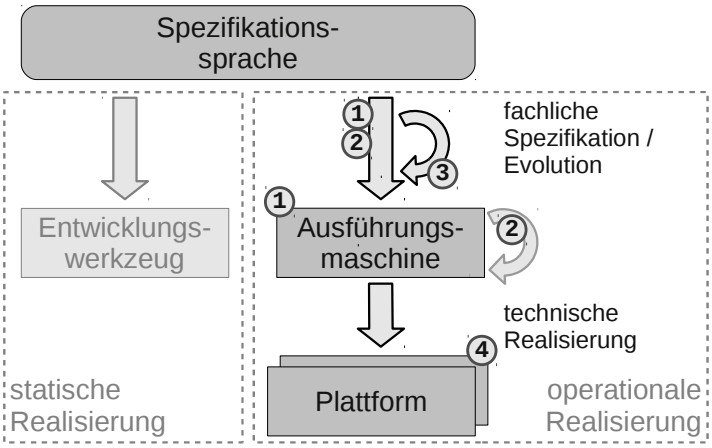


Abbildung 1.4: Skizzierte Zielsetzung mit Querbezügen zu Anforderungen

1.4.1 Überblick

Die Übersichtsdarstellung in Abbildung 1.4 zeigt die Realisierung einer Spezifikationssprache bezüglich zwei verschiedener Aspekte. Dabei wird der links dargestellte statische Anteil, wozu insbesondere Entwicklungswerkzeuge zur leichteren Nutzung der Spezifikationssprache zählen, hier nicht behandelt. Hierzu sind bereits eine Reihe erprobter Rahmenwerke vorhanden, was exemplarisch in Kapitel 3 aufgezeigt wird. Die vorliegende Arbeit konzentriert sich dagegen auf den rechten Teil der Abbildung, der die operationale Realisierung einer Spezifikationssprache repräsentiert. Die Abbildung zeigt hierzu die groben Teilbereiche die sich in eine fachliche und eine technische Realisierung zerlegen lassen. Eingekreist dargestellt sind jeweils die zugeordneten Anforderungen des vorangegangenen Abschnitts.

Zur Ausführung der operationalen Realisierung einer Spezifikationssprache wird eine *Ausführungsmaschine* bereitgestellt, die im unteren Teil der Abbildung dargestellt wird. Dieser Teil bildet den Ansatzpunkt der vorliegenden Arbeit, da eine solche Maschine insbesondere aus Kostengründen nicht spezifisch für einzelne Spezifikationssprachen realisiert werden sollte. Stattdessen wird eine solche Realisierung durch Wiederverwendung einer geeigneten allgemeinen Ausführungsmaschine unterstützt. Die im Kontext dieser Arbeit erstellte Maschinerie soll eine leichtere Umsetzung sprachspezifischer Lösungen ermöglichen, und unterliegt daher insbesondere Anforderung 1 (Verhaltensmodellierung).

Zur Nutzung dieser Ausführungsmaschine wird eine Abbildung von sprachspezifischen Konzepten auf allgemeine Grundlagen benötigt. Bei dieser Abbildung handelt es sich, wie in der Abbildung dargestellt wird, um eine *fachliche* Realisierung, da zunächst die Bedeutung einzelner Sprachkonstrukte plattformunabhängig festgelegt

wird. Die Möglichkeit, diese Abbildung angemessen zu beschreiben, wird durch die Anforderungen 1 bzw. 2 (Abstraktionsniveau) bestimmt. Zur Evolution der Spezifikationssprache wird daher zusätzlich eine Evolution dieser Abbildung erforderlich, wodurch sich ein Querbezug zu Anforderung 3 (Anpassbarkeit) ergibt. Ferner soll die Ausführungsmaschine an sich anpassbar und erweiterbar gestaltet werden, sollte eine Spezifikationssprache die Realisierung nicht bzw. nur umständlich ausdrückbare Konstrukte erfordern.

Als letztes Element der Skizze wird die *technische* Realisierung der Ausführungsmaschine gezeigt. An dieser Stelle erfolgt der Anschluss des Systems an eine spezifische Plattform, also konkrete Betriebssysteme, Datenbanken, etc. Idealerweise sollte diese Realisierung gemäß Anforderung 4 (Plattformunabhängigkeit) für das modellierte System unabhängig austauschbar sein.

1.4.2 Beispielszenario

Betriebswirtschaftliche *Informationssysteme* [MBP+04] nehmen unter aktuellen Softwareentwicklungsprojekten einen bedeutenden Anteil ein. Zur Realisierung der hierzu erforderlichen Funktionalität ist es häufig nötig, komplexe Datensätze zu analysieren und verdichtete Informationen zu extrahieren. Im Rahmen eines fiktiven Verwaltungssystems zur Projektadministration wird somit Funktionalität gewünscht, um den Zeitbedarf eines Projekts zu ermitteln. Auch die Auslastung der Mitarbeiter oder die an einem Projekt bisher verrichtete Arbeit sind wichtige Informationen für die Nutzer eines solchen Systems.

Anfragen bilden also einen wesentlichen Bestandteil der Anwendungslogik eines solchen Administrationswerkzeugs. Um deren Realisierung zu unterstützen, bietet sich der Einsatz einer *Anfragesprache* auf vorwiegend fachlichem und hochsprachlichem Niveau an, um von technischen Eigenschaften wie bspw. dem zugrundeliegenden Datenzugriff zu abstrahieren. Gängige Programmiersprachen und Datenbankanfragesprachen auf niedrigem Abstraktionsniveau erfordern dagegen eine vergleichsweise umständliche Formulierung der Geschäftslogik.

Im Folgenden wird die Möglichkeit untersucht, mittels des in dieser Arbeit vorgestellten Ansatzes eine ausführbare Spezifikationssprache für Anfragen zu erstellen und eine entsprechende spezifische Ausführungsmaschine zu realisieren. Hierbei handelt es sich insoweitum eine domänenspezifische Sprache, als dass sie sich auf die technische Domäne der Datenanfragen beschränkt. Zugleich eignet sich die handhabbare Komplexität einer Anfragesprache gut zu Illustrationszwecken des Ansatzes.

1.4.3 Unterstützung zur Sprachrealisierung

Als ersten der beiden wesentlichen Bereiche die durch den Ansatz der vorliegenden Arbeit unterstützt werden zeigt Abbildung 1.5 die Unterstützung zur Sprachrealisierung. Dieser Bereich findet Anwendung zur *Spezifikationszeit*, d.h. bei der Verwendung der zu realisierenden Anfragesprache in Form einer fachlichen Spezifikation. Der obere Teil der Abbildung zeigt diejenigen Anteile, die für eine zu realisierende Sprache jeweils individuell bereitzustellen sind. Der untere Teil zeigt Komponenten die im Rahmen dieser Arbeit entwickelt worden sind, oder auf die als am Lehrstuhl verfügbare Infrastruktur zurückgegriffen wird.

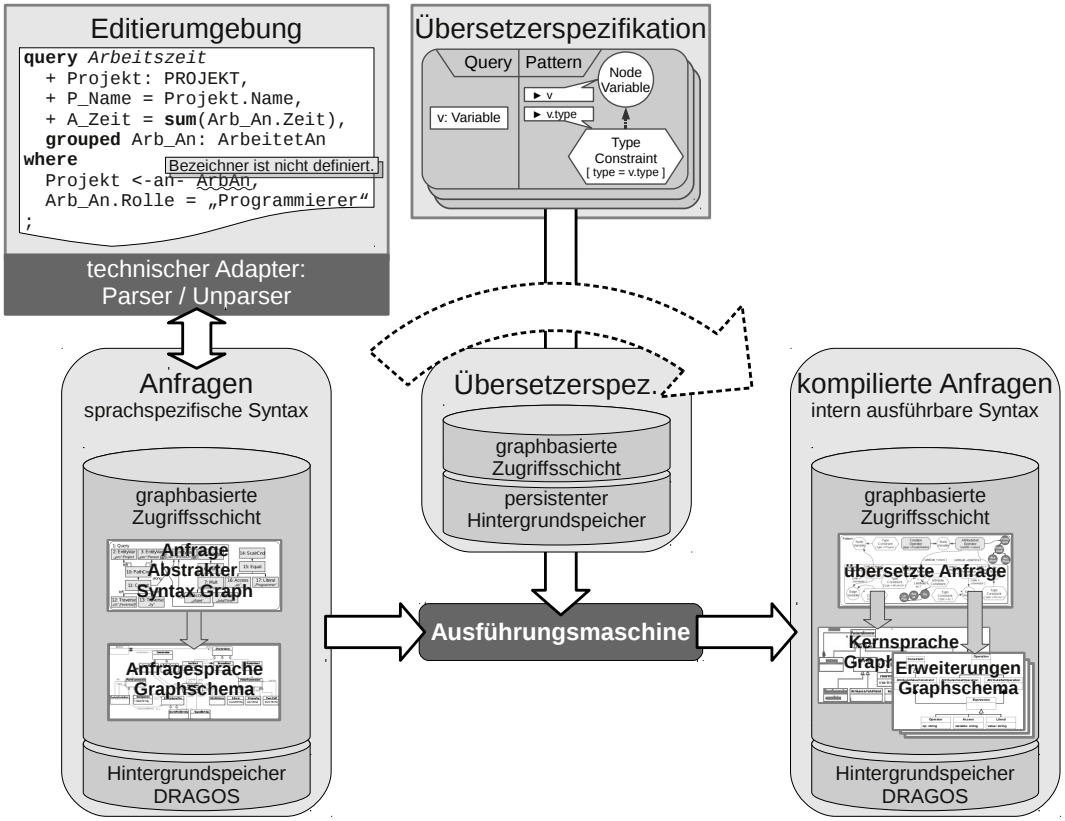


Abbildung 1.5: Realisierung einer Anfragesprache

Zur Realisierung von Anwendungslogik eines Informationssystems mittels hochsprachlichen Anfragen wird grundsätzlich die gleiche Werkzeugunterstützung benötigt wie für Entwicklungsumgebungen herkömmlicher Programmiersprachen. Wie oben links in der Abbildung dargestellt kann dies die syntaktische Hervorhebung oder die Formatierung der textuellen Anfragen umfassen. Auch ist eine Prüfung der Anfragen gegen die Wohlgeformtheitsbedingungen der Anfragesprache sinnvoll um

Spezifikationsfehler frühzeitig, idealerweise während der Eingabe durch den Modellierer, aufzudecken. Die Schnittstelle der Entwicklungsumgebung zum vorgestellten Rahmenwerk kann auf verschiedene Weise, etwa durch einen Parser/Unparser realisiert werden. Ziel dieses Vorgehens ist es, eine syntaktisch sprachspezifische, jedoch technisch uniforme Präsentation der modellierten Spezifikationen zu erhalten. Konkret bedeutet dies, dass ein Parser eine abstrakte Darstellung, bspw. einen abstrakten Syntaxbaum (AST) einer Anfrage ermittelt und abspeichert. Dies erfolgt in sprachspezifischer Syntax insoweit, als dass die Gestalt des ASTs frei gewählt werden kann, also nicht auf ein übergreifendes Datenmodell abgebildet werden muss. Als technische Speicherlösung wird auf das Graphspeichersystem DRAGOS (siehe 2.4) zurückgegriffen, das im Rahmen der Arbeit [Bö06] am Lehrstuhl für Informatik 3 entwickelt worden ist.

DRAGOS bietet aufgrund des graphbasierten Datenmodells eine universelle Plattform zur Speicherung komplexer Spezifikationsdokumente. Im vorliegenden Fall werden die ASTs der Anfragespezifikationen – einschließlich Querbeziehungen – im Graphspeicher persistent abgelegt. Durch die austauschbaren Hintergrundspeicher des DRAGOS Systems entsteht dabei keine Festlegung auf eine konkrete Speicherlösung, wie bspw. ein bestimmtes Datenbanksystem.

Zur eigentlichen Realisierung der Anfragesprache wird neben der syntaktischen Analyse eine Übersetzung der sprachspezifischen Syntax in eine ausführbare Form benötigt. Übliche leicht umzusetzende Ansätze wie die Generierung von Quellcode sind hierzu nur bedingt geeignet, da eine Anfrage durch eine Abfolge komplexer Zugriffsoperationen auf die zugrundeliegende Datenquelle umzusetzen ist. Stattdessen stellt die vorliegende Arbeit eine *Kernsprache* bereit, mittels derer Anfragen und Veränderungen von Graphen auf hohem Abstraktionsniveau beschrieben werden können. Neben ihrem hohen Abstraktionsgrad bietet die Kernsprache Erweiterungsmöglichkeiten zur Ergänzung und Anpassung der anwendungsseitig benötigten Funktionalität, wodurch sie einerseits kompakt, andererseits auch leicht nutzbar gestaltet werden kann. In obiger Abbildung wird diese in Kapitel 4 eingehend behandelte Sprache auf der rechten Seite als kompilierte Anfragen angedeutet. Das Übersetzungsresultat wird auch hier als Graph im DRAGOS Graphspeicher abgelegt.

Auch die eigentliche Übersetzung von der vorgestellten Anfragesprache in die ausführbare Kernsprache wird werkzeugtechnisch abgedeckt: Durch eine regelbasierte Übersetzungsmaschine [Wei09] werden Anfragen in sprachspezifischer Syntax gelesen und in einer für die Kernsprache geeigneten Form ausgegeben. Diese Übersetzungsmaschine wird regelbasiert konfiguriert, was durch die bereitgestellten Werkzeuge auch visuell ermöglicht wird. Die in der Abbildung lediglich angedeutete Syntax der Übersetzungsregeln wird in Kapitel 6 eingehend behandelt.

1.4.4 Unterstützung zur Laufzeit

Abbildung 1.6 zeigt den zweiten Teilbereich der bereitgestellten Realisierungsunterstützung, der die Laufzeit der Anfragesprache und damit die konkrete Auswertung von Anfragen abdeckt. Eine hierfür erstellte Benutzerschnittstelle kann beispielsweise in das oben diskutierte Informationssystem integriert werden, um dessen Benutzern direkten Zugriff auf die modellierten Anfragen zu ermöglichen. Auch denkbar wäre es, auf eine Endbenutzerschnittstelle zu verzichten und die ermittelten Ergebnisse lediglich für interne Berechnungen weiterzuverwenden.

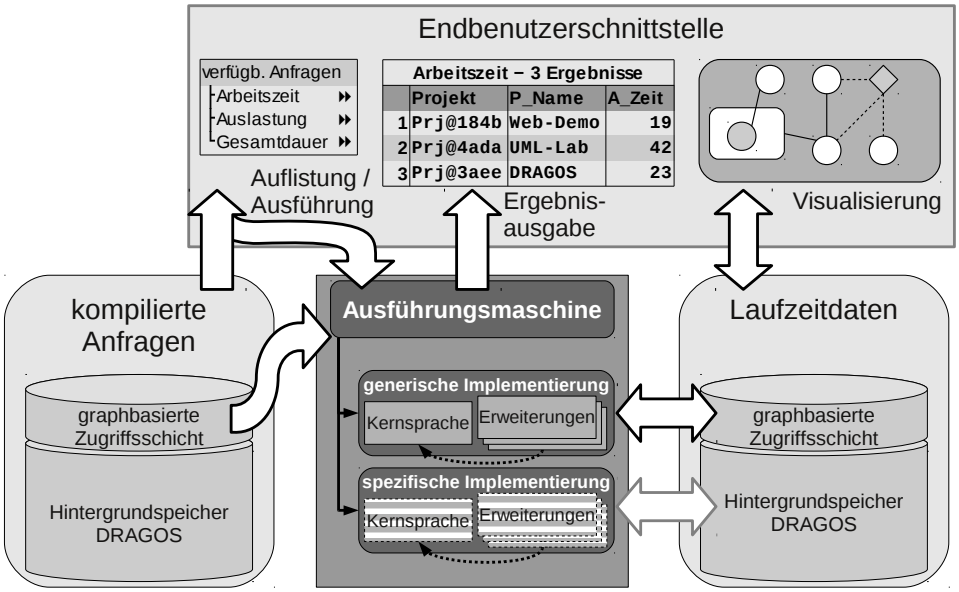


Abbildung 1.6: Einsatz der realisierten Anfragesprache

Der erste Schritt seitens der Benutzerschnittstelle besteht darin, die im verwendeten Graphspeicher verfügbaren Anfragen zu ermitteln. Hierbei wird auf die im vorherigen Schritt *übersetzten Anfragen* zurückgegriffen, so dass jegliche Weiterverarbeitung unabhängig von der Syntax der Anfragesprachen erfolgen kann. Erweiterungen und Veränderungen der Sprache sind also ohne Einfluss auf die Endbenutzerschnittstelle möglich.

Nach Auswahl einer Anfrage durch den Benutzer sowie ggf. der Angabe von Parametern, wird diese der bereitgestellten allgemeinen Ausführungsmaschine [Wei07] übergeben. Diese liest den Inhalt der Anfrage aus dem Graphspeicher aus, und führt die Anfrage gemäß der verfügbaren Sprachrealisierung aus. Hierbei werden zwei verschiedene Varianten angeboten, um die in Abbildung 1.5 angedeutete Kernsprache sowie mögliche Erweiterungsmodule ausführen zu können.

Die als *generische Implementierung* bezeichnete erste Variante greift auf die auszuwertenden Laufzeitdaten über die graphbasierte Schnittstelle des DRAGOS Graphspeichers zu. In diesem Fall werden also die Laufzeitdaten ebenfalls als Graphen interpretiert, und durch einzelne Graphoperationen, bspw. der Traversierung von Verbindungsstrukturen, analysiert. Der Vorteil dieses Ansatzes besteht darin, dass auf Grund der graphbasierten Schnittstelle vielfältige Datenquellen uniform behandelt werden können, hauptspeicherbasierte Lösungen ebenso wie relationale Datenbanksysteme. Nachteilig hieran ist, dass diese Abstraktionsschicht hohen Aufwand zur Laufzeit verursacht, da ein zunächst nicht weiter bestimmtes Datenformat in eine graphbasierte Sicht überführt werden muss.

Als zweite Variante wird die Möglichkeit einer *spezifischen Implementierung* angeboten. Anstelle der graphbasierten Zugriffsschicht wird direkt auf den Hintergrundspeicher und dessen jeweilige Funktionalität zugrückgegriffen. Falls die zu verarbeitenden Laufzeitdaten beispielsweise in einer relationalen Datenbank abgelegt werden, kann die gewählte Anfrage durch Erzeugung von SQL Anweisungen spezifisch für diesen Hintergrundspeicher ausgeführt werden. Da in dieser Variante komplexe Operationen wie die Navigation entlang Verbindungsstrukturen lokal im Hintergrundspeicher ausgeführt werden, kann dessen Optimierungspotenzial besser genutzt werden als in der ersten Variante. Nachteilig hieran ist die komplexe Realisierung einer solch spezifischen Sprachimplementierung, weshalb diese Variante *optional* ist. Auch sind nicht alle Sprachkonstrukte der Kernsprache oder ihrer Erweiterungen in SQL ausdrückbar, so dass hierfür auch auf die generische Implementierung zurückgegriffen werden muss.

Unabhängig von den beiden vorgenannten Möglichkeiten steht für Spracherweiterungen eine dritte Realisierungsvariante zur Verfügung. Wie durch die gestrichelten Pfeile angedeutet wird, können Sprachkonstrukte eines Erweiterungsmoduls mittels Rückführung auf die Kernsprache oder andere Erweiterungen umgesetzt werden [Wei08a]. Dieses in der Literatur als „Desugaring“[3] bezeichnete Vorgehen [van08] dient dazu, Implementierungsaufwand für Sprachkonstrukte zu vermeiden, in dem auf bestehende Funktionalität zurückgegriffen wird. Da nicht alle Spracherweiterungen praktikabel auf bestehende Konstrukte aufgesetzt werden können oder Effizienzargumente diesem Vorgehen widersprechen, ist eine manuelle Implementierung gemäß obiger Varianten dennoch anzubieten. Gleichzeitig ist auch die Bereitstellung von Sprachkonstrukten sinnvoll, die direkt auf Elemente der Kernsprache zurückgeführt werden können. Hierdurch kann die zur Sprachrealisierung erforderliche Übersetzung vereinfacht werden. Zugleich ist auch eine Wiederverwendung der so verkapselten Funktionalität durch andere Übersetzer möglich.

[3] engl. für die Entfernung syntaktischen „Zuckers“ aus einer Spezifikation

Im weiteren Verlauf der Anfrageausführung werden die berechneten Ergebnisse durch die Endbenutzerschnittstelle aufbereitet und dem Nutzer präsentiert. Hierzu kann bspw. eine tabellarische Aufstellung genutzt werden. Parallel dazu kann auch eine Visualisierung der Laufzeitdaten erfolgen falls dies gewünscht und, gemessen an der Größe dieser Daten, praktikabel ist. Eine rudimentäre Visualisierungsumgebung unter Nutzung der von DRAGOS bereitgestellten graphbasierten Zugriffsschicht ist als Hilfswerkzeug im Rahmen dieser Arbeit bereits entstanden.

1.4.5 Anforderungsabgleich & Beitrag der Arbeit

Zum Abschluss der Kurzbeschreibung des Lösungsansatzes wird dieser mit den in Abschnitt 1.3.1 auf Seite 11 erstellten grobgranularen Anforderungen in Bezug gesetzt sowie von weiterführenden Zielsetzungen abgegrenzt. Dieser Abschnitt schließt mit einer Zusammenfassung der erwarteten Resultate dieser Arbeit.

Anforderungsabgleich

Die in Abbildung 1.5 zur Sprachrealisierung vorgestellte Unterstützung bedient alle wesentlichen Aspekte der gestellten Anforderungen. Entsprechend ergänzt die vorhandene Unterstützung zur Laufzeit die verbleibenden Anforderungen hinsichtlich der erleichterten Ausführbarkeit der erstellten Spezifikationen.

In Bezug auf Anforderung 1 (*Unterstützung zur Verhaltensmodellierung*) steht mit der aufgezeigten *Kernsprache* eine Plattform zur Verfügung, auf deren Basis verhaltensorientierte Spezifikationen modelliert werden können. Die zugehörige *Ausführungsmaschine* setzt dies zur Laufzeit des modellierten Systems um. Das hohe Niveau der Kernsprache sowie die Möglichkeit zur *regelbasierten Übersetzerspezifikation* berücksichtigen Anforderung 2 bezüglich des betrachteten Abstraktionsgrads.

Die Anpassbarkeit der Lösung sowie des Gesamtsystems zur Realisierung weiterer Spezifikationssprachen deckt sich mit Anforderung 3. Einerseits ermöglicht die regelbasierte Übersetzerspezifikation eine leichte Anpassbarkeit des Übersetzers im Falle einer Evolution der realisierten Sprache. Andererseits erlaubt die Erweiterbarkeit der Kernsprache eine leichte Abdeckung neuer zu unterstützender Sprachkonstrukte sowie die Realisierung weiterer Spezifikationssprachen.

Schließlich wird zur Laufzeit das Graphdatenbanksystems DRAGOS als graphbasierte Zugriffsschicht eingesetzt. Dieses ermöglicht den Einsatz unterschiedlicher Speichertechnologien zur persistenten Datenablage, wodurch das resultierende System gemäß Anforderung 4 an keine spezifische Plattform gebunden ist.

Nicht behandelte Aspekte

Aufgrund der Komplexität der zugrundeliegenden Thematik kann diese Arbeit keine in sich vollständige Lösung zur Entwicklung ausführbarer Spezifikationssprachen bieten. Um eine Eingrenzung der im Folgenden behandelten Thematik zu ermöglichen, werden die getroffenen Einschränkungen hier kurz diskutiert.

Unberücksichtigt bleibt insbesondere der obere Teil von Abbildung 1.4, der die *Ausarbeitung* von Spezifikationssprachen sowie den Bau von *Editoren* umfasst. Dies schließt auch die *statische Analyse* von erstellten Spezifikationen ein. Die hierfür erforderliche Funktionalität wird von verschiedenen Rahmenwerken bereits angeboten die mit den eingesetzten Technologien verknüpft werden können. Kapitel 3 zeigt dies für gängige Lösungen auf Basis der Eclipse-Plattform [ML05] auf.

Die *formale Definition* von Spezifikationssprachen liegt nicht im Schwerpunkt dieser Arbeit, da hierzu bereits zahlreiche theoretische Arbeiten existieren. Der Fokus liegt hier stattdessen auf einer praktikablen Unterstützung des Werkzeugbaus. Zudem wird die formale Definition einer Spezifikationssprache in praktischen Szenarien insoweithäufig erschwert, als dass die Struktur der zu verarbeitenden Daten erst zur Laufzeit bekannt ist. Beispielsweise ist dies für eine informationsverarbeitende Anfragesprache üblich. Weitere Ansätze wie etwa [Var02, KGKK02] stellen Formalisierungsansätze für die UML oder auch für domänenspezifische Sprachen vor. Zwar ermöglichen diese eine formale Analyse der Spezifikationsdokumente, allerdings streben diese keine werkzeugtechnische Ausführung an.

Bezüglich der Art der unterstützten Spezifikationssprachen ist anzumerken, dass vornehmlich operationale Sprachen bzw. Teilsprachen durch die präsentierten Konzepte unterstützt werden. Zur Realisierung statischer Spezifikationssprachen bieten vorlagenbasierte Quelltextgeneratoren bereits ein bequem nutzbares und industriell erprobtes Mittel an. Wie in Abschnitt 1.2 aufgezeigt wurde, sind diese allein jedoch nicht zur Realisierung verhaltensorientierter Sprachen geeignet.

Technisch basiert die hier vorgestellte Lösung auf der am Lehrstuhl für Informatik 3 bereits verfügbaren *nicht-standard Infrastruktur*. Obwohl, wie im folgenden Kapitel erläutert wird, dadurch keine Einschränkung für das modellierte System hinsichtlich Anforderung 4 entsteht, könnte der Einsatz standardbasierter Lösungen dennoch als vorteilhaft erachtet werden. Tatsächlich ist die implementierte Lösung als Prototyp zu verstehen, der zwar Interoperabilität mit entsprechenden Standards anbietet, dennoch aber nicht nativ auf diesen basiert. Allerdings bietet die verfügbare Infrastruktur Vorteile gegenüber einschlägigen Standards, die zur Realisierung der hier vorgestellten Konzepte genutzt werden konnten. Eine Vertiefung dieses Aspekts erfolgt in Abschnitt 2.4.

Erzielter Beitrag

Als Resultat dieser Arbeit können verhaltensorientierte Spezifikationen in der MDSE besser nutzbar gemacht werden. Trotz zahlreicher bestehender Sprachen und zugehöriger Ausführungsmaschinen wird in dieser Hinsicht ein Fortschritt erzielt, da die Schaffung neuer und insbes. domänenspezifischer Sprachen erleichtert wird. Hierdurch kann existierenden Vorbehalten etwa im Bezug auf das zu erhaltende Abstraktionsniveau einer Spezifikation begegnet werden. Die Art der Unterstützung ist in dieser Hinsicht zweiteilig: Einerseits werden verschiedene *Werkzeuge* bereitgestellt, mit denen sich eine ausführbare Spezifikationssprache zumindest bis zu einer prototypischen Form realisieren lässt. Andererseits entsteht durch die ausgearbeiteten *Konzepte* die Möglichkeit, existierende Lösungen entsprechend zu erweitern oder eine Neurealisierung bspw. unter geänderten Rahmenbedingungen wie der gewählten Programmiersprache vorzunehmen. Dies wäre insbesondere unter der Zielsetzung einer industriell nutzbaren Werkzeugkette zum Bau verhaltensorientierter Spezifikationssprachen sinnvoll. In diesem Zusammenhang könnten auch die oben aufgezählten Einschränkungen überprüft und ggf. behoben werden, um eine weiter reichende Lösung zu realisieren.

Neben dem soeben behandelten softwaretechnischen Beitrag behandelt diese Arbeit auch verschiedene wissenschaftliche Fragestellungen aus dem Bereich der *Graphtransformationen* bzw. *Graphgrammatiken*. So wird untersucht, in wie weit ausführbare Spezifikationen mittels einer *modularen Sprache* aufgebaut werden können, um eine leichte Anpassbarkeit zu ermöglichen. Existierende Spezifikationssprachen sehen zwar teilweise Erweiterungsmöglichkeiten vor, doch insbesondere die Möglichkeit zur integrativen Auswertung solcher erweiterten Sprachkonstrukte wird bislang kaum geboten. Die *variable Sprachrealisierung* ermöglicht zudem, verschiedene Ansätze zur Ausführung von Graphtransformationsregeln zu verknüpfen. Sowohl die hier vornehmlich betrachtete interpretative Variante als auch die Quellcodegenerierung etwa für SQL Datenbanken werden bereits in der Literatur behandelt. Neu hinzugekommen ist an dieser Stelle die Möglichkeit, die jeweilige Herangehensweise erst bei der Ausführung des modellierten Systems zu wählen. Ferner wird eine Möglichkeit zur *Übersetzerspezifikation* bereitgestellt, die über derzeit verbreitete Ansätze hinausgehen. Insbesondere werden Spezifika dieses Sachgebiets untersucht, welche die Übersetzung von Spezifikationssprachen in eine vorgegebene Zielsprache betreffen, anstelle zwischen allgemeinen graphischen Strukturen zu transformieren.

1.5 Struktur der Arbeit

Diese Arbeit gliedert sich wie folgt:

Kapitel 2 beschreibt die für das Verständnis dieser Arbeit notwendigen Grundlagen. Aus dem Bereich graphbasierter Werkzeuge zählen hierzu Graphtransformationen einerseits sowie Graphspeicherlösungen andererseits.

Kapitel 3 vertieft den Überblick des vorangegangenen Abschnitts 1.4 und beschreibt insbesondere die Randbereiche der Zielsetzung bzw. der hier einsetzbaren existierenden Technologien. Das oben eingeführte Beispiel einer zu realisierenden Anfragesprache, auf das sich die weiteren Kapitel beziehen, wird an dieser Stelle ausführlich beschrieben.

Kapitel 4 behandelt die durch das vorgestellte Rahmenwerk bereitgestellte Kernsprache. Hierzu werden die kontextfreie Syntax, kontextsensitive Wohlgeformtheitsbedingungen sowie die operationale Semantik der einzelnen Sprachkonstrukte formal definiert. Neben der mehrschichtig konzeptionierten Spracharchitektur stellt dieses Kapitel auch die erreichte Erweiterbarkeit der Sprache vor.

Kapitel 5 geht auf die Ausführung der in Kapitel 4 vorgestellten Kernsprache ein. Schwerpunkt hier ist die Architektur dieser Maschine sowie deren Erweiterbarkeit bzgl. Spracherweiterungen.

Kapitel 6 erläutert den in dieser Arbeit vorgestellten Ansatz, operationale Spezifikationssprachen regelbasiert zu übersetzen. Die zu diesem Zweck eingerichtete Übersetzungssprache wird vorgestellt und erläutert.

Kapitel 7 fasst die Resultate der Arbeit zusammen und gibt Ansatzpunkte für zukünftige Arbeiten.

2 Grundlagen

Dieses Kapitel führt zunächst theoretische und technische Grundlagen ein auf denen die vorliegende Arbeit basiert. Da die in der MDSE erstellten Modelle hauptsächlich diskrete, jedoch komplex strukturierte Sachverhalte beschreiben, bieten sich *Graphen* als grundlegendes Ausdrucksmittel an. Um verhaltensorientierte Spezifikationen hierauf zu beschreiben werden strukturierte Veränderung von Graphen benötigt, die in Form von *Graphgrammatiken* dargestellt werden können. Weitere Möglichkeiten zur verhaltensorientierten Spezifikation stellen die Übersetzung und die Auswertung von Modellen dar, für die ebenfalls graphorientierte Ansätze vorhanden sind.

Im Anschluss an die konzeptionelle Einführung graphbasierter Technologien in Abschnitt 2.1 nennt Abschnitt 2.2 verfügbare Werkzeugimplementierungen aus diesem Bereich. Verwandte industrielle Standards werden in Abschnitt 2.3 aufgegriffen. Gegenstand der Diskussion in Abschnitt 2.4 ist der Graphspeicher DRAGOS, der als technische Grundlage der in den folgenden Kapiteln erläuterten Konzepte dient.

2.1 Graphbasierte Spezifikationssprachen

Formales Spezifizieren mit Hilfe von Graphtransformationen geht auf die Arbeiten von Pfaltz und Rosenfeld [PR69] sowie Schneider [Sch70] zurück, die diese in den 1970ern unabhängig voneinander beschrieben haben. Grundsätzlich wird dabei die Ableitung einer Zielgraphstruktur aus einer Ursprungsgraphstruktur durch die unterscheidenden Teilmengen beider Seiten dargestellt. Wie in Abbildung 2.1 gezeigt wird ersetzt ein Ableitungsschritt die graphisch hervorgehobene Teilmenge des Ursprungsgraphen durch die entsprechende Teilmenge des Zielgraphen. Die im unteren Teil der Abbildung gezeigten Ausschnitte bilden zusammen eine *Graphersetzungsregel*. Dabei stellt die gestrichelte Linie eine sogen. Einbettungsüberführung dar, die ein unverändertes Beibehalten dieses Knotens während der Transformation bewirkt. Insbesondere ist der dargestellte Knoten auf Ursprungs- und Zielseite mit den gleichen Kanten verbunden die selbst nicht Teil der ersetzten Teilmenge sind.

Eine Menge von Graphersetzungsregeln definiert als Graph*grammatik* alle hiermit erzeugbaren Graphstrukturen und damit eine formale Graph*sprache*. Auch hier bestehen die üblichen Unterscheidungen zwischen kontextfreien und kontextsensiti-

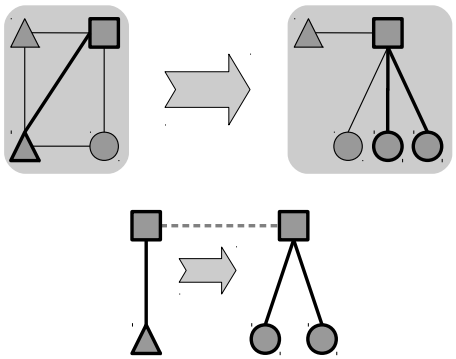


Abbildung 2.1: Einfache Graphersetzung

ven Grammatiken. Während kontextfreie Graphgrammatiken lediglich ein einzelnes quellseitiges Element aufweisen, können kontextsensitive Graphgrammatiken komplexere Suchmuster wie in obigem Beispiel einsetzen. Obwohl nur im kontextfreien Fall bspw. das Wortproblem effizient entscheidbar ist, hat sich in praktischen Anwendungen die Ausdruckskraft kontextsensitiver Graphgrammatiken durchsetzen können. Werden die möglichen Ableitungen einer Graphgrammatik durch zusätzliche Kontrollflussspezifikationen weiter eingeschränkt wird diese als *Graphtransformationssystem* bezeichnet. Hiermit können vielfältige softwaretechnische Sachverhalte spezifiziert werden, wie etwa Zustandsübergänge in einem Softwaresystem.

Zur formalen Beschreibung kontextsensitiver Graphgrammatiken sind eine Reihe von Ansätzen vorgeschlagen worden, die sich insbesondere in ihrer Herangehensweise und der erzielbaren Ergebnisse unterscheiden. Der auf mathematischen Algebren basierende Double-Pushout-Approach [CMR+97] ist aufgrund seiner nachweisbaren Eigenschaften der verbreitetste Ansatz. Dem gegenüber steht im mengenorientierten Ansatz nach Nagl [Nag78] bzw. den darauf basierenden Arbeiten von Schürr [Sch91] zur programmierten Graphersetzung ein erweiterter Sprachumfang und damit die praktische Anwendbarkeit im Vordergrund. Ferner hat mit den sogen. Hypergraphgrammatiken [HK86] auch ein Formalismus speziell für kontextfreie Graphgrammatiken Verbreitung gefunden.

Im Folgenden werden einige für den weiteren Verlauf der vorliegenden Arbeit besonders relevanten Grundbegriffe weiter vertieft. Da an dieser Stelle keine ausführliche Darstellung des theoretischen Unterbaus sowie der darauf basierenden Werkzeuge möglich ist, wird als einleitende Literatur insbesondere auf die ersten Bände der *Handbooks of Graph Grammar* [Roz97, EEKR99] verwiesen.

2.1.1 Einfache Graphmodelle

Ein Graphmodell definiert die Beschaffenheit einer Graphstruktur und legt insbesondere die hierin auftretenden Elemente fest. Als das wohl einfachste praktisch angewendete Graphmodell wird hier zunächst das Modell *gerichteter, knoten- und kantenmarkierter Graphen* vorgestellt. Zusätzlich werden Erweiterungen des Grundmodells umrissen, die im Graphmodell des DRAGOS-Graphspeichers zusammengeführt werden.

Gerichtete knoten- und kantenmarkierte Graphen

Ein gerichteter, knoten- und kantenmarkierter Graph (gkk-Graph) ist über einem endlichen Beschriftungsalphabet $\Sigma := \langle \Sigma_V, \Sigma_E \rangle$ als Tripel

$$G := (V, E, l^V)$$

wie folgt definiert:

- V ist eine endliche Menge von Knotenbezeichnern
- $E \subseteq V \times \Sigma_E \times V$ eine ebenfalls endliche *Menge* beschrifteter Kanten zwischen je einem Start- und einem Zielknoten. Aufgrund der Mengeneigenschaft können in diesem Graphmodell somit keine parallelen Kanten gleicher Beschriftung zwischen zwei Knoten existieren.
- $l^V : V \to \Sigma_V$ ist eine Beschriftungsfunktion, die einem Knoten einen Knotenbezeichner zuordnet. Die Abbildung wird üblicherweise nicht injektiv gewählt und erlaubt quasi eine einfache Charakterisierung bzw. Typisierung von Knoten.

Die Menge aller gerichteter knoten- und kantenmarkierter Graphen über einem Beschriftungsalphabet Σ wird als $\mathcal{G}_\Sigma^{gkk}$ bezeichnet. Im Folgenden nicht näher bezeichnete Graphen entstammen implizit dieser Menge.

Graphen mit Kantenidentität

Eine alternative Definition des einfachen Graphmodells sieht vor, Kanten zu identifizieren und somit als eigenständige „First-Class Citizens“ zu behandeln. Hierdurch werden parallele, identisch beschriftete Kanten zwischen zwei Knoten ermöglicht. Die Definition erfolgt durch ein 6-Tupel der Form :

$$G := (V, E, s, t, l^V, l^E)$$

Im Unterschied zu oben stellt E lediglich eine endliche Kantenbezeichnermenge dar. Ferner sind

- $s, t : E \to V$ Abbildungen einer Kante auf ihre Quell- und Zielknoten, sowie
- $l^E : E \to \Sigma_E$ eine Abbildung zur Herstellung einer Kantenbeschriftung.

Attributierte Graphen

Üblicherweise reicht die Wahl einer atomaren und zudem über den Lebenszyklus eines Knotens konstante Beschriftung nicht zur Modellierung komplexer Sachverhalte aus. *Attributierte Graphen* – analog zu oben auch als *gakk-Graphen* bezeichnet – können daher um zusätzliche Informationen angereichert werden, die sich aus einer eigenen Attributalgebra ergeben. Das eingangs beschriebene Graphmodell wird somit um drei zusätzliche Komponenten erweitert. Eine detaillierte Erläuterung attributierter Graphen – und der damit zu modifizierenden Graphgrammatiken – gibt [EEPT06, Kap. 8].

- A bestimmt eine Algebra zur Darstellung von Attributwerten.
- $a_v : V \to \mathcal{U}(A)$ und $a_e : E \to \mathcal{U}(A)$ weist Knoten bzw. Kanten einen Attributwert aus A zu.

Hierarchische Graphen

In vielen praktischen Anwendungsfällen entsteht der Bedarf, *Hierarchien* zwischen einzelnen Teilstrukturen abbilden zu können. Hierunter wird insbesondere eine Existenzabhängigkeit verstanden, so dass bei Löschung eines übergeordneten Elements alle enthaltenen ebenfalls gelöscht werden. Auch Sichtbarkeitsregeln lassen sich hierdurch spezifizieren, bspw. in dem Verbindungen zwischen Graphelementen nur auf gleicher Hierarchieebene innerhalb eines übergeordneten Elements zugelassen werden. Eine einfache, lediglich strukturbezogene Definition hierarchische gkk-Graphen ist damit

$$G := (V, E, l^V, g)$$

mit der Funktion $g : G \to (P)(\mathcal{G}_\Sigma^{hgkk})$ die einem Graphen die Menge der enthaltenen Graphen zuzuordnen.

2.1.2 Graphgrammatiken

Unter Zuhilfenahme obiger Graphmodelle sowie deren zugeordneter Graphmorphismen lassen sich die Ableitungsschritte einer Graphgrammatik formal darstellen. Ausgehend von den Arbeiten von Schneider und Ehrig [SE76] hat sich hierzu eine kategorientheoretische Beschreibung von Graphtransformationen durchsetzen können.

Hierbei werden Graphen als abstrakte *Objekte* aufgefasst, die mittels entsprechender *Homomorphismen* einander zugeordnet werden.

Graphmorphismen

Als Homomorphismus wird eine Funktion bezeichnet, die zwei mathematische Objekte strukturerhaltend miteinander in Relation setzt. Für *Graphen* als spezielle Objekte werden Homomorphismen anhand des jeweils verwendeten Graphmodells definiert, wobei im Folgenden von einfachen Graphen der Familie $\mathcal{G}_{\Sigma}^{gkk}$ ausgegangen wird. Als Minimaleigenschaften eines solchen Homomorphismus $h : G \xrightarrow{h} H$ wird vorausgesetzt, dass dieser

1. beschriftungserhaltend ist, also sowohl Knoten- als auch Kantenbeschriftungen beibehält, und
2. verbindungserhaltend ist, das Bild einer Kante also inzident zu den Bildern ihrer Quell- und Zielknoten ist.

Ein Homomorphismus h für Graphen der Familie $\mathcal{G}_{\Sigma}^{gkk}$ besteht aus zwei Komponenten $\langle h_V, h_E \rangle$ jeweils zur Abbildung der Knoten- und Kantenmengen. Formal gilt hierbei:

- $\forall v \big(v \in V_G \rightarrow h_V(v) \in V_H \big)$
- $\forall v \big(v \in V_G \rightarrow l_G(v) = l_H(h_V(v)) \big)$
- $\forall (s_G, l_G, t_G) \big((s_G, l_G, t_G) \in E_G \rightarrow (h_V(s_G), h_E(l_G), h_V(t_G)) \in E_H \big)$

In vielen praktischen Anwendungsfällen ist es zweckmäßig, Morphismen auf *injektive* Abbildungen zu beschränken, bei der jedes Bildelement höchstens ein Urbild besitzt. Diese Variante wird demzufolge als *Monomorphismus* bezeichnet. Ist die Abbildung zudem surjektiv, so dass zu jedem Element aus H ein Urbild in G existiert, so heißt h *Isomorphismus*. In diesem Fall beschreibt h eine eineindeutige Zuordnung zwischen allen Elementen aus G und H.

Formalisierung von Graphgrammatiken

Eine Graphtransformationsregel besteht aus einer Quellseite beschrieben durch einen Graphen L und einer Zielseite in Graph R, die mittels injektiver Homomorphismen und eines Zwischengraphen K in Beziehung gesetzt werden: $L \xleftarrow{l} K \xrightarrow{r} R$. L und R modellieren einen Transformationsschritt als Vor- und Nachbedingungen des zugrundeliegenden Arbeitsgraphen. K enthält als *Klebegraph* die Schnittmenge der jeweiligen

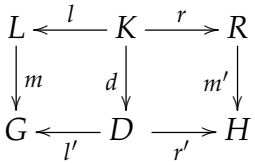


Abbildung 2.2: Pushout-Diagramm einer Graphtransformationsregel

Elemente und beschreibt somit die identisch zu ersetzenden Teile der Transformationsregel. Elemente aus L, die *nicht* in K enthalten sind, werden bei Anwendung der Graphtransformationsregel aus dem Arbeitsgraphen *gelöscht*. Elemente aus R, die nicht in K enthalten sind, dagegen *erstellt*.

Die Anwendung einer Graphtransformationsregel auf den Arbeitsgraph G wird durch eine *Auffindungsstelle* beschrieben, die als Monomorphismus m zwischen L und G definiert ist. Voraussetzung zur Anwendung der Transformationsregel $L \leftarrow K \rightarrow R$ ist – neben der Existenz von m – eine Einbettung des Klebegraphen durch den Morphismus $d : K \xrightarrow{d} D$ sowie eine Einbettung von D in G durch $l' : D \xrightarrow{l'} G$. Kommutiert der linke Teil von Abbildung 2.2, d.h. gilt für jedes $k \in K : m\big(l(k)\big) = l'\big(d(k)\big)$, und handelt es sich bei dieser Teilstruktur um einen *Pushout* für den G bis auf Isomorphie eindeutig definiert ist, so lässt sich dieser Ausgangsgraph konstruktiv und eindeutig in einen Ergebnisgraphen H überführen [EPS73]. Dieser ergibt sich als Vervollständigung der Elemente $D \leftarrow K \rightarrow R$ zu einem zweiten kommutierenden Diagramm (Pushout) mit H. Der hierbei geschilderte Graphersetzungsansatz wird demnach als *Double-Pushout-Approach* (DPO) bezeichnet.

Vorteilhaft an der algebraischen Definitionsweise ist einerseits die Abstraktion vom zugrundeliegenden Graphmodell zu werten, das durch geeignete Wahl der Morphismen sowohl einfache gerichtete Graphen als auch bspw. hierarchische Strukturen umfassen kann. Entsprechend allgemeine Grundlagen hierzu legen die sogen. *adhäsiven Ersetzungssysteme höherer Ordnung* [EHKPP90, EHPP04]. Ferner können zahlreiche formale Eigenschaften und Beweise aus anderen Kategorien auf Graphersetzungssysteme übertragen werden, bspw. die Church-Rosser Eigenschaft zur Ermittlung kanonischer Ableitungssequenzen analog zu Termersetzungssystemen [Ros75].

Nichtsdestotrotz sind Graphtransformationen gemäß dem DPO-Ansatz hinsichtlich ihrer Anwendbarkeit eingeschränkt. Dies ergibt sich indirekt aus der Pushout-Eigenschaft des linken Diagrammteils, welche die sogen. *Gluing-Condition* impliziert: Eine Anwendung der Transformationsregel kann nur dann ein Element, bspw. einen Knoten *löschen*, wenn alle inzidenten Kanten ebenfalls in der linken Regelseite aufgeführt – und ggf. ebenfalls gelöscht – werden. Dies stellt aus Praxissicht eine erhebliche Einschränkung dar, da somit keine Graphersetzungen in unbekannten Kontexten

spezifiziert werden können. Ein in dieser Hinsicht verallgemeinerter Ansatz stellt der Single-Pushout-Approach [EHK+97] dar, bei dem die Ableitung der Ausgangs- zur Ergebnisstruktur in einem einzelnen Pushout beschrieben wird. Hierdurch ergibt sich jedoch eine ärmere formale Basis, da bspw. die eindeutige Invertierbarkeit der Regelanwendung verloren geht.

Zur Erhöhung ihrer Ausdruckskraft sind verschiedene Erweiterungen der algebraischen Graphtransformationsansätze vorgeschlagen worden. Hierzu zählt die Einbeziehung *amalgamierter* Teilgraphen, bei denen Teilregeln mehrfach innerhalb eines Transformationsschritts angewendet werden [TB93]. Dieser Ansatz lässt sich auch zur Beschreibung *verteilter Graphtransformationen*, die simultan sogen. Schnittstellengraphen und lokale Graphen transformieren [Tae94], einsetzen. Ferner wird die Verwendung von Graphtransformationen in unbekannten Kontexten von [EHL+98] mittels sogen. *Double-Pullback*-Transformationen unterstützt, die anders als im DPO-Ansatz von simultanen, jedoch unbekannten Veränderung des Arbeitsgraphen ebenfalls im Kontext verteilter Systeme ausgehen. Als Erweiterungen der Ausdrucksfähigkeit werden in [HHT96] *negative* Anwendungsbedingungen vorgestellt, die bei Vorhandensein einer Auffindungsstelle die Transformationsausführung verhindern. Zudem wird in [Gd07] eine Möglichkeit zur *rekursiven* Spezifikation von DPO-Ersetzungsregeln angegeben.

2.1.3 Programmierte Graphersetzung

Graphgrammatiken weisen insbesondere für Anwendungen im softwaretechnischen Umfeld ein hohes Potenzial auf, da in diesem Kontext häufig graphbasierte Datenstrukturen verarbeitetet werden. In praktischen Szenarien würden jedoch eine Vielzahl komplexer Graphersetzungsregeln zur Modellierung benötigt werden, wodurch der gewonnene Abstraktionsgrad verloren geht. Ein wesentlicher Beitrag für die praktische Anwendbarkeit von Graphgrammatiken stellt somit die programmierte Graphersetzung dar, bei der einzelne Ersetzungsregeln bspw. durch Sequenzen, Alternativen oder Iterationen verschaltet werden können. Hierdurch fallen die einzelnen Ersetzungsregeln kompakter aus, da nicht pro Regel eine allgemeine Anwendbarkeitsbedingung spezifiziert werden muss. Stattdessen wird eine Ersetzungsregel nur dann angewendet, wenn neben ihrer Vorbedingung auch der modellierte Kontrollfluss eine Anwendung erlaubt. Dem praktischen Gewinn steht gegenüber, dass die theoretischen Ergebnisse, bspw. des DPO-Ansatzes, nicht oder nur teilweise auf ein solches Graphersetzungssystem übertragbar sind.

Moderne in der Praxis etablierte Graphersetzungssprachen zählen i.d.R. zur Kategorie programmierter Graphersetzungen. Dies gilt einerseits für die allgemein anwendbaren Sprachen PROGRES, Fujaba, VMTS und VIATRA, andererseits auch für

auf Spezialdomänen ausgerichtete Lösungen wie GROOVE und GReAT. Nichtsdestotrotz hat mit AGG auch eine Spezifikationssprache Verbreitung gefunden, die nicht dieser Kategorie zuzuordnen ist, sondern allein auf den Grundlagen der Graphgrammatiken basiert.

Die formale Beschreibung einer solchen Spezifikationssprache erfolgt für PROGES in [Sch91]. Dabei wird eine ganzheitliche syntaktische und semantische Beschreibung des Typsystems, einzelne Ersetzungsregeln sowie deren Kombination durch Kontrollflusselemente vorgestellt. Schürr fasst die verarbeiteten Graphen als prädikatenlogische Formeln auf und definiert den Kontrollfluss der Regelanwendungen durch eine Least-Fixpoint-Semantik. Im Kontext von Fujaba definiert [Zü01] die Semantik der sogen. Story-Diagramme[FNTZ98] durch Mengenoperationen auf einem zugrundeliegenden Arbeitsgraphen. Kontrollflussbeziehungen zwischen Regeln werden dabei induktiv ebenfalls über Mengenoperationen definiert.

2.1.4 Graphübersetzung

Die schablonenartige Transformation zwischen Graphstrukturen, die auch als Übersetzung zwischen zwei Graphsprachen verstanden werden kann, hat sich in der jüngeren Vergangenheit zu einem softwaretechnischen Standardverfahren entwickelt. Erste Anwendungen eines graphbasierten Ansatzes gehen bspw. auf die Kopplung unterschiedlicher Softwarewerkzeuge auf einer gemeinsamen Datenbasis zurück [NS94]. Auch das Reengineering von Datenbanken und eine damit einhergehende Schemamigration ist bereits frühzeitig verfolgt worden [Jah99]. Ferner lassen sich unspezifisch definierte Modellierungssprachen wie bspw. UML Aktivitätsdiagramme dadurch mit einer konkreten Bedeutung hinterlegen, indem eine Übersetzung in formale Algebren angegeben wird. Wie in [VAB+07] gezeigt, sind hierfür eine Vielzahl von Graphtransformationswerkzeugen mit unterschiedlicher Eignung vorhanden, die zu unterschiedlich kompakten und verständlichen Übersetzerspezifikationen führen.

Kernaspekt einer Graphübersetzung ist die Beschreibung einer Korrespondenz unterschiedlicher Graphstrukturen, bei der Graphgrammatiken zum Einsatz kommen. Der wohl verbreitetste Ansatz in diesem Kontext sind die Tripel-Graph-Grammatiken (TGG) nach Schürr [Sch94]. Neben den beiden in Bezug zu setzenden Graphen wird hierbei ein expliziter dritter Korrespondenzgraph zur Festlegung der jeweiligen Zusammenhänge verwendet. Eine TGG definiert Graphersetzungen auf allen drei Graphen simultan und beschreibt so deren gemeinsamen Aufbau ausgehend von den jeweiligen Startgraphen.

Durch eine TGG wird grundsätzlich eine bidirektionale Abbildung beschrieben, die miteinander in Bezug gesetzte Graphsprachen können somit jeweils ineinander übersetzt werden. Ferner eignen sie sich einerseits für „batchartige“ Übersetzung, ande-

rerseits ist auch ein inkrementelles Verfahren unter Berücksichtigung von *Änderungen* in den jeweiligen Dokumenten möglich. Hieraus lassen sich wiederum durchzuführende Änderungen in den anderen Graphen ableiten.

2.1.5 Graphanfragen

Schließlich können Graphstrukturen wie in der klassischen Informationsverarbeitung hinsichtlich vorgegebener Kriterien analysiert werden. Im Kontext des Reengineering bestehender Softwaresysteme ist dazu bspw. die Anfragesprache GReQL[KW99] vorgestellt worden, die somit auch Endbenutzern des Systems höherwertige Auswertungsfunktionen anbieten kann. Bspw. können so Informationen über ein analysiertes Softwaresystem unter Zuhilfenahme von Graphtraversierungen verdichtet werden. Die Relevanz einer Klasse könnte somit numerisch über die Anzahl aller statischen Aufrufe ihrer Methoden definiert werden.

Neben der Informationsgewinnung ist auch eine Prüfung von Graphstrukturen hinsichtlich der Wohlgeformtheitsbedingungen einer Graphsprache denkbar. So kann eine Graphsprache dahingehend eingeschränkt werden, das alle in Form von Graphanfragen formulierte Wohlgeformtheitsbedingungen für alle enthaltenen Graphen das boolesche Ergebnis *wahr* liefern.

2.2 Verfügbare Werkzeuge

Derzeit ist eine Vielzahl graphbasierter Werkzeuge verfügbar, von denen nur einige im Hinblick auf obige Themenbereiche vorgestellt werden können.

2.2.1 Beschreibung von Graphmodellen & Persistierung

Die Beschreibung von Datenstrukturen mittels Graphmodellen (s. Abschnitt 2.1.1) wird durch zahlreiche Modellierungsrahmenwerke und -umgebungen unterstützt. Hierzu zählt beispielsweise das Eclipse Modeling Framework (EMF) [SBPM08], das eine zentrale Rolle in der Eclipse Entwicklungsumgebung einnimmt. Mittels EMF können Datenstrukturen durch Metamodelle beschrieben und über standardisiert generierten Code leicht programmatisch nutzbar gemacht werden. Zudem erlaubt eine reflektive Zugriffsschicht sowie die Unterstützung dynamischer Metamodelle ohne Notwendigkeit zur Codegenerierung, eine generische aber dennoch typsichere Verarbeitung von Modellen. Zur persistenten Speicherung von Instanzmodellen verwendet EMF standardmäßig eine Serialisierung in das XML-basierte Austauschformat XMI

[OMG05]. Über das sogen. Connected Data Objects Model Repository lassen sich jedoch u.a. auch relationale Datenbanken einbinden.

Einen ähnlichen Ansatz verfolgt das Universal Data Model (UDM) [MBL+03] der Vanderbilt University. Im Unterschied zu EMF bietet das UDM eine leichtere Nutzung persistenter Speicherlösungen neben der einfachen Serialisierung von Modellen. Zudem wird als Zielsprache der generierten API auch C++ unterstützt.

Traditionell im Datenbankkontext verankert ist Graph Storage (GRAS) [KSW95], ein Datenbankmanagementsystem das explizit zur Verwaltung von Graphen gestaltet worden ist. Neben deren persistenten Ablage bietet GRAS auch Nebenläufigkeitskontrolle und Versionsverwaltung an. GRAS wurde in zahlreichen Revisionen weiterentwickelt, und ermöglicht seitdem den Einsatz in verteilten Systemen (RGRAS) sowie die Verwaltung hierarchischer Graphen (GRAS3, [Bau99]). In seiner neuesten Ausprägung, genannt DRAGOS [Bö06], werden austauschbare Hintergrundspeicher ähnlich dem UDM angeboten. Jedoch stellt DRAGOS komplexere Ausdrucksmöglichkeiten für Metamodelle als EMF und UDM bereit, da neben hierarchischen Graphen auch bspw. attributierte und identifizierte Kanten erlaubt werden. Als Grundlage der vorliegenden Arbeit wird DRAGOS ausführlicher in Abschnitt 2.4 behandelt.

2.2.2 Ausführung von Graphspezifikationen

Graphgrammatiken (Abschnitt 2.1.2) zur konsistenzerhaltenden Editierung von Spezifikationen werden durch zahlreiche Graphtransformationswerkzeuge bereitgestellt. Bei der Formulierungsweise wird zwischen verschiedenen Ansätzen von Graphtransformationen unterschieden: Als Vertreter des *algebraischen* Ansatzes ist beispielsweise AGG [Tae99] zu nennen, das auf dem Single-Pushout-Approach basiert. Dem gegenüber vertritt PROGRES [Sch90, Win00, RW07] den *logikorientierten* Ansatz, der gegenüber dem SPO Kontrollstrukturen zur programmierten Graphersetzung (Abschnitt 2.1.3) einführt, allerdings schwächere formale Eigenschaften aufweist. Fujaba [NNZ00] ist ein UML-basiertes Graphtransformationswerkzeug dessen Spezifikationssprache *Story-Driven Modeling* (SDM) [FNTZ98] ebenfalls der programmierten Graphersetzung zuzuordnen ist.

Viele Graphtransformationswerkzeuge sind eng mit einer Umgebung verbunden, die eine Visualisierung der jeweiligen Instanzgraphen erlaubt. Das in die Eclipse Entwicklungsumgebung integrierte TIGER [EEHT05] wird zur visuellen Ausführung von AGG Spezifikationen angeboten. Im PROGRES Projekt wird für diesen Zweck das UPGRADE Rahmenwerk [BJSW02] bereitgestellt. Ferner existieren Werkzeuge, die eine integrierte Spezifikations- und Darstellungsumgebung umfassen. Hierzu zählen VMTS [LLMC04] in Verbindung mit dessen Darstellungsumgebung VPF [MML08] sowie AToM^3 [dV02]. In beiden Systemen ist es möglich visuelle Spezifikationsspra-

chen sowohl zu beschreiben als auch innerhalb desselben Werkzeugs – teils sogar ohne dessen Neustart – zu benutzen.

2.2.3 Übersetzung von Graphstrukturen

Insbesondere zur Transformation von Graphstrukturen sind aufgrund vielfältiger Anwendungsbereiche in den letzten Jahren zahlreiche Werkzeuge vorgeschlagen worden. Allein zur Anwendung von Tripel-Graph-Grammatiken (TGG) sind eine ganze Reihe von Implementierungen vorgestellt worden, die sich z.T. in Details wie dem verwendeten Metamodell zur Datenbeschreibung unterscheiden. Ein stark an industriellen Standards ausgerichtete Implementierung stellt bspw. das Werkzeug MOFLON [AKRS06] bereit. Gemeinsam ist TGG-basierten Werkzeugen die Möglichkeit, Instanzmodelle zwischen verschiedenen Metamodellen kontinuierlich und bidirektional abzugleichen. So können neben einer batch-orientierten initialen Übersetzung nachträgliche Änderungen sowohl an Quell- als auch an Zielmodellen in das jeweilige Gegenstück überführt werden, wodurch Modelle gemäß eines definierten Regelsatzes synchron gehalten werden.

Tripel-Graph-Grammatiken sowie der TGG-ähnliche Ansatz BOTL [BM03], werden in Abschnitt 6.9.1 als Bestandteil verwandter Arbeiten näher erläutert. Eine ausführliche Klassifizierung von Werkzeugen zur Graphübersetzung kann zwar an dieser Stelle nicht gegeben werden, jedoch wird insbesondere auf die Arbeiten von Andries, et al. [AEH+99], Czarnecki [CH06], und van Gorp [van08, Kapitel 2] verwiesen. Einen ausführlichen Vergleich von VMTS, AToM^3, VIATRA [CHM+02, VP03] und AGG bietet [TEG+05].

2.2.4 Auswertung durch Graphanfragen

Werkzeuge für Graphanfragen gemäß Abschnitt 2.1.5 sind weniger verbreitet als beispielsweise Vertreter für Graphtransformationen, obwohl auch in diesem Kontext bereits zahlreiche Implementierungen entstanden sind. Frühe Ansätze sind hier in der Sprache G [ACS90] zu finden, die mit visuellen und musterbasierten Anfragen eine Alternative zu relationalen Anfragesprachen bereitstellen sollte. Neuere Ansätze finden sich beispielsweise mit der Sprache GReQL [KW99], die insbesondere Expertennutzern der Reengineringumgebung GUPRO Informationen zu einem analysierten Softwaresystem liefern soll [EKRW02].

Ferner besitzen Anfragesprachen für graphbasierte Daten im Bereich wissensbasierter Systeme und insbesondere dem Semantic Web, abseits der modellgetriebenen Softwareentwicklung, große Bedeutung. In diesem Kontext sind sowohl Anfragesprachen

für die Web Ontology Language [FHH03] als auch, auf entsprechend niedrigerem Abstraktionsniveau, Sprachen für das Resource Descriptor Framework [SPA08, Bro03] entstanden.

2.3 Industrielle Standards

Oben genannte Werkzeuge sind vornehmlich in universitärem Kontext oder innerhalb von Forschungsprojekten entstanden und besitzen daher grundsätzlich eher prototypischen Charakter. Auch kann bei Einsatz dieser Werkzeuge oft nicht gewährleistet werden, dass die so erstellten Spezifikationsdokumente in andere Formate, etwa zur Weiterverarbeitung mittels anderer Werkzeuge, übertragen werden können. Dem gegenüber hat die Object Management Group (OMG) als Industriekonsortium eine Reihe von Standards verabschiedet, unter anderem die bereits erwähnte UML. Mittels dieser Standards können Werkzeuge zur modellgetriebenen Entwicklung vereinheitlicht werden [1], wodurch übergreifende Werkzeugketten gebildet werden können. Außer die Übertragbarkeit von Spezifikationen sicherzustellen, fördern anerkannte Standards auch deren Verständlichkeit durch Entwickler unter Verwendung werkzeugunabhängiger Sprachen. So werden etwa UML Diagramme in allen verfügbaren Werkzeugen nahezu identisch dargestellt, und können demzufolge übergreifend verstanden und bearbeitet werden.

Obgleich bisherige OMG Spezifikationen keinerlei direkten Bezug auf graphbasierte Werkzeuge und Techniken nehmen, sind dennoch viele Konzepte aufgrund der Korrespondenz zwischen Modellen und Graphen übertragbar. An dieser Stelle werden daher Standards angesprochen, die seitens der OMG zur Metamodellierung (MOF), Transformation (QVT) sowie für Anfragen auf Modellen (OCL) verabschiedet worden sind. Im Folgenden wird untersucht, in wie weit diese Standards die o.g. graphbasierten nicht-standard Werkzeuge ablösen können.

2.3.1 Meta Object Facility

Die Meta Object Facility (MOF) [OMG06a] stellt einen Standard zur Definition statischer Modelle bereit. Anders als die UML, der ein breiter Ansatz zur Abdeckung unterschiedlicher Systemaspekte zugrunde liegt, enthält die MOF lediglich Elemente um statische Strukturen, ähnlich der UML Klassendiagramme zu modellieren. Dagegen spezifiziert MOF weder Ausdrucksmittel auf Planungsebene wie bspw. UML

[1] Im Bereich MDSE handelt es sich bei der OMG derzeit um die einzige Organisation mit normungsähnlicher Tätigkeit – andererseits nimmt sie keine offizielle Funktion wie bspw. VDE oder ISO ein.

Anwendungsfalldiagramme, noch legt es im Kern Beschreibungsmöglichkeiten für Systemverhalten fest.

Das mit der MOF verfolgte Hauptanwendungsgebiet ist die syntaktische Definition von *Spezifikationssprachen*, bspw. der UML, dem Common Warehouse Metamodel (CWM), oder auch der MOF selbst. Im Kontext dieser Arbeit bietet sich MOF als Standard zur Speicherung von Graphstrukturen an, da hierfür eine syntaktische Repräsentation von Modellen erforderlich ist. Tatsächlich wird ein vereinfachter Kernteil der an sich komplexen MOF, die sogenannte *Essential MOF* (EMOF), direkt von einigen Werkzeugen und Rahmenwerken unterstützt. Beispielsweise zählt hierzu das bereits erwähnte Rahmenwerk EMF.

Nachteilig ist anzumerken, dass MOF und insbesondere EMOF lediglich objektorientierte Konzepte anbieten. Hierarchische Strukturen sowie identifizier- und attributierbare Referenzen zwischen Objekten werden von MOF nicht angeboten. Dedizierte Graphspeicherlösungen wie GRAS3 oder DRAGOS bieten an dieser Stelle ein mächtigeres Datenmodell an.

2.3.2 Query/View/Transformation

Der Query/View/Transformation Standard (QVT) [OMG08] ergänzt die MOF um eine Teilsprache zur automatisierten Verarbeitung von Modellen. Obwohl der Name QVT eine breite Aufstellung vermuten lässt, liegt das Hauptaugenmerk dieses Ansatzes auf der *Übersetzung* von Modellen. Nicht beachtet in der gegenwärtigen Fassung werden Modellsichten, mittels derer abgeleitete Modelle von einem Ursprungsmodell gebildet werden können. Anfragen, die Informationen aus einem Modell extrahieren, werden durch die unten erläuterte OCL abgedeckt.

QVT unterstützt drei verschiedene Ansätze zur Modellierung von Übersetzern. Die Teilsprachen *Core* sowie *Operational Mappings* gewährleisten dabei einen vergleichsweise *geringen Abstraktionsgrad* von der technischen Ausführungsmaschine. Insbesondere beruhen sie auf expliziten Operationen, etwa zum Traversieren und Erstellen von Modellen. Da diese Sprachen zudem eine rein textuelle Syntax verwenden, sind sie konzeptionell vergleichbar mit herkömmlichen Programmiersprachen. Zumindest wird jedoch eine Abstraktion von der zugrundeliegenden Persistenzlösung angeboten, die durch das jeweilige Ausführungswerkzeug bereitgestellt werden muss.

Die Teilsprache *Relations* erlaubt dagegen eine *deklarative* und auf Wunsch teilweise *visuelle* Spezifikation von Modellkorrespondenzen. Die dazu modellierten Regeln können von einer standardkonformen Ausführungsmaschine zur Übersetzung von Modellen interpretiert werden. Dies erfolgt gemäß Standard *inkrementell*, sodass Veränderungen am Ausgangsmodell stückweise in das Zielmodell überführt werden. Da Zielmodelle eigenständig bearbeitet werden können, ist diese Eigenschaft nicht nur

vom Effizienzstandpunkt, sondern auch bezüglich der Benutzbarkeit von großer Bedeutung. Ferner können Übersetzungen grundsätzlich auch in umgekehrter Richtung erfolgen, also bspw. von einem vorab erzeugten Zielmodell zurück zu einem korrespondierenden Quellmodell. Der QVT Standard trifft hier jedoch keine hinreichend präzise Festlegung bezüglich der zu diesem Zweck erforderlichen Funktionalität, etwa dem Zusammenführen paralleler Veränderungen an beiden Modellen.

Insgesamt ist der QVT Standard als ambitionierter Ansatz zu werten um ausführbare Modelle im Kontext der MOF zu definieren. QVT ist zunächst auf Übersetzungen zwischen Modellen ausgelegt, könnte jedoch auch für Transformationen innerhalb einzelner Modelle verwendet werden und ist somit allgemein zur Modellverarbeitung einsetzbar. Derzeit besteht jedoch keine nennenswerte Verbreitung umfassender QVT Werkzeuge. Lediglich die auf niedrigem Niveau angesiedelten Sprachen *Core* und *Operational Mappings* werden bislang in größerem Umfang unterstützt. Nicht beachtet vom QVT Standard werden verschiedene unter graphbasierten Werkzeugen bereits verbreitete Sprachkonstrukte. Dazu zählt beispielsweise das rekursive Auffinden von Strukturen im Quellmodell, die in Form von *Pfaden* unter graphbasierten Werkzeugen etabliert sind. Im Bereich graphbasierter Werkzeuge verfolgen *Tripel-Graph-Grammatiken* (TGGs, so.) [Sch94] eine mit dem QVT Standard vergleichbare Zielsetzung. TGGs besitzen darüber hinaus eine klar definierte Semantik und werden von einer Reihe von Werkzeugrealisierungen wie etwa dem oben genannten MOFLON unterstützt.

2.3.3 Object Constraint Language

Die Object Constraint Language (OCL) [OMG06b] ist historisch als Bestandteil der UML entstanden, um mittels dieser erstellte Spezifikationen präziser und damit verständlicher formulieren zu können. Eine Anwendung der OCL besteht darin, Klassen eines Klassendiagramms mit Invarianten zu belegen, wodurch die Menge ihrer gültigen Instanzen gemäß der Vorstellung des Entwicklers formal eingeschränkt werden kann. Analog dazu kann dynamisches Verhalten, beispielsweise von Methoden, durch Angabe von Vor- und Nachbedingungen nach dem Design-by-Contract Prinzip spezifiziert werden.

Mittlerweile wird die OCL als eigenständiger Standard geführt. Neben ihrer ursprünglichen Verwendung in UML Diagrammen können auch MOF Spezifikationen um OCL Bedingungen angereichert werden. Da MOF wiederum als Grundlage zur Definition der UML dient, wird auch die UML selbst durch OCL Invarianten präzisiert. Zusätzlich kommt die OCL zur Formulierung von Modelltransformationen mittels QVT zum Einsatz.

Im Bezug auf graphbasierte Werkzeuge bietet sich die OCL grundsätzlich dazu an,

um Anfragen auf Graphstrukturen zu formulieren. Derzeit bietet OCL allerdings auch in der aktuellen Version 2.0 keine Unterstützung für verschiedene Sprachkonstrukte, die in diesem Zusammenhang benötigt werden. Wie bereits von Schürr in [Sch01] diskutiert, zählt hierzu beispielsweise der transitive Abschluss von Graphmustern. Mittels OCL ist dies nur durch Verkapselung des abgeschlossenen Teilmusters in einer eigenen Hilfsfunktion und deren rekursivem Aufruf möglich. Ebenfalls wenig unterstützt werden Aggregationsfunktionen auf Basis von Wertemengen [HKSS08]. Der Standard sieht hier lediglich die Funktionen `sum` und `count` vor. In ihrer derzeitigen Form kann die OCL daher zwar als Grundlage für Graphanfragen verstanden werden, nicht jedoch als adäquate und bequem anwendbare Spezifikationssprache.

Zwischenfazit: Seitens der OMG sind bereits verschiedene Standards verabschiedet worden, die Funktionalität häufig im Forschungskontext entwickelter Werkzeuge standardisieren. Allerdings decken die Standards häufig nur einen Teil der erforderlichen Ausdrucksmöglichkeiten, bspw. in der OCL ab. Gleichzeitig stellt die hohe Komplexität der Standards, im Fall der UML eine 800-seitige Spezifikation, ein Problem für unterstützende Werkzeuge dar, die daher häufig nur einen bestimmten Teil eines Standards abdecken. Insbesondere im Bereich formaler Verhaltensmodellierung kann daher noch nicht von einem zufriedenstellenden Grad der Standardisierung ausgegangen werden. Graphbasierte Werkzeuge werden also auch in Zukunft weiterhin zur fallweisen Lösung von softwaretechnischen Problemen entwickelt und herangezogen werden.

2.4 Der Graphspeicher DRAGOS

Die praktische Umsetzung von Graphtransformationssystemen erfordert eine Möglichkeit zur Verwaltung und persistenten Ablage der verarbeiteten Instanzgraphen. Analog zu herkömmlich realisierten Informationssystemen, bei denen häufig relationale Datenbanken für den Datenzugriff zum Einsatz kommen, sind auch bei Graphtransformationssystemen Performanz, Speicher- und Nebenläufigkeitsverhalten zu berücksichtigen. Aus diesem Grund werden am Lehrstuhl für Informatik 3 Nichtstandard Datenbanksysteme entwickelt und eingesetzt, die auf ein graphorientiertes Datenmodell zurückgreifen. Hierdurch lassen sich die speziellen Anforderungen graphbasierter Anwendungen optimal abdecken.

Das Graphspeichersystem DRAGOS, auf dessen Technologie die Ergebnisse dieser Arbeit beruhen, stellt die jüngste Inkarnation dieser Klasse von Datenbanksystemen

dar. Ausgehend von dem Bedarf die bislang architektonisch wenig flexiblen Vorgängersysteme abzulösen, führt DRAGOS eine komponentenbasierte Strukturierung zur Verstärkung der Parametrisierbarkeit ein. Somit kann bspw. in einfachen lokalen Umgebungen auf komplexe transaktionale Datenspeicher zugunsten einer höheren Performanz verzichtet werden. Dagegen kann durch Umkonfiguration auch auf komplexe Komponenten kommerzieller Anbieter gewechselt werden, um bspw. industriellen Anforderungen gerecht zu werden.

Im Folgenden wird kurz die Gesamtarchitektur des Systems vorgestellt sowie das verwendete Graphmodell detailliert erläutert. Letzteres wird zudem formal beschrieben um in Kapitel 4 zur Definition der Kernsprache darauf zurückgreifen zu können. Für eine ausführliche Darstellung des DRAGOS Graphspeichers wird auf [Bö06] verwiesen.

2.4.1 Gesamtarchitektur

DRAGOS liegt eine komponentenbasierte Architektur zugrunde, die eine Zerlegung in voneinander entkoppelte Bestandteile ermöglicht. Insbesondere können dadurch Implementierungen ausgetauscht und an spezifische Anforderungen angepasst werden.

Einen Überblick der Gesamtarchitektur des DRAGOS Graphspeichers zeigt Abbildung 2.3. Die zuunterst dargestellte Schicht beinhaltet extern bereitgestellte Dienstkomponenten, wie bspw. relationale Datenbanken als persistente Datenspeicher. Diese bieten i.d.R. auch erweiterte Funktionalität wie Ereignis- und Transaktionsmanager an. In verteilten Systemen ist an dieser Stelle der Einsatz einer Middlewarelösung üblich.

Die zweite Schicht bildet das eigentliche DRAGOS-System, den *Kern*, ab. Hierin werden einheitliche Schnittstellen für den Zugriff auf die referenzierten Basisdienste definiert. Insbesondere beinhaltet diese Schicht das Graphmodell und das Schemamodell, die im folgenden Unterabschnitt erläutert werden. Durch geeignete Implementierungen dieser Schnittstellen werden die jeweils gewünschten extern bereitgestellten Komponenten angebunden. DRAGOS stellt einfache Standardimplementierungen zur Datenspeicherung und Transaktionssteuerung bereit die keine externen Systeme voraussetzen und primär für Testzwecke ausgelegt sind. Ferner können die SQL-basierten Datenbanken PostgreSQL, MySQL, und DerbyDB sowie der Persistenzmechanismus JPOX verwendet werden. Zudem ist es möglich, mit DRAGOS auf beliebige Instanz- und Metamodelle des Eclipse Modeling Frameworks zuzugreifen, und die dort objektorientiert dargestellten Strukturen in eine graphbasierte Form aufzubereiten. Auf diese Weise lässt sich die Funktionalität von DRAGOS auf einer Vielzahl technischer Plattformen nutzen.

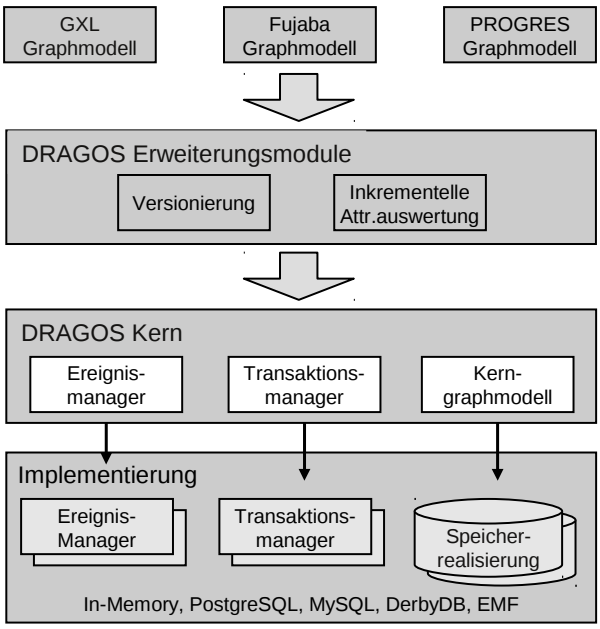


Abbildung 2.3: Schichtenarchitektur des DRAGOS-Systems, reproduziert aus [Bö06]

Außerhalb des DRAGOS-Kerns werden *Erweiterungsmodule* aufgeführt, die separat zu einem Anwendungssystem hinzugeschaltet werden. Wesentlich sind hier die Erweiterung zur Versionierung von Graphstrukturen und eine Unterstützung zur inkrementellen Berechnung abgeleiteter Attributwerte. Diese Trennung vom Kern besitzt wiederum den Vorteil, dass lediglich benötigte Funktionalität in ein bereitzustellendes Gesamtsystem aufgenommen werden muss. Kann bspw. auf die Versionierung von Graphen und somit das Wiederherstellen früherer Zustände verzichtet werden, kann durch Abschalten dieser Erweiterung erheblicher Laufzeitaufwand eingespart werden, da keine Protokollierung von Änderungsoperationen erfolgen muss.

Oberhalb des eigentlichen DRAGOS-Systems liegen die sogenannten *spezifischen Graphmodelle*. Diese realisieren unter Nutzung des Kernmodells eigene, für spezielle Anwendungsdomänen zugeschnittene Graphmodelle. So sind bspw. Graphmodelle für die Graphtransformationswerkzeuge PROGRES und Fujaba sowie für das Graphaustauschformat GXL[HWS00] definiert worden.

2.4.2 Einsatzmöglichkeiten und Einschränkungen

Im Folgenden wird der aktuelle Einsatzzweck des DRAGOS Graphspeichersystems erläutert, der sich auf die generative Werkzeugentwicklung bezieht. Abseits dieser Kernkompetenz zeigen sich jedoch Funktionalitätseinschränkungen, die eine allge-

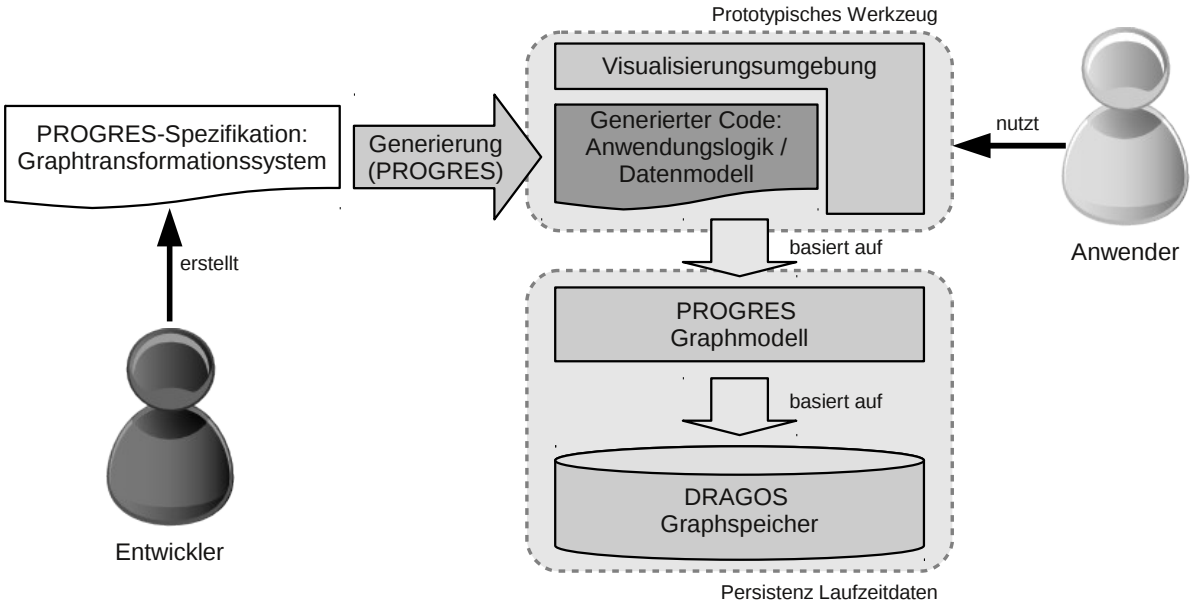


Abbildung 2.4: Graphbasiertes Rapid-Prototyping mit PROGRES

meine Nutzung des DRAGOS Systems erschweren. Mit Hilfe der Ergebnisse der vorliegenden Arbeit wird diesen Einschränkungen zum Teil begegnet.

Aktuelle Verwendung zur graphbasierten Werkzeugentwicklung

Derzeit wird DRAGOS als Persistenzsystem für generativ realisierte Graphtransformationssysteme verwendet. In diesem als *Graph-Grammar-Engineerings* [ELS86] bezeichneten Prozess werden kompakte und problemorientierte Spezifikationen durch ein Graphtransformationswerkzeug in ausführbare prototypische Systeme übersetzt. Schematisch wird dieses Vorgehen in Abbildung 2.4 gezeigt: Ausgehend von einer Spezifikation in Form einer Graphgrammatik bzw. eines Graphtransformationssystems wird mit Hilfe eines Graphtransformationswerkzeug wie etwa PROGRES Quellcode generiert. Dieser implementiert die modellierten Graphersetzungsregeln sowie das zugehörige Datenmodell. Die Laufzeitdaten des so erstellten Werkzeugs werden durch DRAGOS verwaltet und der generierten Anwendungslogik durch eine graphorientierte Schnittstelle zur Verfügung gestellt. Üblicherweise werden beim Graph-Grammar-Engineering zudem die graphbasierten Laufzeitdaten durch eine Visualisierungsumgebung aufbereitet und dem Endbenutzer präsentiert.

Durch den Prozess des Graph-Grammar-Engineerings ist eine rasche Entwicklung prototypischer Softwarewerkzeuge, sogen. Rapid-Prototyping, zur zeitnahen Erprobung möglich. Am Lehrstuhl für Informatik 3 sind auf diese Weise Softwarewerk-

zeuge unterschiedlicher Anwendungsdomänen entstanden, bspw. zum Reengineering von Telekommunikationssystemen [MW03], zum Prozessmanagement [HJS+04], oder zur Kopplung bestehender Werkzeuge durch Integratoren [BHLW07].

Als Graphtransformationswerkzeug ist bislang hauptsächlich das am gleichen Lehrstuhl entwickelte PROGRES eingesetzt worden. Jedoch bieten DRAGOS sowie das Visualisierungsrahmenwerk mittlerweile auch Unterstützung für das auf UML-Syntax basierende Werkzeug Fujaba an [SRB06].

Beschränkung der Funktionalität

Je nach Gestaltung der Codegenerierung in obigem Ablauf kann eine leicht programmatisch nutzbare Schnittstelle für das spezifizierte Datenmodell und die zugehörigen Graphtransformationsregeln durch das Graphtransformationswerkzeug erstellt werden. Im Unterschied zu den o.g. Technologien EMF und UDM bietet DRAGOS jedoch keine integrierte Funktionalität zum Erzeugen entsprechender Programmierschnittstellen an. Stattdessen basiert die Programmierschnittstelle von DRAGOS allein auf dessen Graph- bzw. Schemamodell, und ist daher unabhängig vom jeweiligen Graphschema. Zwar kann auf diese Weise reflektiv auf Instanzdaten zugegriffen werden, dies ist programmatisch verglichen mit einer graphschemaspezifisch generierten API jedoch umständlich und somit fehleranfällig.

Ferner bietet DRAGOS derzeit lediglich atomare Operationen zur Speicherung und Abfrage von Graphstrukturen an. Komplexere Anfragen die über einstellige Ergebnismengen hinausgehen würden, bspw. also Paare von verbundenen Knoten, sind nicht möglich. Eine hierzu erarbeitete Anfragesprache GRAS QL ist dabei nur konzeptionell definiert, jedoch nicht umgesetzt worden. Stattdessen erfolgt solche Auswertung der gespeicherten Graphen bislang lediglich durch den generierten Code im Rahmen von Graphtransformationssystemen. Dies stellt insbesondere für Anfragen zur Analyse und Auswertung von Graphen ein Problem dar, da diese idealerweise auch ad-hoc vom Endbenutzer erstellt werden sollten. Der zum Rapid-Prototyping verfolgte Ansatz, aus einer entsprechenden Spezifikation Quellcode zu erzeugen, zu kompilieren und an die Infrastruktur des DRGAOS-Systems und zu binden bietet hierzu nicht die notwendige Flexibilität.

2.4.3 Graph- und Schemamodell

Im Anschluss an die bisherige allgemeine Einleitung des DRAGOS Systems erfolgt an dieser Stelle ein detaillierter technischer Einstieg insbesondere in das zur Speicherung von Graphen bereitgestellte Datenmodell. Das Modell eines Graphspeichers definiert alle darstellbaren Graphstrukturen und hat somit wesentlichen Einfluss auf dessen

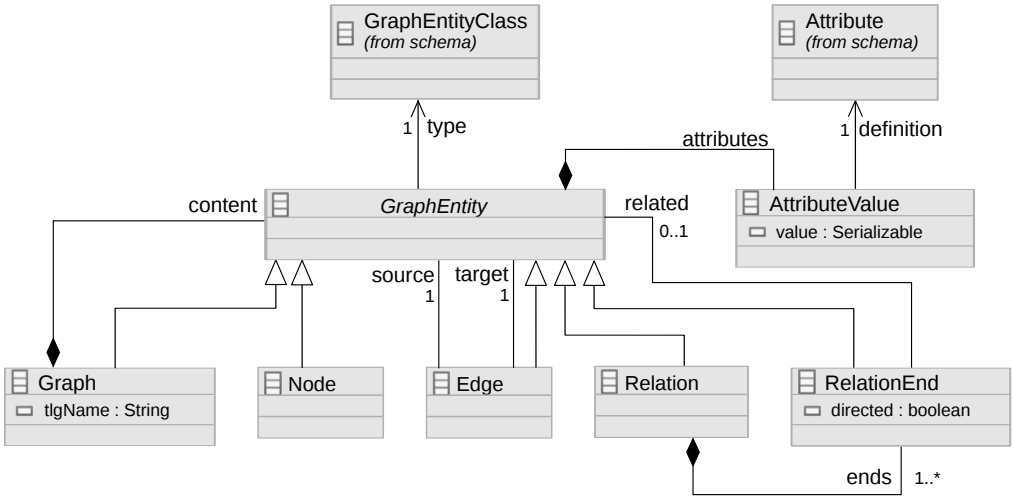


Abbildung 2.5: DRAGOS Graphmodell, reproduziert aus [Bö06] mit Anpassungen

praktische Anwendbarkeit. Zu unterscheiden ist dabei zwischen dem Metamodell der Instanzgraphen (Graphmodell) sowie dem Metamodell des zugeordneten Graphschemas (Schemamodell). Ein Instanzgraph ist technisch eine Instanz des Graphmodells, bspw. instanziiert jeder Knoten eines Graphen das Graphmodellelement *Knoten*. Das DRAGOS Graphmodell ist grundsätzlich fest zwischen Anwendung und Datenspeicherimplementierung vereinbart und daher nicht anwendungsspezifisch änderbar. Allerdings bietet DRAGOS einen Mechanismus, der mit Hilfe eines sogen. Graphschemas die Menge aller möglichen Instanzgraphen auf gemäß Anwendungsvorgaben konforme einschränkt. Hierzu zählt die Festlegung erlaubter Elementtypen sowie darauf aufsetzend Vorgaben zur Konnektierung und Attributierung von Graphelementen. Ein in DRAGOS abgelegter Graph ist somit eine *Instanz* des Graphmodells und *konform* zu seinem Graphschema. Graphschemata sind wiederum Instanzen des DRAGOS-eigenen Schemamodells, das ebenso wie das Graphmodell im Folgenden erläutert wird.

Das Graphmodell des DRAGOS Graphspeichers wird in Abbildung 2.5 gezeigt. DRAGOS erlaubt demnach den Aufbau hierarchischer Graphstrukturen, das `Graph`-Elemente beliebige Entitäten und somit auch weitere Graphen enthalten können. Mittels n-ärer Relationen lassen sich zudem sogen. Hypergraphen ausdrücken, wobei aus Performanzgründen auch binäre Relationen in Form von Kanten direkt unterstützt werden. Sowohl Relationen als auch Kanten können auf beliebige Graphentitäten, und somit bspw. auch auf andere Kanten, als Quelle und Ziel verweisen. Zudem können alle Graphentitäten attributiert werden. Eine Typisierung ist stets erforderlich.

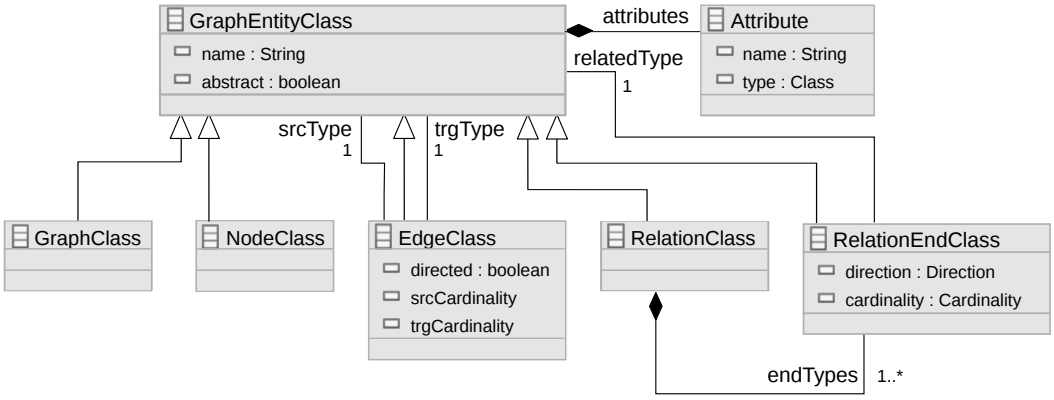


Abbildung 2.6: DRAGOS Schemamodell, reproduziert aus [Bö06] mit Anpassungen

Analog zum DRAGOS-Graphmodell zeigt Abbildung 2.6 das zugehörige Schemamodell. Hier werden Entitätsklassen für alle Entitätsarten aus Abbildung 2.5 definiert. Somit existiert zum Modellelement Node das Schemamodellelement NodeClass, das zur Typisierung von Knoten verwendet wird[2]. Mehrfachvererbung ist im DRAGOS Schemamodell zulässig, zudem können abstrakte, also nicht instanziierbare Typen festgelegt werden. Verschiedene kontextsensitive Einschränkungen legen bspw. die Zyklusloseigkeit der Vererbungsbeziehung fest. Für eine ausführliche Darstellung wird auf [Bö06] verwiesen.

2.4.4 Formale Darstellung des Graphmodells

Im weiteren Verlauf dieser Arbeit wird eine prädikatenlogische Formalisierung der dort vorgestellten Kernsprache für Graphtransformationen durchgeführt, die entsprechende Grundlagen seitens des DRAGOS Graphmodells benötigt. Das in [Bö06] in Form obiger UML-Diagramme und zusätzlicher Wohlgeformtheitsbedingungen definierte Graph- bzw. Schemamodell wird an dieser Stelle daher um eine mengentheoretische Formalisierung ergänzt.

Graphmodell

Instanzgraphen im Sinne des DRAGOS Graphmodells werden als *Universum* $\mathcal{U}$ analog zum Klassendiagramm aus Abbildung 2.5 als disjunkte Mengenvereinigung aller

[2]Der im Folgenden verwendete Begriff *Graphklasse* bezieht sich auf das hier gezeigte Schemamodellelement GraphClass und nicht auf eine graphentheoretische Charakterisierung einer bestimmten Form von Graphstrukturen.

Graphen, Knoten, Relationen, Kanten sowie Relationsenden definiert:

$$\mathcal{U} := \mathcal{G} \cup \mathcal{N} \cup \mathcal{R} \cup \mathcal{E} \cup \mathcal{RE}$$

Verbindungen von Graphelementen werden durch die Inzidenzrelation Inc_G ausgedrückt. Hierdurch werden konnektierende Elemente, d.h. Kanten und Relationsenden (zweite Komponente der Relation Inc_G), mit beliebigen Quell- und Zielelementen in Beziehung gesetzt (erste bzw. dritte Komponente von Inc_G).

$$\underset{G}{\text{Inc}} : \mathcal{U} \times \{\mathcal{E} \cup \mathcal{RE}\} \times \mathcal{U}$$

Relationsenden referenzieren die sie enthaltende Relation entweder als Quell- oder als Zielelement. Somit gilt für Relationsenden:

$$\forall s, re, t \Big((s, re, t) \in \underset{G}{\text{Inc}} \rightarrow \big(s \in \mathcal{R} \wedge s \triangleright_{\mathcal{R}} re \big) \vee \big(t \in \mathcal{R} \wedge t \triangleright_{\mathcal{R}} \big) \Big)$$

Zur vereinfachten Handhabung werden im Folgenden zwei Hilfsfunktionen source_G bzw. target_G verwendet, mittels derer Quell- und Zielelemente einer Verbindung ermittelt werden können. Für eine Kante e, ihrem Startelement u, und der Graphstruktur G gilt also:

$$u = \underset{G}{\text{source}}(e) \Leftrightarrow \exists v \big((u, e, v) \in \underset{G}{\text{Inc}} \big) \quad \text{sowie}$$
$$v = \underset{G}{\text{target}}(e) \Leftrightarrow \exists u \big((u, e, v) \in \underset{G}{\text{Inc}} \big)$$

Die Enthaltenseinsbeziehung des Graphmodells zwischen Graphelementen wird durch die Relation $\triangleright \subseteq \mathcal{G} \times (\mathcal{U} \setminus \mathcal{RE}) \cup \mathcal{R} \times \mathcal{RE}$ dargestellt. Einerseits enthalten somit Graphen beliebige Graphelemente mit Ausnahme von Relationsenden, andererseits enthalten Relationen direkt ihre Relationsenden. Spezifische Einschränkungen auf nur eine dieser Enthaltenseinsbeziehungen werden durch die Indizes $\triangleright_{\mathcal{G}}$ bzw. $\triangleright_{\mathcal{R}}$ ausgedrückt. Die Schreibweise $u \triangleright v$ sei im Folgenden äquivalent zu $(u, v) \in \triangleright$. Die Enthaltenseinsbeziehung ist azyklisch, d.h. die nicht-reflexive transitive Hülle $\triangleright^+$ enthält ein Element nicht in beiden Komponenten: $\nexists v (v \triangleright^+ v)$. Ferner ist die Enthaltenseinsbeziehung in der ersten Komponente, also dem Behälter, eindeutig: $\forall u, v, w (u \triangleright v \wedge w \triangleright v \rightarrow u = w)$

Übergeordnete Graphen sind ein Teilmenge der Graphen, die selbst keinen enthaltenden Graphen besitzen:

$$\mathcal{TLG} = \{ g | g \in \mathcal{G} \wedge \nexists g' (g' \triangleright_{\mathcal{G}} g) \}$$

Diesen wird in DRAGOS eine Beschriftungsfunktion $\text{name}_{\mathcal{G}} : \mathcal{TLG} \rightarrow \{\perp \cup \mathbb{A}^+\}$ zugeordnet, die einem Graphen einen eindeutigen Namen aus dem alphanumerischen Alphabet zuweist. Die Beschriftungsfunktion $\text{name}_{\mathcal{G}}$ ist eindeutig auf allen übergeordneten Graphen: $\forall g_1, g_2 \in \mathcal{TLG}(\text{name}_{\mathcal{G}}(g_1) = \text{name}_{\mathcal{G}}(g_2) \rightarrow g_1 = g_2)$

Graphschema

Analog zum Graphmodell wird die Menge gültiger Graphschemata $\mathcal{GS}$ als disjunkte Vereinigung aller Typen aufgefasst. Da das Schemamodellelement GraphEntityClass im Gegensatz zu seinem Pendant GraphEntity in Abbildung 2.6 instanziiert werden kann[3], wird hierfür eine zusätzliche Teilmenge $\mathcal{GEC}$ aufgeführt.

$$\mathcal{GS} := \mathcal{GC} \cup \mathcal{NC} \cup \mathcal{RC} \cup \mathcal{EC} \cup \mathcal{REC} \cup \mathcal{GEC}$$

Das Graphschemamodell definiert mit $u^{GS} \prec_{\mathcal{GS}} v^{GS}$ eine Generalisierungsbeziehung, die wie folgt als Relation definiert wird. u^{GS} ist hierbei die speziellere Unterklasse der Relation, wohingegen v^{GS} die allgemeinere Oberklasse darstellt.

$$\begin{aligned} \prec_{\mathcal{GS}} \subseteq\ & \mathcal{GC} \times (\mathcal{GC} \cup \mathcal{GEC}) \\ & \cup \mathcal{NC} \times (\mathcal{NC} \cup \mathcal{GEC}) \\ & \cup \cdots \\ & \cup \mathcal{GEC} \times \mathcal{GEC} \end{aligned}$$

Somit kann eine Graphklasse sowohl Graph- als auch Graphentitätsklassen, nicht jedoch bspw. Knotenklassen als Generalisierungen besitzen. Die transitiv-reflexive Hülle der Generalisierungsrelation wird mit $\prec^*_{\mathcal{GS}}$, die nicht-reflexive Variante mit $\prec^+_{\mathcal{GS}}$ bezeichnet. Wie obige Enthaltenseinsbeziehung im Graphmodell ist auch die Generalisierungsbeziehung azyklisch. Eine beliebige Klasse ist somit nicht in der nicht-reflexiv transitiven Hülle der Generalisierungsbeziehung enthalten: $\nexists c(v \prec^+_{\mathcal{GS}} v)$

Die zulässigen Quell- und Zielknotenklassen einer Kantenklasse bzw. einer Relationsendenklasse werden durch die Relation $\text{Inc}_{GS} : \mathcal{GS} \times {\mathbb{N}_0}^2 \times \{\mathcal{EC} \cup \mathcal{REC}\} \times {\mathbb{N}_0}^2 \times \mathcal{GEC}$ beschrieben. Die numerischen Paare bezeichnen dabei Intervalle zulässiger Verbindungszahlen am jeweils gegenüberliegenden Begrenzer, wie im folgenden Abschnitt beschrieben wird.

Typisierung von Instanzgraphen

Die Typisierungseigenschaft des DRAGOS Graphmodells besagt, dass jedem Graphelement genau ein Schemaelement zugeordnet ist. Dies wird hier durch die Funktion

[3] Instanzen von GraphEntityClass stellen abstrakte Klassen innerhalb eines Schemas dar.

type erreicht, welche die Abbildung der einzelnen Modellelemente auf ihre entsprechenden Schemaelemente beschreibt:

$$type : \big(\mathcal{G} \to \mathcal{GC} \cup \mathcal{N} \to \mathcal{NC} \cup \mathcal{E} \to \mathcal{EC} \cup \mathcal{R} \to \mathcal{RC} \cup \mathcal{RE} \to \mathcal{REC}\big)$$

Kanten sowie Relationsenden können nur zwischen Instanzen der im Graphschema festgelegten Klassen oder deren Unterklassen angelegt werden. Daher setzt die Inzidenzrelation Inc das Vorhandensein entsprechender Typen im Graphschema u_{GS}, v_{GS} voraus.

$$\begin{aligned}(u,e,v) \in \text{Inc} \to \exists u'_{GS}, v'_{GS} \Big(& type(u) \prec^* u'_{GS} \\ & \wedge\, type(v) \prec^* v'_{GS} \\ & \wedge\, \big(u'_{GS}, type(e), v'_{GS}\big) \in \underset{GS}{\text{Inc}} \Big)\end{aligned}$$

Obige Kardinalitätsbedingungen legen zudem fest, dass zu einem gegebenen Graphelement u mindestens c^{min} und höchstens c^{max} Kanten e und Zielelemente v existieren.

$$\begin{aligned}\big(\_,\_,e_{GS},(c^{min},c^{max}),t_{trg}\big) \in GS \to \\ \forall u \Big(c^{min} \leq \big|\{(e,v) | (u,e,v) \in \text{Inc} \wedge type(e) = e_{GS}\}\big| \leq c^{max} \Big)\end{aligned}$$

Analog hierzu wird die Bedingung für die quellseitige Kardinalitätsbedingung definiert.

Obige Definitionen des Graph- und Schemamodells werden im Folgenden miteinander kombiniert. Hierzu wird eine sogen. *Graphsprache* in Anlehnung an die Graphgrammatiken aus Abschnitt 2.1.2 verwendet, die sich wie folgt darstellt:

$$LG(GS) : GS \to \mathcal{P}(\mathcal{U})$$

Es gelten dabei obige Einschränkungen bzgl. des Graphschema GS:

$$\begin{aligned}\mathcal{U} \in LG(GS) \Leftrightarrow \forall u \exists u^{GS} \big(u \in \mathcal{U} \to type(u) = u^{GS}\big) \\ \wedge\, \forall u,e,v \exists u^{GS}, e^{GS}, v^{GS} \big((u,e,v) \in \underset{\mathcal{U}}{\text{Inc}} \\ \to (u^{GS}, e^{GS}, v^{GS}) \in \underset{GS}{\text{Inc}}\big)\end{aligned}$$

DRAGOS nimmt bewusst keine Einschränkung dahingehend vor, welche Typen von Elementen in einer Graphklasse enthalten sein können. Die Enthaltenseinsrelation des Graphmodells ist daher nicht Gegenstand der hier vorgenommenen Typisierung und einer entsprechenden Einschränkung schemakonformer Graphsprachen.

Attributierung

Abseits der oben behandelten Graphentitäten und -klassen beinhaltet das Modell Möglichkeiten zur Attributierung von Graphstrukturen. Die formale Spezifikation dieses Bereichs kann hier jedoch nicht vollständig erfolgen, da hierzu eine hinreichend allgemeine Algebra zur Darstellung von Attributwerten benötigt würde. Es wird daher auf eine vereinfachte Darstellung zurückgegriffen.

Für Instanzgraphen wird eine partielle Attributierungsfunktion *attr* für eine Attributdefinition A derart definiert, dass sie einem Graphelement aus $\mathcal{U}$ höchstens ein Element des Wertebereichs von A zuweist.

$$attr_{\mathcal{A}} : \mathcal{U} \dashrightarrow dom(A)$$

Schemaseitig ist festzulegen, welche Attributdefinitionen seitens einer Klasse definiert werden. Jedem Element des Graphschemas kann hierzu über die Funktion $attr_{GS}$ eine Menge von Attributdefinitionen zugeordnet werden.

$$attr_{GS} : GS \rightarrow \mathcal{P}(\mathcal{A})$$

Die Kombination der instanz- und schemaseitigen Attributfunktionen erfolgt wiederum durch die Typisierung von Graphelementen. Dazu wird festgelegt, dass $attr_{\mathcal{A}}$ höchstens denjenigen Graphelementen $\mathcal{U}$ einen Wert zuordnet für die gilt:

$$\exists t \big(type(\mathcal{U}) \prec^* t \wedge (t, \mathcal{A}) \in attr_{GS} \big)$$

2.5 Zusammenfassung

In diesem Kapitel sind zunächst grundlegende Begriffe aus dem Bereich graphbasierter Spezifikationssprachen eingeführt worden, die sowohl den fachlichen Hintergrund als auch die technische Infrastruktur der vorliegenden Arbeit bereitstellen. Das folgende Kapitel zeigt zunächst ein Anwendungsszenario das eine Anfragesprache für graphbasierte Datenstrukturen umfasst. Die nachfolgend behandelte Realisierung dieser Sprache greift auf Graphtransformationen als Ausführungsformalismus zurück. Der technische Zugriff auf die zu verarbeitenden Daten erfolgt durch Nutzung des DRAGOS Graphspeichers, womit die hier vorgestellten Technologien auf unterschiedlichen Ebenen zum Einsatz kommen.

3 Überblick & Anwendungsszenario

Dieses Kapitel gibt einen Überblick über den Ansatz der vorliegenden Arbeit, die Realisierung verhaltensorientierter Spezifikationssprachen zu unterstützen. Neben einem technischen Systemüberblick wird ein schematischer Entwicklungsprozess vorgestellt, der die Nutzung der bereitgestellten Werkzeugplattform beschreibt. Ferner stellt dieses Kapitel das im Weiteren verwendete Beispiel einer textuellen Modellanfragesprache vor.

3.1 Technische Systemübersicht

Ausgehend von der Gesamtübersicht aus Abschnitt 1.4 führt dieser Abschnitt die hierzu bereitgestellten technischen Komponenten ein. Diese werden in Abbildung 3.1 aufgezeigt und im Folgenden erläutert.

Die zentrale Ebene der *Sprachdefinition* umfasst unter anderem eine ausführbare Kernsprache sowie darauf aufsetzende Spracherweiterungen. Aus technischer Sicht ist hierunter lediglich eine rein syntaktische Definition zu verstehen, da das im weiteren Verlauf formal definierte Ausführungsverhalten der Kernsprache nur implizit über die Sprachrealisierung einfließt. Im folgenden Kapitel 4 wird die Kernsprache ausführlich vorgestellt und formal definiert. Auch die in Kapitel 6 zur Realisierung von Übersetzern bereitgestellte Spezifikationssprache ist auf dieser Ebene angesiedelt.

Die nächst tiefere Schicht beinhaltet verschiedene Aspekte zur Sprachrealisierung. Dies umfasst eine Programmierschnittstelle zur Steuerung der Ausführungsmaschine, die Implementierungen der bereitgestellten Kernsprache sowie von Spracherweiterungen beinhaltet. Dies entspricht dem Laufzeitsystem der so realisierten Spezifikationssprachen aus Abbildung 1.6 auf Seite 21. Dieser Bereich ist Gegenstand von Kapitel 5. Ferner findet sich auf dieser Ebene auch die Steuerung der Übersetzersprache, mittels derer Spezifikationsdokumente gemäß Abbildung 1.5 auf Seite 19 in eine ausführbare Form überführt werden können. Zur Übersetzersprache existiert keine eigene Realisierung, stattdessen wird auf die Kernsprache und deren Erweiterungsmodule zurückgegriffen.

Die unterste dargestellte Schicht bildet die notwendige Infrastruktur ab, um graphbasierte Daten zu speichern und zugreifbar zu machen. Hierzu wird das Graphspei-

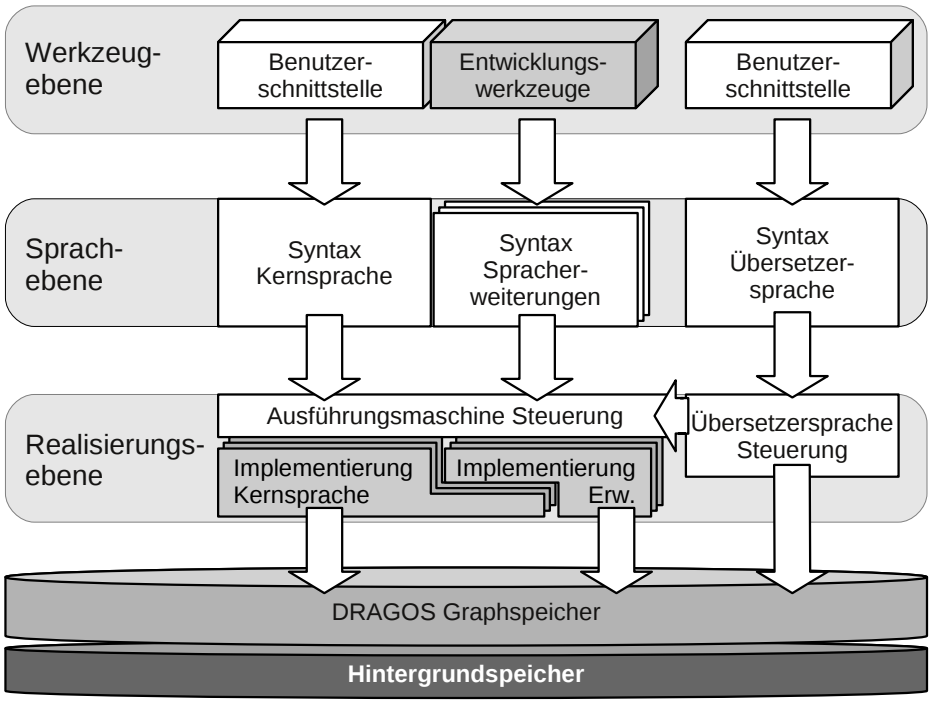


Abbildung 3.1: Systemüberblick

chersystem DRAGOS als bestehende Technologie verwendet. Diese dient als technische Grundlage zur Speicherung der auszuführenden Spezifikationen sowie dem vereinheitlichten, graphbasierten Zugriff auf die dabei zu verarbeitenden Daten.

In der zuoberst dargestellten *Werkzeugebene* werden schließlich Hilfsmittel zur vereinfachten Nutzung des vorgestellten Realisierungsrahmenwerks angeboten. Hierzu zählen Benutzerschnittstellen, mit denen Spezifikationsdokumente der Kernsprache und der Übersetzersprache erstellt werden können. Diese Werkzeuge erlauben einem Entwickler, in beiden Sprachen mit Hilfe visueller konkreter Syntax zu arbeiten. Zusätzlich existieren Entwicklungswerkzeuge, um Erweiterungsmodule für die Kernsprache syntaktisch zu entwerfen und durch Quellcodegenerierung in ein Implementierungsgerüst zu überführen. Die Werkzeugunterstützung erfolgt hier also nur zur syntaktischen Sprachbeschreibung, wohingegen das Ausführungsverhalten vorrangig mittels herkömmlicher Programmierung festgelegt werden muss. Eine Ausnahme bildet die in Abschnitt 1.4 erwähnte Reduktion von Sprachkonstrukten, die in Abschnitt 5.5 als Teil der erweiterbaren Ausführungsmaschine behandelt wird.

3.2 Realisierungsprozess für operationale Spezifikationssprachen

Entwurf und Realisierung einer operationalen Spezifikationssprache erfolgen üblicherweise in mehreren aufeinander aufbauenden Schritten, die teilweise individuell und fallspezifisch gestaltet sein können. Diese Arbeit fokussiert den Realisierungsaspekt, für den eine Vorgehensweise zur effektiven Nutzung der bereitgestellten Technologien aufgezeigt wird. Die hierzu vorgeschlagenen Arbeitsschritte werden in Abbildung 3.2 in einer groben Übersicht aufgezeigt.

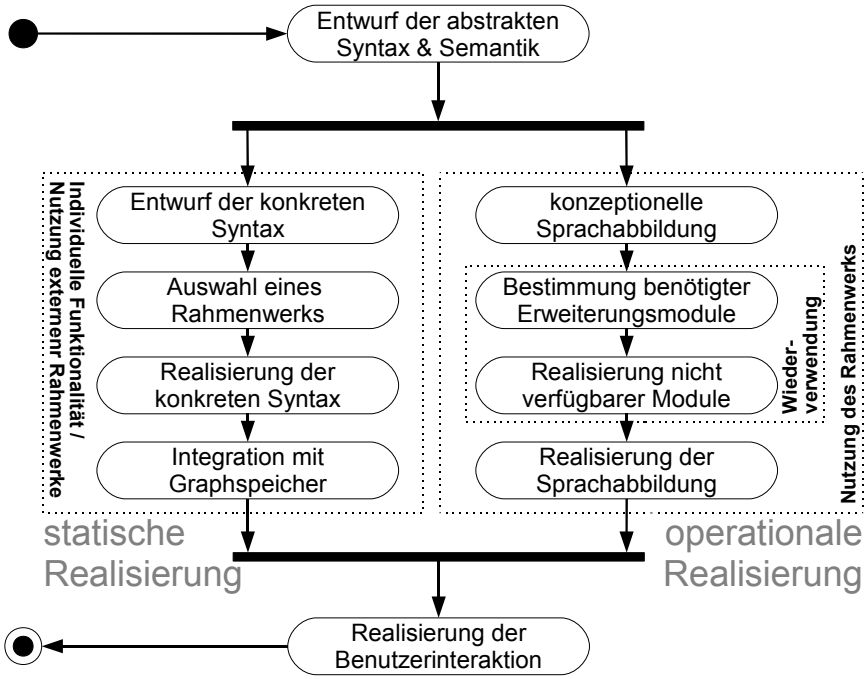


Abbildung 3.2: Realisierungsprozess einer Spezifikationssprache

Der dargestellte Prozess beginnt zunächst mit dem syntaktischen und semantischen Entwurf der zu realisierenden Spezifikationssprache. Dies ist grundsätzlich der komplexeste Arbeitsschritt, da hierunter der vollständige konzeptionelle Sprachentwurf bis hin zu einer formalisierbaren Syntax und Semantik erfolgt. Die Abbildung verdichtet dies zu einer einzelnen Aktivität, da die vorliegende Arbeit nicht die Entwurfsphase betrachtet, sondern auf die zugehörigen Realisierungsaspekte konzentriert ist.

3.2.1 Statische Realisierung

Im Anschluss an den Entwurf erfolgt eine Verzweigung der Entwicklung, da statische und operationale Aspekte weitestgehend unabhängig voneinander bearbeitet werden können. Der statische Teil beginnt zunächst den Entwurf einer konkreten Syntax, so denn dies nicht bereits zum eingangs behandelten Sprachentwurf erfolgt ist. Tatsächlich sind hier verschiedene Ansätze denkbar: Soll bspw. eine vorhandene diagrammartige Beschreibungssprache formalisiert und zum Zweck der direkten Ausführung verfeinert werden, so ist die konkrete Syntax bereits vorab weitestgehend vorhanden. Dagegen verfolgen gängige Werkzeuge zum Entwurf domänenspezifischer Spezifikationssprachen, darunter MetaEdit+, VMTS und AToM^3, einen umgekehrten Entwurfsprozess. Hierbei steht abstrakte Sprachsyntax im Vordergrund, auf deren Basis nachgelagert eine konkrete Visualisierung aufgesetzt wird. Der in Abbildung 3.2 dargestellte Prozess folgt also dieser zweiten Variante.

An den Entwurf der konkreten Syntax schließt sich die Auswahl einer geeigneten Plattform wie etwa eines Visualisierungsrahmenwerks sowie die tatsächliche Realisierung des Frontends an. Hierunter sind auch erweiterte Funktionen zur Unterstützung des Entwicklers zu verstehen, bspw. also das Prüfen erstellter Dokumente gegen die statische Sprachdefinition.

Als letzter Teilaspekt der statischen Realisierung ist eine Integration der erstellten Komponenten mit dem dieser Arbeit zugrundeliegenden Graphspeicher DRAGOS zu bewerkstelligen. So wird eine lose Kopplung mit den im zweiten Teil des Realisierungsprozesses umgesetzten operationalen Aspekten erreicht. Praktisch erfolgen kann die erforderliche Integration bspw. derart, dass die aus konkreten Spezifikationen gewonnenen abstrakten Syntaxbäume graphbasiert in der Datenbank abgelegt werden.

Während die oberen Teilaspekte größtenteils individuell für eine gegebene Spezifikationssprache umgesetzt werden müssen, kann für diese letzte Aufgabe häufig Wiederverwendung betrieben werden. So kann eine Integration spezifisch für ein Visualisierungsrahmenwerk bzw. der darin eingesetzten Technologie erfolgen, wie dies in Abschnitt 3.3.4 der Fall ist.

3.2.2 Dynamische Realisierung

Der operationale Anteil zur Realisierung einer Spezifikationssprache gemäß des Ansatzes dieser Arbeit wird im rechten Teil von Abbildung 3.2 dargestellt. Wie im statischen Fall ist dabei sowohl sprachspezifische Funktionalität nötig, allerdings kann auch ggf. Wiederverwendung bereits realisierter Komponenten betrieben werden.

Der erste Schritt zur operationalen Realisierung besteht in einer zunächst konzep-

tionellen Sprachabbildung. Ziel dabei ist es, die umzusetzende Spezifikationssprache auf die seitens des vorgestellten Rahmenwerks angebotene Kernsprache zurückzuführen. Eine mögliche Vorgehensweise besteht darin, zunächst allgemeine Sprachkonzepte auf beiden Seiten zu identifizieren, und anschließend eine vorläufige Abbildung der jeweiligen abstrakten Syntaxmodelle vorzunehmen. Dieser Aspekt wird nach Vorstellung der Kernsprache in Kapitel 6 behandelt.

Falls zwischen der zu realisierenden Spezifikationssprache und der Kernsprache ein konzeptioneller Niveauunterschied besteht, kann eine komplexe Realisierung der benötigten Abbildung notwendig sein. In diesem Fall kann durch Einsatz von Spracherweiterungen die Kernsprache angepasst und erweitert werden, um diese Komplexität zu reduzieren. Geeignete Erweiterungsmodule zu bestimmen, bzw. noch nicht verfügbare Module zu definieren, stellen weitere Tätigkeiten im Entwurfsprozess dar. Durch Entwicklung zusätzlicher Sprachmodule anstelle der Realisierung einer komplexen Abbildung kann zudem Funktionalität für vergleichbare Spezifikationssprachen verfügbar gemacht und wiederverwendet werden. Abschließend soll die Sprachabbildung ggf. unter Nutzung von Erweiterungsmodulen und der angebotenen Kernsprache erfolgen.

3.2.3 Abschluss der Realisierung

Abschließend werden beide Realisierungszweige zusammengeführt um eine funktionsfähige Lösung zu erhalten. Hierbei sind abschließende Arbeiten, wie die Integration der Endbenutzerschnittstelle mit dem bereitgestellten Rahmenwerk, durchzuführen. Je nach Anwendungsszenario der realisierten Spezifikationssprache und der damit einhergehenden Expertise der Endbenutzer muss deren Benutzerschnittstelle fallweise ausgestaltet werden. Im Falle einer rein programmatischen Nutzung der modellierten Funktionalität kann dagegen eine einfache sprachspezifische Programmierschnittstelle ausreichen.

Zwischenfazit: Der vorgeschlagene Ansatz zur Realisierung von Spezifikationssprachen basiert auf Techniken, die im Bereich domänenspezifischer Werkzeuge hinreichend erprobt wurden. Unterstützend hinzu kommt die operationale Realisierung auf Basis einer Kernsprache, die in den folgenden Kapiteln detailliert vorgestellt wird. Um die Anwendbarkeit dieses Ansatzes zu verdeutlichen führt der folgende Abschnitt 3.3 zunächst ein praktischen Anwendungsbeispiel ein.

3.3 Exemplarisches Anwendungsszenario

Die im weiteren Verlauf dieser Arbeit herangezogenen Beispiele beziehen sich vornehmlich auf das in Abschnitt 1.4 auf Seite 16 skizzierte Anwendungsszenario, einer *textuellen Anfragesprache* zur Datenauswertung in einem Informationssystem. Diese Sprache lehnt sich konzeptionell an existierende Vertreter aus dem Bereich der Datenverarbeitung an. Hier ist insbesondere die standardisierte Anfragesprache SQL [KKH05] zu nennen, die im Bereich relationaler Datenbanken eine herausragende Rolle einnimmt. Verschiedene Rahmenwerke bieten davon abgeleitete Techniken an, darunter EJB-QL im Bereich der Enterprise Java Beans (EJB) [1]. Auch im Bereich graphbasierter Werkzeuge sind Anlehnungen an SQL zu finden, hierunter hauptsächlich das in Abschnitt 2.1.5 genannte GReQL.

Die in diesem Kapitel vorgestellte und im Weiteren zu Illustrationszwecken verwendete Sprache wird EMAA – *Einfache Modell-Anfragen & Auswertungen* – genannt. Diese verknüpft die deklarative Pragmatik der SQL mit graphbasierten Konzepten, bspw. also der Auswertung von Verbindungsstrukturen.

Im Vergleich zu GReQL, welches mit der gleichen Zielsetzung entwickelt worden ist, unterstützt EMAA ein komplexeres Graphmodell, welches der DRAGOS Graphdatenbank entstammt (siehe dazu Abschnitt 2.4). Dadurch wird bspw. die Identifizierung und Attributierung von Kanten ermöglicht, wodurch sogenannte *Multigraphen* angefragt und ausgewertet werden können. Zusätzlich verwendet EMAA eine vereinfachte syntaktische Struktur, die eine knappere Formulierung gängiger Anfragen erlaubt. Die bereits für DRAGOS konzeptionell entwickelte Anfragesprache GRAS QL wird dagegen nur teilweise nachgebildet. Der Grund hierfür liegt in der hohen syntaktischen Komplexität dieser Sprache die eine Illustration der Umsetzung erschweren. EMAA bietet dagegen trotz einer reduzierten Syntax sämtliche wesentliche Funktionalität für graphorientierte Anfragen und skalare Ausdrücke an.

Im Folgenden wird zunächst die Syntax von EMAA erläutert. Im Anschluss an ihre statische Definition werden einige Beispiele aufgezeigt, anhand derer die Funktionsweise der Sprache erläutert wird. Hieraus ergeben sich die zur weiteren Präzisierung des Ausführungsverhaltens benötigten Voraussetzungen für ein unterstützendes Rahmenwerk, welches im weiteren Verlauf dieser Arbeit vorgestellt wird.

3.3.1 Konzeptuelle statische Sprachdefinition

Zur detaillierten Darstellung der betrachteten Anfragesprache erfolgt zunächst ihre statische Definition. Diese besteht aus zwei Aspekten: Zunächst wird die *syntaktische*

[1] Bei EJB handelt es sich um einem Standard zur Gestaltung komponentenbasierter Systeme, der häufig bei der Entwicklung von Informationssystemen Anwendung findet.

Struktur festgelegt, wodurch die Formulierung von Anfragen beschrieben wird. Ferner wird mittels *Wohlgeformtheitsbedingungen* festgehalten, welchen Einschränkungen die so erstellten Syntaxmodelle genügen müssen, um eine gültige Anfrage darzustellen.

Metamodellbasierte syntaktische Definition

Die formale syntaktische Definition von EMAA erfolgt anhand eines Metamodells, welches in Abbildung 3.3 dargestellt wird. Innerhalb eines Spezifikationsdokuments (Document) können mehrere benannte Anfragen abgelegt werden. Diese bestehen aus mindestens einem Ergebnis (Result) sowie einer optionalen Menge von Restriktionen (Restriction).

Anfrageergebnisse können zweierlei Gestalt haben: Entitäten (Entity) bezeichnen *Graphelemente*, bspw. also Knoten oder Kanten aus den angefragten Laufzeitdaten. Durch Angabe eines Typs können die möglichen Ergebnisse direkt auf einen bestimmten Typ beschränkt werden. Skalare Werte werden demgegenüber unter Angabe eines Ausdrucks *berechnet*. In beiden Fällen werden Ergebnisse durch einen Namen identifiziert. Ferner kann ein Ergebnis als öffentlich markiert werden, wodurch es in das Gesamtergebnis der Anfrage aufgenommen wird. Andernfalls dient ein Ergebnis lediglich der internen Verarbeitung und wird nicht ausgegeben.

Analog zu Ergebnissen existieren zweierlei Arten von Restriktionen: Graphrestriktionen sichern die Gültigkeit einer GraphExpression zwischen den angegeben begrenzenden Knoten. Dabei stehen Ausdrucksmittel zur Verfügung um einzelne Verbindungen in der gewünschten Richtung zu traversieren (Traversal), mehrere Teilpfade zu konkatenieren (Concatenation), oder einen transitiven bzw. reflexiv-transitiven Abschluss eines Pfades zu bestimmen (Closure). Letzterer bewirkt, dass der referenzierte Unterausdruck unbestimmt lange iteriert wird, um eine Verbindung aufzufinden. Dieses Sprachkonstrukt entspricht dem Kleene-Stern regulärer Ausdrücke [Kle67]. Bei der Konkatenation von Graphausdrücken kann optional ein *Typ* angegeben werden, den die traversierten „Zwischenentität“ instanziieren müssen. Möglich ist auch die Angabe einer vorgegebenen *Ergebnisvariablen*, deren zugeordnete Graphentität auf dem geprüften Pfad liegen muss.

Skalare Ausdrücke beziehen sich auf Zahlwerte oder Zeichenketten, im Unterschied zu den oben betrachteten graphischen Elementen. Die im unteren Teil der Abbildung dargestellten Metamodellelemente definieren die hierfür verfügbare Funktionalität. Neben konstanten Werten (Literal) sowie Referenzen auf berechnete skalare Ergebnisse (ScalarAccess) ist auch der Zugriff auf Attribute von Graphentitäten möglich (AttributeAccess). Hierfür sind eine Ergebnisentität sowie das auszulesende Attribut anzugeben. Über binäre Ausdrücke (BinaryExp) können Teilausdrücke kombiniert wer-

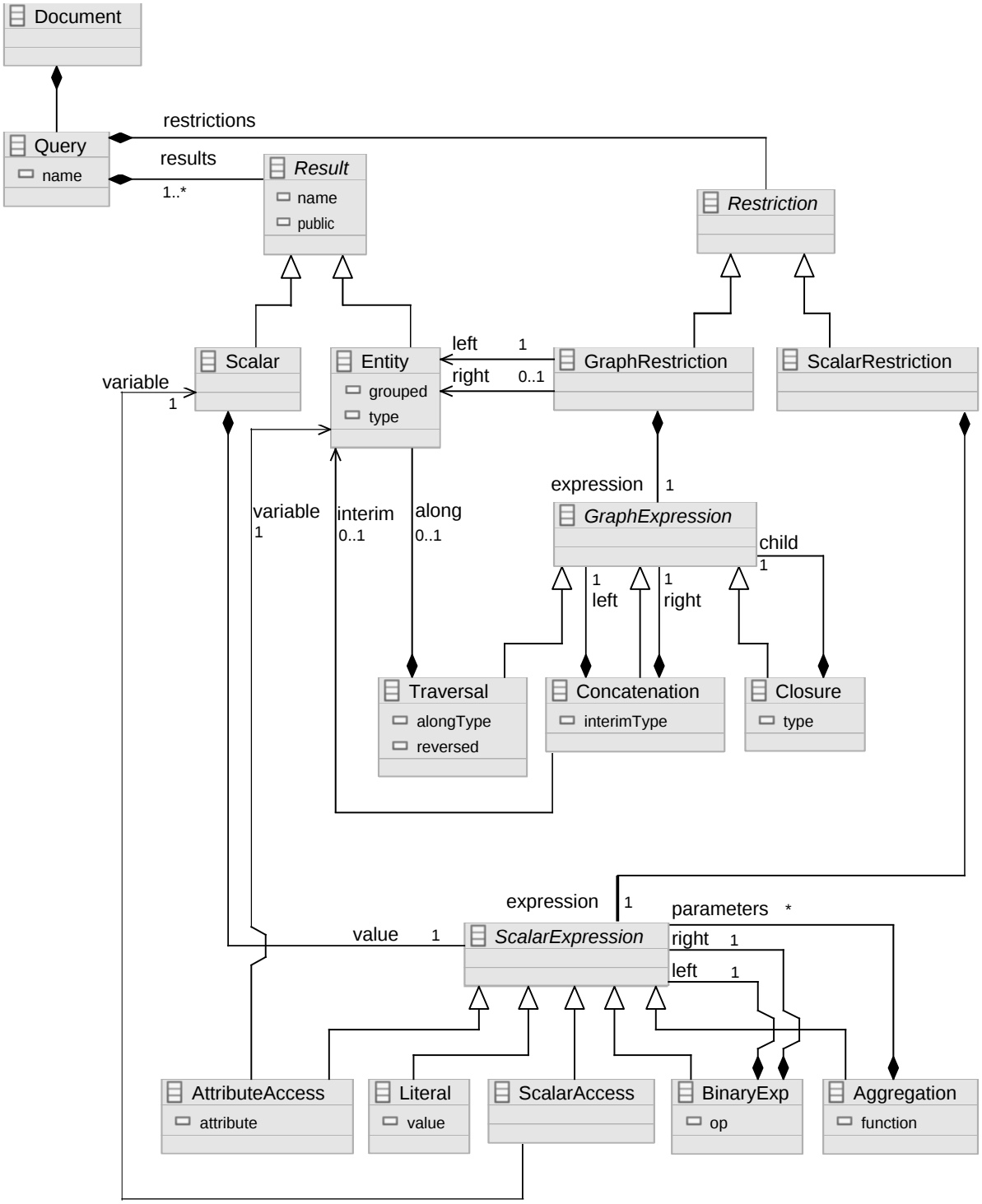


Abbildung 3.3: Metamodell der Beispielsprache EMAA

den, bspw. um mathematische Berechnungen durchzuführen. Der zu verwendende Operator wird dabei über das Attribut op angegeben. Als letzter Teil der Sprachsyntax dienen Aggregationsfunktionen (Aggregation) zur Auswertung mengenwertiger Teilergebnisse. Über die angegebene function können so bspw. Summen und Mittelwerte von Attributwerten berechnet werden.

Die in einer Aggregation zusammenzufassenden mengenwertigen Ergebnisse werden durch die grouped Markierung von Entitäten bestimmt. Das Vorgehen besteht dabei darin, Ergebnisse mit identischer Belegung der *nicht-gruppierten* Entitäten zusammenzufassen. Hierdurch entsteht für alle gruppierten Entitäten eine Sequenz von Belegungen, die als mengenwertiges Ergebnis weiterverarbeitet wird. Demzufolge fragt bspw. ein Attributzugriff nicht den Attributwert einer *einzelnen* Graphentität ab, sondern *alle Werte* des mengenwertigen Ergebnisses. Aggregationen reduzieren diese Wertemenge auf einen einzelnen Wert, je nach Operatortyp durch Aufsummierung, Durchschnittsbildung, etc.

Statische Wohlgeformtheitsbedingungen

Das Metamodell aus Abbildung 3.3 erlaubt die syntaktische Beschreibung der hier betrachteten Sprache und damit im formalen Sinne eine kontextfreie Beschränkung aller darin gültigen Sätze. Das Metamodell geht in seinem Formalisierungsgrad dabei bereits über klassische Ansätze zur Definition textueller Sprachen hinaus, da es keine rein baumartigen Strukturen beschreibt. Beispielsweise werden zur Beschreibung von Graphbeziehungen (GraphRestriction) die begrenzenden Entitäten über die Assoziationen left und right definiert. Stattdessen würde eine Beschreibung auf Basis der Backus-Nauer-Form (BNF, [Knu64]) oder vergleichbarer grammatikbasierter Verfahren eine baumförmige Syntaxdefinition, d.h. insbesondere ohne diese Querbeziehungen ergeben. Anstellte o.g. Assoziationen würden angewandte Auftreten von Bezeichnern verwendet werden, um die jeweiligen Entitäten zu referenzieren. Die tatsächliche Existenz einer definierenden Entität mit gleichem Namen kann bei Grammatiken, anders als im vorliegenden Fall, nicht gewährleistet werden.

Dennoch können viele Sachverhalte auch in einem metamodellbasierten Ansatz nicht direkt beschrieben werden. Daher sehen gängige Modellierungsansätze und -werkzeuge vor, das Metamodell durch *Wohlgeformtheitsbedingungen* weiter zu präzisieren. Gängige Praxis ist dabei der Einsatz der Object Constraint Language, die bereits in Abschnitt 2.3 erwähnt worden ist. Da die so gestellten Bedingungen gegen alle erstellten Instanzmodelle anstelle des Metamodells geprüft werden, können beispielsweise Eindeutigkeitsbedingungen bezüglich der vergebenen Bezeichner ausgewertet werden. Mittels der folgenden OCL Bedingung werden daher eindeutige Namen für alle Resultate einer Anfrage gefordert. Nichtsdestotrotz können Resultate

unterschiedlicher Anfragen innerhalb eines Spezifikationsdokuments identische Namen besitzen.

```
context Query
inv: self.results→isUnique(r | r.name)
```

Verschiedene Attribute einzelner Metamodellelemente können nur unter wechselseitigem Ausschluss gesetzt werden. So kann eine Entität nicht gleichzeitig als Teil des Gesamtergebnisses zurückgegeben werden, und zwecks Verwendung in Aggregationsfunktionen gruppiert werden. Bei der Konkatenation von Pfadausdrücken kann zudem entweder eine zu traversierende Entität, oder alternativ ein Typ des zu traversierenden Elements angegeben werden, nicht jedoch beides zugleich.

```
context Entity
inv: not (self.grouped and self.public)

context Concatenation
inv: self.interimType.oclIsUndefined() or
       self.interim.oclIsUndefined()
```

Als weitere Einschränkung gilt, dass die Berechnungsfunktion skalarer Werte keine Abhängigkeit zu sich selbst aufweisen darf. Von daher ist bei der Spezifikation eines Scalar sicherzustellen, dass kein ScalarAccess auftritt der sich auf das zu berechnende Ergebnis bezieht. Dies ist auch über in der Baumstruktur verschachtelte Teilausdrücke, bspw. als Operanden eines binären Operators zu prüfen. Im Folgenden OCL Ausdruck wird daher zunächst eine Hilfsfunktion allReferenced definiert, so dass die nachstehende Invariante vergleichsweise einfach angegeben werden kann[2].

```
context ScalarExpression
def: allReferenced : Set(Scalar) = Set {}

context ScalarAccess
def: allReferenced : Set(Scalar) = Set {self.variable}

context BinaryExp
def: allReferenced : Set(Scalar) = self.left.allReferenced→
       union(self.right.allReferenced)

context Aggregation
def: allReferenced : Set(Scalar) = self.parameters→
```

[2]Die hier verwendete Notation geht der Übersichtlichkeit halber von einer polymorphen Überladung von Hilfsdefinitionen aus. Da dies im Standard nicht explizit vorgesehen ist, müsste eigentlich eine if-blockbasierte Typunterscheidung im Kontext von ScalarExpression erfolgen.

```
        collect(p | p.allReferenced)→flatten()

context Scalar
inv: not self.value.allReferenced→includes(self)
```

Zu den zu gewährleistende Eigenschaften zählt ferner, dass eine Wertebeschränkung in Form einer ScalarRestriction ausschließlich über Ausdrücke mit booleschem Ergebnis definiert werden darf. Als zulässige, durch eine ScalarRestriction referenzierte Ausdrücke kommen damit binäre Ausdrücke (BinaryExp) mit booleschem Ergebnis in Frage, worunter bspw. Gleichheitstests und Größenvergleich von Zahlwerten fallen. Auch boolesche Attribute von Graphentitäten oder skalare Resultate sowie die Literale `true` und `false` können verwendet werden. Da zum Zeitpunkt der Anfragedefinition das Graphschema der angefragten Laufzeitdaten üblicherweise nicht vorliegt, ist eine vollständige statische Überprüfung dieser Typbedingung in ersteren beiden Fällen allerdings nicht möglich. Weitere Bedingungen, die Kenntnisse des Graphschemas der Laufzeitdaten voraussetzen, können nur unter entsprechender Einbettung in das Laufzeitsystem sichergestellt werden. Da die zur Spezifikationszeit prüfbaren Bedingungen vergleichsweise einfach gestaltet sind, werden diese hier ausgelassen.

3.3.2 Beispielanfragen und Auswertung

Im Folgenden werden einige Beispiele aufgezeigt, welche die Anwendung von EMAA demonstrieren, und deren beabsichtigte Semantik skizzieren. Die angefragten Datensätze entstammen hierbei einem fiktiven Informationssystem zum Projekt- und Ressourcenmanagement das sich konzeptionell an existierende Systeme wie [NWS03] anlehnt. So zeigt Abbildung 3.4 verschiedene Projekte, die eine Reihe direkt zugeordneter Aufgaben enthalten. Diese können sich wiederum in Teilaufgaben zergliedern. Die jeweils notierte Bearbeitungsdauer betrifft daher nur die jeweils betrachtete Aufgabe, nicht jedoch mögliche Unteraufgaben. Im unteren Abschnitt sind Personen aufgezeigt, die jeweils einem oder mehreren Projekten zugeordnet sind. Eine Person leistet eine bestimmte Menge an wöchentlicher Arbeitszeit, die anteilig auf die von ihr bearbeiteten Aufgaben verteilt werden kann. Bei der Übernahme von Tätigkeiten kann eine Person zudem verschiedene Rollen annehmen, bspw. dies eines Programmierers oder Softwaretesters.

Einfache Attributanfrage

Das erste in Abbildung 3.5 gezeigte Beispiel zählt die Namen aller vorhandenen Projekte sowie der darin enthaltenen Aufgaben auf. Nebenstehend sind die diesbezüglich

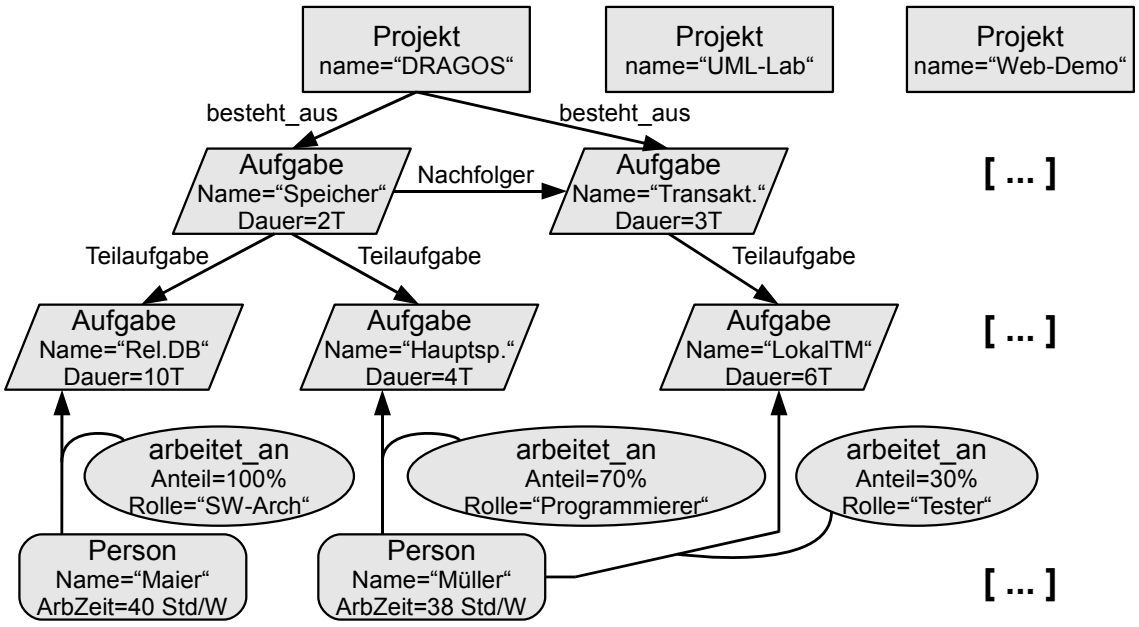


Abbildung 3.4: Beispieldaten des fiktiven Projektmanagements

aufgefundenen Ergebnisse und die entsprechend verwendeten Elemente des Beispielgraphen dargestellt. Der obere Teil der Anfrage enthält zwei skalare Werte sowie zwei Entitäten als Ergebnisse. Nur die skalaren Werte werden aufgrund ihrer „+"-Markierung als öffentlich sichtbar in die Ergebnistabelle aufgenommen. Ihre Werte bestimmen sich aus dem `Name`-Attribut der Entitäten. Im unteren Bereich wird die Belegung der beiden Entitäten gemäß ihrer Verbindungsstruktur eingeschränkt. Gefordert wird, dass eine Traversierung einer Kante vom Typ `besteht_aus` sowie einer beliebigen Anzahl des Typs `Teilaufgabe` möglich ist. Die Ergebnistabelle enthält hierzu jeweils ein Beispiel für die Pfadlänge `0` bzw. `1`. Insgesamt kann diese Anfrage $m \times n$ Ergebnisse liefern, wobei m die Anzahl der Projekte und n die Anzahl der Aufgaben ist. Durch die Abfrage der Konnektierungsstruktur und der Annahme, dass jede Aufgabe genau einem Projekt zugeordnet ist, entstehen jedoch nur n gültige Belegungen.

Werteaggregation

Abbildung 3.6 zeigt eine Variante des vorherigen Beispiels, wobei hier die Entität `aufgabe` gruppiert wird. In diesem Fall werden alle gemäß der **`where`**-Klausel gültigen Variablenbelegungen bestimmt. Die Aggregationsfunktion **`sum`** ermittelt hiervon die Summe der jeweiligen `Dauer`-Attribute. Insgesamt liefert die Anfrage m Ergebnisse, da lediglich nicht-gruppierte Entitäten die Menge eindeutiger Ergebnisse bestim-

```
query Projektstruktur
  + P_Name = projekt.Name,
  + A_Name = aufgabe.Name,
  projekt : Projekt,
  aufgabe : Aufgabe
where
  projekt -/besteht_aus/->
    ( -/Teilaufgabe/-> )* aufgabe
;
```

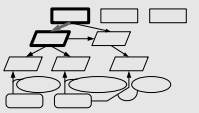
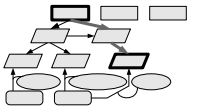
P_Name	A_Name	Auswahl
DRAGOS	Speicher	
DRAGOS	LokalTM	
[...]	[...]	

Abbildung 3.5: EMAA Anfrage: Projektstruktur

men.

```
query Gesamtdauer
  + P_Name = projekt.Name,
  + P_Dauer = sum (aufgabe.Dauer),
  projekt : Projekt,
  grouped aufgabe : Aufgabe
where
  projekt -/besteht_aus/->
    ( -/Teilaufgabe/-> )* aufgabe
;
```

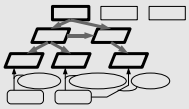
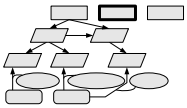
P_Name	P_Dauer	Auswahl
DRAGOS	25T	
UML-Lab	0T	
[...]	[...]	

Abbildung 3.6: EMAA Anfrage: Berechnung der Gesamtdauer

Identifizierte Navigation & Mehrfachaggregation

Die Abfrage in Abbildung 3.7 zeigt ein Beispiel, in dem das komplexe Graphmodell des DRAGOS Datenbanksystems hilfreich eingesetzt werden kann. Als Ergebnisentitäten werden in dieser Anfrage zunächst alle Projekte sowie deren beteiligte Mitarbeiter ermittelt. Eine Besonderheit stellt dabei die Entität `arbAn` dar, da ihr Typ `arbeitet_an` gemäß Abbildung 3.4 eine Verbindung repräsentiert. Der **`where`**-Bereich formuliert in dieser Anfrage die Bedingung, dass der Pfad zwischen `projekt` und `person` *entlang* dieser Verbindung verlaufen muss. Dies wird durch den Teilausdruck `<-arbAn-` formuliert, wohingegen Ausdrücke der Form `-/besteht_aus/->` lediglich den *Typ* der zu traversierenden Verbindung festlegen. Durch die so erfolgte Identifizierung der traversierten Verbindung wird ein Zugriff auf ihre Attributwerte ermöglicht, was zur Einschränkung der Tätigkeit auf Programmierer sowie der Ermittlung

der anteilig erbrachten Arbeitszeit benötigt wird. Das wesentliche Ergebnis dieser Anfrage ergibt sich aus der ermittelten Arbeitszeit (`A_Zeit`), die pro Person als Produkt der persönlichen Arbeitszeit, und des als Programmierer eines jeweiligen Projekts tätigen Zeitanteils gebildet wird. Das Gesamtergebnis für ein Projekt ergibt sich demzufolge aus der Summe dieser individuellen Arbeitszeiten alle beteiligten Personen. Die Anfrage bildet eine Mehrfachaggregation in der Hinsicht, dass diesmal *zwei* Variablen gruppiert und deren Attribute ausgelesen werden. Bei der Auswertung ist zu beachten, dass diese Werte korreliert verarbeitet werden. Im vorliegenden Fall müssen die multiplizierten Arbeitszeiten und Zeitanteile einem gemäß der **`where`**-Klausel gültigen Paar von `person` und `arbAn` Entitäten entstammen. Wie im vorherigen Fall entstehen *m* Ergebnisse zu dieser Anfrage.

```
query Arbeitszeit
  + P_Name = projekt.Name,
  + A_Zeit =
      sum(arbAn.Anteil
        * person.ArbZeit),
  projekt: Projekt,
  grouped arbAn: arbeitet_an,
  grouped person: Person
where
  projekt -/besteht_aus/->
    ( -/Teilaufgabe/-> )* <-arbAn-
  person,
  arbAn.Rolle = „Programmierer“
;
```

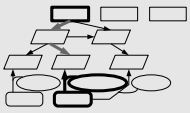
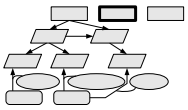
P_Name	A_Zeit	Auswahl
DRAGOS	26,6h	
UML-Lab	0h	
[...]	[...]	

Abbildung 3.7: EMAA Anfrage: Durch Programmierer erbrachte Arbeitszeit

3.3.3 Vorgehensweise zur statischen Sprachrealisierung

Im Anschluss an die syntaktisch formale Definition von EMAA, und ihrer aufgezeigten beispielhaften Verwendung, entsteht die Fragestellung der weiteren Realisierung dieser Spezifikationssprache gemäß des linken Zweiges von Abbildung 3.2. Hierzu werden eine Reihe technischer Komponenten benötigt:

Visualisierung: Die Visualisierung einer EMAA-Spezifikation kann aufgrund der textuellen Syntax mittels beliebiger Texteditoren erfolgen. Ein sprachspezifischer Editor ist jedoch hilfreich, um die in den Abbildungen angedeutete Syntaxhervorhebung zu erreichen. Auch die Markierung syntaktischer Fehler sowie Verlet-

zungen der damit verwandten Wohlgeformtheitsbedingungen können auf diese Weise realisiert werden. Zu diesem Zweck können verschiedene existierende Techniken und Rahmenwerke eingesetzt werden. Eine beispielhafte Anwendung eines verbreiteten Rahmenwerks zeigt der folgende Abschnitt.

Persistente Speicherung: Auch das Abspeichern der mit EMAA spezifizierten Dokumente kann problemlos mittels regulärer Textdateien erfolgen. Zur weiteren Verarbeitung wird jedoch auch eine formale Repräsentation benötigt, auf deren Basis die Spezifikation programmatisch analysiert werden kann. Hierzu bieten sich graphbasierte Modelle gemäß des Metamodells aus Abbildung 3.3 an, die aus textuellen Spezifikationen durch einen EMAA Parser erzeugt werden können. Zur Ablage der hieraus ermittelten Graphen kann die DRAGOS Graphdatenbank eingesetzt werden, die auch im Folgenden Grundlage zur operationalen Weiterverarbeitung ist.

Statische Prüfung: Abgespeicherte Spezifikationsdokumente müssen gegen die syntaktische Definition bestehend aus dem Metamodell und zusätzlichen Wohlgeformtheitsbedingungen überprüft werden können. Andernfalls würden fehlerhafte Anfragen verarbeitet werden, die ggf. unerwartetes Fehlverhalten oder falsche Ergebnisse bei der späteren Auswertung liefern. Daher sind geeignete Mittel bereitzustellen, um den Entwickler frühzeitig auf Fehler hinweisen zu können.

Im Anschluss an diese Übersicht wird die Realisierung einer textuell-konkreten Syntax an einem verbreiteten Rahmenwerk illustriert. Ferner wird untersucht, welchen Anforderungen die Anfragesprache EMAA an eine allgemeine Ausführungsmaschine stellt, auf deren Basis sie realisiert werden kann. Hierdurch ergeben sich Ansprüche an die im nächsten Kapitel vorgestellte Kernsprache für Graphtransformationen.

3.3.4 Realisierung einer textuell-konkreten Syntax

Abbildung 3.8 zeigt die Möglichkeit auf, eine EMAA-Editierumgebung auf Basis des openArchitectureWare-Rahmenwerks [oAW10] zu erstellen. Hierbei kommen zwei Bestandteile zum Einsatz: Mittels der Beschreibungssprache *Xtext*[3] wird die konkrete Sprachsyntax in grammatik-ähnlicher Form notiert. Am Beispiel dargestellt sind die ersten Elemente des Metamodells aus Abbildung 3.3. Hierbei wird bspw. für Anfragen festgelegt, dass diese mit dem Schlüsselwort **`query`** gefolgt von einem Bezeichner

[3]Xtext stellt ein Entwicklungswerkzeug für textuelle Spezifikationssprachen zur Verfügung, ähnliche Ansätze verfolgen bspw. MontiCore[KRV07] und EMFText[HJK+09].

beginnen sowie zumindest ein Ergebnis und optional eine komma-getrennte Liste von Restriktionen beinhalten. Auf eine detaillierte Vorstellung der Xtext-Sprachkonstrukte wird hier verzichtet, diese kann jedoch der Referenzliteratur [Xte10] entnommen werden.

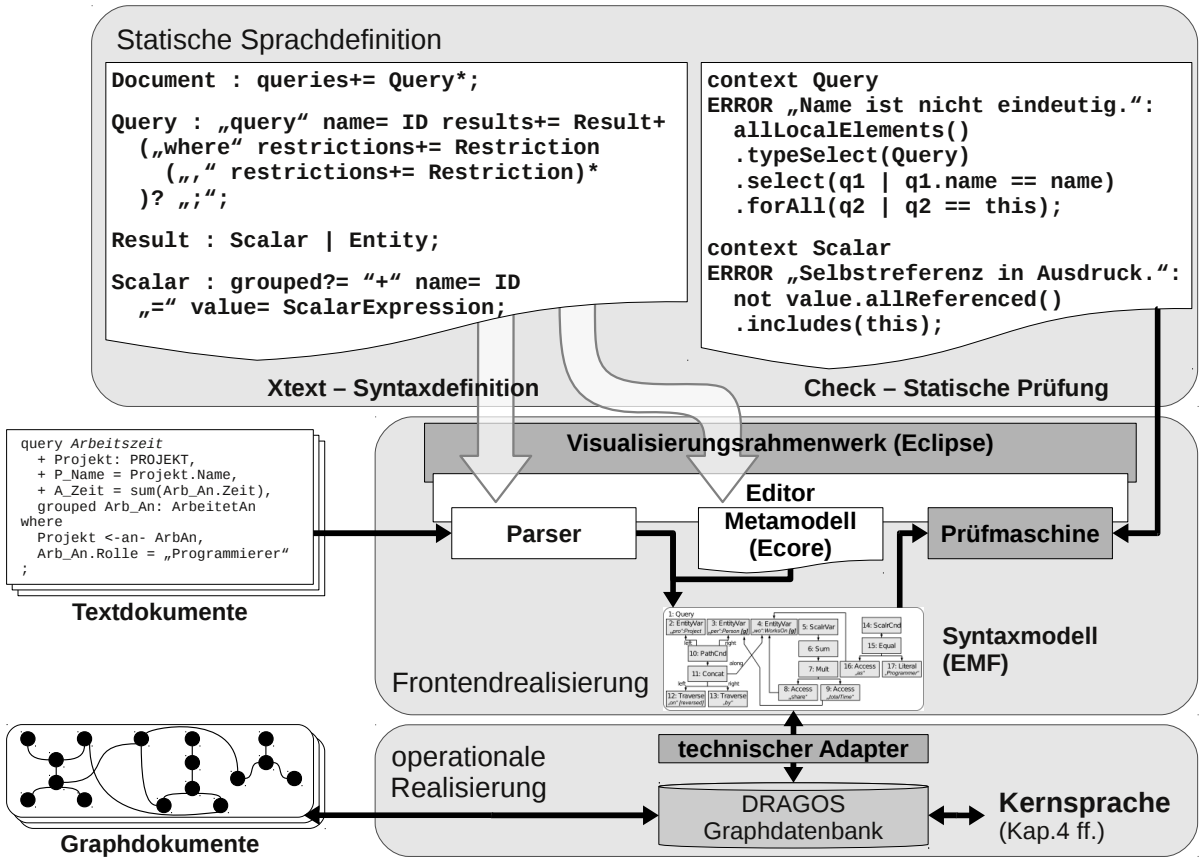


Abbildung 3.8: Frontendrealisierung mittels openArchitectureWare Xtext

Aus der vorliegenden Spezifikation werden zweierlei Dokumente generiert. Hierbei handelt es sich um einen *Parser*, der textuelle Eingaben einliest und gemäß der modellierten Syntax analysiert. Ferner wird ein *Metamodell* unter Nutzung des Eclipse-Modeling-Frameworks (EMF) [SBPM08] erzeugt, welches zur typisierten Darstellung der analysierten Eingabedokumente dient. Dieses wird vom Parser zum Erzeugen entsprechender Instanzmodelle benötigt, welche die Ausgabe der syntaktischen Analyse darstellen. Neben ihrer klassischen Aufgabe, Eingabedokumente gemäß grammatikalischer Regeln zu strukturieren, beinhalten Xtext-generierte Parser eine automatische Erstellungsroutine für *Querverweise* in den Instanzmodellen. Hierbei werden angewendete Bezeichner aufgelöst in dem Verweis auf die zugehörigen definierenden Bezeichner in das Instanzmodell eingetragen werden.

Der generierte Parser bedient alle weiteren Verarbeitungsschritte auf Basis der textuellen Spezifikation. Eine persistente Ablage der daraus ermittelten Modelle erfolgt nicht – Textdateien bilden in Xtext also die primär verarbeiteten Dokumente.

Zur Überprüfung statischer Wohlgeformtheitsbedingungen stellt das openArchitectureWare-Rahmenwerk eine weitere spezialisierte Sprache, *Check*, bereit. Ein Beispiel hierzu zeigt die rechte Seite der Sprachdefinition, welches die zu obigen OCL-Bedingungen passenden Check-Bedingungen formuliert. Diese Prüfbedingungen werden zur Laufzeit der Editierumgebung durch eine vom Rahmenwerk bereitgestellten Prüfmaschine interpretativ ausgewertet, unterliegen also keiner Codegenerierung. Die Prüfung erfolgt auf Basis des Instanzmodells der Spezifikation, welches vom Parser zuvor erzeugt wurde.

Aus der syntaktischen Sprachbeschreibung der Xtext-Grammatik generiert das Rahmenwerk eine Editierumgebung auf Basis der Entwicklungsumgebung Eclipse. Neben der Darstellung textueller Dokumente ermöglicht dieser Editor die Hervorhebung syntaktischer Schlüsselwörter sowie deren Vervollständigung während der Eingabe durch den Entwickler. Durch Interaktion mit dem generierten Parser und o.g. Prüfmaschine können zudem Spezifikationsfehler visuell markiert werden.

Die weitere operationale Realisierung von EMAA erfolgt auf Basis des hier vorgestellten Rahmenwerks. Zu diesem Zweck müssen die durch den Parser ermittelten abstrakten Syntaxdokumente in die Graphdatenbank DRAGOS überführt werden. Dies ist Aufgabe des in Abbildung 3.8 aufgezeigten *technischen Adapters*. Im vorliegenden Fall Xtext-generierter Editoren liegen Instanzmodelle in Form von EMF-Modellen vor. Für diese standardisierte Technologie wird ein allgemeiner, sprachunabhängiger Adapter bereitgestellt, der EMF Instanzmodelle einerseits und zu diesen korrespondierenden Graphstrukturen andererseits synchronisiert.

3.3.5 Abschließende Realisierungsschritte

Gemäß Abbildung 3.2 bleibt nach Abschluss sowohl der statischen als auch der operationalen Realisierung eine Integration beider Zweige vorzunehmen. Konkret erfolgt dies durch Bereitstellung einer Benutzerschnittstelle, um Anfragen wie in Abbildung 1.6 auf Seite 21 auswählen, ausführen, und deren Ergebnis visualisieren zu können.

Die für diesen letzten Schritt notwendigen Realisierungskomponenten sind stark auf die jeweilige Spezifikationssprache spezialisiert. Im Fall von EMAA würde bspw. eine Auflistung verfügbarer Anfragen sowie eine tabellarische Ergebnisdarstellung in Frage kommen. In anderen Szenarien kämen anders geartete Benutzerschnittstellen zum Einsatz. Im Kontext des Werkzeugbaus mittels Graphtransformationssystemen, in denen Sprachen wie PROGRES und Fujaba bereits erfolgreich eingesetzt worden

sind, ist beispielsweise eine flexible parametrisierbare Graphvisualisierung essenziell. Zusätzlich müssen Transformationsregeln durch einen Benutzer angesteuert und mit Eingabeparametern versehen werden. Dagegen spielt die Darstellung von Ausführungsergebnissen, abseits der Anzeige des Laufzeitgraphen, keine Rolle. Werkzeuge, die bspw. zur Übersetzung zwischen verschiedenen Modellierungssprachen unter Nutzung von Graphtransformationsregeln erstellt werden, benötigen oftmals grundsätzlich keine Benutzerschnittstelle, da sie rein programmatisch etwa durch bestehende Endbenutzeranwendungen angesteuert werden.

Aufgrund der vielfältigen Ausprägungsmöglichkeiten dieses letzten Realisierungsschritts ist eine sinnvolle Rahmenwerksunterstützung schwer zu erreichen, und wird daher im Kontext dieser Arbeit nicht angestrebt. Die jeweilige fallspezifische Realisierung erfordert hauptsächlich die Nutzung der von der Ausführungsmaschine der Kernsprache bereitgestellten Programmierschnittstelle, und ggf. geeigneter Realisierungsrahmenwerke. Da dies einen hochgradig technischen Systemaspekt darstellt, wird er hier nicht weiter vertieft.

3.3.6 Anforderungen zur operationalen Realisierung

Zum Abschluss des Beispielszenarios erfolgt an dieser Stelle ein Vergleich der Anforderungen aus Abschnitt 1.3.1 auf Seite 10 an eine geeignete Realisierungsinfrastruktur für EMAA.

Zu Anforderung 1 – Unterstützung der Verhaltensmodellierung: EMAA bietet verschiedene Sprachkonstrukte die typisch für eine Anfragesprache sind, jedoch nicht notwendigerweise in anderen graphbasierten Spezifikationssprachen Relevanz besitzen. Hierzu zählt bspw. die Werteaggregation für skalare Ausdrücke. Ein geeignetes Grundgerüst zur Realisierung dieser Sprache sollte dennoch in der Lage sein, die benötigte Funktionalität unter angemessenem Aufwand bereitzustellen.

Zur Realisierung des graphbasierten Funktionsbereichs erfordert EMAA einerseits einfache Navigation entlang Verbindungsstrukturen. Andererseits zählen hierzu auch der transitive Abschluss von Pfadausdrücken sowie die Verarbeitung attributierter und identifizierter Verbindungsstrukturen. Ferner benötigt eine EMAA Realisierung geeignete Funktionalität um skalare Werte und Aggregation von Wertemengen zu berechnen.

EMAA Spezifikationen sind zudem nicht abgeschlossen, d.h. sie beinhalten keine Definition des Metamodells der angefragten Daten. Stattdessen können erst während der Ausführung bestimmte Konsistenzbedingungen überprüft werden, bspw. ob der Typ einer angefragten Entität existiert. Seitens der Realisierungsumgebung ist zu beachten, dass eine entsprechend flexible Lösung ermöglicht wird.

Zu Anforderung 2 – Erhalt des Abstraktionsniveaus: Das Abstraktionsniveau, das von der Sprache EMAA angestrebt und realisiert wird, entspricht sequenziell definierten und abgearbeiteten Klauseln, die einer herkömmlichen Programmiersprache ähneln. Dennoch bietet die Sprache Abstraktion von konkreten Verarbeitungsvorschriften, da bspw. die technische Ausgestaltung des Datenzugriffs und der darauf basierenden Verarbeitung von Verwendern der Sprache verborgen bleibt. Die Erhaltung des Abstraktionsniveaus sollte bei der Realisierung von EMAA keine besondere Rolle spielen, da der vorgeschlagene Ansatz bereits auf graphbasierten Datenstrukturen aufsetzt.

Zu Anforderung 3 – Anpassbarkeit der Lösung: Da EMAA nicht auf eine spezifische Anwendungsdomäne oder ein gegebenes Datenmodell festgelegt ist, kann die benötigte Grundfunktionalität fest definiert werden. Anpassungen oder Erweiterungen sollten dennoch ermöglicht werden, bspw. um die Auswertung komplexerer Navigation entlang von Graphstrukturen zu unterstützen. Auch denkbar ist die Ergänzung weiterer Aggregationsfunktionen, bzw. der Anschluss externer Bibliotheken für statistische Funktionen.

Zu Anforderung 4 – Unabhängigkeit von technischen Plattformen: Als Anfragesprache sollte EMAA nach Möglichkeit Daten unterschiedlicher Quellen verarbeiten können, so denn diese eine graphische Struktur aufweisen. Wie es beispielsweise bei der SQL für relationale Datenbanken der Fall ist, kann natürlich für jede mögliche Datenquelle eine gesonderte Implementierung erfolgen. Zur Beschleunigung der Sprachrealisierung bietet es sich jedoch an, eine gemeinsame Schnittstelle für Datenquellen einzuführen, auf welcher die Realisierung der Anfragesprache basiert. Das DRAGOS Graphspeicher liefert eine solche graphbasierte Schnittstelle für eine Reihe existierender Speicherlösungen.

Zwischenfazit: Das vorgestellte Anwendungsszenario zeigt einen realistischen Einsatz der im weiteren Verlauf dieser Arbeit vorgestellten Technologien. Einerseits demonstriert es den Entwurf einer dedizierten Anfragesprache um Applikationslogik kompakt zu beschreiben. Hierdurch werden Entwickler direkt unterstützt. Andererseits kann die EMAA den Bedürfnissen der Softwareentwickler weiter angepasst und spezialisiert werden. Durch die Anwendung eines modellgetriebenen Ansatzes werden sowohl die initiale Realisierung, als auch nachgelagerte Erweiterungen und Veränderungen effektiv mitgetragen.

3.4 Weitere unterstützte Anwendungsszenarien

Im Weiteren wird untersucht, welche zusätzlichen Anforderungen bei der Realisierung operationaler Spezifikationssprachen entstehen. Da insbesondere die im vorherigen Kapitel vorgestellten *Graphersetzungssprachen* bereits vielfältig zur operationalen Spezifikation von Softwaresystemen eingesetzt werden, erfolgt eine genauere Untersuchung ihrer jeweiligen Realisierungsanforderungen. Dies liegt darin begründet, dass zukünftig zu realisierende Spezifikationssprachen mutmaßlich bestehende und verbreitete Technologien adaptieren werden, und somit vergleichbare Anforderungen aufweisen. Zielsetzung dieser Diskussion ist es, geeignete Funktionalität zu identifizieren, die als Teil eines Realisierungsrahmenwerks bereitgestellt werden sollte. Darauf aufbauend wird das folgende Kapitel eine *Kernsprache* vorstellen, die einen sinnvollen Teil dieser notwendigen Funktionalität bereitstellt, und als Realisierungsgrundlage für operationale Spezifikationssprachen dienen kann.

3.4.1 Graphgrammatiken: Mustersuche

Die im vorangegangen Kapitel vorgestellten Graphgrammatiken stellen zahlreiche Anforderungen an eine werkzeugtechnische Realisierung. Aufgrund dieser Vielzahl wird zunächst nur der Aspekt der *Mustersuche* gängiger Systeme sowie einige verbreitete Erweiterungen in diesem Bereich diskutiert. Wie in Abschnitt 2.1.2 erläutert wird, definieren Graphgrammatiken eine Menge von kontextsensitiven Ersetzungsregeln, die als solche Teilmengen des zu verarbeitenden Laufzeitgraphen ermitteln.

Spezifika

Je nach gewähltem Ansatz kann die Mustersuche verschiedene Konstrukte beinhalten. So werden in der Urform lediglich ungetypte Kanten zur Mustersuche verwendet, wohingegen aktuelle Sprachen getypte Graphmodelle [CEL+94] einsetzen. Ferner werden Kanten in den algebraischen Ansätzen identifiziert. Hierdurch werden Multigraphen ermöglicht, in denen Kanten gleichen Typs und mit identischen Start- und Zielknoten existieren. Die dem algorithmischen Ansatz zugeordneten Varianten schließen derlei Multigraphen dagegen aus.

Unterschiede bestehen zudem in der Unterstützung von Attributen, die einerseits Knoten, selten aber auch Kanten zugewiesen werden können. Häufig können Attributwerte als Teil einer Anfrage eingesetzt werden, um die Menge der ermittelten Graphelemente einzuschränken. Einige Spezifikationssprachen bieten hierfür Anschluss an gängige Programmiersprachen, insbesondere falls diese als Zielsprache einer nachgeschalteten Quelltextgenerierung dienen. In anderen Fällen wird dagegen eine eige-

ne Teilsprache zur Attributberechnung oder standardisierte Lösungen wie die OCL angeboten.

Neben diesen grundlegenden Konzepten für Suchmuster bieten gängige graphbasierte Sprachen verschiedene Erweiterungen an. Hierzu zählen *Pfadausdrücke*, mit denen Graphstrukturen unbekannten Umfangs zur Mustersuche herangezogen werden können. Pfadausdrücke werden dabei sehr unterschiedlich ausgestaltet, wobei allen Varianten die Möglichkeit zur beliebig häufigen Wiederholung von Teilausdrücken gemein ist. Zusätzlich bieten einige Sprachen auch Verzweigungen sowie Bedingungen, bspw. auf Basis von Attributwerten, zur Gestaltung von Pfadausdrücken an. Pfadausdrücke wurden bereits in einigen Arbeiten verallgemeinert, so bspw. durch „Stern"-Regionen in [LLP07] oder durch rekursive Suchmuster [VHV07]. Allgemein wird hier im Folgenden von *transitiven Ausdrücken* gesprochen.

Eine ebenfalls verbreitete Erweiterungsmöglichkeit besteht in der *Negation* von Teilausdrücken eines Graphmusters, mittels derer die Anwendbarkeit einer Ersetzungsregel eingeschränkt werden kann. Hier sind unter gängigen Sprachen die Negation einzelner Elemente eines Suchmusters in Form von Knoten oder Kanten, einer Erweiterung dieser auf negierte Pfadausdrücke sowie schließlich die Negation kompletter Suchmuster verfügbar.

Als letztes häufig eingesetztes Sprachkonzept im Bereich Graphgrammatiken sind *mengenwertige* Ausdrücke zu nennen. Diese kommen bspw. in Form amalgamierter Regeln im algebraischen Ansatz zu tragen, und beschreiben das Auffinden der Menge aller möglicher Anwendungsstellen eines Teilmusters bezüglich einer einzelnen Kernanwendungsstelle. In Systemen auf Basis des algorithmischen Ansatzes werden dagegen i.d.R. einzelne Knoten eines Graphmusters als mengenwertig markiert, um genau für diese Elemente alle entsprechenden Belegungen aufzufinden.

Entstehende Anforderungen

Die aufgeführten Eigenschaften gängiger Graphgrammatikansätze erfordert eine breite Palette an Funktionalität, um eine angemessene Realisierungsunterstützung bieten zu können. Hierbei kann zwischen einfacher Funktionalität in Form von Teilgraphensuche und der Unterstützung komplexerer Strukturen wie Negation oder mengenwertiger Ausdrücke unterschieden werden. Grundsätzlich ist eine vollständige rahmenwerksbasierte Unterstützung sämtlicher o.g. Eigenschaften nicht sinnvoll, da hierzu z.T. überlappende Funktionsbereiche einzeln abgedeckt werden müssten.

Im Bezug auf *einfache Funktionalität* besteht der wesentliche Unterschied zwischen einzelnen Sprachen in der Art der Typisierung der angefragten Elemente. Gängige Sprachen erfordern zwingend typspezifische Anfragen, allerdings existieren auch Lösungen welche dies nur optional vorsehen. Im typisierten Fall ist auch die stati-

sche Semantik der Spezifikationssprache zu berücksichtigen, die Typvererbung mittels Einfach- oder Mehrfachvererbung vorsehen kann. Je nach beabsichtigter dynamischer Semantik sind Untertypen in das Anfrageergebnis aufzunehmen, oder davon auszuschließen. Hieraus ergibt sich eine erforderliche Parametrisierbarkeit hinsichtlich der Anfrage von Elementen.

Die Behandlung von Attributwerten stellt stark unterschiedliche Anforderungen an eine mögliche Lösung, insbesondere da in einigen Fällen Programmiersprachen mit sehr breiter angebotener Funktionalität eingesetzt werden. Fallspezifische Lösungen sind hierbei unumgänglich, könnten aber durch geeignete Basisfunktionen wie boolesche Ausdrücke o.Ä. unterstützt werden.

Die starke Divergenz der angebotenen *komplexen Strukturen* erfordert ebenfalls eine geeignete Unterstützung spezifischer Lösungen sowie Parametrisierbarkeit generischer Funktionalität. Eine möglichst allgemeine Lösung ist dabei zu bevorzugen: So stellen bspw. negierte Teilmuster eine echte Generalisierung einzelner Konstrukte wie negierter Knoten oder Kanten dar. Obwohl letztere mit einer allgemeinen Lösung umständlicher zu modellieren sind, wird Redundanz in den bereitgestellten Konstrukten vermieden. Dies gilt gleichermaßen für die Behandlung von Pfadausdrücken und verwandten Konstrukten.

3.4.2 Graphgrammatiken: Graphersetzung

Neben der oben diskutierten Mustersuche erfordert die Realisierung von graphbasierten Spezifikationssprachen Funktionalität zur *Ersetzung* der gesuchten Teilgraphen.

Spezifika

Die einfachste Form der für Graphgrammatiken benötigten Sprachkonstrukte umfassen das Löschen und die Erzeugung von Graphelementen, je nach verwendetem Graphmodell also von Knoten und Kanten. Hinzu kommt die Veränderung von Attributwerten, die von allen gängigen Werkzeugen unterstützt werden.

Zusätzlicher Bestandteil der algorithmischen Ansätze zu Graphtransformationen sind die sogenannten *Einbettungsüberführungsregeln*. Diese beschreiben die Einbettung der durch eine Regelanwendung betroffenen Knoten in ihre jeweilige Umgebung. Hierzu bietet bspw. PROGRES die Möglichkeit, inzidenten Kanten eines festgelegten Typs auf andere Knoten umzuleiten, Kanten zu löschen oder zu duplizieren.

Die auf dem algebraischen Ansatz beruhende Sprache AGG bietet des weiteren Funktionalität um zwei oder mehrere Knoten zu *verkleben*. Hierbei werden sowohl die inzidenten Kanten der betroffenen Knoten sowie deren Attributwerte auf einem einzelnen Ergebnisknoten vereinigt. Falls die zu verschmelzenden Knoten unterschiedli-

che Attributwerte für ein gegebenes Attribut aufweisen, so muss mittels Benutzerinteraktion einer dieser Werte ausgewählt werden [TB93].

Neben der mengenwertigen Suche, die im vorherigen Unterabschnitt behandelt wurde, entsteht der Bedarf nach ebenso mengenwertiger Manipulation von Graphstrukturen. Hierunter fallen geeignete Änderungsoperationen, die auf Mengen anstelle einzelner Elemente operieren können. Wie im Fall der Mustersuche kann dies durch amalgamierte Ersetzungsregeln oder auch durch rekursive Muster erreicht werden.

Entstehende Anforderungen

Wie bereits bei der Mustersuche in Abschnitt 3.4.1 aufgezeigt wurde, entsteht je nach betrachtetem Werkzeug eine Vielzahl z.T. sehr spezifischer Anforderungen an eine geeignete Realisierungsplattform. Die eingangs erwähnten einfachen Veränderungsoperationen sind dabei leicht bereitzustellen, da diese Funktionalität in den üblichen Spezifikationssprachen gleichermaßen verwendet wird. Auch Einbettungsüberführungsregeln können auf diese Weise ausgedrückt werden. Da diese allerdings Graphersetzungen in unbekanntem Kontext, und damit beliebig vielen verbundenen Elementen beschreiben, sind hierzu Veränderungsoperationen auf mengenwertigen Strukturen erforderlich.

Fallspezifische Anpassungen sind dagegen wieder für hochgradig spezialisierte Konstrukte, bspw. der Verklebeoperation in AGG, nötig. Da diese u.a. Benutzerinteraktion vorsieht, ist die Einbettung in eine geeignete Anwenderschnittstelle spezifisch zu realisieren.

3.4.3 Programmierte Graphersetzung

Die oben diskutierten Graphgrammatiken bieten grundsätzlich Funktionalität zur Verarbeitung von Graphstrukturen. Dennoch ist ihre Anwendbarkeit auf praktische Szenarien jedoch beschränkt, da die so erstellten Spezifikationen lediglich aus Sammlungen einzelner Ersetzungsregeln bestehen.

Spezifika

Programmierte Graphersetzungssysteme nutzen *Kontrollstrukturen*, um Graphersetzungen explizit steuern zu können. Die Vielfalt der möglichen Strukturen variiert dabei stark: Einige Systeme wie bspw. VMTS bieten einen beschränkten Satz an Sprachkonstrukten an, die UML Aktivitätsdiagrammen entlehnt sind. Fujaba bietet zusätzlich sogenannte *Kollaborationsmuster* innerhalb von Transformationsregeln an, mittels

derer andere Transformationsregeln aufgerufen werden können. Auch Iterationen können so realisiert werden. PROGRES stellt zudem nicht-deterministische Regelanwendungen bereit. Entsprechend existieren Sprachkonstrukte, die Regelanwendungen als Alternativen oder in beliebiger, nicht definierter Reihenfolge ermöglichen.

Neben der eigentlichen Ablaufsteuerung wird in allen genannten Systemen zugleich auch Datenfluss spezifiziert, was häufig in Form von Ein- und Ausgabeparameter einer Graphersetzungsregel erfolgt. VMTS setzt hierzu visuelle Datenspeicherkonstrukte ein, mit denen schreibende und lesende Transformationsregeln verbunden werden.

Entstehende Anforderungen

Zur Realisierung programmierter Graphersetzungen auf Basis einer graphbasierten Ausführungsmaschine muss diese im Prinzip keine eigene Unterstützung für Kontrollstrukturen anbieten. Stattdessen kann die notwendige Funktionalität simuliert werden, indem bspw. zusätzliche Knoten eine Markierung der aktuell ausführbaren Transformationsregeln realisieren. Hierbei entstünde allerdings ein hoher Realisierungsaufwand um zusätzlich benötigte Transformationsregeln zu erzeugen. Um Anforderung 2 abdecken zu können, sollte eine Zielsprache für programmierte Graphersetzungssprachen daher Kontrollstrukturen nativ bereitstellen.

Nichtdeterministische Kontrollstrukturen, die bspw. von PROGRES angeboten werden, erfordern aus dem gleichen Grund eine geeignete Unterstützung in der Ausführungsmaschine. So könnte zwar Nichtdeterminismus ebenso simuliert werden, dies würde aber zu hochgradig komplexen und damit unterwünschten Sprachabbildungen führen. Da eine üblicherweise durchgeführte Implementierung jedoch Backtracking einsetzen und somit Graphveränderungen protokollieren müsste, ist diese Unterstützung aus Performanzgründen modular zu gestalten.

3.4.4 Modellübersetzung

Im Unterschied zu den vorgenannten Fällen wird Modellübersetzung vornehmlich nicht innerhalb eines Dokuments eingesetzt, sondern für den Abgleich zwischen verschieden strukturierten. Die Gewährleistung eines geeigneten Abstraktionsniveaus macht auch hier spezielle Sprachkonzepte notwendig.

Spezifika

Eine wichtige Eigenschaft spezialisierter Übersetzungssprachen ist die Nachverfolgbarkeit der durchgeführten Übersetzung, die sogen. *Traceability*. Hierdurch können

sequenzielle Übersetzungsschritte auf die Ergebnisse ihrer Vorgänger zugreifen. Zur weiteren Verarbeitung kann zudem ein Ergebnis, das auf Basis des Zielmodells ermittelt wurde, mit entsprechenden Eingabedaten verbunden werden.

Eine oftmals geforderte Eigenschaft von Modellübersetzungen ist die *Bidirektionalität*, mittels derer die übersetzten Modelle in ihre Ursprungsform zurück übertragen werden. Hierdurch wird eine Veränderbarkeit der Resultate ermöglicht. Auch kann damit eine kontinuierliche Übersetzung zwischen Quell- und Zielmodellen erreicht werden, wodurch eine weitere Problematik entsteht: Durch parallele Veränderung von Quell- und Zielmodellen können *Inkonsistenzen* entstehen, falls die jeweiligen Änderungen zueinander in Konflikt stehen.

Insbesondere durch Unterstützung bidirektionaler und kontinuierlicher Übersetzung kann in vielen Anwendungsfällen keine vollautomatische Realisierung erfolgen. Stattdessen ist eine *Interaktion* mit dem Werkzeugnutzer erforderlich, der in Konfliktfällen eine Entscheidung mittels des ihm eigenen Expertenwissens trifft.

Entstehende Anforderungen

Aus den spezifischen Eigenschaften für Modellübersetzungen entstehen einige Anforderungen an eine mögliche Realisierungsplattform. Da diese z.T. deutlich von den bisher behandelten Aspekten divergieren, ist grundsätzlich eine Auslagerung der für diese Systemart bereitgestellten Funktionalität sinnvoll. Auf diese Weise kann eine Überladung der Realisierungsplattform vermieden werden.

Teil einer solch spezifischen Lösung wäre die automatisierte Verwaltung der Nachverfolgbarkeit. Bezüglich Bidirektionalität kann durch geeignete Funktionalität eine modellierte Übersetzung als Synchronisation zwischen Modellelementen realisiert werden. Interaktion sowie die Behandlung von Inkonsistenzen müssten in diesem Zusammenhang durch geeignete Benutzerschnittstellen unterstützt werden. Hierbei kann generische Rahmenwerksfunktionalität jedoch nur begrenzt sinnvoll bereitgestellt werden.

3.4.5 Zusammenfassung der Anforderungen

Die folgende Tabelle fasst die bisher ermittelten Anforderungen zusammen. Zur Redundanzvermeidung werden wiederholte Aspekte nur einmalig aufgezählt.

Anforderung	Szenario	Wichtigste Spezifika
1 – Verhaltensmodellierung	Abschnitt 3.3	*Navigation* entlang v. Verbindungsstrukturen; *Attributabfragen* und *Werteaggregation*; Behandlung *nicht abgeschlossener Spezifikationen*
	Abschnitte 3.4.1, 3.4.2	*Mustersuche* einschl. komplexer Konstrukte, bspw. Negation und Rekursion; *Transformation* durch Erzeugung, Löschung, Änderung von Elementen
	Abschnitt 3.4.3	*Ablaufsteuerung* für komplexe Anwendungen, insbesondere Kontroll- und Datenfluss
	Abschnitt 3.4.4	Möglichkeit zur *Korrespondenzmodellierung*, Gewährleistung der *Nachverfolgbarkeit*, ggf. *Umkehrbarkeit* der Übersetzung
2 – Abstraktionsniveau	Abschnitt 3.3	Abstraktion von *Datenzugriff* und *Navigation* entlang Verbindungsstrukturen
	Abschnitte 3.4.1, 3.4.2	*Graphmustersuche* anstelle technischer Basisoperationen; Nichtdeterministische *Transformation* von Teilgraphen
	Abschnitt 3.4.3	Abstraktion der Ausführungsreihenfolge durch *nichtdeterminitische Kontrollstrukturen*
	Abschnitt 3.4.4	*Anwendungsreihenfolge* einzelner Übersetzungsschritte; implizite *Nachverfolgbarkeit*
3 – Anpassbarkeit	Abschnitt 3.3	Erweiterte Pfadausdrücke; Spezifische Aggregationsfunktionen, alternativ Zugriffsmöglichkeit auf externe Bibliotheken
	Abschnitte 3.4.1, 3.4.2	Unterstützung spezifischer Funktionalität, bspw. Pfadausdrücke, Verklebung, etc.
	Abschnitt 3.4.3	Parametrisierbare Unterstützung von Nichtdeterminismus; ggf. Ergänzbarkeit weiterer Konstrukte zur Kontrollflussmodellierung
	Abschnitt 3.4.4	Spezifisch für gegebene Dokumententypen, bspw. in Form von Traversierungsstrategien

fortgesetzt auf folgender Seite

Anforderung	Szenario	Wichtigste Spezifika
4 – Plattform-unabhängigkeit	Abschnitt 3.3	Technologieunabhängige *graphbasierte Datenrepräsentation*; *Navigation* entlang Verbindungsstrukturen, *Auswertung* von Attributwerten, etc.
	Abschnitte 3.4.1, 3.4.2	zusätzlich *verändernder Zugriff* auf Datenbasis
	Abschnitt 3.4.3	(keine weiteren Anforderungen)
	Abschnitt 3.4.4	Gemeinsame Nutzung *mehrere Datenbasen* für Quell- und Zieldokumente sowie zur Verwaltung von Nachverfolgbarkeitsinformationen

Tabelle 3.1: Zusammenfassung der entstehenden Anforderungen

3.5 Zusammenfassung

Dieses Kapitel hat zunächst ein realistisches Anwendungsszenario des Ansatzes der vorliegenden Arbeit vorgestellt. Ziel ist dabei die zeitgünstige Realisierung einer Anfragesprache für graphbasierte Daten. Im weiteren Verlauf dieser Arbeit wird aufgezeigt, wie dies mit Hilfe des bereitgestellten Rahmenwerks erreicht wird. Hierzu folgt zunächst eine detaillierte Vorstellung der zu diesem Zweck angebotenen Kernsprache sowie der zugehörigen Ausführungsmaschine. Anschließend wird die Möglichkeit aufgezeigt, einen Übersetzer zur Nutzung dieser Kernsprache regelbasiert zu spezifizieren.

4 DRAGULA - Eine Kernsprache für Graphtransformationen

Dieses Kapitel stellt die als *DRAGULA* bezeichnete Graphtransformationssprache vor, die Kernbestandteil der vorliegenden Arb. Der Name DRAGULA ergibt sich aus der technisch engen Anbindung der Sprachrealisierung an den Graphspeicher DRAGOS bzw. deren gemeinsam genutztes Graphmodell, und kürzt daher den Begriff *DRAGOS Unified Language* ab.

Der angestrebte Nutzen der Sprache DRAGULA, der bereits in Abschnitt 1.4 auf Seite 16 erläutert worden ist, wird in Abbildung 4.1 noch einmal verdeutlicht. Zur Realisierung einer Spezifikationssprache erfolgt eine Übersetzung, oder Kompilation, dieser Sprache nach DRAGULA. Durch die bereitgestellte Ausführungsmaschine, die Gegenstand von Kapitel 5 ist, führt diese Übersetzung zu einer ausführbaren Lösung.

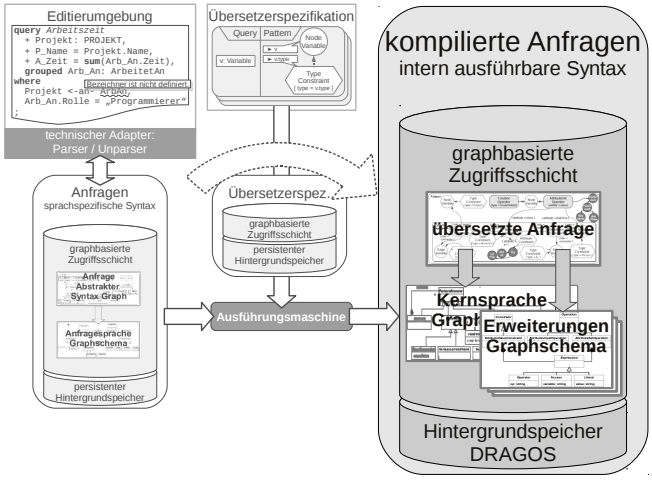


Abbildung 4.1: Einordnung der Kernsprache (analog zu Abbildung 1.5)

DRAGULA ist unter verschiedenen Gesichtspunkten als *Kernsprache* konzipiert, was sich insbesondere in folgenden Eigenschaften zeigt:

Reduzierter Sprachumfang: Die bereitgestellten Sprachkonstrukte werden auf wesentliche Elemente beschränkt, so dass Redundanzen in der Sprachdefinition

vermieden werden. Auf diese Weise kann die Ausführungsmaschine kompakt gestaltet werden. Zudem entsteht so nicht die Notwendigkeit, für eine Vielzahl erdenklicher Anwendungsszenarien entsprechende Sprachkonstrukte bereitstellen zu müssen.

Fokussierung auf Verhaltensanteile: DRAGULA ist mit der Zielsetzung entwickelt worden, verhaltensorientierte Spezifikationssprachen abbilden zu können. Hierzu u.a. die Anfragesprachen wie GReQL oder der Beispielsprache EMAA, die als reine verhaltensorientierte Sprache keine eigenen Schemadefinitionssprachen anbieten. Um auch diese Sprachen abdecken zu können, wird DRAGULA ebenfalls als offene Sprache *ohne Sprachkonstrukte zur Schemadefinition* konzipiert. Typinformationen werden somit ausschließlich zur Laufzeit aus der jeweiligen Datenquelle bezogen. In Bezug auf die DRAGULA Sprachdefinition werden Schemainformationen somit nicht als Teil einer Spezifikation berücksichtigt, was die statische Prüfbarkeit der erstellten Spezifikationen allerdings erschwert.

Erweiterbarkeit und Parametrisierbarkeit: Trotz der bewusst vorgenommenen Einschränkungen kann DRAGULA erweitert werden, um ergänzende Sprachkonstrukte anzubieten. Dies ist bspw. im Bereich domänenspezifischer Systeme relevant, die häufig wiederkehrende Funktionalität aufweisen. So wird erreicht, dass Familien spezifischer Systeme die erweiterte Sprache DRAGULA mit angemessenem Aufwand als Realisierungsgrundlagen nutzen können.

Zielgruppe: DRAGULA ist nicht primär für die direkte Benutzung durch Entwickler, sondern stattdessen als Zielsystem für generierende Werkzeuge wie Compiler konzipiert. So weist die Sprache lediglich eine *abstrakte Syntax* auf, bietet also keine auf Verständlichkeit oder Ästhetik bedachte Visualisierung an. Eine grundlegende Editierumgebung, die sich eng an der abstrakten Syntax orientiert, wird allerdings nichtsdestotrotz angeboten.

Das Kapitel gliedert sich im Folgenden wie folgt: Zunächst erfolgt eine einleitende Übersicht der jeweiligen Teilaspekte um verschiedene Komponenten von DRAGULA zu identifizieren. Hieran schließt sich deren schrittweise Untersuchung, die sich jeweils in einem Beispiel sowie der anschließenden syntaktischen und semantischen Definition unterteilen. Abschließend erfolgen Literaturvergleich und Zusammenfassung.

4.1 Strukturübersicht

Der erste Abschnitt dieses Kapitels führt zunächst geeignete Notationen ein um Syntax und Semantik der Kernsprache darstellen zu können. Anschließend wird der komponentenweise Aufbau von DRAGULA vorgestellt an dem sich die weitere Beschreibung orientiert.

4.1.1 Eingesetzte Notationen

Sowohl die syntaktische als auch die semantische Spezifikation der Kernsprache DRAGULA wird im Folgenden mit Hilfe möglichst einfacher Notationsformen vorgenommen. Dabei liegt der Schwerpunkt auf einer interpretationsfreien Darstellung der Sachverhalte, wohingegen ein formal abgeschlossener Kalkül bspw. zum formalen Beweisen nicht im Fokus dieser Arbeit liegt. Auf diese Weise wird die Komplexität der Sprachbeschreibung beherrschbar, wohingegen eine entsprechend reichhaltige algebraische Definition mehrere Teildefinitionen vereinigen müsste, etwa hinsichtlich Typgraphen, attributierten Graphgrammatiken oder transitiv abgeschlossenen Graphmustern.

Syntax

Zur *syntaktischen Definition* werden im Verlauf dieser Arbeit standardisierte Sprachen der OMG eingesetzt, da diese Notationen allgemein verbreitet und für versierte Leser leicht verständlich sind. Die *kontextfreie* Syntax wird mittels UML Klassendiagrammen spezifiziert, die somit ein Metamodell der Sprache DRAGULA angeben. Eine formal präzise Spezifikation der UML, die zu einer widerspruchsfreien Interpretation der Sprachbeschreibung nötig wäre, liegt nicht im Rahmen dieser Arbeit. Allerdings stellt [RK08] einen Ansatz auf Basis von *Typgraphen* bereit, der als formale Grundlage der im Folgenden dargestellten UML Klassendiagramme dient. Die nötigen geringfügigen Ergänzungen dieses Ansatzes werden fallweise kommentiert.

Alternativ könnte auch eine syntaktische Beschreibung mittels *EBNF-Grammatiken* und prädikatenlogischer Regeln zur Bezeichnerbindung analog zur Sprachbeschreibung von PROGRES [Sch91] erfolgen. Dies würde allerdings dem grundsätzlich graphischen Ansatz von DRAGULA widersprechen und die mögliche visuelle Ausgestaltung von Beispielen behindern. Auch der Einsatz von *Graphgrammatiken* wird an dieser Stelle nicht weiter verfolgt, da der Metamodellierungsansatz aufgrund höherer Verständlichkeit in der Literatur mittlerweile dominiert.

Die *kontextsensitive* Syntax wird in Form von OCL Ausdrücken angegeben. Der Vorteil von OCL liegt in seiner direkten Anbindung an die UML Infrastruktur, so

dass syntaktische Klassendiagramme auf einfache Weise um ergänzende Wohlgeformtheitsbedingungen angereichert werden können. Demgegenüber würden prädikatenlogische Formeln mit gleicher Zielsetzung zunächst eine Überführung der UML Diagramme in eine entsprechende Formelmenge erfordern.

Auch für die OCL gelten o.g. Einschränkungen bezüglich ihres Grads der Formalisierung, der seitens der OMG nur teilweise erfolgt ist. Dennoch sind hinreichend weite Teile dieser Sprache in einigen Arbeiten bereits formal ausgestaltet worden [Baa02], so dass sie als geeignete Grundlage zur Definition von DRAGULA eingesetzt werden kann. Im Übrigen wird auf das OCL-Schlüsselwort `allInstances`, welches je nach verwendeter semantischer Domäne einer Allquantifizierung über unendliche Strukturen entspricht [BW02], verzichtet. Ebenfalls nicht verwendet wird das Schlüsselwort `any` zur nichtdeterministischen Auswahl, welches eine strittige semantische Definition besitzt [Baa05]. Auch werden zur Vermeidung möglicher Mehrdeutigkeiten keine verkürzten Notationen wie in Abschnitt 3.3 eingesetzt.

Als *korrekt* wird eine DRAGULA-Spezifikation im Folgenden bezeichnet, wenn (1.) ihre Elemente gemäß der üblichen Formalisierung eine Instanz des Metamodells darstellen, und (2.) alle Wohlgeformtheitsbedingungen erfüllt sind. Korrektheit bezieht sich damit ausschließlich auf statische, nicht jedoch auf zur Laufzeit prüfbare Eigenschaften.

Semantik

Zur semantischen Definition der Kernsprache DRAGULA werden im Rahmen dieser Arbeit hauptsächlich Formeln der Prädikatenlogik erster Stufe eingesetzt. Zurück gegriffen wird dabei auf die in Abschnitt 2.4 gegebene formale Darstellung des DRAGOS Graphmodells und damit auf die Instanzdaten der zu verarbeitenden Datenbasis. In dieser Hinsicht erfolgt die Angabe einer *operationellen Semantik*, in dem syntaktische Elemente mit ihrem Verhalten bezüglich des zugrundeliegenden Graphspeichers in Beziehung gesetzt werden.

Auch wenn hierbei ähnliche Bezeichner wie in der Beschreibung von PROGERS [Sch91] eingesetzt werden, so ist die Beschreibungsweise dennoch grundsätzlich unterschiedlich. Die Semantik von PROGRES wird auf *denotationale* Weise festgelegt, in dem deren jeweiligen syntaktischen Konstrukte auf die Semantik des Kalküls programmierbarer Graphersetzungssysteme rückgeführt wird. Die vorliegende Arbeit ist von ihrem Ansatz her dagegen eher zu vergleichen mit der Formalisierung von Fujaba [Zü01], die ebenfalls einen operationellen Ansatz wählt. Ferner kann die semantische Beschreibung nicht allein durch Rückführung auf algebraische Ansätze, etwa dem Single-Pushout-Approach [EHK$^+$97] formuliert werden. Dies widerspräche der angestrebten Erweiterbarkeit des Ansatzes, da hierbei die gesamte semantische Abbildung

in der Formulierung eines geeigneten Morphismus erfolgen würde.

Um eine enge Kopplung prädikatenlogischer Formeln auf Semantikseite mit der syntaktischen Beschreibung in Form von Metamodellen und OCL-Bedingungen zu ermöglichen, wird eine *implizite Definition* bestimmter Formeln angenommen. So wird vereinbart, dass Klassenobjekte des Metamodells als nullstellige Relationssymbole aufzufassen sind, welche die Menge aller Instanzen festlegen. Assoziationen werden als einstellige Relationssymbole interpretiert, die unter Angabe eines Ausgangspunkts die Menge aller per Traversierung erreichbaren Instanzen ergeben.

4.1.2 Sprachstruktur

Abbildung 4.2 zeigt die Grobstruktur DRAGULAs in Form einzelner Sprachkomponenten. Innerster Bestandteil ist diejenige Funktionalität, die zur Formulierung *einfacher Suchanfragen* benötigt wird. Der Begriff „einfach“ bezieht sich dabei auf die bereitgestellten Sprachkonstrukte, die gewissermaßen den kleinsten gemeinsamen Anteil graphbasierter Spezifikationssprachen umfassen. Auf diesem Kernbestandteil basieren *zusammengesetzte Suchmuster*, die erweiterte Funktionen bspw. in Form boolescher Musterkombinationen ergänzen. *Transformationsregeln*, die ebenfalls auf die einfache Mustersuche zurückgreifen, stellen Sprachkonstrukte zur Veränderung der aufgefundenen Graphstrukturen bereit. Als äußere Komponente ermöglicht die *Ablaufsteuerung* schließlich die schrittweise Abarbeitung einzelner Anfragen und Transformationen.

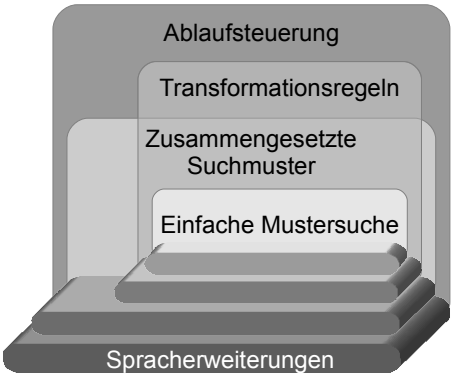


Abbildung 4.2: Übersicht der Sprachstruktur

Wie die Abbildung ferner andeutet, können sämtliche der dargestellten Komponenten durch Erweiterungsmodule ergänzt werden. Die Erweiterbarkeit ist Gegenstand von Abschnitt 4.6 aus konzeptioneller, und von Abschnitt 5.5 aus werkzeugtechnischer Sicht.

4.2 Einfache Mustersuche

Als erste Komponente wird die einfache Mustersuche behandelt, die grundlegende Anfragefunktionalität etwa zur Realisierung von Graphgrammatiken umfasst. Im Anschluss an ein einfaches Anwendungsbeispiel folgt die syntaktische und semantische Definition der dabei eingesetzten Sprachkonstrukte.

4.2.1 Anwendungsbeispiel

Ein initiales Anwendungsbeispiel der Sprache DRAGULA zeigt der obere Teil von Abbildung 4.3. Informell ausgedrückt werden hierdurch alle Knoten des Typs Projekt ermittelt, die im Graphspeicher abgelegt sind. Syntaktisch unterscheidet DRAGULA zwei Klassen von Sprachkonstrukten zur einfachen Mustersuche – Variablen und Bedingungen. Während *Variablen* die abzufragenden Elemente festlegen denen zur Laufzeit der Anfrage entsprechende Werte zugewiesen werden, schränken *Bedingungen* die hierfür zulässigen Werte ein. Die spezifizierte Anfrage ermittelt somit alle in der Datenbasis verfügbaren Knoten deren Typisierung dem nebenstehenden TypeConstraint genügen. Dies ist gemäß dem Attribut für Knoten vom Typ Projekt der Fall. Der untere Teil der Abbildung zeigt den betrachteten Laufzeitgraphen. Die möglichen Zuweisungen der darin enthaltenen Knoten zu der Knotenvariable der DRAGULA-Spezifikation ist im mittleren Teil der Abbildung dargestellt. Hierbei fasst eine Auffindungsstelle (Match) Zuweisungen (Assignment) für alle angegebenen Variablen zusammen.

Das zweite betrachtete Beispiel in Abbildung 4.4 modelliert die Suche nach miteinander verbundenen Graphelementen. Hierzu wird ein sogen. IncidenceConstraint mit drei Variablen verbunden. Diese Verbindungen werden attributiert um verschiedene Rollen der adjazenten Variablen zu definieren. Im vorliegenden Fall werden Belegung der Kantenvariable mit Rollenbezeichnung connector gesucht, welche von der Belegung der linken Knotenvariable ausgehen und zur Belegung der rechten Variable hin gerichtet sind. Die nebenstehende Spezifikation zeigt eine verdichtete Darstellung ohne Rollenbezeichner, welche die gesuchte Richtung in Form visueller Elemente festlegt. Analog zu obigem Beispiel wird das Auffinden geeigneter Elemente in der Datenbasis durch Auffindungsstellen markiert. Die jeweils zugewiesene Variable ergibt sich in der Abbildung zur besseren Übersicht aus der analogen Anordnung von Auffindungsstelle und Suchmuster.

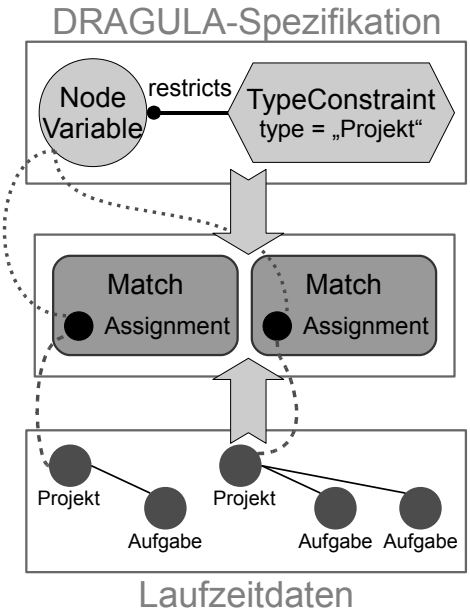


Abbildung 4.3: Beispielspezifikation zur Typprüfung

4.2.2 Syntaktische Definition

In Abbildung 4.5 wird der bisher von DRAGULA verwendete Sprachumfang beschrieben. Dargestellt sind die im Folgenden behandelten *Muster* zur syntaktischen Darstellung von Anfragen sowie den darin enthaltenen Variablen und Bedingungen. Im weiteren Verlauf werden zusätzliche Ausdrucksmittel zur Wertzuweisung und damit zur dynamischen Auswertung von DRAGULA Anfragen eingeführt.

Muster

Ein Muster (Pattern) enthält demnach eine Menge von Elementen (PatternElement), die wie bereits aus den Beispielen ersichtlich in Variablen und Bedingungen untergliedert sind. Jede Bedingung ist an wenigstens eine Variable über eine Verbindung der Assoziation Restricts verbunden. Diese Verbindung kann optional einen Rollennamen aufnehmen, im Standardfall ist dieser leer.

Die kontextsensitive Syntax zu diesem Metamodellfragment legt fest, dass Variablen und Bedingungen stets in einem Muster enthalten sein müssen. Dies könnte zwar auch direkt durch eine entsprechende Kardinalitätsangabe im Metamodell definiert werden, allerdings wird diese Typhierarchie im Folgenden noch erweitert werden. Ferner gilt, dass Restricts-Beziehungen stets Elemente eines Musters miteinander in Beziehung setzen. Dies ist eine *vorläufige* Bedingung, die im nachfolgenden Abschnitt redefiniert werden wird.

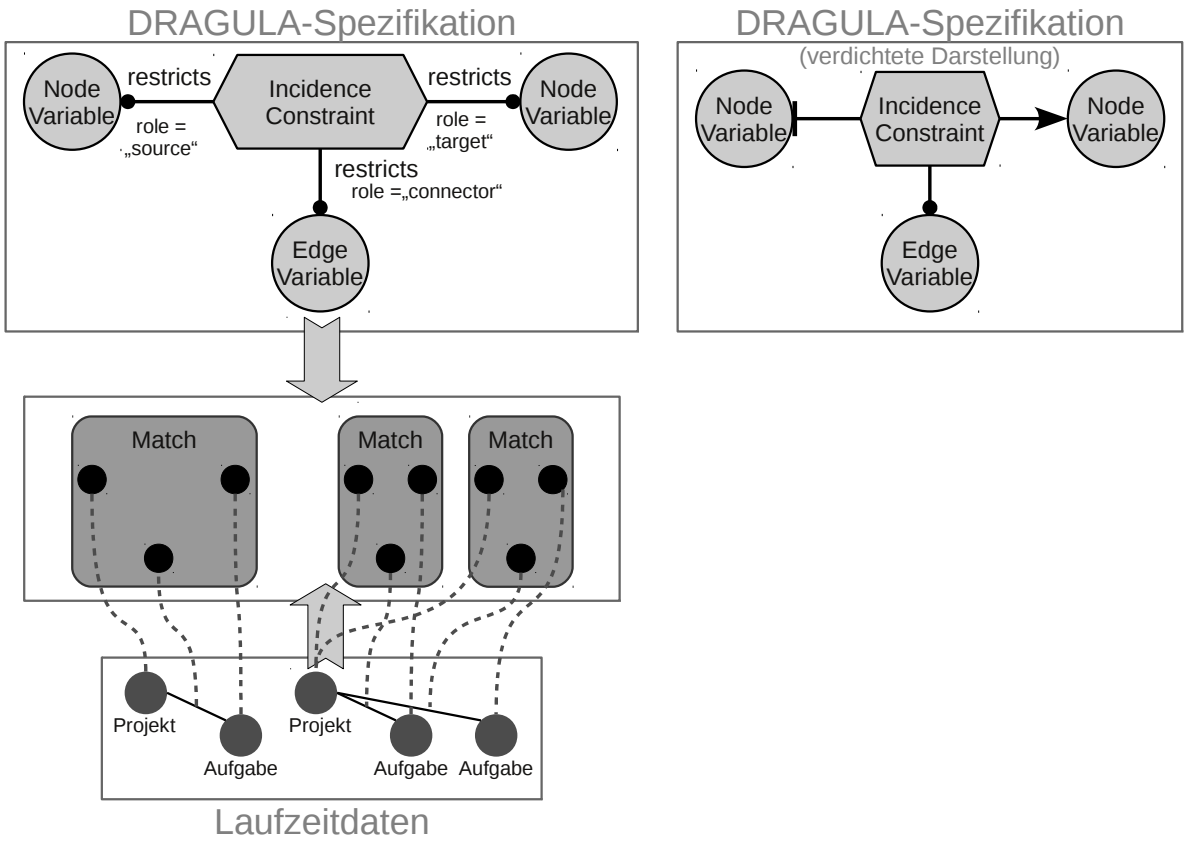


Abbildung 4.4: Beispielspezifikation zur Kantentraversierung

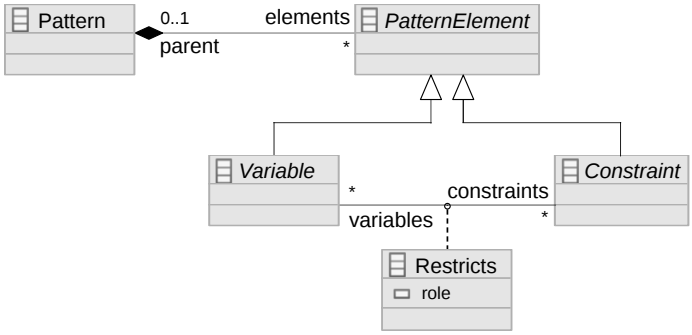


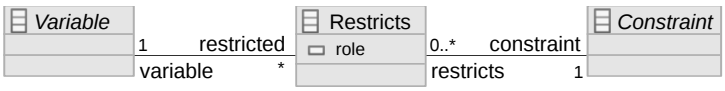
Abbildung 4.5: Definition Suchmuster

```
context Variable
inv nullParent : not self.parent.oclIsUndefined()

context Constraint
inv nullParent : not self.parent.oclIsUndefined()
```

```
context Restricts
inv hierarchy : self.constraints.parent = self.variables.pattern
```

Bemerkung zur formalen Definiertheit: Die Nutzung der Assoziationsklasse Restricts, obgleich dies zur Lesbarkeit des Metamodells und dessen Wohlgeformtheitsbedingungen beiträgt, wird nicht im gewählten Formalisierungsansatz nach [RK08] berücksichtigt. Um dennoch eine eindeutige Interpretation zu gewährleisten, wird diese gemäß Abbildung 4.6 durch *Rückführung* auf eine reguläre Klasse abgebildet. Dadurch wird bei der in Wohlgeformtheitsbedingungen verwendeten Navigation ein Zwischenschritt benötigt, wie exemplarisch als Hilfsfunktion variables in der Abbildung gezeigt wird.




```
context Constraint
def: variables : Set(Variable) =
        self.restricts→collect(r | r.variable)
```

Abbildung 4.6: Rückgeführtes Metamodellfragment und Hilfsfunktion

Variablen

Variablen gliedern sich gemäß des DRAGOS Graphmodells (s. Abschnitt 2.4 auf Seite 41) in die unterschiedlichen Untertypen aus Abbildung 4.7. Hierüber werden im Folgenden unterschiedliche Graphelemente von der zugrundeliegenden Datenbasis abgefragt. Neben den spezifischen Varianten bspw. für Knoten oder Graphen ist auch ein allgemeiner Variablentyp GraphEntityVariable vorhanden, der alle verfügbaren Elemente als Wert aufnehmen kann. Sowohl Kanten- als auch Relationsenden werden in DRAGOS als Verbindungsstrukturen angesehen, weshalb die zugehörigen Variablen einen gemeinsam Obertyp ConnectionVariable besitzen.

Bedingungen

Des Weiteren definiert DRAGULA eine Reihe von Bedingungsklassen, die der Abbildung 4.8 zu entnehmen sind. Hierbei handelt es sich zunächst nur um einen minimalen Satz um die Funktionalität des DRAGOS Graphmodells abdecken zu können. Die Zuständigkeiten der dargestellten Bedingungen werden im Weiteren erläutert.

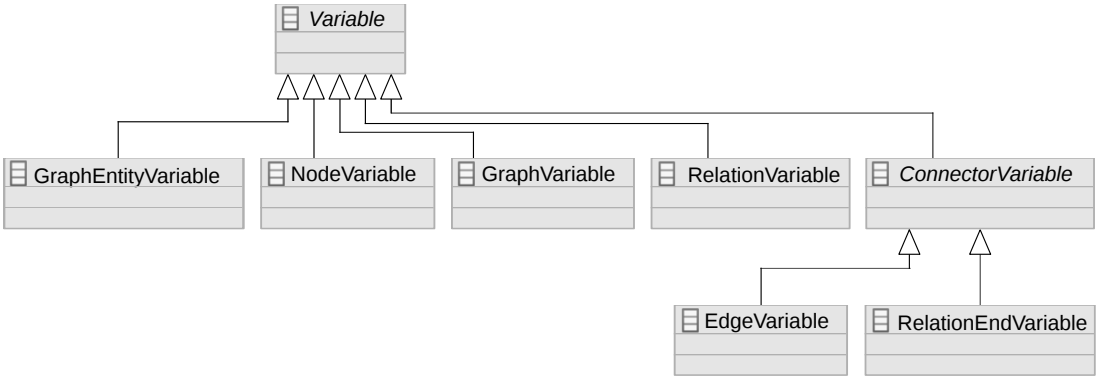


Abbildung 4.7: Typhierarchie der Variablen

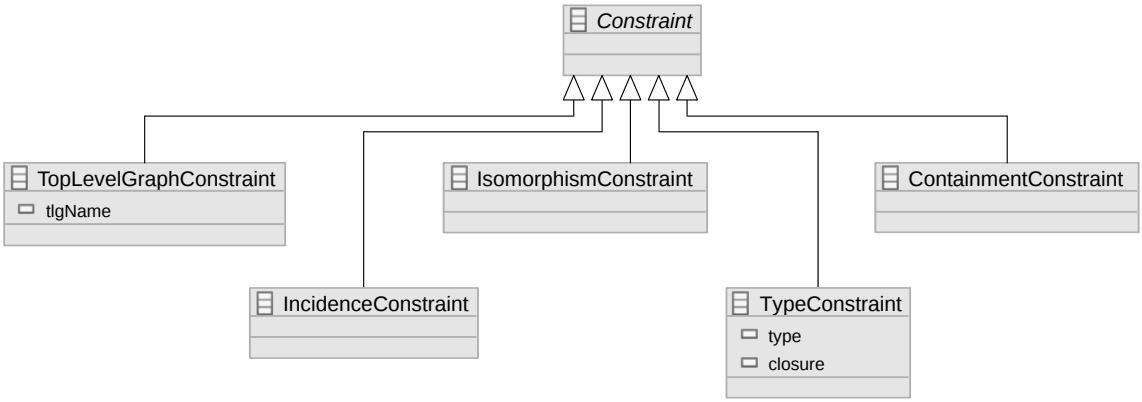


Abbildung 4.8: Typhierarchie der Bedingungen

Typisierung: Das TypeConstraint schränkt verbundene Variablen auf den angegebenen Typ type ein. Dabei besteht die Wahl, ob lediglich direkte Instanzen der angegeben Klassen akzeptiert werden, oder Instanzen der Unterklassenhierarchie (Attribut closure) eingeschlossen werden.

Inzidenz: Das IncidenceConstraint prüft Verbindungsstrukturen zwischen den Belegungen einzelner Variablen. Wie bereits anhand des Beispiels in Abbildung 4.4 gezeigt, werden hierfür Start- und Zielentitäten sowie eine Variable für die gesuchte Verbindung mit einer solchen Bedingung versehen.

Zusätzlich gelten die folgenden kontextsensitiven Einschränkungen: Variablen, die mit einem IncidenceConstraint verbunden sind, müssen über die Rollenbeschriftung source, target, oder connector angeschlossen sein – andere oder keine angegebene Beschriftungen sind nicht zulässig. Die so identifizierten Variablen müssen eindeutig bestimmt sein. Die Quell- oder die Zielvariable, nicht jedoch beide, können ausgelassen

werden. Bei der Verbindung handelt es sich um eine Variable vom Typ ConnectorVariable.

```
context Constraint
def varByRole (role : String) : Set(Variable) =
        self.restricts→select(r1 | r1.role = role)→
        collect(r2 | r2.variables)

context IncidenceConstraint
inv allRoles : self.restricts→forAll(r | r.role=„source“ or
        r.role=„target“ or r.role=„connector“)
inv uniqueRoles : self.varByRole(„source“)→size()≤1 and
        self.varByRole(„target“)→size()≤1 and
        self.varByRole(„connector“)→size()=1
inv minRoles : not self.varByRole(„source“).oclIsUndefined()
        or not self.varByRole(„target“).oclIsUndefined()
inv connector : self.varByRole(„connector“)
                    .oclIsTypeOf(ConnectorVariable)
```

Enthaltensein: Ähnlich zur Inzidenz prüft das ContainmentConstraint die Enthaltenseinsbeziehung zwischen einem Graphen und darin enthaltenen Elementen. Hier gilt die Einschränkung, dass genau eine angehängte Variable mit der Rolle „container“ beschriftet ist, und es sich dabei um eine GraphVariable handelt. Als Rollenbeschriftungen sind lediglich die Werte container und containee zulässig.

```
context ContainmentConstraint
inv allRoles : self.restricts→forAll(r | r.role=„containee“ or
        r.role=„container“)
inv uniqueContainer : self.varByRole(„container“)→size()=1
inv containerVar : self.varByRole(„container“).
        oclIsTypeOf(GraphVariable)
```

Isomorphie: Mittels des IsomorphismConstraint wird die paarweise ungleiche Belegung aller Variablen sichergestellt. Die dadurch entstehenden Zuweisungen bilden eine isomorphe Abbildung vom dargestellten Muster in einen Teilgraphen der Datenbasis, was diesen Namen begründet.

Übergeordnete Graphen: Graphen, die in der DRAGOS Graphdatenbank in keinem anderen Graph enthalten sind, werden als sogen. *Top-Level-Graphen* bezeichnet. Lediglich solch übergeordnete Graphen können mittels eines eindeutigen Namens identifi-

ziert werden. Diese Namen werden durch Angabe eines TopLevelConstraints überprüft. Ähnlich zu oben gilt hier die Einschränkung, dass es sich bei allen verbundenen Variablen um GraphVariable handelt.

```
context TopLevelGraphConstraint
inv varType : self.variables→
        forAll(v | v.oclIsTypeOf(GraphVariable))
```

Wertezuweisung

Um die bisher behandelten syntaktischen Sprachkonstrukte zur Anfrage von Graphstrukturen auswerten zu können, werden zusätzliche Mittel zur Darstellung von Wertezuweisungen benötigt. Auf diesen in Abbildung 4.9 explizit eingeführten Datenstrukturen erfolgt im weiteren Verlauf die semantische Definition von Suchanfragen. Demnach umfasst eine Auffindungsstelle (Match) eines Musters eine Menge von Zuweisungen (Assignment), die jeweils einer Variablen des Musters genau einen Wert aus dem Graphspeicher zuordnet. Hierdurch entsteht eine syntaktische Abbildungsrelation von den Elementen einer DRAGULA-Spezifikation zum Inhalt der Graphdatenbank.

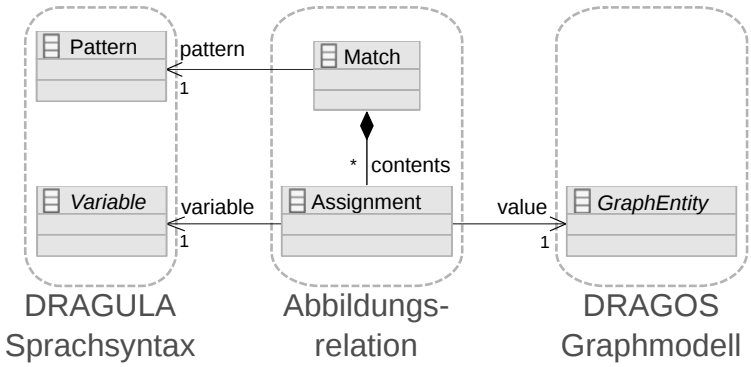


Abbildung 4.9: Verknüpfung des Sprachmodells mit Laufzeitdaten

Die Abbildungsrelation unterliegt einer Reihe von Wohlgeformtheitsbedingungen, die, da es sich um eine rein syntaktische Beziehung handelt, der kontextsensitiven Syntax zuzuordnen sind. Den linken und mittleren Teil der Abbildung betreffend wird gefordert, dass eine Auffindungsstelle lediglich Zuweisungen zu Variablen enthalten darf, die im zugehörigen Muster liegen. Diese Eigenschaft wird als *Lokalitätsbedingung* bezeichnet.

```
context Match
inv locality :
```

```
        self.contents→forAll(a | a.variable.parent = self.pattern)
```

Als *Eindeutigkeitsbedingung* gilt, dass jede Variable innerhalb einer Auffindungsstelle höchstens eine Zuweisung besitzt. Diese Einschränkung ist notwendig um die Semantik von Mustern und Bedingungen definieren zu können. Auf diese Weise können also keine mengenwertigen Belegungen ausgedrückt werden[1].

```
context Match
inv uniqueness :
        self.contents→collect(a | a.variable)→isUnique()
```

Die *Vollständigkeitsbedingung* einer Auffindungsstelle besagt, dass alle Variablen des zugehörigen Musters, welche durch mindestens eine Bedingung beschränkt werden, mittels einer Zuweisung an einen Wert gebunden werden. Zu beachten ist, dass hierdurch unbelegte Variablen als Teil einer Auffindungsstelle zugelassen werden, falls diese keine angeschlossene Bedingung besitzen. Dies ist zur Definition von Änderungsoperatoren (vgl. Abschnitt 4.4) notwendig.

```
context Match
inv completeness :
        self.pattern→collect(pe | pe.oclIsTypeOf(Variable))→
          oclAsType(Variable)→
          select(v1 | not v1.constraint.oclIsUndefined())→
          forAll(v2 | self.contents→exists(a | a.variable = v2))
```

Als letzte Wohlgeformtheitsbedingung einer Auffindungsstelle wird ihre *Korrektheit* spezifiziert. Diese lässt Zuweisungen nur zwischen *sortenkompatiblen* Variablen und Entitäten zu. Bspw. können zu Knotenvariablen ausschließlich Knoten zugewiesen werden. Dies wird für alle Variablentypen aus Abbildung 4.7 definiert.

```
context Assignment
def correct : Boolean =
        if variable.oclIsType(NodeVariable) then
          value.oclIsTypeOf(Node)
        else if variable.oclIsType(EdgeVariable) then
          value.oclIsTypeOf(Edge)
        else [...]
        endif endif
inv correctness : self.correct()

context Match
inv correctness :
        self.contents→forAll(a | a.correct())
```

[1]Die hierfür angebotenen Sprachkonstrukte sind Gegenstand von Abschnitt 4.3.

4.2.3 Semantische Definition

Die semantische Beschreibung der einfachen Mustersuche in DRAGULA erfolgt anhand der Funktionenfamilie SEM, die jeweils ein Objekt des syntaktischen Metamodells mit einer Menge zulässiger Interpretationen in Verbindung setzt. Die im vorangegangenen Unterabschnitt eingeführte Definition von *Auffindungsstellen* ersetzt dabei den in der Literatur häufig anzutreffenden *Morphismus* mittels dessen Muster- auf Arbeitsgraphen abgebildet werden. Diese Entscheidung liegt darin begründet, dass die syntaktische Struktur von DRAGULA keine kompakte Benennung einer hinreichend einschränkenden Funktion erlaubt. So könnte zwar die Menge der Variablen eines Musters mit den Elementen der Datenbasis in Relation gesetzt werden. Allerdings müsste hierzu die Semantik sämtlicher Bedingungen berücksichtigt und als Nebenbedingung in die Definition aufgenommen werden. Auch im Hinblick auf die angestrebte Erweiterbarkeit der Kernsprache ist dieser Ansatz nicht optimal, da hierzu eine Redefinition des Morphismus nötig wäre.

Suchmuster

Um die Beschreibung einfacher Mustersuchen von später erfolgenden Transformationen abzugrenzen, wird zunächst die Funktion $SEM_{\substack{Pattern \\ \textbf{search}}}$ wie folgt definiert:

$$SEM_{\substack{Pattern \\ \textbf{search}}} : Pattern \rightarrow LG(GS) \times \mathcal{P}(Match)$$

Somit weist die Funktion jedem Muster ein Tupel zu, welches eine Menge von Graphelementen der Datenbank sowie einer Menge von Auffindungsstellen umfasst. Als Graphelemente sind hierbei nur solche Strukturen zulässig, die gültige Instanzen des Graphschemas darstellen. Auffindungsstellen müssen gültig bezüglich obiger syntaktischer Definition sein. Unter diesen Voraussetzungen besagt die Semantik eines Musters, dass jede der Auffindungsstellen eine Zuweisung von Graphelementen zu Variablen darstellt, die den Bedingungen des Musters entsprechen. Hierzu darf keine Bedingung c des Musters verletzt sein, was durch den Ausdruck **false** in der Semantik des Musters dargestellt wird. Zusätzlich wird gefordert, dass die Menge vollständig ist, $SEM_{\substack{Pattern \\ \textbf{search}}}$ also alle zulässigen Auffindungsstellen ermittelt. Formal sei hierzu $G \in LG(GS)$ ein Laufzeitgraph, $M \in \mathcal{P}(Match)$ die zugehörige Menge von Auffindungsstelle des Musters $p \in Pattern$:

$$\begin{aligned}(G, M) \in SEM_{\textbf{search}}[\![p]\!] \quad \Leftrightarrow \quad & \forall m \forall c (m \in M \wedge c \in Constraint \cap elements(p) \rightarrow \\ & \nexists v_1 \ldots v_n \in Variable \cap elements(p) \\ & \wedge (m, v_1, \ldots, v_n, \textbf{false}) \in SEM[\![c]\!]) \\ \wedge \quad & \nexists M' \supsetneq M \wedge (G, M') \in SEM_{\textbf{search}}[\![p]\!]\end{aligned}$$

Variablen

Die semantische Funktion für Variablen definiert wie folgt deren Abbildung auf Tupel von Auffindungsstellen und *einzelnen* Graphelementen:

$$SEM_{Variable} : Variable \rightarrow Match \times \mathcal{U}$$

Diese Funktion wird derart definiert, dass ein Tupel bestehend aus einer Auffindungsstelle $m \in M$ und einem Graphelement $u \in U$ genau dann als Variablenbelegung gilt, wenn eine Zuweisung a von m dies entsprechend festlegt.

$$\begin{aligned}(m,u) \in SEM_{Variable}[\![v]\!] \quad &\Leftrightarrow \quad v \in elements\big(pattern(m)\big) \\ &\wedge \quad \exists a\big(a \in contents(m) \wedge variable(a) = v \wedge value(a) = u\big)\end{aligned}$$

Bedingungen

Die eigentliche Bedeutung einer Mustersuche wird in DRAGULA mittels Bedingungen festgelegt, welche gültige Variablenbelegungen einschränken. Das Ergebnis der hierzu gehörigen semantischen Funktion $SEM_{Constraint}$ wird dabei als Tupel definiert, welches die zu prüfende Auffindungsstelle, die zu beschränkenden Variablen v_i sowie einen Wahrheitswert einschließt.

$$SEM_{Constraint^{(n)}} : Constraint \dashrightarrow Match \times (Variable)^n \times \{\mathbf{true}, \mathbf{false}\}$$

$SEM_{Constraint}$ ist im Unterschied zu obigen Varianten aus zwei Gründen als *partielle* Funktion definiert: Erstens kann so eine *dreiwertige* Logik unterstützt werden, d.h. eine Bedingung kann als weder erfüllt noch verletzt betrachtet werden. Diese Eigenschaft ist im folgenden Abschnitt nützlich, da während der *komponierten Mustersuche* ggf. nicht zugewiesene Variablen bei der Auswertung berücksichtigt werden müssen. Zweitens kann so eine leichte Erweiterung der semantischen Definition erfolgen, wenn neue syntaktische Elemente als Spezialisierung der Klasse Constraint erstellt werden. Die hierzu gehörige semantische Auswertungsfunktion ist dann zunächst, in Übereinstimmung mit ihrer Signatur, undefiniert.

Die Nichterfülltheit einer Bedingung wird für alle Ausprägungen allgemein dadurch festgestellt, dass für alle Variablen eine Belegung zu ermitteln ist, dennoch aber keine Tupel mit positivem Wahrheitswert als Ergebnis der semantischen Funktion vorliegen. Für eine Auffindungsstelle m, eine beliebige Bedingung $c \in Constraint$, und deren zu beschränkenden Variablen $v_i \in \text{restricts}(c)$ gilt demnach:

$$\begin{aligned}(m, v_1, \ldots, v_n, \mathbf{false}) \in SEM[\![c]\!] \quad &\Leftrightarrow \quad \forall v_i \exists u((m,u) \in SEM[\![v_i]\!]) \\ &\wedge \nexists (m, v_1, \ldots, v_n, \mathbf{true}) \in SEM[\![c]\!]\end{aligned}$$

Die (positive) Erfülltheit einer Bedingung wird im Folgenden typspezifisch für die einzelnen Untertypen aus Abbildung 4.8 definiert.

Typisierung: Semantisch wird die Typisierungsprüfung dadurch formuliert, dass die Eigenschaft $\text{type}_{\mathcal{G}}(u)$ des einer Variable zugewiesenen Graphelements mit dem gewünschten Typ $\text{type}(tc)$ der Bedingung $tc \in \mathit{TypeConstraint}$ verglichen wird. Sind diese identisch, oder handelt es sich beim Typ des Elements um einen Untertyp des angefragten bei gesetztem closure-Attribut der Typbedingung, so ergibt der Ausdruck ein positives Ergebnis.

$$(m, v_1, \ldots, v_n, \mathbf{true}) \in SEM[\![tc]\!] \;\Leftrightarrow\; \forall v_i \exists u \big((m,u) \in SEM[\![v_i]\!]\big) \wedge \begin{cases} \text{type}_{\mathcal{G}}(u) = \text{type}(tc) & \text{falls } \neg \mathit{closure}(tc) \\ \text{type}_{\mathcal{G}}(u) \prec^*_{\mathcal{GS}} \text{type}(tc) & \text{falls } \mathit{closure}(tc) \end{cases}$$

Inzidenz: Ähnlich zu obigem Fall vergleicht die Inzidenzbedingung die Konnektierung von Graphelementen mit den entsprechend verknüpften Variablen des Musters. Sei $ic \in \mathit{IncidenceConstraint}$ eine solche Bedingung sowie $v_s \in \text{restricts}(ic, \mathit{source})$, $v_c \in \text{restricts}(ic, \mathit{connector})$, $v_t \in \text{restricts}(ic, \mathit{target})$ die adjazenten Variablen mit entsprechend beschrifteten Verbindungsobjekten.

$$\begin{aligned}(m, v_s, v_c, v_t, \mathbf{true}) \in SEM[\![ic]\!] \;\Leftrightarrow\; & \exists u_c \big((m,u_c) \in SEM[\![v_c]\!]\big) \\ & \wedge \Big(\exists u_s \big((m,u_s) \in SEM[\![v_s]\!]\big) \rightarrow \underset{\mathcal{G}}{\text{source}}(u_c) = u_s\Big) \\ & \wedge \Big(\exists u_t \big((m,u_t) \in SEM[\![v_t]\!]\big) \rightarrow \underset{\mathcal{G}}{\text{target}}(u_c) = u_t\Big)\end{aligned}$$

Enthaltensein: Auch Enthaltenseinsbeziehungen werden auf Basis eines Vergleichs der Graph- und Musterstrukturen erzielt. Voraussetzung für ein positives Prüfergebnis ist hier, dass die Enthaltenseinsbeziehung der Variablenbelegungen der durch die Bedingung geforderten Hierarchie *direkt* folgt – transitive Beziehungen sind also nicht zugelassen. Sei $cc \in \mathit{ContainmentConstraint}$, die Variable $v_{co} \in \text{restricts}(cc, \mathit{container})$ der zugeordnete Container, und $v_{cn_i} \in \text{restricts}(cc, \mathit{containee})$ die darin enthaltenen Variablen:

$$\begin{aligned}(m, v_{co}, v_{cn_1}, \ldots, v_{cn_n}, \mathbf{true}) \in SEM[\![cc]\!] \;\Leftrightarrow\; & \exists u_{co} \Big((m,u_{co}) \in SEM[\![v_{co}]\!] \\ & \wedge \forall v_{cn_i} \exists u_{cn_i} \big((m,u_{cn_i}) \in SEM[\![v_{cn_i}]\!] \\ & \quad \wedge u_{co} \triangleright_{\mathcal{G}} u_{cn_i}\big)\Big)\end{aligned}$$

Isomorphie: Isomorphiebedingungen greifen weder auf Eigenschaften des Laufzeitgraphen noch des Schemas zurück, sondern stellen lediglich die paarweise Verschiedenheit der ermittelten Variablenbelegungen fest. Im Folgenden sei hierzu $isoc \in IsomorphismConstraint$ und v_i die damit verbundenen Variablen:

$$(m, v_1, \ldots, v_n, \mathbf{true}) \in SEM[\![isoc]\!] \quad \Leftrightarrow \quad \exists u_i \big((m, u_i) \in SEM[\![v_i]\!] \\ \wedge u_1 \ldots u_n \text{ paarweise verschieden } ... \big)$$

Übergeordnete Graphen: Diese Bedingung prüft die Eigenschaft eines Graphen in keinem anderen enthalten zu sein, und damit letztlich auf die Nichtexistenz eines solchen. Falls angegeben wird auch der durch die Bedingung geforderte Graphname mit dem Namen des aufgefundenen übergeordneten Graphen verglichen. Sei $tlgc \in TopLevelGraphConstraint$, $v_i \in \text{restricts}(tlgc)$ die damit verbundenen Variablen:

$$(m, v_1, \ldots, v_n, \mathbf{true}) \in SEM[\![tlgc]\!] \quad \Leftrightarrow \quad \forall v_i \exists u_i ((m, u_i) \in SEM[\![v_i]\!] \\ \wedge \nexists (g \in \mathcal{G} \wedge g \triangleright_{\mathcal{G}} u_i) \\ \wedge (\text{tlgName}(tlgc) \in \{\bot, \text{name}_{\mathcal{G}}(u_i)\}))$$

Zwischenfazit: Bisher ist eine grundlegende Menge von Sprachkonstrukten eingeführt worden, mit denen einfache Graphanfragen in DRAGULA beschrieben werden können. Variablen stellen Platzhalter für Elemente des zugrundeliegenden Graphspeichers dar, die hierzu orthogonal von Bedingungen eingeschränkt werden. Muster aggregieren beide Elementarten zu einfachen Anfragen.
Die in DRAGULA erreichte *Trennung der Zuständigkeiten* zwischen Variablen und Bedingungen erlaubt die Unterstützung des komplexen DRAGOS Graphmodells ohne Redundanz in der Sprachdefinition oder der Realisierung einzuführen. Gleichzeitig können einzelne Sprachkonstrukte leicht spezialisiert und so um zusätzliche Funktionalität erweitert werden. Die entstehenden Spezifikationsdokumente sind im Vergleich zu gängigen Sprachen zwar deutlich umfangreicher, was Les- und Wartbarkeit einschränken kann. Durch die Ausrichtung DRAGULAs als *Kernsprache* und Realisierungsplattform wird ihr direkter Einsatz durch Endbenutzer jedoch nicht angestrebt.

4.3 Komponierte Mustersuche

Die bisher dargestellte Sprachstruktur bietet die Möglichkeit, Anfragen durch Variablen und an deren Werte geknüpfte Bedingungen zu spezifizieren. Bisher nicht mög-

lich ist die Kombination solcher Ausdrücke, insbesondere um *Alternativen* unter den geforderten Graphstrukturen zu formulieren. Auch die im Zuge der Anforderungsermittlung aus Abschnitt 3.4.1 genannten Sprachkonstrukte zur Negation oder der transitiven Mustersuche sind bislang nicht berücksichtigt worden. Dieser Abschnitt stellt daher die hierfür seitens DRAGULA angebotenen Strukturen vor, die auf der *Schachtelung* von Mustern basieren.

4.3.1 Geschachtelte Muster

Als Grundlage komplexer Kompositionsstrukturen stellt dieser Abschnitt zunächst *geschachtelte Muster* vor. Ein stark vereinfachtes Anwendungsbeispiel hierzu zeigt Abbildung 4.10[2]. Im Unterschied zu Abbildung 4.4 auf Seite 88 werden hierbei zwei Variablen in einem *geschachtelten Muster* verkapselt.

Die Mustersuche ermittelt im Beispiel der Abbildung 4.10 zunächst Auffindungsstellen des *äußeren Musters*. Da dies keinerlei Einschränkungen definiert werden demnach fünf verschiedene Belegungen der Knotenvariable ermittelt. Für das innere Muster werden nur solche Variablenbelegungen ermittelt, die zusammen mit einer jeweiligen Auffindungsstelle des äußeren Musters zu keinen verletzten Bedingungen führen. Im vorliegenden Fall werden für das innere Muster diejenigen Entitäten ausgewählt, die mit den Variablenbelegungen des äußeren Musters verbunden sind. Im ersten Fall existiert nur eine zu dieser Belegung inzidente Kante, weswegen die geschachtelte Auffindungsstelle eindeutig bestimmt ist. Im zweiten Fall werden zwei innere Auffindungsstellen ermittelt. Die weiteren Auffindungsstellen des äußeren Musters besitzen keine inzidenten auslaufenden Kanten, weshalb keine Auffindungsstellen des inneren Musters ermittelt werden können.

Ein weiteres Beispiel für geschachtelte Muster zeigt Abbildung 4.11. An diesem Beispiel wird verdeutlicht, dass die Suche nach Auffindungsstellen zwischen mehreren geschachtelten Mustern unabhängig voneinander abläuft. Die dargestellte Musterstruktur besagt dabei intuitiv, dass aufgrund der Isomorphiebedingung paarweise verschiedene Belegungen der drei Variablen ermittelt werden sollen. Tatsächlich werden bei der Suche nach Auffindungsstellen der geschachtelten Muster lediglich Belegungen des *äußeren* Musters, nicht jedoch anderer Muster auf gleicher Schachtelungsebene berücksichtigt. Die entstehenden Auffindungsstellen stellen daher lediglich paarweise verschiedene Belegungen der äußeren Variable sowie *einer* inneren Variable dar. Die dargestellte Isomorphiebedingung wird dabei nicht verletzt: Während der Mustersuche des äußeren Musters sind die inneren Variablen unbelegt und

[2]Im Weiteren wird stets die verdichtete Darstellung von DRAGULA-Spezifikationen eingesetzt. Zusätzlich werden Bezeichner abgekürzt falls die Bedeutung eindeutig ist.

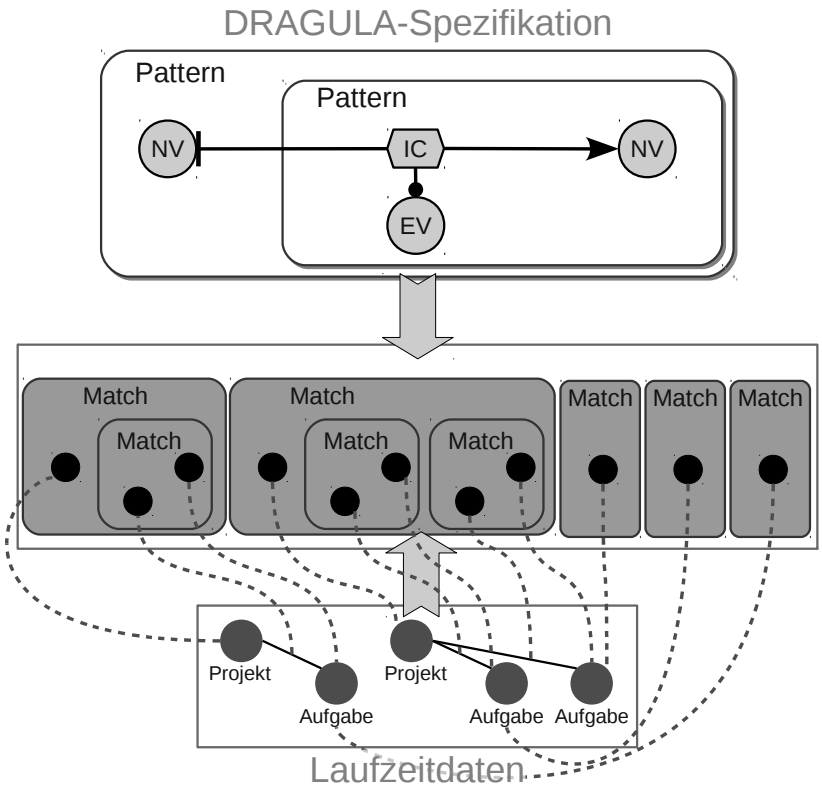


Abbildung 4.10: Anwendungsbeispiel für komponierte Muster

führen daher zu keiner Verletzung der Isomorphiebedingung – was der Definition gültiger Auffindungsstellen aus Abschnitt 4.2.3 genügt. Bezüglich der geschachtelten Muster sind lediglich deren eigene Belegungen sowie die der übergeordneten Auffindungsstellen zu berücksichtigen.

Syntaktische Definition

Eine syntaktische Unterstützung geschachtelter Muster wird in Abbildung 4.12 auf einfache Weise dadurch erreicht, dass die Klasse Pattern zur Spezialisierung von PatternElement deklariert wird. Damit ist analog zum *Composite*-Entwurfsmuster [GHJV94] eine strukturelle Schachtelung entsprechender Objekte möglich. Analog hierzu wird die Definition von Auffindungsstellen durch eine reflexive Assoziation ergänzt, so dass auch diese geschachtelt strukturiert werden können.

Zur Ergänzung der kontextsensitiven Syntax wird gefordert, dass die Enthaltenseinsbeziehung auf Mustern azyklisch ist, diese sich also nicht selbst enthalten können. Erweitert wird ferner die Wohlgeformtheitsbedingung hierarchy der Assoziationsklasse Restricts. So gilt in der redefinierten Form, dass die jeweiligen Muster der

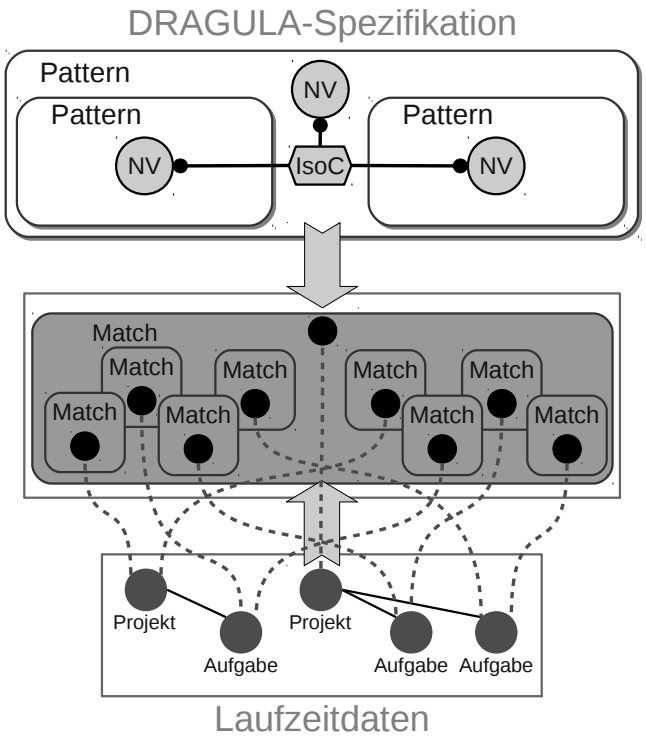


Abbildung 4.11: Unabhängigkeit von Mustern auf gleicher Ebene

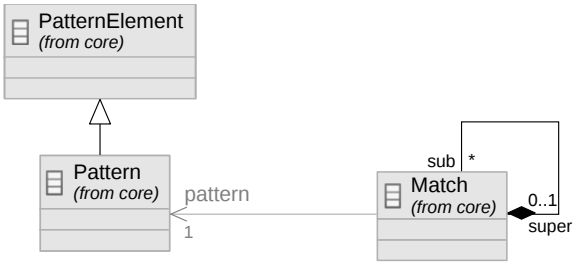


Abbildung 4.12: Metamodell für geschachtelte Muster und Auffindungsstellen

verbundenen Variablen und Bedingungen in einer Enthaltenseinshierarchie zueinander stehen. Es wird also nicht mehr gefordert das Variablen und Bedingungen stets in einem gemeinsamen Muster liegen. Bezüglich geschachtelter Auffindungsstellen gilt die Einschränkung, dass deren Hierarchie stets der Hierarchie ihrer zugehörigen Muster folgt.

```
context Pattern
def allElements : Set(PatternElement) = self.elements→
      union(self.elements→collect(e | e.allElements))
```

```
inv acyclicelements : not self.allElements→contains(self)

context Restricts
inv hierarchy : self.constraints.parent.allElements→
      contains(variable) or
      self.variables.parent.allElements→contains(variable)

context Match
inv hierarchy : self.super.pattern = self.pattern.parent
```

Semantische Definition

Die Ermittlung geschachtelter Auffindungsstellen ist stets an deren jeweilige übergeordnete Auffindungsstelle gebunden. So erfolgt eine Mustersuche für geschachtelte Strukturen nur dann, wenn für deren übergeordnetes Muster zumindest eine Auffindungsstelle ermittelt werden kann. Ist dies der Fall, so erfolgt eine Mustersuche für alle enthaltenen Muster in nicht festgelegter Reihenfolge. Eine Besonderheit stellen in dieser Hinsicht Muster dar die *keine Variablen* enthalten. Falls die lokalen Bedingungen bezüglich übergeordneter Variablen erfüllt sind, wird genau eine Auffindungsstelle für ein solches Muster gefunden.

Bezüglich der Semantik von Bedingungen gilt folgende weitere Eigenschaft: Für die Bestimmung einer Auffindungsstelle eines Musters dürfen

- keine Bedingungen des Musters verletzt werden.
- keine Bedingungen eines übergeordneten Musters verletzt werden.

In beiden Fällen wird die Verletztheit von Bedingungen bezüglich der Variablenbelegung des zu prüfenden Musters sowie der aller transitiv übergeordneten Muster bestimmt. Die Formalisierung dieser Eigenschaften erfolgt unter Rückgriff auf die Ergebnisse des vorherigen Abschnitts. Zunächst ist die semantische Funktion für *Variablen* geeignet zu erweitern, um eine *Sichtbarkeitsregel* für übergeordnete Muster aufzustellen. Hierbei wird die Semantik von Variablen in hierarchischen Mustern, $SEM_{\substack{Variable \\ hrcl}} \rightarrow Match \times \mathcal{U}$, wie oben definiert, allerdings um eine Erweiterung ergänzt. Diese besagt, dass auch übergeordnete Auffindungsstellen zu einer gültigen Variablenbelegung beitragen können.

$$\begin{aligned}(m,u) \in SEM_{hrcl}[\![v]\!] \quad &\Leftrightarrow \quad (m,u) \in SEM[\![v]\!] \\ &\vee \quad \exists m_1 \ldots m_n \big(m \in \mathrm{sub}(m_1) \wedge \ldots \wedge m_{n-1} \in \mathrm{sub}(m_n) \\ &\quad \wedge (m_n,u) \in SEM[\![v]\!]\big)\end{aligned}$$

Als zweite Ergänzung ist die Berücksichtigung „nicht verarbeiteter“ Bedingungen nötig, also solchen, die Teil eines Musters sind, jedoch keine Variablen dieses Musters einschränken. Ein häufig auftretender Anwendungsfall besteht darin, dass Bedingungen ausschließlich Elemente tiefer geschachtelter Muster referenzieren. Wie oben sei $SEM_{\substack{hrcl \\ \textbf{search}}}[\![p]\!] : Pattern \to LS(GS) \times \mathcal{P}(Match)$. Ein Tupel bestehend aus einem (Teil-)Graphen und einer Menge von Auffindungsstellen ist genau dann in der Relation SEM enthalten, wenn folgendes gilt: (G, M) bilden sowohl eine gültige Belegung für das Muster p, als auch, falls vorhanden, für das übergeordnete Muster p'.

$$\begin{aligned}(G,M) \in SEM_{\substack{hrcl \\ \textbf{search}}}[\![p]\!] \quad &\Leftrightarrow \quad (G,M) \in SEM_{\textbf{search}}[\![p]\!] \\ &\wedge \quad \forall p' \big(p \in \text{elements}(p') \to (G,M) \in SEM_{\substack{hrcl \\ \textbf{search}}}[\![p']\!]\big)\end{aligned}$$

4.3.2 Gültigkeit von Auffindungsstellen

Im bisherigen Verlauf dieses Abschnitts ist die einfache Mustersuche um geschachtelte Konstrukte erweitert worden. Zwar können hierdurch Anfragen strukturiert werden, dennoch fehlen weitere Auswertungsmöglichkeiten der ermittelten Auffindungsstellen. Die eingangs erwünschte boolesche Verknüpfung von Teilmustern ist auf diese Weise noch nicht möglich.

Im Weiteren werden geschachtelte Muster um *Annotationen* und in diesem Zusammenhang um *Musterbedingungen* erweitert, mittels derer eine flexible Kombination unterschiedlicher Teilmuster ermöglicht wird. Hierzu zählen einerseits *logische Ausdrücke*, wie bspw. alternative Strukturen. Auch möglich ist damit die Zusicherung einer bestimmten *Anzahl von Auffindungsstellen*, wodurch obligatorische, optionale, und negierte Teilausdrücke abgedeckt werden können. Zur Vorbereitung dessen führt der aktuelle Abschnitt zunächst eine verallgemeinerte Grundlage zur *Gültigkeitsprüfung* von Auffindungsstellen ein.

Eine Auffindungsstelle soll im Folgenden als *gültig* erachtet werden, wenn sie den ihrem Muster mittels Annotationen zugeordneten Musterbedingungen genügt. Sollte die Auswertung dieser Bedingungen bezüglich einer Auffindungsstelle das Ergebnis *ungültig* ergeben, so gilt diese Stelle als nicht existent. Es ist daher unerheblich ob für ein gegebenes Musters keine Auffindungsstelle *gefunden* werden kann, oder ob ausschließlich *ungültige* Stellen existieren. Zusätzlich kann die Mustersuche eines Teilmusters in Bezug auf die Auffindungsstelle eines übergeordneten Musters insgesamt als gültig oder ungültig bewertet werden.

Formal wird die Gültigkeit einer Auffindungsstelle als semantische Funktion definiert: $SEM_{\substack{valid \\ \textbf{search}}} : Pattern \to LG(GS) \times \mathcal{P}(Match)$ Hierzu werden zwei Hilfsfunktionen eingesetzt, die im folgenden Abschnitt spezifisch für einzelne Musterannotationen

ausgestaltet werden.

- Die Funktion $valid_{Match} : \mathcal{P}(Match) \rightarrow \mathcal{P}(Match)$ ermittelt aus einer gegebenen Menge von Auffindungsstellen die maximale Teilmenge *gültiger* Stellen. $valid_{Match}$ fungiert daher als *Filterfunktion*.
- Die Funktion $valid_{Pattern} : Pattern \times Match \rightarrow \{\mathbf{true}, \mathbf{false}\}$ wertet die Mustersuche für das Muster p und eine Auffindungsstelle m seines *enthaltenden* Musters aus. Hierzu werden alle in m enthaltenen und gemäß $valid_{Match}$ gültigen Auffindungsstellen von p herangezogen.

Mit Hilfe obiger noch zu beschreibenden Funktionen kann $SEM_{\substack{valid \\ \mathbf{search}}}$ auf folgende Weise definiert werden:

$$\begin{aligned}(G, M) \in SEM_{\substack{valid \\ \mathbf{search}}}[\![p]\!] \quad \Leftrightarrow \quad & \exists M'\big((G, M') \in SEM_{\substack{hrcl \\ \mathbf{search}}}[\![p]\!] \wedge M = valid_{Match}(M')\big) \\ \wedge \quad & \forall p', m\big(p' \in elements(p) \wedge m \in M \rightarrow valid_{Pattern}(p', m)\big)\end{aligned}$$

Eine Menge von Auffindungsstellen M ist für einen gegebenen Laufzeitgraphen G und ein Suchmuster p demzufolge genau dann gültig wenn M die Teilmenge gültiger Auffindungsstellen der ungeprüften Mustersuche $SEM_{\substack{hrcl \\ \mathbf{search}}}[\![p]\!]$ darstellt und jede dieser Auffindungsstellen $m \in M$ gültig bezüglich aller in p enthaltenen Untermuster ist. Der zweite Teil dieser Definition fordert insbesondere nicht zwangsläufig das Vorhandensein von Auffindungsstellen für die Untermuster, sondern lediglich eine Gültigkeit gemäß $valid_{Pattern}$. Es können somit auch Untermuster ohne Auffindungsstellen ein gültiges Ergebnis ergeben.

Ausgehend von dieser allgemeinen Grundlage führt der folgende Abschnitt verschiedene *Musterannotationen* ein, mit deren Hilfe die Bedeutung geschachtelter Musterstrukturen festgelegt werden können. Die jeweilige semantische Definition wird dabei auf dem Begriff der Gültigkeit von Auffindungsstellen aufgesetzt. Um die komponierte Mustersuche auf *gültige* Auffindungsstellen einzuschränken, nehmen die folgenden Abschnitte Bezug auf $SEM_{\substack{valid \\ \mathbf{search}}}$ anstatt der bisher eingeführten, nicht prüfenden Funktion $SEM_{\substack{hrcl \\ \mathbf{search}}}$.

4.3.3 Musterannotationen

Basierend auf der bereits erläuterten Möglichkeit Suchmuster geschachtelt zu strukturieren, werden an dieser Stelle sogen. *Annotationen* definiert, mit denen die Bedeutung der Schachtelungsstruktur präzisiert werden kann. Die hierzu vorgenommene Erweiterung des DRAGULA Metamodells zeigt Abbildung 4.13. Annotationen werden syntaktisch an den betreffenden Mustern eingetragen, wie die Kompositionsbeziehung

annotations definiert. Die Abbildung zeigt drei hauptsächliche Arten von Annotationen, die im Weiteren erläutert werden.

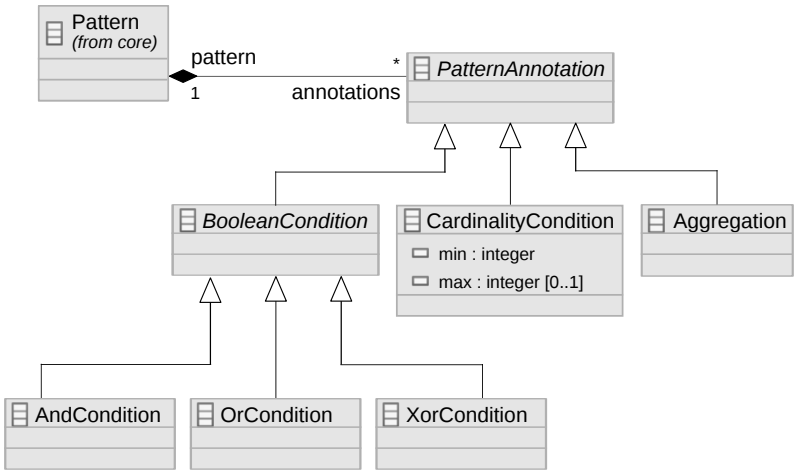


Abbildung 4.13: Annotationen für Muster

Logische Musterbedingungen: Diese Gruppe beeinflusst das Verhalten der Mustersuche bezüglich der in einem Muster *enthaltenen Untermuster*. So wird durch die logische `und`-Bedingung (AndCondition) festgelegt, dass für derart annotierte Muster nur solche Auffindungsstellen ermittelt werden, die ihrerseits Auffindungsstellen *für alle Untermuster* enthalten. Analog hierzu werden Annotationen für ein- und ausschließende `oder`-Bedingungen (OrCondition bzw. XorCondition) bereitgestellt.

Ein Beispiel für logische Musterbedingungen zeigt Abbildung 4.14. Das hier dargestellte Muster ermittelt Tupel aus Projekten, Personen sowie Verbindungen des Typs arbeitetAn. Hierbei besteht die Alternative, dass eine Person direkt an der jeweiligen Aufgabe, oder an einer Unteraufgabe arbeitet. Mögliche Auffindungsstellen dieser Musterstruktur sind im unteren Teil der Abbildung dargestellt. Wie angedeutet können je nach Laufzeitdaten unterschiedliche geschachtelte Auffindungsstellen für die jeweiligen Muster aufgefunden werden. Die Gültigkeit mindestens einer dieser Stellen vorausgesetzt, gilt die übergeordnete Auffindungsstelle gemäß der angegebenen *Logisch-Oder* Verknüpfung. Dies ist lediglich im ersten Beispiel *nicht* der Fall, obwohl die Variablenbelegung des äußeren Musters eine gültige Auffindungsstelle darstellt. Im Übrigen kann im dargestellten Fall keine Auffindungsstelle beide Untermuster zugleich erfüllen, da die zu ermittelnde Kante hierzu zwei verschiedene Quellen aufgrund der angeschlossenen Variablen besitzen müsste[3].

[3]Der zwar technisch, nicht jedoch fachlich zulässige Fall einer reflexiven Teilaufgabenbeziehung ist hierbei nicht berücksichtigt.

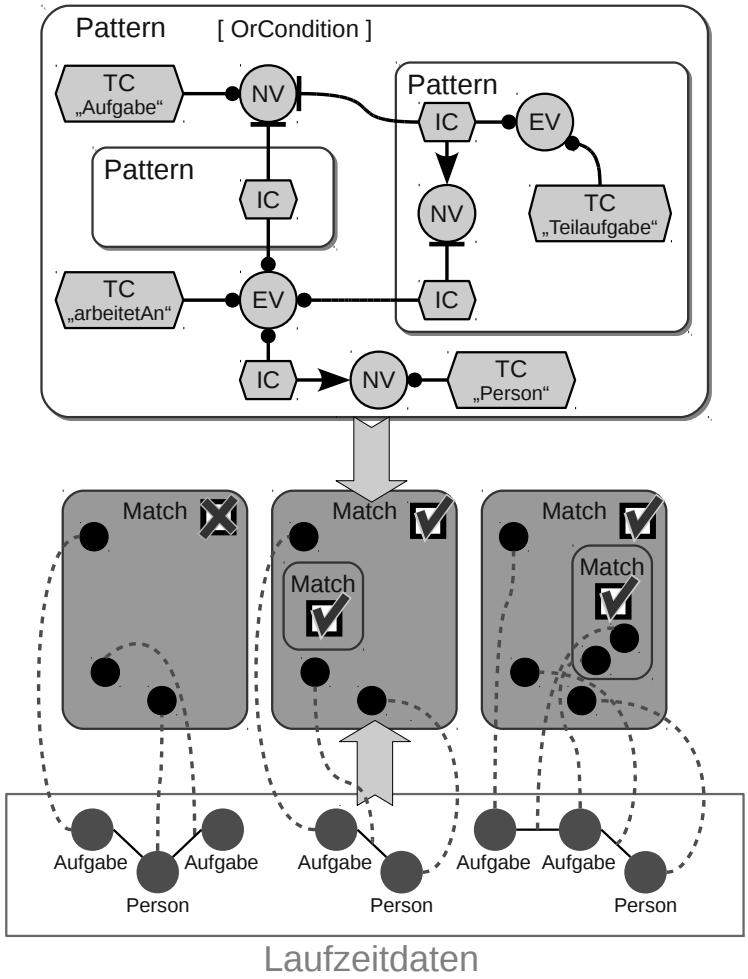


Abbildung 4.14: Anwendungsbeispiel für logische Musterbedingung

Die kontextsensitive Syntax für logische Musterbedingungen besagt, dass jeder Einsatz einer BooleanCondition weitere solche Bedingungen innerhalb eines Musters ausschließt.

```
context Pattern
inv exclusiveBooleanCondition : self.annotations→
      select(a | a.oclIsKindOf(BooleanCondition)→size()≤1)
```

Die semantische Definition logischer Musterbedingungen basiert auf der Funktion $valid_{Match}$, also der Gültigkeitsbestimmung für einzelne Auffindungsstellen. Seien hierzu $p = pattern(m)$ für eine Auffindungsstelle $m \in M$ eindeutig bestimmt sowie $\overline{p_1}, \ldots, \overline{p_n}$ eine Menge von Untermustern mit $\overline{p_i} \in elements(p) \cap Pattern$. Ferner sei $a \in annotations(pattern(m)) \cap BooleanCondition$ eine logische Musterbedingung,

welche aufgrund obiger Wohlgeformtheitsbedingung entweder nicht vorhanden, oder ebenfalls eindeutig bestimmt ist. Der folgende Ausdruck besagt, dass eine *logisch-Und* Bedingung genau dann erfüllt ist, falls für jede Auffindungsstelle in M *alle* Untermuster gültig sind im Sinne der Funktion $valid_{Pattern}$.

$$\begin{aligned} M = valid_{Match}(M') \quad \Leftrightarrow \quad & M \subseteq M' \wedge \Big(a \in AndCondition \rightarrow \\ & \forall m \in M\big(valid_{Pattern}(m, \overline{p_1}) \\ & \quad \wedge \quad \ldots \\ & \quad \wedge \quad valid_{Pattern}(m, \overline{p_n})\big)\Big) \end{aligned}$$

Die übrigen Bedingungstypen für *logisch-Oder* Verknüpfungen werden analog definiert. Für die nicht ausschließende Oder-Bedingung werden die Teilausdrücke

$$valid_{Pattern}(m, \overline{p_1}), \ldots, valid_{Pattern}(m, \overline{p_1 n})$$

per Disjunktion verknüpft, bei ausschließender Oder-Bedingung wird die Gültigkeit eines Untermusters bei gleichzeitiger Ungültigkeit aller anderen Untermuster geprüft.

Bedingung der Auffindungsstellenanzahl: Die Bedingung der *Auffindungsstellenanzahl*, genauer eines Gültigkeitsbereichs dieser Anzahl, wird mittels einer CardinalityCondition definiert. Diese besagt, dass zu einem gegebenen Muster mindestens min und, falls definiert, höchstens max Auffindungsstellen gefunden werden dürfen. Ist diese Kardinalitätsbedingung bezüglich der Auffindungsstelle des übergeordneten Musters verletzt, so werden alle gefundenen Auffindungsstellen als *ungültig* erachtet und nicht weiter verarbeitet. Zugleich genügt die Erfüllung der Kardinalitätsbedingung um im Sinne logischer Musterbedingungen ein positives Ergebnis zu erhalten.

Eine CardinalityCondition wird häufig für zweierlei Aufgaben eingesetzt: Durch die Angabe einer min-Kardinalität 0 wird das Nicht-Auffinden eines Untermusters als zulässiges Ergebnis gewertet – dies entspricht also einem *optionalen* Untermuster. Bei Angabe einer max-Kardinalität 0 führt das Auffinden eines Musters zu einem negativen Ergebnis bezüglich der logischen Musterbedingung des übergeordneten Musters – dies entspricht einer negativen Anwendbarkeitsbedingung.

Da diese Art der Bedingung nur im Bezug auf eine übergeordnete Auffindungsstelle gelten kann, fordert die kontextsensitive Syntax die Existenz eines entsprechend übergeordneten Musters. Ferner gilt auch hier die Maßgabe der Eindeutigkeit.

```
context Pattern
inv exclusiveCardinalityCondition : self.annotations→
      select(a | a.oclIsTypeOf(CardinalityCondition) )→size()≤1
```

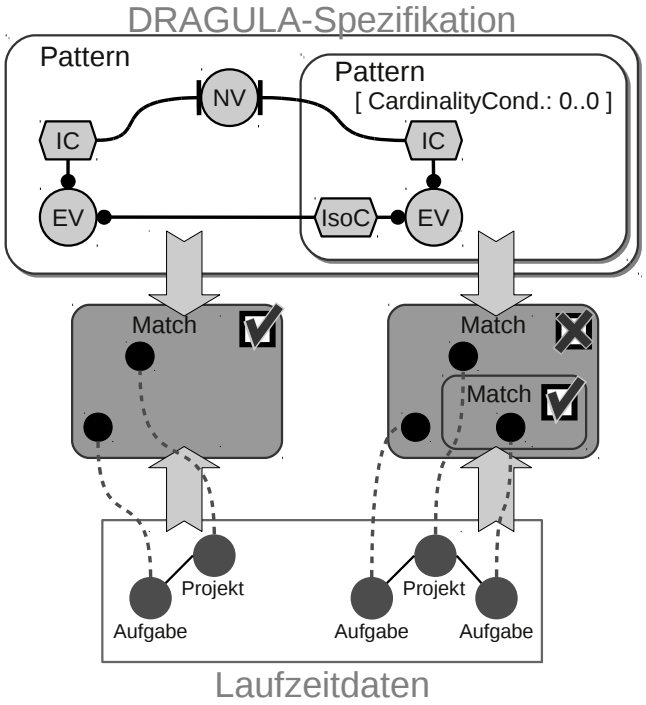


Abbildung 4.15: Anwendungsbeispiel für bedingte Auffindungsstellenanzahl

```
inv cardinalityConditionExPattern : self.annotations→
      exists(a | a.oclIsTypeOf(CardinalityCondition)) implies
      not self.pattern.parent.oclIsUndefined()

context CardinalityCondition
inv wellformedBounds : not self.max.oclIsUndefined() implies
      self.max≥self.min
```

Semantisch festgelegt werden Bedingungen der Auffindungsstellenzahl mittels der Funktion $valid_{Pattern}$, welche die Gültigkeit eines *Musters* im Bezug auf alle ermittelbaren Auffindungsstellen beschreibt. Hierzu sei $p \in Pattern$ ein Untermuster, für dessen übergeordnetes Muster eine Auffindungsstelle m gegeben ist. Die Bedingung besagt diesbezüglich, dass die *Anzahl* der für p in m auffindbaren Auffindungsstellen innerhalb eines vorgegebenen Intervalls zulässig sein soll.

$$valid_{Pattern}(p,m) \quad = \quad \begin{cases} \textbf{true} & \text{falls } \forall a \Big(a \in CardinalityConstraint \rightarrow \\ & \exists M, G \big((M,G) \in SEM_{\substack{valid \\ search}} [\![p]\!] \wedge M \subseteq \text{sub}(m) \big) \\ & \wedge \quad min(a) \leq |M| \leq max(a) \Big) \\ \textbf{false} & \text{sonst} \end{cases}$$

Mengenwertige Verarbeitung: Die Annotation Aggregation legt fest, dass *alle gültigen* Auffindungsstellen dieses Musters zusammengefasst und gemeinsam verarbeitet werden. Dies wird im Weiteren bspw. zur Berechnung von aggregierten Attributwerten eingesetzt, wie in Abschnitt 4.6 erläutert werden wird. Auch möglich ist hiermit die simultane Transformation von Graphstrukturen bzgl. *aller* ermittelten Auffindungsstellen.

Die Aggregation Annotation ist verträglich mit o.g. Annotationen, so dass keine weitere Einschränkung durch die kontextsensitive Syntax erfolgt. Einzig die Verwendung mit Kardinalitätsbedingungen der Obergrenze max=0 führt zu keinen sinnvollen Ergebnissen, stellt jedoch keine Verletzung der korrekten Anwendung dieses Sprachkonstrukts dar. Mengenwertige Teilausdrücke werden an dieser Stelle nicht formal definiert, stattdessen erfolgt dies bei der Verarbeitung der ermittelten Strukturen.

Standardverhalten: Um DRAGULA-Spezifikationen nicht durch wiederkehrende Konstrukte unnötig zu überfüllen, werden für obige Annotationen Standardverhaltensregeln vereinbart. Bezüglich der Musterbedingungen gilt – falls nicht explizit angegeben – die AndCondition mit der Forderung, dass alle Untermuster erfüllt sein müssen. Die Auffindungsstellenanzahl wird standardgemäß mit der CardinalityCondition in der Belegung min=1 sowie max undefiniert behandelt. Ferner erfolgt standardgemäß *keine* mengenwertige Auswertung.

4.3.4 Musterreferenzen und rekursive Muster

Bislang sind in der Sprachbeschreibung von DRAGULA lediglich statisch strukturierte Muster behandelt worden. Diese können zwar, wie bisher in diesem Abschnitt gezeigt, auch durch Schachtelung zu logischen Ausdrücken kombiniert werden. Dennoch sind die so darstellbaren Anfragen stets in ihrer Größe, etwa bezüglich der Anzahl der zu belegenden Variablen, beschränkt. Nicht modellierbar sind daher bspw. *Pfadausdrücke*, die i.d.R. einen transitiven Abschluss der zu untersuchenden Teilausdrücke erlauben. Um dies mittels DRAGULA auszudrücken wäre daher eine vorab nicht begrenzte Anzahl an Variablen und Bedingungen nötig. Ein zusätzliches Problem entsteht durch die Beschränkung auf statische Musterstrukturen insoweit, als dass keinerlei *Wiederverwendung* zwischen Mustergrenzen möglich ist. Hierdurch entsteht die übliche softwaretechnische Problematik der erschwerten Wartbarkeit und gesteigerten Fehleranfälligkeit einer Spezifikation.

Als Lösung für diese Einschränkung wird in DRAGULA das Konzept der *Musterreferenzen* eingeführt, mittels dessen die starre Abgrenzung zwischen diesen syntaktischen Elementen aufgehoben werden kann. Konzeptionell stellt eine Musterreferenz

einen Verweis auf ein beliebiges Muster dar, dessen enthaltene Elemente als *Bestandteil* des referenzierenden Musters aufgefasst werden sollen. Hierbei werden im Weiteren auch rekursive Referenzen zugelassen. Zur sinnvollen Anwendung von Musterreferenzen müssen die referenzierten Strukturen mit dem referenzierenden Muster in Beziehung gesetzt, also eine *Einbettung* hergestellt werden. Dies erfolgt mittels einer durch den Entwickler explizit zu spezifizierenden Abbildungsbeziehung zwischen den jeweiligen Elementen.

Anwendungsbeispiel: Wiederverwendung

Abbildung 4.16 zeigt auf der linken Seite ein einleitendes Anwendungsbeispiel zum Einsatz von Musterreferenzen. Betrachtet wird hier zunächst der Anwendungsfall der *Wiederverwendung*, also der Einbeziehung vordefinierter Teilstrukturen in ein auszuwertendes Muster. Ziel der dargestellten Struktur ist es, zwei parallele Kanten des Typs arbeitetAn mit gemeinsamer Quelle zu ermitteln. Hierzu wird ein Teilmuster zur Kantennavigation *zweimal* referenziert, und mit der Variable des Startknotens verknüpft. Zusätzlich wird eine Isomorphiebedingung eingesetzt um die Verschiedenheit der gefundenen Kanten sicherzustellen.

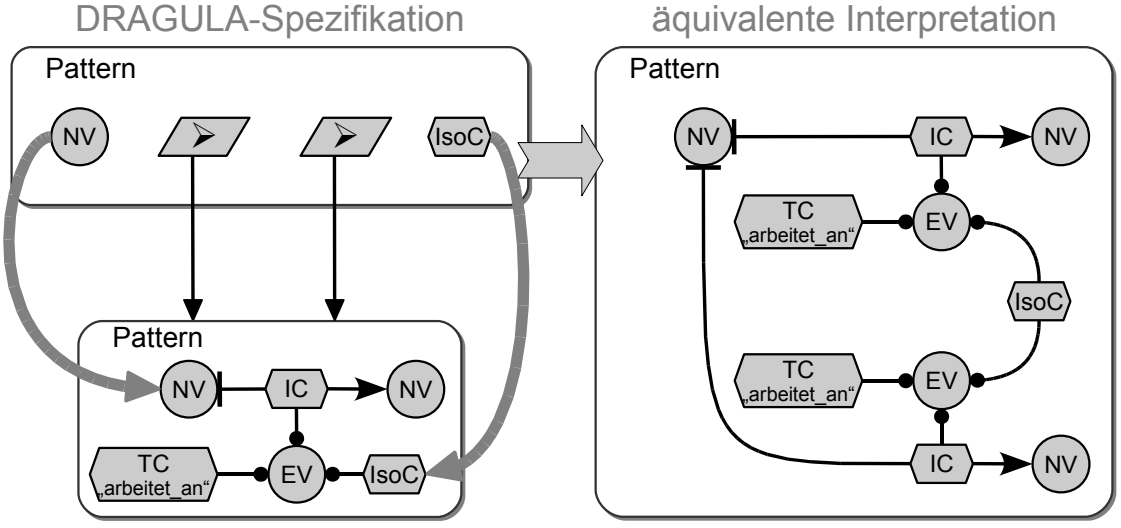


Abbildung 4.16: Anwendungsbeispiel für Musterreferenzen / Interpretation

Die rechte Seite stellt eine äquivalente Fassung dieser Struktur ohne Musterreferenzen dar – bei der Auswertung beider Muster werden also identische Auffindungsstellen gefunden. Die Verbindungsprüfung, bestehend aus Bedingungen und Variablen für die zu ermittelnden Ziel- und Verbindungselemente, tritt hierbei zweifach

auf. Durch Angabe der als graue Pfeile dargestellten *Abbildungen* zwischen Musterelementen werden der Startknoten sowie die Isomorphiebedingung *nicht* verdoppelt. Stattdessen greifen beide Teilmuster auf diese Elemente gemeinsam zu.

Informelle Interpretation

Die bislang informelle Erläuterung von Musterreferenzen lässt zwei unterschiedliche Interpretationen bzw. Realisierungsvarianten zu: Einerseits kann ein referenziertes Muster ähnlich einem *Unterprogrammaufruf* getrennt vom aufrufenden Muster verarbeitet werden. Ein solcher Aufruf erfolgt nach Belegung aller lokalen Variablen des referenzierenden Musters. Hierbei werden Werte von Variablen, die gemäß der angegebenen Abbildungsrelationen an Variablen des referenzierten Musters gebunden sind, als „Aktualparameter“ übergeben. Nach Abarbeitung des Aufrufs werden die ermittelten Auffindungsstellen in die des referenzierenden Musters zurück übertragen. Bei Ermittlung einer *einzelnen Auffindungsstelle* erfordert dies lediglich eine Erweiterung der Auffindungsstelle des Aufrufers um zusätzliche Zuweisungen. Andernfalls wird die Auffindungsstelle des referenzierenden Musters pro Auffindungsstelle des referenzierten *vervielfältigt* und entsprechend um Zuweisungen ergänzt. Abschließend ist sicherzustellen das Bedingungen des aufrufenden Musters, die per Abbildungsrelation mit dem aufgerufenen Muster in Beziehung gesetzt sind, nicht verletzt werden. In obigem Beispiel kann die Isomorphiebedingung erst mit Ermittlung der Auffindungsstellen der referenzierten Muster geprüft werden.

Andererseits kann eine explizite Strukturkopie des Inhalts eines referenzierten Musters erstellt und im Aufrufer abgelegt werden, wie bereits in Abbildung 4.16 graphisch dargestellt wird. Durchgeführt würde diese Expansion *zur Laufzeit* der Mustersuche und unter der Voraussetzung, dass eine Belegung der Variablen des enthaltenden Musters gefunden werden kann. Treten mehrere Musterreferenzen auf, so werden diese in nicht festgelegter Reihenfolge verarbeitet.

Die zweite Variante hat den Vorteil einer leichteren Verständlichkeit, da die Bedeutung von Musterreferenzen, mit Ausnahme des Kopierprozesses, auf bereits behandelte Sprachkonstrukte von DRAGULA zurückgeführt wird. Eine Implementierung dieser Variante wäre dagegen deutlich ineffizienter, da eine wiederholte Veränderung der verarbeiteten Musterstruktur, einschließlich komplexer Kopieroperationen, erfolgt. Aufgrund der direkten Verständlichkeit dieser Variante sowie der Möglichkeit einer vereinfachten formalen Definition, wird im Folgenden stets die Strukturkopie als Interpretation einer Musterreferenz verwendet. Die in Kapitel 5 behandelte Realisierung setzt dagegen aus Effizienzgründen erstere Variante des Unterprogrammaufrufs ein.

Syntaktische Definition

Die für Musterreferenzen nötige Ergänzung des DRAGULA Metamodells ist in Abbildung 4.17 dargestellt. Das hinzugefügte Element für PatternReference referenziert ein Muster unter Angabe einer Menge von Elementabbildungen (ElementMapping). Diese legen fest, wie das zu übertragende Muster in die Umgebung des Zielmusters eingebunden wird. In der visuellen Notation des obigen Beispiels wird diese Abbildung als Pfeil vom kopierten Element in Richtung des entsprechenden Originalelements dargestellt. Falls keine explizite Angabe erfolgt, sind alle dargestellten Elementabbildungen allen Musterreferenzen gleichermaßen zugeordnet.

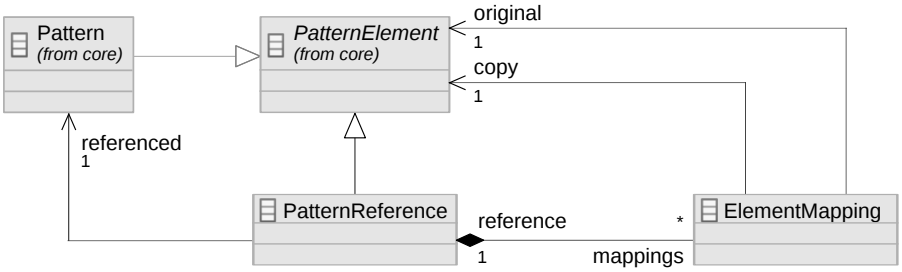


Abbildung 4.17: Metamodell für Musterreferenzen

Als kontextsensitive Einschränkung von Musterreferenzen wird die Gestalt der Elementabbildungen konkretisiert. Gefordert wird durch uniqueMap, dass die in einer einzelnen PatternReference enthaltenen Abbildungen die jeweiligen Quellelemente eindeutig referenzieren. Nicht zulässig ist daher, dass mehrere Originalobjekte einer gemeinsamen Kopie zugeordnet werden. Je nach Interpretation würde dies verschiedene fehlerhafte Situationen bewirken: Im Fall des Unterprogrammaufrufs käme dies einer mehrdeutigen Belegung eines Aktualparameters gleich, wohingegen im Fall der Strukturkopie ein kopiertes Element in zwei getrennten Kontexten zugleich verwendet würde.

```
context PatternReference
inv uniqueMap : self.mappings→collect(m | m.original)→isUnique()
```

Graphisch werden zulässige Elementabbildungen sowie deren Verarbeitung anhand von Abbildung 4.18 demonstriert. Der erste Fall zeigt eine quell- und zielseitig eindeutige Abbildung wie im einleitenden Beispiel, die eine isomorphe Einbettung des referenzierten Musters bewirkt. Im zweiten Beispiel besitzt die quellseitige Variable zwei Abbildungen auf Variablen des referenzierten Musters. Hierdurch werden alle zielseitigen Verbindungen auf diese Variable vereinigt, wodurch im Beispiel eine reflexive Verbindungsprüfung entsteht. Der umgekehrte Fall, ein Element des Originalmusters auf zwei Elemente der Zielumgebung abzubilden, ist wie oben erläutert

nicht zulässig. Der vorliegende Fall könnte entweder dahingehend interpretiert werden das die kopierte Typbedingung an *alle* Elemente gebunden wird, oder mehrere Kopien der Bedingung angefertigt werden sollen.

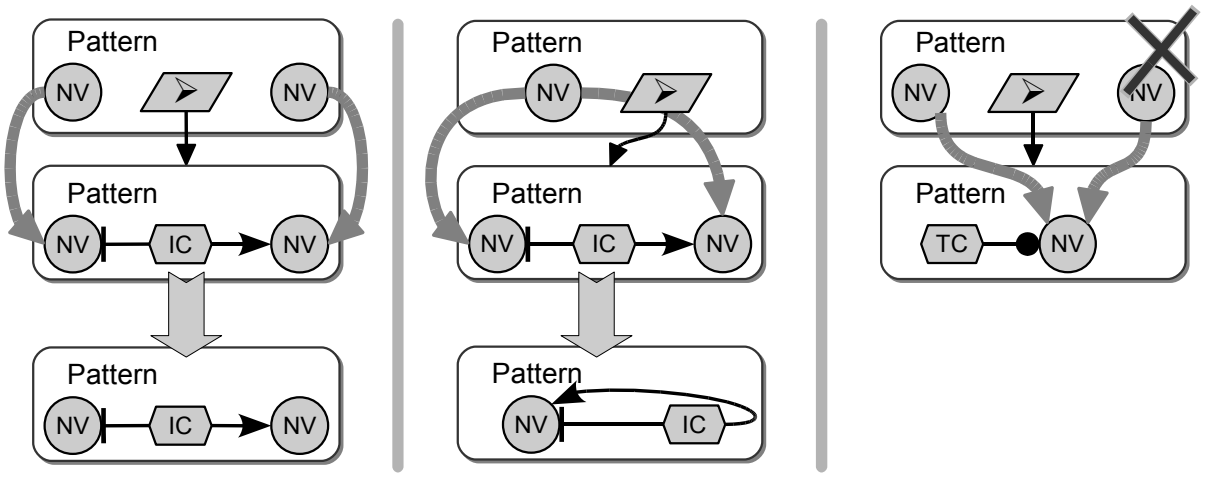


Abbildung 4.18: Zulässige und unzulässige Elementabbildungen

Eine spezielle Behandlung bei der Referenzierung eines Musters erfordern dessen sogen. „externe Elemente“. Diese stellen Elemente dar, die zwar mit denen eines Musters verbunden sind[4], nicht jedoch in selbigem Muster oder einem transitiv darin enthaltenen liegen. Eine Referenzierung dieser Elemente seitens der Kopie ist aufgrund fehlender Enthaltenseinshierarchien zwischen referenzierendem und referenziertem Muster aufgrund obiger Wohlgeformtheitsbedingungen i.A. nicht möglich. Bezüglich der Interpretation als Unterprogrammaufruf wäre darunter zu verstehen, dass Eingabe-Formalparameter nicht an entsprechende Aktualwerte gebunden werden. Aus diesem Grund wird gefordert, dass externe Entitäten mittels der bereits eingeführten *Abbilungsrelationen* auf entsprechend geeignete Elemente des aufrufenden Musters abgebildet werden. Für externe Entitäten wird allerdings insbesondere verlangt, dass diese erzwungenermaßen abgebildet werden. Definiert wird diese Forderung durch die folgende Hilfsfunktion `refs`, die zu einem gegebenen Musterelement die Menge der adjazenten Elemente ermittelt. Diese Liste wird in der Invariante `allMapped` auf *außerhalb* des referenzierten Musters liegende Element eingeschränkt. Für die verbleibenden Elemente wird das Vorhandensein einer Elementabbildung gefordert.

```
context Pattern
def refs : Collection(PatternElement) =
```

[4]Bspw. in Form von Restricts Kanten zwischen Variablen und Bedingungen

```
        self.elements→collect(pe |
          if (pe.oclIsKindOf(Variable)) then
            pe.oclAsType(Variable).constraints
          else if (pe.oclIsKindOf(Variable)) then
            pe.oclAsType(Constraint).variables
          else if (pe.oclIsTypeOf(Pattern)) then
            pe.oclAsType(Pattern)→refs()
          else if (pe.oclIsTypeOf(PatternReference)) then
            pe.oclAsType(PatternReference).references
          endif [...] endif )

context PatternReference
inv allMapped : self.references.refs()→
      reject(loc | self.references.allElements.contains(loc))→
      forAll(ext | self.mappings→exists(m | m.original = ext))
```

Die zunächst letzte kontextsensitive Einschränkung von Musterreferenzen betrifft die Abbildungen zwischen Original und übertragenem Element. Zu beachten ist hierbei, dass beide Elemente des gleichen Typs sind, bspw. darf also eine Knotenvariable nur auf eine weitere Knotenvariable abgebildet werden. In Ermangelung an OCL-Sprachkonstrukten zur Analyse der Typisierung von Objekten ist dabei eine vollständige Aufsummierung von kompatiblen Typen nötig. Der Einfachheit halber wird hier vorausgesetzt, dass die jeweiligen Typen stets identisch sein müssen – die Abbildung von Instanzen allgemeiner auf Instanzen spezialisierter Typen wird daher nicht unterstützt.

```
context MappingElemente
inv compatibleMapping : self.original.isOclTypeOf(Pattern).
        implies(self.copy.isOclTypeOf(Pattern))
      or self.original.isOclTypeOf(NodeVariable).
        implies(self.copy.isOclTypeOf(NodeVariable))
      or [...]
```

Semantische Definition

Die semantische Definition von Musterreferenzen erfolgt anhand ihrer Interpretation als Strukturkopie. Demnach ist eine Formalisierung des Kopierbegriffs unter Berücksichtigung der o.g. Abbildungsrelationen nötig. Diese erfolgt in mehreren Schritten: Zunächst wird eine erweiterte Semantikdefinition für *Muster* eingeführt um eine Rückführung von Mustern mit Referenzen auf entsprechend modifizierte Varianten ohne Referenzen vorzunehmen. Hierzu wird der zweite Teil der Formalisierung, eine

Funktion zur *Expansion* von Musterreferenzen benötigt. Diese stützt sich wiederum auf eine Funktion zur *Strukturkopie* von Musterstrukturen ab.

Sei $SEM_{\substack{ref \\ \textbf{search}}}[\![p]\!] : Pattern \to LS(GS) \times \mathcal{P}(Match)$, p ein Muster und $pr_1 \ldots pr_n \in \text{elements}(p) \cap PatternReference$ Musterreferenzen. Eine Expansion dieser Referenzen erfolgt wenn eine Auffindungsstelle für p ohne Berücksichtigung von Referenzen oder geschachtelter Muster[5] gemäß $SEM_{\substack{hrcl \\ \textbf{search}}}[\![p]\!]$ gefunden werden kann – hier dargestellt durch die Menge M'. Die endgültige Menge von Auffindungsstellen M wird zu p nach Expansion aller Musterreferenzen bestimmt.

$$\begin{aligned}(G,M) \in SEM_{\substack{ref \\ \textbf{hrcl} \\ \textbf{search}}}[\![p]\!] \quad &\Leftrightarrow \quad (G,M) \in SEM_{\substack{valid \\ \textbf{search}}}[\![expand_{pr_1} \circ \ldots \circ expand_{pr_n}(p)]\!] \\ &\wedge \quad \exists M' \big((G,M') \in SEM_{\textbf{search}}[\![p]\!] \big)\end{aligned}$$

Die Hilfsfunktion expand : $PatternReference \times Pattern \to Pattern$ formuliert die Expansion von Musterreferenzen. Diese Funktion ersetzt in einem Muster eine gewählte Referenz durch ihr expandiertes Äquivalent. Für eine Musterreferenz $pr \in \text{elements}(p) \cap PatternReference$ gilt hierbei:

$$\underset{pr}{\text{expand}}(p) = p' \quad \Leftrightarrow \quad \text{elements}(p') = \text{elements}(p) \cup \{pe_1, \ldots, pe_n\}$$

Das Muster p wird in p' also um zusätzliche Elemente $pe_1, \ldots, pe_n$ ergänzt. Hierbei handelt es sich um *Kopien* der Elemente des referenzierten Musters:

$$\forall pe_i \exists \widehat{pe_i} \Big(\widehat{pe_i} \in \text{elements}\left(referenced\,(pr)\right) \wedge pe_i = copy_{pr}\,(\widehat{pe_i}) \Big)$$

Unter einer Kopie wird in diesem Sinne eine *strukturerhaltene Abbildung* zwischen denjenigen Musterelementen verstanden die transitiv im Original bzw. in der erzeugten Kopie enthalten sind. Die Originalelemente gehören dem referenzierten Muster an, Zielelemente liegen im Muster welches die Referenz enthält. Aus der syntaktischen Definition wird die Hilfsfunktion allElements eingesetzt um transitiv enthaltene Elemente zu bestimmen. Ferner wird eine Kopie an eine zu expandierende Musterreferenz gebunden um auf die zugeordneten Abbildungsrelationen Bezug nehmen zu können.

$$\begin{aligned}copy \quad &: \quad PatternReference \times PatternElement|_{allElements(referenced(pr))} \\ &\to \quad PatternElement|_{allElements(parent(pr))}\end{aligned}$$

[5] Jedoch unter Berücksichtigung übergeordneter Auffindungssstellen.

Die jeweiligen quell- und zielseitigen Elemente einer Kopie werden derart in Relation gesetzt das Verbindungsstrukturen und Attributwerte erhalten bleiben. Zur vollständigen Definition müssen dazu neben der Attributierung und Typisierung aller möglichen Elemente auch Verbindungsstrukturen, bspw. zwischen Variablen und deren einschränkenden Bedingungen, überprüft werden. Auch die Enthaltenseinsbeziehung zwischen Mustern und Musterelementen ist zu berücksichtigen um eine Strukturkopie zu erhalten. Die konkreten Eigenschaften ergeben sich aus den einzelnen Metamodellfragmenten der Abbildungen 4.5, 4.13, und 4.17. Da dies eine langwierige Aufzählung zur Folge hätte, werden an dieser Stelle nur wesentliche Teile der Definition dargestellt. Beispielsweise besagt die erste Teilbedingung das ein Abbild einer Variablen mit Abbildern aller zugeordneten Bedingungen verbunden ist.

$$
\begin{aligned}
\widehat{pe} = copy_{pr}(pe) \quad \Rightarrow \quad & \Big(pe \in Variable \rightarrow \\
& \quad \forall c \left(c \in constraint(pe) \rightarrow copy_{pr}(c) \in constraint(\widehat{pe}) \right) \Big) \\
\wedge \quad & (pe \in Constraint \rightarrow \ldots) \\
\ldots & \\
\wedge \quad & \Big(pe \in Pattern \rightarrow \\
& \quad \forall pe' \left(pe' \in \text{elements}(pe) \rightarrow copy_{pr}(pe') \in \text{elements}(\widehat{pe}) \right) \Big)
\end{aligned}
$$

Die Kopie eines Musters sollte naturgemäß äquivalent zum jeweiligen Original sein, eine Eigenschaft, die i.d.R. durch die *Isomorphie* einer Abbildung zwischen zwei Strukturen definiert wird. Um diese Eigenschaft zu gewährleisten, wird verlangt das $copy_{pr}$ *injektiv* ist, für Elemente pe_1, pe_2 des Originalmusters also gilt:

$$copy_{pr}(pe_1) = copy_{pr}(pe_2) \Rightarrow pe_1 = pe_2$$

Aufgrund dieser Forderung ergibt sich eine isomorphe Abbildungen zwischen einem referenzierten Muster und einer geeigneten Teilmenge der Zielumgebung, für die $copy_{pr}$ eine surjektive Funktion ist. Einzuschränken ist die Isomorphieeigenschaft auf solche Musterelemente, die nicht Teil der spezifizierten Elementabbildung der Referenz sind. Zu fordern ist daher, dass

$$copy_{pr}|_{\{pe|pe \notin original(mappings(pr))\}} \text{ injektiv ist.}$$

Elementabbildungen stellen wie oben erläutert ein Hilfsmittel zur präzisen Einbettung eines kopierten Musters in einen geeigneten Kontext bereit. Da diese auch nicht-injektiv sein können ist eine entsprechende Berücksichtigung durch *copy* nötig.

Der folgende Ausdruck legt dabei fest, dass explizit aufeinander abgebildete Elemente durch die Funktion respektiert werden. Dies erfolgt über Wertepaare $(pe, \widehat{pe})$, deren zweite Komponente als *Kopie* einer expliziten Abbildung referenziert werden muss, falls die erste Komponente als *Original* referenziert wird. Gemäß obiger Festlegung muss es sich hierbei nicht um eine injektive Abbildung handeln, wodurch auch nicht-injektive Elementabbildungen unterstützt werden.

$$\widehat{pe} = copy_{pr}(pe) \Rightarrow \forall m \Big(\big(m \in mappings(pr) \wedge pe = original(m) \big) \rightarrow \widehat{pe} = copy(m) \Big)$$

Rekursive Musterreferenzen

Primäre Motivation zur Einführung von Musterreferenzen, wie bereits eingangs erläutert, ist die Erweiterung DRAGULAs bezüglich unbeschränkter, bspw. transitiv abgeschlossener Musterstrukturen. Zu diesem Zweck wird zugelassen das Musterreferenzen *rekursiv* definiert werden. Dies erfolgt durch Bezugnahme einer Musterreferenz auf ein Muster in dem sie selber transitiv enthalten ist. Die bereits eingeführte Syntax und Semantik von Musterreferenzen muss zu diesem Zweck nicht angepasst werden, da bereits eine durch Laufzeitzustände bedingte Expansion von Musterreferenzen vorgesehen ist.

Ausgeschlossen ist bezüglich rekursiven Musterreferenzen, dass diese sich *direkt* auf ihr übergeordnetes Muster beziehen. Wäre dies zugelassen, würde jede aufgelöste Musterreferenz als Kopie wieder in das referenzierende Muster eingeführt, und somit eine nicht terminierende Verarbeitung bewirken. Für die implementierungsseitige Interpretation als Unterprogrammaufruf käme dies einer fehlenden Abbruchbedingung gleich. Wie die folgenden Beispiele zeigen, werden daher *zwischengeschaltete Muster* genutzt, um die zu ersetzende Referenz geeignet zu verkapseln. Die Variablen und Bedingungen dieser Muster dienen dabei als Abbruchbedingung, da bei Nicht-auffinden einer Auffindungsstelle keine Verarbeitung der enthaltenen Musterreferenzen zu erfolgen braucht.

Zur Vermeidung endloser rekursiver Aufrufe wird die folgende kontextsensitive Einschränkung getroffen. Diese besagt lediglich, dass Musterreferenzen nicht in dem von ihnen referenzierten Muster enthalten sein dürfen.

```
context PatternReference
inv notDirectRecursion : not self.parent = self.referenced
```

Anwendungsbeispiel: Kantenbüschel

Ein Anwendungsbeispiel für rekursive Suchmuster zeigt Abbildung 4.19. Zur Ermittlung *aller* zu einem Knoten inzidenten Kanten, sogen. *Kantenbüscheln*, wird eine Selbs-

treferenz auf das dementsprechend suchende Muster eingesetzt. Ein mittels Kardinalitätsangaben als *optional* markiertes Muster ermittelt im oberen Bereich der Abbildung zunächst eine einzelne Kante des angegebenen Typs `besteht_aus`. Die Musterreferenz innerhalb des geschachtelten Musters bewirkt, dass der Inhalt des navigierenden Musters strukturerhaltend kopiert wird, und so eine rekursive Vervielfältigung dieses Teilmusters erfolgt.

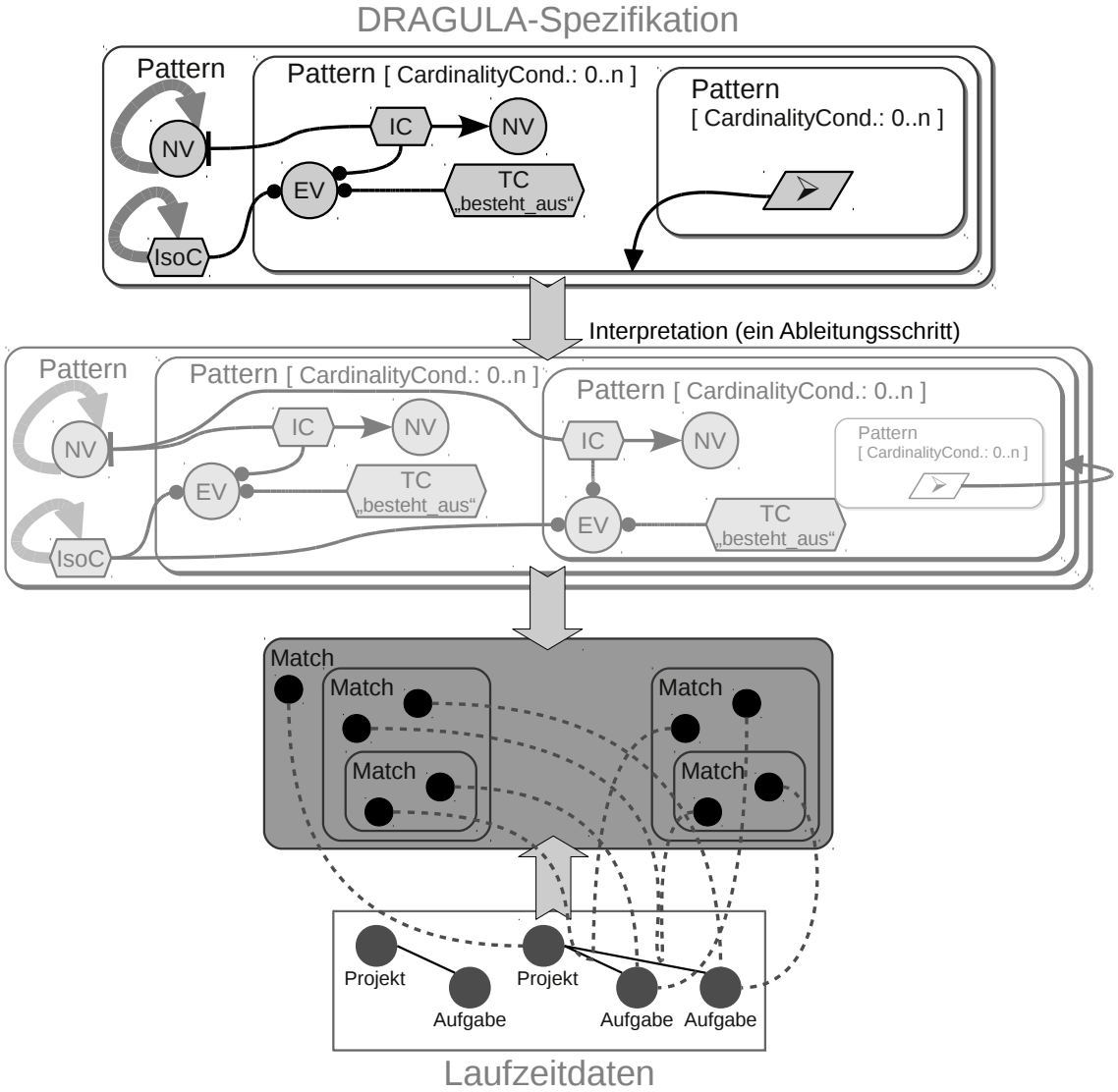


Abbildung 4.19: Rekursives Muster zur Bestimmung von Kantenbüscheln

Der schrittweise Ablauf der Mustersuche gestaltet sich ausgehend von der oberen Struktur in Abbildung 4.19 wie folgt: Nach Auffinden einer Variablenbelegung für das

äußere Muster wird mit dem direkt darin enthaltenen Muster fortgefahren. Hierfür sind im dargestellten Beispieldatensatz zwei Auffindungsstellen verfügbar. In beiden Fällen wird mit dem auf dritter Ebene befindlichen Muster fortgefahren, für das jeweils eine zunächst leere Auffindungsstelle ermittelt wird, da es keinerlei Variablen enthält.

Anschließend wird aufgrund dessen die Musterreferenz expandiert, wobei eine Kopie der Elemente des referenzierten Musters angelegt und in das referenzierende Muster eingebettet wird. Die explizit *reflexiv* abgebildeten externen Elemente werden dabei, wie in der zweiten Schicht der Abbildung dargestellt, beibehalten und somit von der erzeugten Kopie ebenfalls referenziert. Dies entspricht der Intention eines Kantenbüschels, da der jeweilige Startknoten eindeutig, und alle ermittelten Kanten unterschiedlicher Identität sein sollten.

Nach Ermittlung der Auffindungsstellen des so erweiterten Musters wird mit dem neu hinzugekommenen Muster der vierten Eben analog verfahren. Nach einem weiteren Expansionsschritt wird die Mustersuche feststellen, dass keine Variablenbelegung existiert, welche die äußere Isomorphiebedingung nicht verletzt. Somit erfolgt keine weitere Expansion, und die Mustersuche terminiert. Alle auf diese Weise gefundenen Auffindungsstellen sind ferner *gültig*, da alle beteiligten Muster als *optional* markiert sind, und somit keine zusätzlichen Einschränkungen einbringen.

Als Ergebnis der Mustersuche wird eine Struktur geschachtelter Auffindungsstellen ermittelt, wie Abbildung 4.19 an einem Beispiel illustriert. Die Reihenfolge, in der Kanten und deren inzidente Zielknoten bei der Suche aufgefunden werden, beeinflusst deren Zuordnung zu den dargestellten Variablen. Im konkreten Fall werden also zwei verschiedene Auffindungsstellen des äußeren navigierenden Musters gefunden, obwohl diese identische Kantenbüschel repräsentieren, und sich lediglich in der hier irrelevanten Variablenanordnung unterscheiden. Durch eine geeignete Parametrisierung der Ausführungsmaschine kann allerdings erreicht werden das lediglich eine *einzelne* Auffindungsstelle verarbeitet wird, falls diese als gültig erkannt wird. Hierdurch kann die Effizienz der Verarbeitung erhöht und die Menge der ausgegebenen Auffindungsstellen auf relevante Informationen beschränkt werden.

Anwendungsbeispiel: Pfadausdrücke

Abbildung 4.20 zeigt eine Anwendung rekursiver Musterreferenzen zur Modellierung von transitiven Abschlüssen in Pfadausdrücken. Im dargestellten Beispiel wird eine einzelne Kante vom Typ Teilaufgabe zwischen null und beliebig vielen Schritten navigiert um alle Bestandteile einer Aufgabe zu ermitteln. Die linke äußere Variable repräsentiert die übergeordnete Aufgabe, die rechte äußere Variable einen transitiv darin enthaltenen Bestandteil. Zusätzlich findet sich im äußersten Muster eine

Isomorphiebedingung die zur Erkennung von Zyklen in den navigierten Strukturen benötigt wird.

In diesem äußeren Muster befindet sich ein geschachteltes Muster mit der Annotation OrCondition. Dieses Muster ist trivial erfüllt, da es selbst keine Variablen enthält. Bei Ausführung der Mustersuche wird deshalb mit den beiden enthaltenen Mustern fortgefahren. Aufgrund der booleschen Musterbedingung genügt es, wenn zu wenigstens einem dieser Muster eine gültige Auffindungsstelle gefunden wird, um die gesamte Pfadsuche als erfolgreich zu werten.

Auf der dritten Ebene der Musterschachtelung findet sich ein negiertes Muster mit Kardinalitätsbedingung 0..0. Dies bewirkt eine Negation der enthaltenen Isomorphiebedingung, was in einer Gleichheitsprüfung der verbundenen Variablen resultiert. Dieses Muster ist also insgesamt erfüllt, wenn beiden äußeren Variablen der selbe Wert zugeordnet ist. Das untere Muster führt schließlich die gewünschte Navigation entlang einer Kantenvariablen durch.

Die Rekursion des Pfadausdrucks wird schließlich durch die dargestellte Musterreferenz modelliert. Wird die Musterreferenz als *Strukturkopie* interpretiert, so wird dadurch der Inhalt des *ersten geschachtelten* Musters in das Muster der Referenz übertragen. Mittels der Abbildungsrelationen wird festgelegt, dass die kopierte Struktur dieselbe Zielvariable sowie die selbe Isomorphiebedingung verwenden wie das Originalmuster. Dagegen wird als *Anfangsvariable* des neuen Teilmusters die *Zwischenvariable* der Navigation des Originalmusters verwendet. Insgesamt entsteht so die in der unteren Hälfte von Abbildung 4.20 dargestellte Musterstruktur.

Bei der Ermittlung von Auffindungsstellen für dieses Suchmuster wird analog zu Abbildung 4.19 vorgegangen. Das Ergebnis besteht also aus einer Menge geschachtelter Auffindungsstellen die gemäß der Musterstruktur aufgebaut sind. Dabei wird pro Traversierungsschritt *eine Schachtelungsebene* in der Struktur der Auffindungsstellen erzeugt. So zeigt Abbildung 4.20 mögliche Auffindungsstellen für Pfade der Länge null und eins. Für längere Pfade würde zunächst eine weitere Expansion der Musterreferenz erfolgen, um einen Vergleich des zuletzt durch Traversierung ermittelten Elements und dem Wert der äußeren rechten Variable durchzuführen.

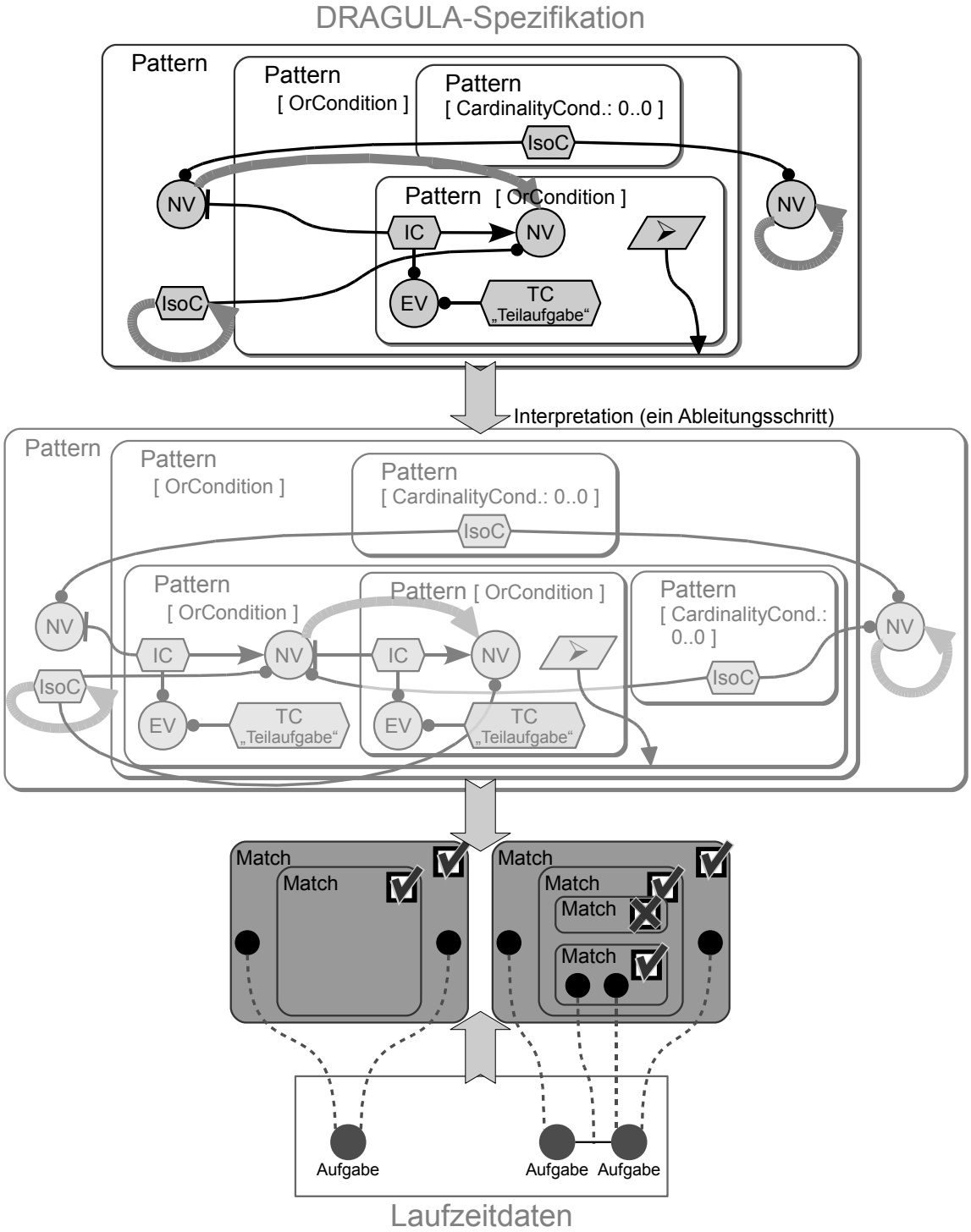


Abbildung 4.20: Transitiver Abschluss einer Verbindungsnavigation

Zwischenfazit: Musterreferenzen bieten ein geeignetes Sprachkonstrukt zur Modellierung verschiedener komplexer Strukturen. Hierzu zählen bspw. mengenwertige Kantenbüschel sowie transitive Abschlüsse von Graphmustern. Nichtsdestotrotz entstehen hierbei schwer verständliche Dokumente die, auch wenn eine Nutzung durch Entwickler nicht im Vordergrund von DRAGULA steht, mit erhöhtem Spezifikationsaufwand verbunden sind. Aus diesem Grund werden die wesentlichen Einsatzzwecke der Musterreferenzen in Abschnitt 4.6 in Form von *Spracherweiterungen* geeignet verkapselt. Deren Bedeutung kann jedoch auf die formale Beschreibung des im aktuellen Abschnitt vorgestellten Sprachkerns *zurück geführt* werden.

4.4 Transformationen

Die im bisherigen Verlauf dieses Kapitels vorgestellten Sprachkonstrukte umfassen Funktionalität zur Mustersuche und damit zur Analyse graphbasierter Daten. Die Veränderung von Graphstrukturen basierend auf einer zuvor ermittelten Auffindungsstelle wird in diesem Abschnitt behandelt. Hierbei verfolgt DRAGULA einen ähnlichen Ansatz wie in den vorangegangenen Sprachkomponenten. Im Vordergrund stehen dabei die leichte Erweiterbarkeit der angebotenen Konstrukte sowie die Unterstützung des komplexen zugrundeliegenden Graphmodells.

Transformationen werden in DRAGULA durch sogen. *Operatoren* beschrieben. Diese werden syntaktisch als Bestandteile von Mustern aufgefasst um bspw. das Erstellen oder Löschen von Elementen zu spezifizieren. Hierbei wird anders als bei den in Abschnitt 2.1.2 theoretisch eingeführten Graphtransformationen keine linken und rechten Regelseiten unterschieden, sondern stattdessen einzelne Editieroperationen explizit spezifiziert. Diese Entwurfsentscheidung liegt darin begründet, dass aufgrund der in DRAGULA durchgeführten Trennung zwischen Variablen und Bedingungen keine direkte strukturerhaltene Abbildung zwischen einem modellierten Muster und einer Laufzeitdatenstruktur möglich ist. Transformationsschritte können daher ebenfalls nicht direkt mittels Morphismen auf Ausgangs- und Zielgraphen abgebildet werden, wodurch der Vorteil einer Darstellung mit Regelseiten entfällt. Vorteilhaft an der in DRAGULA gewählten expliziten Schreibweise ist dagegen die kompaktere Notationsform im Vergleich zur Darstellung mit Regelseiten.

Semantisch wird im Folgenden eine Transformation durch eine entsprechende Variante der Funktion SEM spezifiziert. Diese führt Veränderungsoperationen bezüglich einer zuvor ermittelten Ausführungsstelle durch. Ein Muster mit Veränderungsoperatoren kann somit zu zweierlei Aufgaben eingesetzt werden: Durch Anwendung der *Suchsemantik* dient das Muster als reine Anfrage, wohingegen die *Transformationsse-*

mantik zugehörige Veränderungen durchführt.

Obwohl die explizite Spezifikation der durchzuführenden Veränderungen den Anschein eines imperativen, und daher vergleichsweise niedersprachlichen Ansatzes vermittelt, so ist die verwendete Variante dennoch als deklarativ einzustufen. So legt der Entwickler die *Reihenfolge* der Operatoranwendung nicht explizit fest, sondern lässt diese bei der Ausführung geeignet ermitteln. Es erfolgt also nach wie vor eine Abstraktion von den letztlich durchgeführten Graphveränderungen.

4.4.1 Anwendungsbeispiel

Als Anwendungsbeispiel für transformierende Muster werden die Laufzeitdaten des fiktiven Projektmanagementsystems regelbasiert verändert. Einen beispielhaften Laufzeitzustand sowie eine darauf anzuwendende DRAGULA Spezifikation zeigt Abbildung 4.21. Die dargestellte Graphstruktur beschreibt eine Aufgabe und eine daran arbeitende Person. Durch eine zweite Kante ist die Aufgabe zudem der Person als aktuelle Aufgabe zugeteilt. Im Zuge der Transformation soll die bestehende Aufgabe gelöscht und eine neue Aufgabe, bspw. in Form einer alternativen Beschäftigung, erzeugt werden. Die ArbeitetAn Kante wird dabei lediglich *umgeleitet*, wodurch ihre Identität und damit mögliche Kantenattribute erhalten bleiben. Dagegen wird die Aktuell Kante implizit mit der inzidenten Aufgabe zusammen gelöscht und im weiteren Transformationsverlauf eine neue Kante dieses Typs angelegt.

Der *suchende* Anteil des Musters umfasst zwei Knoten- und eine Kantenvariable sowie zugehörige Typ- und Inzidenzbedingungen um die oben beschriebene Ausgangsstruktur zu ermitteln. Im Unterschied zu den vorangegangenen Beispielen ist in diesem Zusammenhang die Ermittlung eines enthaltenden Graphen notwendig, da die im Weiteren neu erstellten Elemente gemäß des DRAGOS Graphmodells einem solchen zugeordnet werden müssen. Dies wird durch eine *GraphVariable* und eine Enthaltenseinsbedingung (ContainmentConstraint, CC) erreicht. Als Behälter der neu erstellten Elemente wird demnach der Graph des Aufgabe-Knotens gewählt.

Für den suchenden Anteil des dargestellten Musters kann die in der Abbildung dargestellte Auffindungsstelle ermittelt werden. Die *nicht* mit Bedingungen verbundenen Variablen bleiben dabei zunächst unbelegt. Falls mehrere Auffindungsstellen des Suchmusters ermittelt werden können, so ist zur nachfolgenden Transformation zunächst eine Stelle auszuwählen.

Das in Abbildung 4.21 dargestellte Muster spezifiziert insgesamt fünf Veränderungen der verarbeiteten Auffindungsstelle:

- Erstellung eines Knotens des Typs Aufgabe.
- Erstellung einer Kante des Typs Aktuell

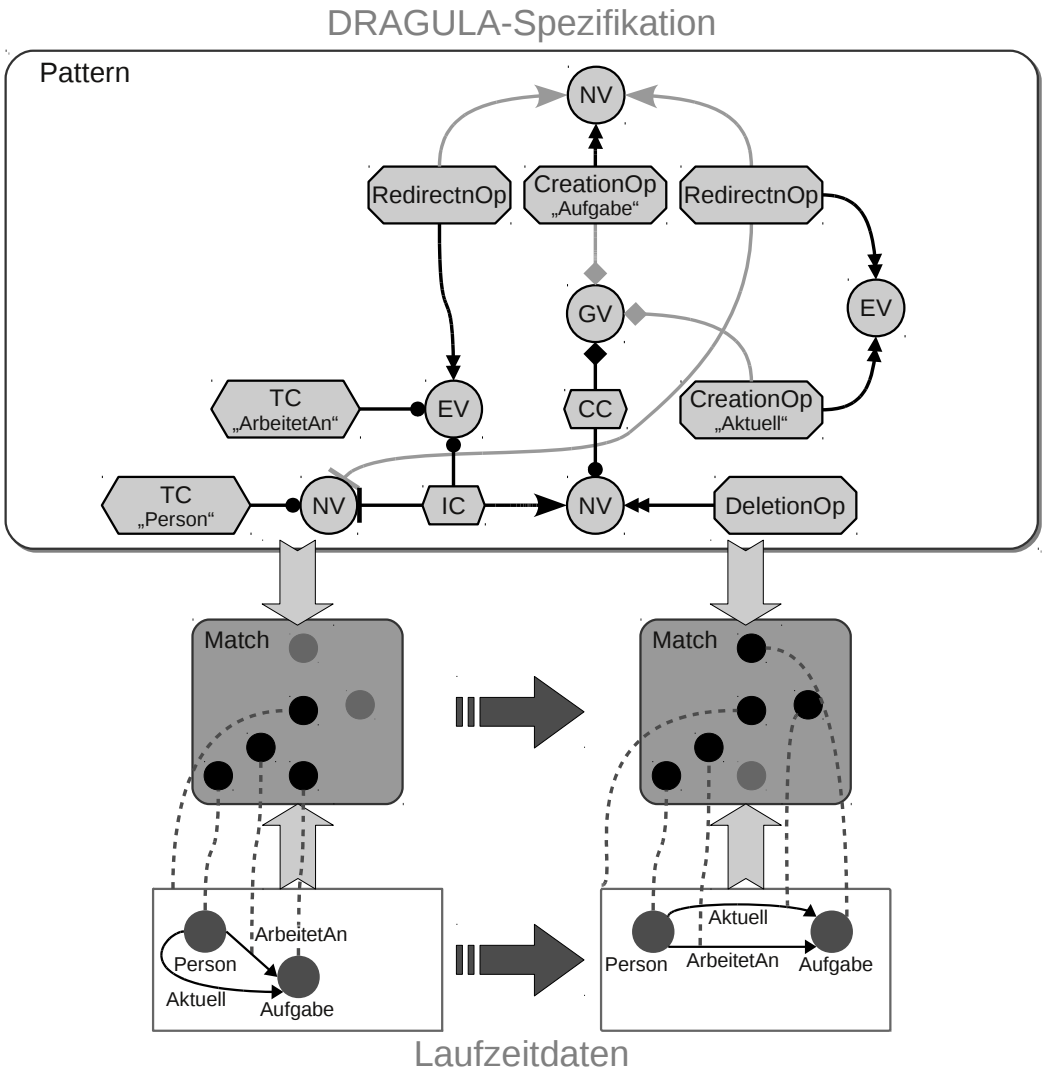


Abbildung 4.21: Beispielspezifikation zur Transformation von Graphstrukturen

- Umleitung dieser Kante von der ermittelten Person zur neu erstellten Aufgabe
- Umleitung einer Kante des Typs ArbeitetAn von der alten zur neuen Aufgabe.
- Entfernung der alten Aufgabe und aller inzidenten Kanten, einschließlich der nicht in der Auffindungsstelle enthaltenen Aktuell-Kante.

Die jeweiligen Operatortypen sowie zugehörige Verbindungsstrukturen, werden im Weiteren detailliert erläutert.

Die Auswirkung der dargestellten Operatoren wird in Form einer veränderten Auffindungsstelle sowie einer veränderten Laufzeitdatenstruktur dargestellt. Dabei wer-

den die neu erstellten Elemente durch Zuweisungen in die Auffindungsstelle aufgenommen. Die Umleitung von Verbindungsstrukturen hat dagegen nur Auswirkungen auf die Laufzeitdaten.

4.4.2 Syntaktische Definition

Zur Unterstützung transformierender Operatoren wird das DRAGULA Metamodell wie in Abbildung 4.22 gezeigt erweitert. Die abstrakte Klasse Operator sieht vor, dass stets genau eine Variable durch die Anwendung einer Operation *beeinflusst* wird. Ferner bieten Operatoren die Möglichkeit, *Verweise* (requires) auf Variablen zu spezifizieren. Diese stellen für die jeweilige Operation benötigte Informationen bereit, bspw. das zu setzende inzidente Graphelement einer Umleitungsoperation. Analog zu restricts Beziehungen sieht auch die Abhängigkeitsbeziehung von Operatoren requires einen Rollenbezeichner (role) vor. Auf diese Weise kann bspw. zwischen Quell- und Zielknoten einer Kantenumleitung unterschieden werden.

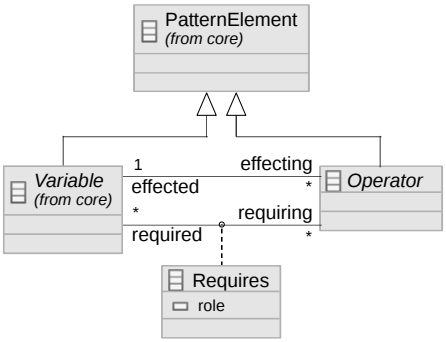


Abbildung 4.22: Transformationsspezifisches Metamodell

Bezüglich der kontextsensitiven Syntax von Operatoren wird festgelegt, dass diese sich im selben Muster wie die jeweilige beeinflusste Variable befinden müssen. Alternativ ist es auch zulässig, dass der Operator sich in einem ggf. transitiven Untermuster befindet. Nicht zulässig ist dagegen bspw., insbesondere im Unterschied zu oben eingeführten Bedingungen, die Referenzierung einer Variablen eines *tiefer* geschachtelten Musters.

```
context Operator
inv hierarchy : self.parent = self.effects.parent or
      self.effects.parent.allElements→contains(self.parent)
```

Die DRAGULA Kernsprache bietet verschiedene *grundlegende* Elemente zur Veränderung von Graphstrukturen. Hierzu zählen die in Abbildung 4.23 aufgezeigten

Operatoren, die aus Maßnahmen zur *Erstellung*, der Erstellung übergeordneter Graphen sowie der *Löschung* von Elementen hervorgehen. Diese Elemente werden im Folgenden einzeln erläutert.

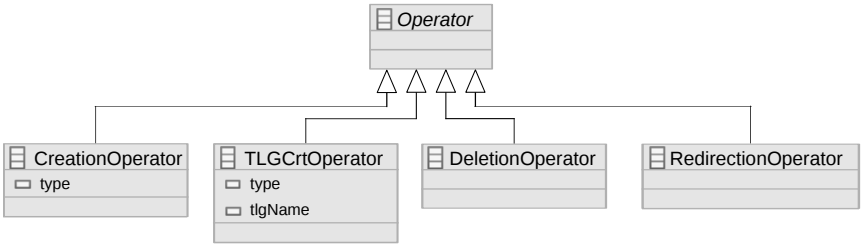


Abbildung 4.23: Typhierarchie der Operatoren

Zur vereinfachten Verwendung des Metamodells wird zusätzlich die Hilfsfunktion varByRole analog zu den Bedingungen aus Abschnitt 4.2 definiert. Diese ermittelt die Menge aller Variablen, die mit einem Operator über einen festgelegten Rollenbezeichner verbunden sind.

```
context Operator
def varByRole (role : String) : Set(Variable) =
        self.requires→select(r1 | r1.role = role)→
        collect(r2 | r2.variables)
```

Erstellung von Elementen: Der CreationOperator beschreibt die Erstellung von Graphelementen, die im Zuge einer Graphtransformation in den Laufzeitdatenstrukturen hinzugefügt und in die verarbeitete Auffindungsstelle aufgenommen werden. Syntaktisch müssen diese Operatoren eine Graphvariable referenzieren, in deren zugeordneten Graph das zu erstellende Element eingefügt wird. In Abbildung 4.21 ist diese Variable durch einen grauen Pfeil nach Art der UML-Kompositionsbeziehung angebunden.

```
context CreationOperator
inv uniqueContainer : self.varByRole(„container“)size()=1

inv properContainer : self.varByRole(„container“)
      .oclIsTypeOf(GraphVariable)
```

Erstellung übergeordneter Graphen: Eine Variante der Elementerstellung stellt der TLGCrtOperator dar, der explizit zur Erstellung *übergeordneter* Graphen dient. Im Unterschied zu obigem Fall erfordert dieser Operator *keine* Graphvariable als Behälter des

neu zu erstellenden Elements. Stattdessen kann ein Name des neuen übergeordneten Graphen festgelegt werden. Der Operator kann nur ausgeführt werden, falls im bearbeiteten Graphspeicher dieser Name nicht bereits vergeben ist. Ferner muss es sich beim angegebenen Typ um eine *Graphklasse* handeln. Da DRAGULA keine Schemaspezifikation ermöglicht, kann diese eigentlich syntaktische Wohlgeformtheitsregel jedoch nicht statisch überprüft und als OCL Ausdruck angegeben werden.

Löschung bestehender Elemente: Zur Löschung von Elementen dient der DeletionOperator. Dieser beeinflusst die Variable, welche den zu löschenden Wert enthält. Ein entsprechend gestalteter Operator ist mit keinem anderen Element verbunden.

```
context DeletionOperator
inv properRequired : self.required→isEmpty()
```

Umleitung von Verbindungsstrukturen: Verbindungselemente, wozu im Falle des DRAGOS Graphmodells sowohl Kanten als auch Relationsenden gehören, können durch den RedirectionOperator auf andere inzidente Elemente umgeleitet werden. Die Variable des umzuleitenden Elements wird mittels einer Verbindung des Typs effects identifiziert. Die visuelle Darstellung erfolgt in Abbildung 4.21 analog zu Inzidenzbedingungen, jedoch in grauer Farbe.

```
context RedirectionOperator
inv properRequired : not ( self.varByRoles(„source“)→isEmpty() and
      self.varByRoles(„target“)→isEmpty() )

inv properSource : self.effects.oclIsTypeOf(RelationEndVariable)
      implies self.varByRoles(„source“)→isEmpty()
```

Das neue Startelement einer umzuleitenden Verbindung wird durch eine Variable mit Rollenbezeichnung (source) bestimmt, das neue Zielelement einer Kante oder eines Relationsendes analog hierzu mit target. Die einem Relationsende zugehörige Relation kann nicht verändert werden, da das Relationsende existenziell von dieser abhängt.

4.4.3 Gewährleistung der Reihenfolgeunabhängigkeit

Eine wesentliche Eigenschaft des Transformationsanteils der Kernsprache DRAGULA ist die *reihenfolgeunabhängige* Spezifikationsweise von Operatoren. Diese gewährleisten eine Abstraktion von den letztendlich durchgeführten Veränderungsschritten, die in gängigen Programmiersprachen als explizite Abfolge festgelegt werden. Nichtsdestotrotz soll das Ausführungsverhalten einer Transformation *deterministisch* bezüglich ei-

ner Auffindungsstelle sein, um das modellierte Verhalten vorhersagen und beeinflussen zu können. Hierzu ist sicherzustellen, dass je nach interner Auswahl einer vorab unbestimmten Operatorreihenfolge das Ergebnis einer Transformation nicht verändert wird.

Zur Lösung dieses Problems führt DRAGULA zweierlei *Randbedingungen* ein, um Operatoren eindeutig ausführen zu können ohne eine explizite Festlegung im Spezifikationsdokument vornehmen zu müssen. Diese sehen eine grobe Einteilung der Operatorklassen in *Phasen* vor, wobei Operatoren einer einzelnen Klasse simultan in zwei *Stufen* ausgeführt werden. Grafisch wird diese Einteilung in Abbildung 4.24 dargestellt.

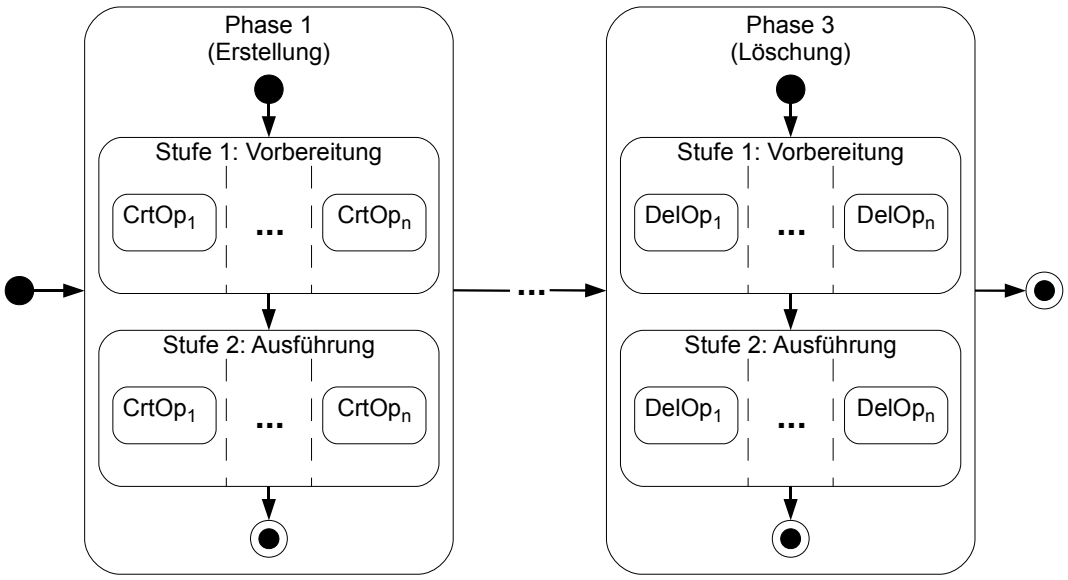


Abbildung 4.24: Phasen- und Stufeneinteilung der Operatorausführung

Phasenbasierte Gliederung: Die phasenbasierte Einteilung von Operatorklassen erlaubt eine grobe Priorisierung einzelner Operatoren. Hierzu wird eine *Halbordnung* vorausgesetzt, in die sich Operatorklassen einordnen lassen. Instanzen einer höher priorisierten Klasse werden demzufolge *vor* niedrigeren Klassen berücksichtigt. Zwischen nicht miteinander in Relation stehenden Operatorklassen wird keine Priorisierung vorgenommen. Die Ausführung dieser Operatoren kann gemäß unten eingeführter Stufen simultan, aber auch in beliebiger sequenzieller Reihenfolge erfolgen.

Die Priorisierung der seitens der Kernsprache bereitgestellten Operatoren aus Abbildung 4.23 gliedert sich wie folgt: Erstellende Operatoren besitzen die höchste Priorität, gefolgt von verändernden Operatoren wie dem RedirectionOperator. Auf diese Wei-

se kann eine Umleitung von Verbindungsstrukturen auch erfolgen, falls das inzidente Element oder der Verbinder erst während der Transformationsausführung erstellt wird. Niedrigste Priorität besitzt der Löschoperator, so dass eine Verbindung bedingungskonform umgeleitet werden kann, falls ein inzidentes Element gelöscht werden soll. Die *Anzahl* der Phasen ist nicht vorab festgelegt, insbesondere also nicht auf die in Abbildung 4.24 angedeuteten und für die Hauptbestandteile DRAGULAs benötigten drei Phasen beschränkt.

Die nachfolgend angegebene Formulierung von Phaseneinteilungen erfolgt gemäß der Notationsweise von OCL als Hilfsfunktion von *Instanzelementen* angegeben. Diese liefert ein positives Ergebnis **true** für `o1. priority (o2)` falls Operator `o1` höher priorisiert als `o2` behandelt werden soll, im umgekehrten Fall dagegen **false**. Falls seitens der Sprachdefinition keine Präzedenz besteht, wird `OclUndefined` verwendet.

```
context CretationOperator
def priority (Operator op) : Boolean =
      if (op.oclIsTypeOf(DeletionOperator) or
            op.oclIsTypeOf(RedirectionOperator)) then true
      else OclUndefined endif

context TLGCrtOperator
def priority (Operator op) : Boolean =
      if (op.oclIsTypeOf(DeletionOperator) or
            op.oclIsTypeOf(RedirectionOperator)) then true
      else OclUndefined endif

context RedirectionOpeator
def priority (Operator op) : Boolean =
      if (op.oclIsTypeOf(DeletionOperator) or
            op.oclIsTypeOf(RedirectionOperator)) then false
      else if (op.oclIsTypeOf(DeletionOperator)) then true
      else OclUndefined endif endif

context DeletionOperator
def priority (Operator op) : Boolean =
      if (op.oclIsTypeOf(DeletionOperator)) then OclUndefined
      else true endif
```

Im nachfolgenden Abschnitt wird die *direkte* Priorisierung von Operatoren ausgedrückt durch die Halbordnungsrelation $\rhd \subset (Operator)^2$. Vergleichend mit der OCL-basierten Darstellung gilt, dass $o_i \rhd o_j \Rightarrow o_i.\mathsf{priority}(o_j)$.

Stufenweisung Verarbeitung: Als zweite Maßnahme werden alle Operatoren einer einzelnen Phase *zweistufig* abgearbeitet. In der ersten Stufe werden die nötigen Änderungen an den Laufzeitdaten vorab bestimmt, ohne diese jedoch direkt anzuwenden. Dies erfolgt erst während der zweiten Stufe, und zwar simultan für alle vorab bestimmten Änderungen. Durch diesen Ansatz ist gewährleistet, dass simultane Operatoren sich nicht gegenseitig beeinflussen, also durch abhängige Variablen auf die Zwischenergebnisse des jeweils anderen zugreifen. Anders als obige Phaseneinteilung organisiert der phasenbasierte Ansatz auch Instanzen einer einzelnen Operatorklasse.

Phasenweise Ausführung des Anwendungsbeispiels: Bezogen auf das eingangs behandelte Beispiel aus Abbildung 4.21 ergibt sich eine phasenweise Ausführung der einzelnen Operatoren, wie Abbildung 4.25 zeigt: Die erste Phase führt lediglich erstellende Veränderungen durch, verbindet jedoch bspw. die erstellte Kante nicht mit den zuzuordnenden Elementen. Dies erfolgt in der nachgelagerten Phase, nachdem alle zu erstellenden Elemente bereit stehen. Im letzten Schritt werden Löschungen durchgeführt, so dass Umlenkungsoperatoren bereits abgearbeitet und Verbindungen entsprechend verändert worden sind.

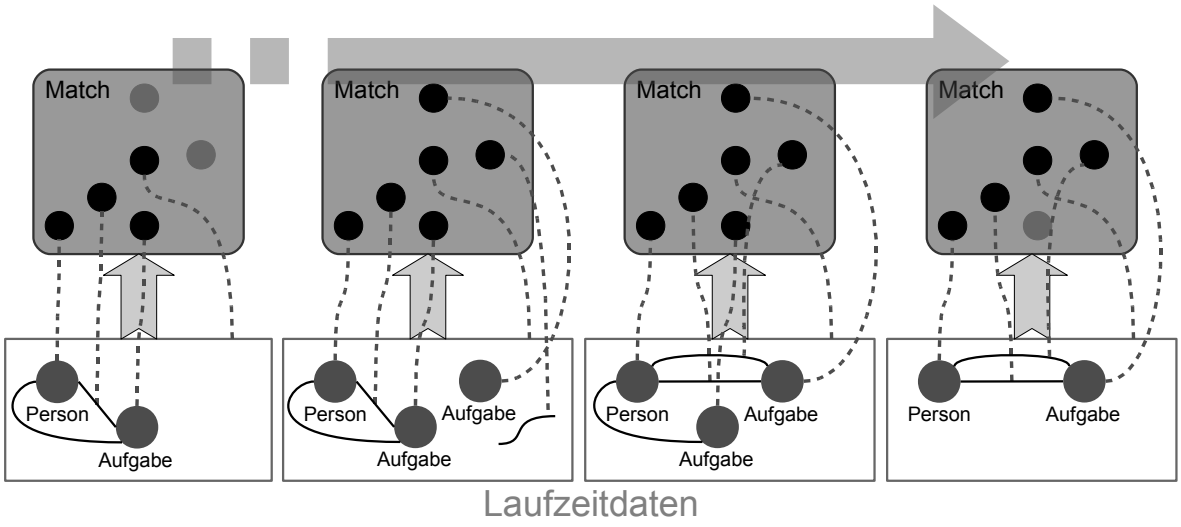


Abbildung 4.25: Phasenweise Verarbeitung der Beispieldaten aus Abbildung 4.21

4.4.4 Semantische Definition

Die semantische Definition des Transformationsverhaltens eines Musters erfolgt anhand einer weiteren Spezialisierung der Relation $SEM[\![p]\!]$. Eine Transformation ver-

ändert simultan die eingegebene Graphstruktur sowie eine zugehörige Auffindungsstelle des Musters p.

$$SEM_{\substack{Pattern \\ \textbf{transform}}} : Match \times LG(GS) \rightarrow LG(GS) \times Match$$

Zu berücksichtigen ist dabei die oben beschriebene Priorisierung von Operatoren. Hierzu seien $o_1, \ldots, o_n \in elements(p) \cap Operator$ die Operatoren eines Musters und $O_1, \ldots, O_m$ eine *Partition* von $elements(p) \cap Operator$ so dass $o_i \rhd o_j \Leftrightarrow o_i \in O_k \wedge o_j \in O_{k+1}$. $G \in LG(GS)$ sei ein Ursprungsgraph sowie $m \in M$ eine ausgewählte Auffindungsstelle der Suchanfrage auf diesem Graphen, die zur Transformation verwendet wird. Die Anwendung der Transformation auf dieser Auffindungsstelle wird beschrieben als Folge von *Zwischenzuständen* $G'_1, \ldots, G'_m$ sowie $M'_1, \ldots, M'_m$, die jeweils das Ergebnis einer Operatorphase darstellen. Das Ergebnis der *letzten* Phase entspricht dem Endergebnis der gesamten Transformation.

$$\begin{aligned}
(m, G, G'_m, m'_m) \in SEM_{\textbf{transform}}[\![p]\!] \quad \Leftrightarrow \quad & \exists M \big(m \in M \wedge (G, M) \in SEM_{\substack{valid \\ \textbf{search}}}[\![p]\!] \big) \\
\wedge \quad & \forall o_i \big(o_i \in O_1 \wedge (m, G, G'_1, m'_i) \in SEM[\![o_i]\!] \big) \\
\wedge \quad & \ldots \\
\wedge \quad & \forall o_j \big(o_j \in O_m \\
& \quad \wedge \quad (m'_{m-1}, G'_{m-1}, G'_m, m'_m) \in SEM[\![o_j]\!] \big)
\end{aligned}$$

Die Semantik einzelner Operatoren wird jeweils auf Basis der Start- und Zielzustände des Arbeitsgraphen sowie der verarbeiteten Auffindungsstelle definiert. Dadurch entsteht eine quasi-parallele Ausführung dieser Operatoren, da jeweils Bezug auf den Startzustand einer Phase genommen wird. Wechselseitige Beeinflussungen zwischen Operatoren einer Phase, die zu unvorhersagbaren Ergebnissen führen könnten, werden somit vermieden.

Erstellung von Elementen: Sei zunächst $co \in elements(p) \cap CreationOperator$, v_{eff} die veränderte Variable sowie v_{cnt} die Graphvariable, welche von co als Container referenziert wird. Die Bedeutung der Erstelloperation ergibt sich aus den Unterschieden der beiden Auffindungsstellen und Graphzuständen, denen eine Zuweisung bzw. ein

Graphelement hinzugefügt wird.

$$
\begin{aligned}
(m, G, G', m') \in SEM[\![co]\!] \quad \Leftrightarrow \quad & \exists u \Big(u \in allElements(G') \setminus allElements(G) \\
\wedge \quad & \underset{\mathcal{G}}{\text{type}}(u) = \text{type}(co) \\
\wedge \quad & \exists g \big(g \in allElements(G) \cap Graph \\
& \wedge \quad g \triangleright_{\mathcal{G}} u \wedge (m, g) \in SEM[\![v_{cnt}]\!] \big) \\
\wedge \quad & (m', u) \in SEM[\![v_{eff}]\!] \Big)
\end{aligned}
$$

Erstellung von Elementen / alternative Definition Problematisch an obiger Definition ist die Tatsache, dass auf diese Weise keine geschachtelten Graphstrukturen in einer einzelnen Transformation erzeugt werden können. Dies wird durch die zweistufige Verarbeitung von Operatoren verursacht, weshalb neu erstellte Graphen nicht als Container eines ebenfalls neu erstellten Elements ausgewählt werden können. Um diese Einschränkung zu umgehen, können Erstelloperationen alternativ dahingehend interpretiert werden, dass Erstellung eines Elements und seine Zuweisung zu einem Container *getrennt* voneinander behandelt werden. Die hierzu nötige Zerlegung der syntaktischen Definition in zwei abstrakte Klassen zeigt Abbildung 4.26. In Bezug auf die Phaseneinteilung des Abschnitts 4.4.3 wird der erstellende Teiloperator BaseCrtOperator ebenso eingeordnet wie obiger CreationOperator, wohingegen der MoveIntoOperator niedriger priorisiert wird. Obige semantische Definition kann auf Basis dieses alternativen Metamodells in zwei Teildefinitionen zerlegt werden, die eine simultane Erstellung hierarchischer Strukturen erlauben. Zur Vermeidung von Redundanz werden diese hier nicht explizit aufgeführt.

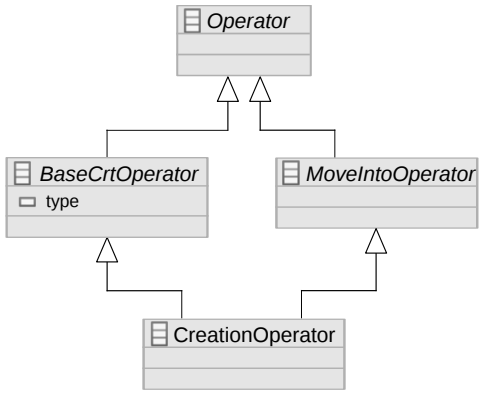


Abbildung 4.26: Alternatives Metamodell für Erstelloperationen

Die Kernsprache DRAGULA bietet trotz der zweiteiligen Definition lediglich das

zusammengefasste Konstrukt CreationOperator an. Die Auslassung einer der beiden Teiloperatoren würde gemäß des DRAGOS Graphmodells direkt zu einem Spezifikationsfehler führen, da ein reguläres Graphelement stets einem Container zugeordnet werden muss. Eine Auftrennung des Erstellungsoperators erscheint aus Benutzersicht von daher nicht sinnvoll.

Erstellung übergeordneter Graphen: Übergeordnete Graphen werden ähnlich zu regulären Graphelementen erstellt, ausgenommen der Festlegung eines enthaltenen Graphen. Zusätzlich besteht die Möglichkeit, den erstellten Graphen zu benennen. Hierzu sei $tco \in elements(p) \cap TLGCrtOperator$ sowie v_{eff} die veränderte Variable.

$$
\begin{aligned}
(m, G, G', m') \in SEM[\![tco]\!] \quad \Leftrightarrow \quad & \exists g \Big(g \in elements(G') \setminus elements(G) \\
\wedge \quad & \underset{\mathcal{G}}{\text{type}}(g) = \text{type}(tco) \\
\wedge \quad & \nexists g \big(g \in allElements(G) \cap Graph \\
& \wedge \quad g \triangleright_{\mathcal{G}} u \wedge (m, g) \in SEM[\![v_{cnt}]\!] \big) \\
\wedge \quad & name(g) = tlgName(tco) \\
\wedge \quad & (m', g) \in SEM[\![v_{eff}]\!] \Big)
\end{aligned}
$$

Löschung bestehender Elemente: Die Löschung eines Graphelements wird lediglich durch Angabe einer zu verändernden Variable beschrieben, deren zugeordnetes Element aus seinem enthaltenden Graphen gelöscht werden soll. Dabei bewirkt diese Operation eine *Kaskadierung* von Elementlöschungen, falls das zu entfernende Element inzidente Verbindungen besitzt, oder als Graph weitere Elemente enthält. Der erste Fall entfernt somit „hängende Kanten“, also Verbindungsstrukturen, deren Quell- oder Zielelement durch die Löschoperation entfernt wird. Formal wird hierzu eine Folge von Verbindungen $\widetilde{l}_1, \ldots, \widetilde{l}_n$ ermittelt, die jeweils einander als Quell- oder Zielelement referenzieren. Das letzte Element der Folge verweist auf das durch die Operation zu löschende Element. Alle Elemente $\widetilde{l}_1, \ldots, \widetilde{l}_n$ werden implizit entfernt, was durch die Mengendifferenz $G \setminus G'$ beschrieben wird. Der zweite Fall bewirkt eine Existenzabhängigkeit aller Elemente von ihrem enthaltenen Graphen, was durch den transitiven Abschluss der Relation $\triangleright_{\mathcal{G}}$ beschrieben wird. In der folgenden Gleichung

sei $do \in elements(p) \cap DeletionOperator$ sowie v_{eff} die veränderte Variable.

$$
\begin{aligned}
(m, G, G', m') \in SEM[\![do]\!] \quad \Leftrightarrow \quad & \exists u \Big(u \in allElements(G) \setminus allElements(G') \\
\wedge \quad & (m, u) \in SEM[\![v_{eff}]\!] \quad \wedge \quad (m', u) \notin SEM[\![v_{eff}]\!] \\
\wedge \quad & \forall \widetilde{l}_1, \ldots, \widetilde{l}_n \Big((\forall_{j<n} \operatorname*{source}_G(\widetilde{l}_j) = \widetilde{l}_{j+1} \vee \operatorname*{target}_G(\widetilde{l}_j) = \widetilde{l}_{j+1}) \\
& \wedge \big(\operatorname*{source}_G(\widetilde{l}_n) = u \vee \operatorname*{target}_G(\widetilde{l}_n) = u \big) \rightarrow \widetilde{l}_i \in G \setminus G' \Big) \\
& \text{(Löschpropagation der Inzidenzeigenschaft)} \\
\wedge \quad & \forall \overline{u} \big(u \triangleright_G^+ \overline{u} \rightarrow \overline{u} \in G \setminus G' \big) \\
& \text{(Löschpropagation der Enthaltenseinsbeziehung)} \Big)
\end{aligned}
$$

Umleitung von Verbindungsstrukturen: Umleitungen von Verbindungsstrukturen werden unter Angabe einer Variablen für das gewünschte Quell- und Zielelement vorgenommen. Hierbei sei $ro \in elements(p) \cap RedirectOperation$, v_{eff} die verarbeitete Variable sowie v_{src}, v_{tgt} die referenzierten Quell- und Zielvariablen. Die Umleitung einer Verbindung l wird auf Basis der Hilfsfunktionen source bzw. target des Graphmodells definiert, die in Abschnitt 2.4 eingeführt worden sind. So wird gefordert, dass bei Auffindung eines zuzuweisenden Quellelements u_{src} die Funktion $\text{source}_{G'}$ angewendet auf die *Zielgraphstruktur* G' dieses als Ergebnis liefert. Die Zuordnung m bleibt an sich unverändert.

$$
\begin{aligned}
(m, G, G', m) \in SEM[\![ro]\!] \quad \Leftrightarrow \quad & \exists l \Big((m, l) \in SEM[\![v_{eff}]\!] \\
\wedge \quad & \forall u_{src} \big((m, u_{src}) \in SEM[\![v_{src}]\!] \rightarrow \operatorname*{source}_{G'}(l) = u_{src} \big) \\
\wedge \quad & \forall u_{tgt} \big((m, u_{tgt}) \in SEM[\![v_{tgt}]\!] \rightarrow \operatorname*{target}_{G'}(l) = u_{tgt} \big) \Big)
\end{aligned}
$$

4.4.5 Transformation geschachtelter Muster

Geschachtelte Muster werden grundsätzlich unter Auswahl *einer* Auffindungsstelle verarbeitet. Hierbei werden alle Operatoren eines geschachtelten Musters ohne Berücksichtigung von Mustergrenzen verarbeitet. Somit erfolgt die phasenweise Ausführung von Operatoren simultan innerhalb der gesamten Musterhierarchie.

Diese Vorgehensweise ist damit zu rechtfertigen, dass geschachtelte Musterstrukturen in erster Linie die Muster*suche* beeinflussen sowie nach deren Abschluss ggf.

zwischen verschiedenen gültigen Auffindungsstellen eines Musters eine Auswahl getroffen werden muss. Eine derart ausgedünnte Struktur kann wiederum als eine einzelne Auffindungsstelle interpretiert werden, die alle geschachtelten Stellen vereinigt.

Aggregierte Teilmuster

Abweichend von diesem Standardverhalten wird unter Angabe der Musterannotation Aggregation eine *quasi-simultane* Verarbeitung aller Auffindungsstellen eines derart gekennzeichneten Musters durchgeführt. Hierdurch werden die modellierten Operatoren auf alle gültigen Belegungen der veränderten Variablen angewendet, was wiederum musterübergreifend unter Beachtung der Phasen- und Stufeneinteilung erfolgt.

Ein Beispiel zur Transformation aggregierter Strukturen ist in Abbildung 4.27 dargestellt. Durch dieses Transformationsmuster werden einer Person alle Aufgaben zugeordnet, die Teilaufgabe einer bereits bearbeiteten sind. Zu beachten ist im dargestellten Ablauf, dass der Erstellungsoperator zunächst in allen Auffindungsstellen angewendet wird, bevor die Umleitung der erstellten Kanten zu ihren jeweiligen Zielknoten erfolgt. Auf diese Weise wird vermieden, dass Zwischenergebnisse aus der Transformation einzelner Auffindungsstellen bei der Bearbeitung weiterer referenziert werden.

Formale Definition

Eine formale Definition der Transformation mittels geschachtelter Muster wird in folgendem Ausdruck angegeben. Dieser setzt für *nicht-aggregierte* geschachtelte Muster voraus, dass zu diesem eine geeignete Auffindungsstelle m_i vorab ausgewählt wird. Die semantische Relation des Untermusters $SEM_{\underset{hrcl}{\textbf{transform}}}[\![p_{sub}]\!]$ beschreibt das Umschreiben der Auffindungsstelle zu m_i'. Für eventuelle vorhandene, tiefer geschachtelte Muster wird eine Folge $m_{i+1}..m_j$ von Auffindungsstellen angegeben. Sowohl für das enthaltende Muster als auch für das untergeordnete wird dabei auf denselben Arbeitsgraphen G zugegriffen. Es erfolgt also, gemäß obiger Beschreibung, keine se-

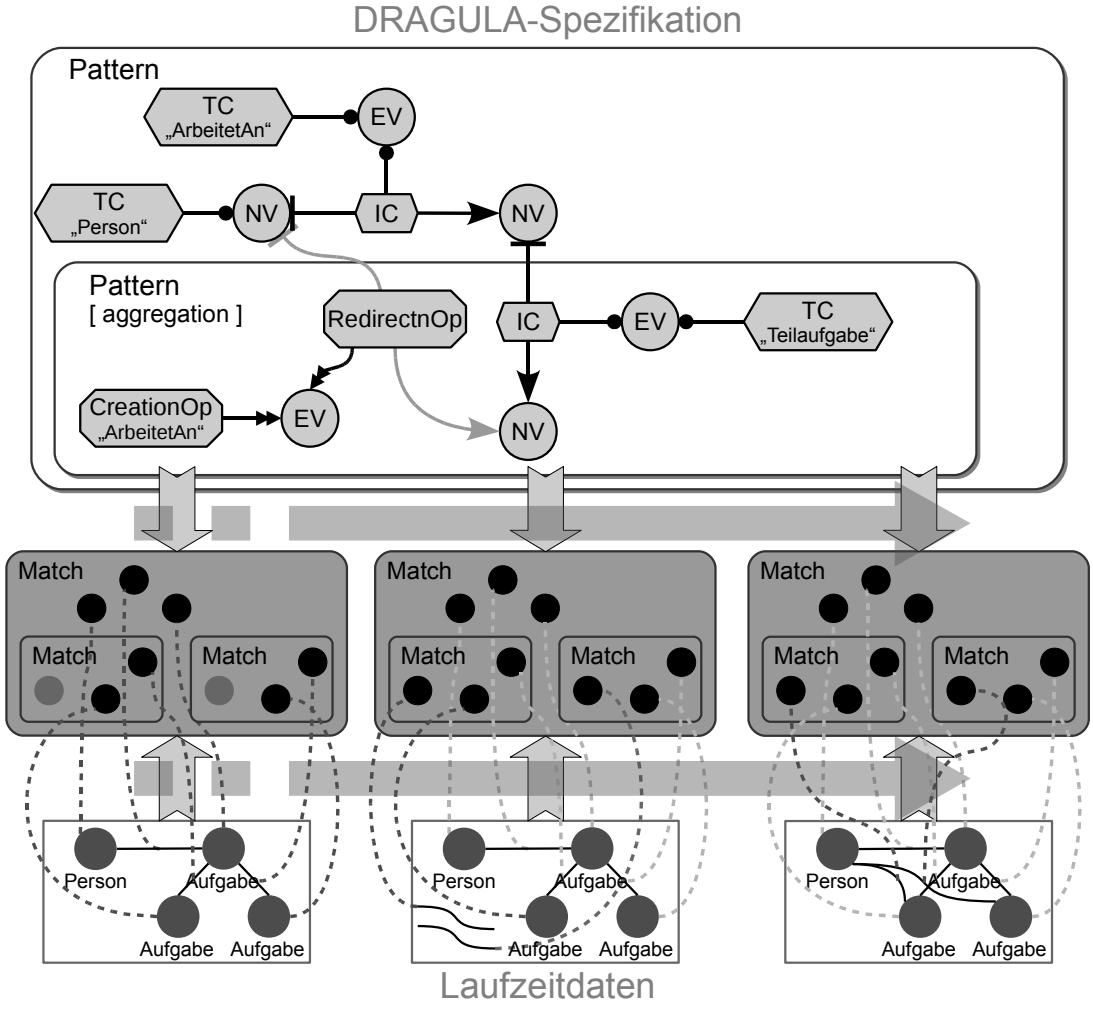


Abbildung 4.27: Beispiel zur aggregierten Transformation

quenzielle Abarbeitung der einzelnen Muster.

$$
\begin{aligned}
(m_{1..n}, G, G', m'_{1..n}) \quad &\in \quad SEM_{\substack{\textbf{transform} \\ hrcl}} [\![p]\!] \\
&\Leftrightarrow \quad (m_1, G, G', m'_1) \in SEM_{\textbf{transform}} [\![p]\!] \\
&\wedge \quad \forall p_{sub} \big(p_{sub} \in elements(p) \cap Pattern \rightarrow \\
&\qquad \exists_{i..j} (m_{i..j}, G, G', m'_{i..j}) \in SEM_{\substack{\textbf{transform} \\ hrcl}} [\![p_{sub}]\!] \big) \\
&\qquad \text{(Transformation von } p_{sub} \text{ auf } \textit{ausgewählter} \text{ Auffindungsstelle)} \\
&\wedge \quad \forall p_{aggr} \Big(p_{aggr} \in elements(p) \cap Pattern \\
&\qquad \wedge \exists a \big(a \in annotations(p_{aggr}) \cap Aggregation \big) \rightarrow \\
&\qquad \exists M_a, M'_a \Big((G, M_a) \in SEM_{\substack{valid \\ \textbf{search}}} [\![p_{aggr}]\!] \\
&\qquad\quad \wedge \forall m_a \exists m'_a \big(m_a \in M_a \rightarrow m'_a \in M'_a \\
&\qquad\qquad \wedge (m_a, m'_a) \in SEM_{\substack{\textbf{transform} \\ hrcl}} [\![p_{aggr}]\!] \big) \Big) \\
&\qquad \text{(Transformation von } p_{aggr} \text{ auf } \textit{allen} \text{ Auffindungsstellen)}
\end{aligned}
$$

Der zweite Teil der Gleichung behandelt die Definition aggregierter Untermuster. Hierbei wird *kein* Bezug auf die vorgegebenen Auffindungsstellen $m_1..m_n$ genommen. Stattdessen wird die Menge aller gültiger Stellen für p_{aggr}, M_a ermittelt. Darauf aufbauend fordert die Gleichung, dass für jedes $m_a \in M_a$ eine entsprechend transformierte Stelle m'_a existiert, die zum Ergebnis der gesamten Transformation von G nach G' beiträgt. Die beschriebene Transformation wird also auf alle gültigen Auffindungsstellen zugleich angewendet.

Zwischenfazit: In DRAGULA werden Graph*transformationen* mittels Operatoren definiert, die jeweils für sich abgeschlossene Veränderungen der Laufzeitdaten beschreiben. Dieser Ansatz ist Vorteilhaft gegenüber einer klassischen Regeldarstellung mittels separater linker und rechter Regelseiten, da keine Wiederholung der beibehaltenen Graphelemente zu erfolgen braucht. Zusätzlich vereinfacht dies eine automatisierte Verarbeitung und Erstellung von Spezifikationen, da hierzu bereits häufig beabsichtigte Graphoperationen bekannt sind, und diese somit direkt nach DRAGULA portiert werden können. Schließlich ist anzumerken, dass DRAGULA durch die reihenfolgeunabhängige Verarbeitung von Operatoren, auch über Mustergrenzen hinweg, eine hinreichende Abstraktion von der eigentlichen Operatorausführung vornimmt.

4.5 Ablaufsteuerung

Die bislang behandelten transformierenden Graphmuster bieten lediglich die Möglichkeit, einzelne Veränderungsschritte an einer Graphstruktur zu bewirken. Ergebnisse eines Transformationsschritts können dagegen bislang nicht *weiterverwendet* werden, um bspw. zyklische Operationen durchzuführen. Zahlreiche Graphalgorithmen, wie bspw. zur Findung kürzester Wege [Dij59], basieren jedoch auf einer iterativen Verarbeitung der untersuchten Strukturen. Die bisher angebotenen Veränderungsoperatoren einschließlich hierarchischer und aggregierter Muster reichen dazu nicht aus, da alle Operationen auf einem einzelnen Zustand der Laufzeitdaten erfolgen.

Dieser Abschnitt führt eine *Ablaufsteuerung* in DRAGULA ein, mittels derer anfragende und verändernde Muster durch *Kontrollflussbeziehungen* kombiniert werden können. Auf diese Weise werden iterative und rekursive Transformationsabläufe ermöglicht, zudem kann anhand des Erfolgs oder Misserfolgs einer Suchanfrage im Kontrollflusspfad verzweigt werden. Mittels *Datenflussbeziehungen* werden Variablenbelegungen zwischen Mustern ausgetauscht. Aufgrund der Pivotisierung vorbelegter Variablen kann so der Suchraum nachfolgender Anfragen reduziert und Graphmuster kompakter gestaltet werden.

Konzeptionell lehnen sich die in DRAGULA bereitgestellten Konstrukte an *hierarchischen Zustandsautomaten*[DHT96] an, mittels derer einzelne Anfrage- und Transformationsmuster sequenziell komponiert und hierarchisch geschachtelt werden können. Neben der regulären **`goto`**-Interpretation der damit modellierten Kontrollflussbeziehungen, kann zusätzlich eine **`call`**-Interpretation genutzt werden. Diese ermöglicht den Aufbau rekursiver Kontrollflussstrukturen. Zur Modellierung von Datenflussbeziehungen werden Konzepte der UML Aktivitätsdiagramme [Fow03, Kap. 11] und vergleichbarer Notationen verwendet.

Im Weiteren erfolgt zunächst die Vorstellung eines initialen Anwendungsbeispiels und die syntaktische Definition der angebotenen Sprachkonstrukte. Daran schließt sich allerdings eine zunächst informelle Vorstellung der Interpretation zur Laufzeit an, wohingegen die semantische Präzisierung nachgelagert erfolgt. Dieses Vorgehen wurde aufgrund der zahlreichen Zusammenhänge der einzelnen Sprachelemente und der daraus entstehenden Komplexität einzelner Erläuterungen gewählt.

4.5.1 Anwendungsbeispiel

Ein zunächst einfaches Anwendungsbeispiel zur Ablaufsteuerung in DRAGULA wird in Abbildung 4.28 gezeigt. Dieses Beispiel zerlegt den suchenden Anteil des äußeren Musters von Abbildung 4.21 in zwei unabhängige Regeln: Im ersten Schritt wird zunächst die Knotenvariable mit einem Element des Typs Person belegt. Dieser Wert

kann durch die dargestellten Datenflussbeziehungen als Parameter der umgebenden komponierten Regel vorbelegt sein, wobei in diesem Fall das Muster eine Typprüfung vornimmt. Schlägt diese Prüfung fehl, scheitert die Ausführung sowohl der geschachtelten als auch der komponierten Regel, andernfalls wird mit der Auswertung regulär fortgefahren. Falls keine Vorbelegung des Variablenwertes vorliegt, wird das Ergebnis der Mustersuche als Wert aller mittels Datenflussbeziehungen verbundenen Variablen verwendet – dies betrifft im dargestellten Fall die nachfolgend ausgeführte Regel.

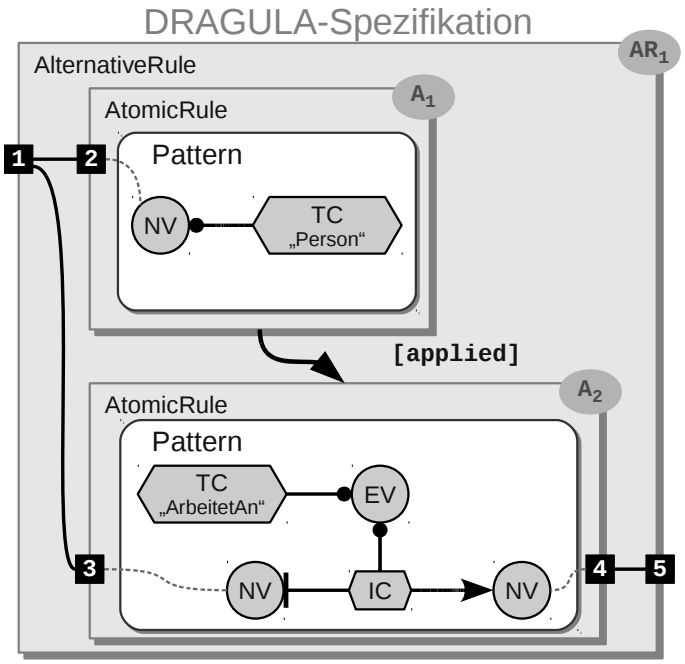


Abbildung 4.28: Initiales Beispiel zur Ablaufsteuerung

Im zweiten Schritt erfolgt die Traversierung der arbeitetAn-Verbindung. Dies erfolgt wiederum als Verbindungsprüfung falls die Zielvariable bislang aufgrund der zugeordneten Datenflussbeziehungen bereits an einen Wert gebunden ist, andernfalls wird ein geeigneter Wert ermittelt.

4.5.2 Syntaktische Definition

Die syntaktische Definition der DRAGULA Kontrollflussfunktionalität wird in Abbildung 4.29 gezeigt. Ein Regelelement (RuleElement) beschreibt eine Ausführungseinheit die durch Angabe einer Kontrollflussbeziehung angesprungen werden kann. Die zugehörige Spezialisierung AtomicRule verweist dabei auf ein Muster, dessen Auffindungsstellen bei Ausführung des Regelelements lediglich gesucht (transform-Attribut

ist nicht gesetzt) oder direkt durch Anwendung seiner Operatoren transformiert werden sollen (transform ist gesetzt). Handelt es sich bei dem transformierenden Muster nicht um eine Aggregation, so wird eine geeignete Auffindungsstelle automatisch und nicht-deterministisch ausgewählt. Bei Fehlschlag einer Mustersuche im weiteren Verlauf der Transformation kann diese Entscheidung zurück genommen und geeignet variiert werden.

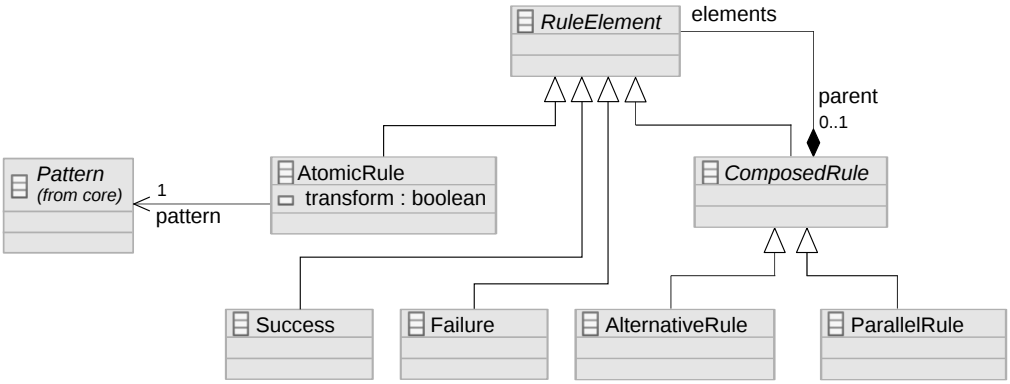


Abbildung 4.29: Metamodell zur Ablaufsteuerung, Regelkomposition

Zusammengesetzte Regeln (ComposedRule) komponieren Regelelemente, um komplexe Kontrollflussstrukturen modellieren zu können. Bei Anspringen eines solchen Elements werden diejenigen enthaltenen Elemente ausgeführt, die *keine* einlaufende Kontrollflussverbindung besitzen. Existieren mehrere in Frage kommende Startelemente, legt der konkrete Typ der zusammengesetzten Regel das genaue Verhalten fest. So ist es möglich mehrfache Startelemente *alternativ* auszuwählen (AlternativeRule), d.h. die Auswertung eines beliebigen Startelements durchzuführen, und bei Erfolg die übrigen infrage kommenden Startelemente zu ignorieren. Ferner wird eine quasiparallele Ausführung angeboten (ParallelRule), die nicht-deterministisch eine geeignete Reihenfolge der möglichen Startelemente ermittelt. Existiert lediglich eine einzige Startregel, unterscheidet sich das Verhalten dieser beiden Varianten nicht. Eine komponierte Regel gilt als abgearbeitet falls keiner der enthaltenen Ausführungspfade weiter abgearbeitet werden kann. Dies ist insbesondere dann der Fall, falls nach Abarbeitung einer Regel keine weiterführende Kontrollflussverbindung besteht.

Ferner stehen zwei gesonderte Regelvarianten Success und Failure zur Verfügung. Diese dienen, wie unten erläutert, zur expliziten Markierung des Erfolgsstatus einer komponierten Regel. Da sie kein eigenständiges Verhalten besitzen, eignen sie sich insbesondere zum Aufbrechen von Schleifen in Kontrollflussbeziehungen, bspw. zur Markierung eines eindeutigen Startelements.

4.5.3 Erfolgsstatus

Zur Verzweigung des Transformationsablaufs sowie zur Steuerung komponierter Regeln wird der Erfolgsstatus eines Regelaufrufs verwendet. Dieser richtet sich für atomare Regeln nach dem Vorhandensein einer gültigen Auffindungsstelle für das auszuwertende Graphmuster.

Für komponierte Regeln gilt der Status der zuletzt abgearbeiteten enthaltenen Regel, wobei durch Backtracking stets ein erfolgreiches Ergebnis herzustellen versucht wird: In einer AlternativeRule genügt dabei die erfolgreiche Abarbeitung ausgehend von einem nichtdeterministisch gewählten Startelement, für eine ParallelRule ist dagegen die erfolgreiche Abarbeitung von allen Startelementen erforderlich. Falls auf diese Weise kein erfolgreicher Regelablauf durch eine komponierte Regel ermittelt wird erfolgt Backtracking, wodurch sukzessive Graphveränderungen rückgängig gemacht und nichtdeterministische Entscheidungen – insbes. die Wahl einer Auffindungsstelle für atomare Regeln – neu getroffen werden. In ParallelRule wird zudem die Abarbeitungsreihenfolge der einzelnen Startelemente variiert um eine erfolgreiche Gesamtauswertung zu erreichen. Erst wenn auch auf diese Weise insgesamt kein erfolgreicher Ablauf herzustellen ist gilt die gesamte komponierte Regel als gescheitert.

Eine Success-Regel wird stets erfolgreich ausgewertet. Eine solche Regel kann daher genutzt werden um den Erfolgsstatus einer umgebenden komponierten Regel festzulegen, und damit Backtracking zu vermeiden. Dagegen schlägt eine Failure-Regel stets fehl, wobei diese *kein* Backtracking auslöst. Hierdurch kann also eine komponierte Regel als fehlgeschlagen markiert werden, ohne alle möglichen Pfade prüfen zu müssen.

4.5.4 Kontrollfluss

Der zweite Teil des DRAGULA Kontrollflusskonzepts formalisiert in Abbildung 4.30 die Begriffe der Kontroll- und Datenflussbeziehungen. Eine ControlFlow-Beziehung setzt jeweils zwei Regeln miteinander in eine entsprechende Beziehung. Hierbei kann der *Erfolgsstatus* der vorgelagerten Regel überprüft werden (Attribut applied), der bei erfolgter Transformation bzw. dem Finden zumindest einer gültigen Auffindungsstelle eines Musters gegeben ist. Der Status komponierter Regeln ergibt sich aus deren *zuletzt* ausgeführten enthaltenen Element. Nur falls der ermittelte Erfolgsstatus und der angegebene Attributwert übereinstimmen wird die Kontrollflussbeziehung zur Weiterschaltung berücksichtigt.

Je nach Kontext der ControlFlow-Verbindung wird diese unterschiedlich interpretiert. DRAGULA unterstützt dabei sogen. Übergangs- als auch Aufrufverweise, die den klassischen Konzepten sequenzieller Ausführung und Unterprogrammaufruf entspre-

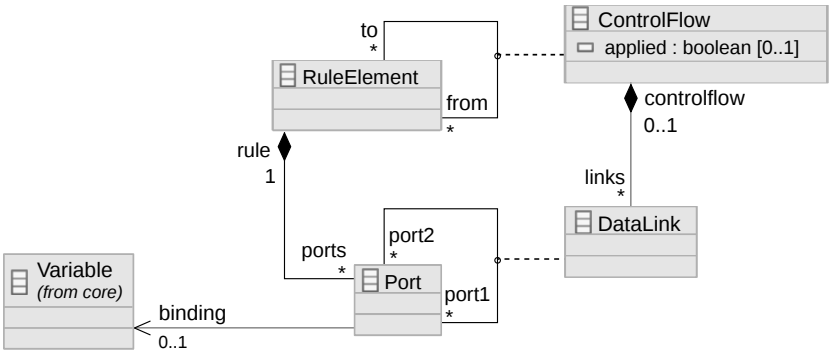


Abbildung 4.30: Metamodell zur Ablaufsteuerung, Kontroll- und Datenfluss

chen. Beide Varianten werden im Folgenden vorgestellt.

Kontrollfluss als Übergangsverweis

Handelt es sich bei Start- und Zielregeln um Elemente der *selben* komponierten Regel, so erfolgt eine sogenannte **goto**-Interpretation. Derartige Verbindungen werden im Folgenden auch als *lokaler Kontrollfluss* bezeichnet. Hierbei wird nach Abarbeitung des Startelements die Kontrolle an die Zielregel übergeben. Sollte diese keine weiteren Nachfolger besitzen, so terminiert die Ausführung der übergeordneten komponierten Regel.

Die in Abbildung 4.28 gezeigte Kontrollflussbeziehung stellt einen Übergangsverweis dar, da beide Regeln in einer komponierten Regel enthalten sind. Einen iterativen Ablauf mittels Aufrufverweise modelliert Abbildung 4.31. Die obere atomare Regel ermittelt alle Teilaufgaben, die über eine Enthaltenseinsstufe einer Person zugeordnet sind. Hierzu wird die Person vorab festgelegt, und von dort aus Kanten des Typs arbeitetAn und Teilaufgabe traversiert. Das Untermuster sichert als negative Anwendbarkeitsbedingung zu, dass die Person nicht bereits an der so ermittelten Teilaufgabe arbeitet.

Falls eine entsprechende Auffindungsstelle gefunden werden kann, führt das zweite Muster die entsprechende Transformation durch. Bei Fehlschlag der Mustersuche wird zu einem Success-Element verzweigt. Da dieses keinen eigenen Nachfolger besitzt, beendet die umgebende komponierte Regel mit erfolgreichem Status. Zur Findung einer geeigneten Startregel ohne einlaufende Kontrollflussbeziehung wird zudem ein zusätzliches Success-Element benötigt.

Anmerkung zu wohlgeformten Ablaufgraphen: Die hier vorgestellte Interpretation von Kontrollflussbeziehungen erlaubt eine vergleichsweise freie Modellierung von

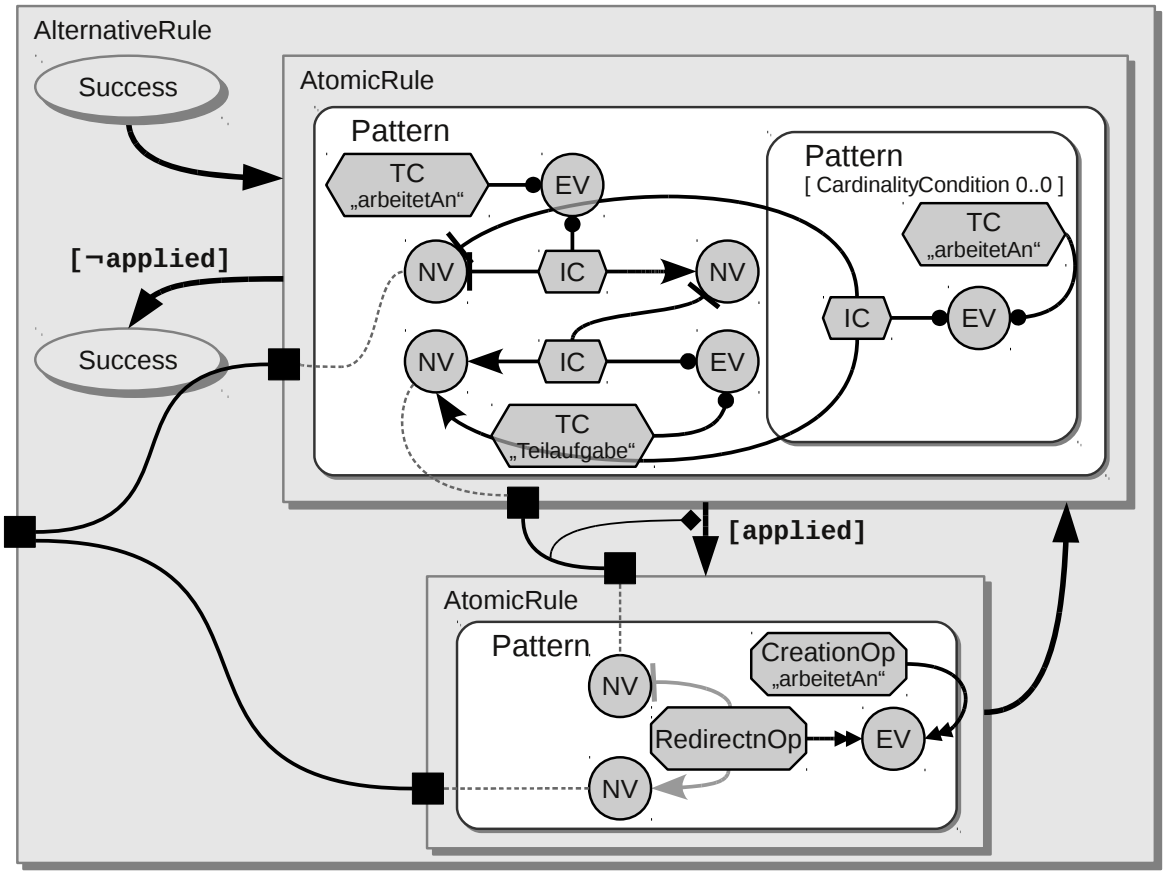


Abbildung 4.31: Iterativer Kontrollfluss mit Erfolgsprüfung

Übergangsrelationen zwischen Regelelementen einer gemeinsamen übergeordneten Regel. Möglich ist dabei auch die Spezifikation von Verbindungsstrukturen, die Wohlgeformtheitsansprüchen im Sinne der strukturierten Programmierung [DDH72] widersprechen. Da solche Strukturen von der bereitgestellten Ausführungsmaschine ausgewertet werden können, wird von deren Verwendung lediglich *informell* abgeraten. Dennoch sollte bei der Spezifikation einer komponierten Transformationsregel seitens des Entwicklers Wert auf Verständlichkeit gelegt werden.

Kontrollfluss als Aufrufverweis

Alternativ zum ersten Fall wird eine **`call`**-Interpretation vorgenommen falls die Kontrollflussbeziehung auf eine Regel einer *anderen* enthaltenden Regel verweist. Anders als beim bisher betrachteten Übergangsverweis verbleibt die Ablaufkontrolle nur solange beim aufgerufenen Regelelement, bis dessen Ausführung terminiert. Anschließend wird mit lokalen Kontrollflussbeziehungen des *aufrufenden* Regelelements fort-

gefahren.

Bei der Modellierung von Aufrufbeziehungen ist zu beachten, dass aufgerufene Regeln *keine auslaufende* Kontrollflussverbindung besitzen. Es kann also kein Regelteil innerhalb einer Kontrollflussbeziehung einzeln ausgeführt werden, da dies insbesondere im Bezug auf Datenflussbeziehungen kaum handhabbar wäre. Zusätzlich wird verlangt, dass es sich bei allen *einlaufenden* Verbindungen um Aufrufe, und nicht etwa um lokale Beziehungen handelt. Intern kann bspw. eine komponierte Regel jedoch beliebig gestaltet sein. Die Einschränkungen werden in den folgenden Wohlgeformtheitsbedingungen formalisiert.

```
context ControlFlow
def isGoto : Boolean = self.to.parent = self.from.parent
def isCall : Boolean = not self.isGoto()

inv validCall : self.isCall().implies(
      self.to.collect( r | r.to )→isEmpty()) and
      self.to.collect( r | r.controlFlow[to] )→
        forAll( cf | cf.isCall()))
```

Kontrollflussbeziehungen müssen grundsätzlich eindeutig bezüglich ihrer Art und des gewählten Erfolgsstatus sein, da andernfalls keine eindeutige Abarbeitungsreihenfolge definiert werden könnte. Somit kann ein Regelelement bspw. höchstens einen ausgehenden Goto-Kontrollfluss besitzen falls dessen applied-Eigenschaft nicht geprüft wird, oder alternativ höchstens je einen solchen Kontrollfluss mit positiver und negativer Prüfung.

```
context RuleElement
inv uniqueSuccessor :
      ( let cf : Collection(ControlFlow) =
          self.controlFlow[to]→select( c | c.isGoto() ) :
        cf→exists( c | c.applied.oclIsUndefined() )
          implies cf→isUnique() and
        cf→isUnique( c | c.applied ))
      and
        -- [...] c.isCall analog --
```

4.5.5 Datenfluss

Zusätzlich Bestandteil der Abbildung 4.30 ist die Definition von Datenflussbeziehungen zwischen DRAGULA Regeln. Dies erfolgt primär durch die Angabe von Schnittstellen (Modellelement Port), die in atomaren Regeln direkt an Variablen des auszu-

wertenden Musters gebunden werden. Wie die nachfolgenden Bedingungen formulieren, kann auch eine Bindung an Variablen geschachtelter Muster erfolgen. Schnittstellen einer atomaren Regel werden dahingehend interpretiert, dass diese die Belegung einer Variable einerseits nach außen sichtbar *freigeben*, andererseits auch eine Belegung von außen *festlegen*.

```
context Port
inv wellformedType : not self.binding.oclIsUndefined() =
      self.rule.oclIsTypeOf(AtomicRule)
inv wellformedBinding : not self.binding.oclIsUndefined() implies
      self.rule.oclAsType(AtomicRule).pattern.allElements()→
        contains(self.binding)
```

Verbindungen zwischen Schnittstellen stellen *Datenflussbeziehungen* (DataLink) dar, womit Variablenbelegungen zwischen Regelelementen übertragen werden können. Diese Beziehungen werden der jeweils traversierten Kontrollflussbeziehung zugeordnet, was graphisch durch einen Kompositionspfeil dargestellt wird. Datenflüsse zwischen ineinander geschachtelten Regeln werden dagegen keinem Kontrollfluss zugeordnet.

Datenflussbeziehungen besitzen in DRAGULA *keinerlei Richtung*, unterscheiden somit also nicht zwischen lesenden und schreibenden Zugriffen. Eine Datenflussbeziehung bewirkt bei Auswertung eine Zuordnung identischer Werte an die jeweils verbundenen Variablen. Bindet also eine erfolgreiche Mustersuche ein Graphelement an eine Variable, so stellt eine darauf verweisende Schnittstelle diesen Wert nach außen zur Verfügung. Dieser Wert wird an alle Schnittstellen propagiert, die über Datenflussbeziehungen des verfolgten Kontrollflusspfades erreichbar sind. Hierzu belegt eine nachfolgende (atomare) Regel die entsprechende Variable des auszuwertenden Musters vor, weist dieser also für die Mustersuche einen konstanten Wert zu.

Datenflussbeziehungen können in zwei Anwendungsfällen auftreten, die beide eine in unten angegebener Wohlgeformtheitsbedingung beschrieben werden. Mittels Datenflussbeziehungen die *keiner* Kontrollflussbeziehung zugeordnet sind (lokaler Datenfluss) kann eine Regel auf Variablenbelegungen ihrer enthaltenden komponierten Regel zugreifen. Dies erfordert, dass die Schnittstelle eines Endes (bspw. port1) zu einer Regel gehört, deren enthaltende Regel die Schnittstelle des anderen Verknüpfungsendes (bspw. port2) enthält. Auf diese Weise werden Variablenbelegungen entlang der Strukturhierarchie der beteiligten Regeln propagiert.

Ferner kann eine Datenflussbeziehung nur bei Traversierung eines zugeordneten Kontrollflusses berücksichtigt werden (Parameter-Datenfluss). Hierbei ist ebenfalls sicherzustellen das die Zielregel der Kontrollflussbeziehung einem Ende der Datenflussbeziehung entspricht. Ferner kann die Datenflussbeziehung lediglich Schnittstel-

len referenzieren von deren zugehörigen Regelelementen aus ein *positiver Kontrollflusspfad* zur Zielregel existiert. Ein solcher Pfad beginnt mit einer positiven Kontrollflussbeziehung, und wird über beliebig viele sowohl positive als auch negative Schritte fortgesetzt. Die Einschränkung auf über positive Pfade verlassene Regeln gewährleistet, dass eine Schnittstellenbelegung der referenzierten Variablen vorliegen *kann*[6], eine Übertragung dieser Belegung auf das Zielelement der Kontrollflussbeziehung also möglich ist. Würde die Ausgangsregel über eine negierte Transition verlassen, würde das die Abwesenheit einer gültigen Auffindungsstelle und damit einer Schnittstellenbelegung bewirken.

```
context RuleElement
def allSucc : Set(RuleElement) = self.to→union(self.to→
      collect( f | f.allSucc()))
def positiveSucc : Set(RuleElement) = self.to→select(s |
      self.controlflow[to].applied)→collect(ps | ps.allSucc())

context DataLink
inv wellformed : ( self.controlflow.oclIsUndefined() and
          self.port1.rule.parent.ports→contains(self.port2) and
          -- [...] Umgekehrte Rollenbezeichner analog --
      ) or ( self.controlflow.to = self.port2.rule and
          self.port1.rule.positiveSucc()→
            contains(self.controlflow.from)
      ) or ( -- [...] Umgekehrte Rollenbezeichner analog --
      )
```

Abbildung 4.32 illustriert zulässige Datenflussbeziehungen im Bezug auf die letzte Kontrollflussbeziehung in der dargestellten Sequenz. Ohne weitere Zuordnung können Schnittstellen der enthaltenden komponierten Regel referenziert werden. Zudem kann die letzte aufgerufene Regel auf das Ergebnis der ersten Regel zugreifen, da ein positiver Kontrollflusspfad vorhanden ist.

Datenfluss bei rekursiven Aufrufverweisen

Ein Beispiel für Datenflussbeziehungen bei *rekursiven* Aufrufen zeigt Abbildung 4.33[7]. Im dargestellten Fall sollen einer Person alle (transitiven) Unteraufgaben der aktuell bearbeiteten Aufgabe zugeordnet werden. Anders als in Abschnitt 4.3.4 wird die Suche nach allen Unteraufgaben nicht mit einem geschachtelten Muster und Mus-

[6] Da u.U. verschiedene potenzielle Kontrollflusspfade vorliegen, handelt es sich hierbei um eine notwendige, jedoch nicht hinreichende Bedingung.

[7] Die dargestellte Nummerierung der Schnittstellen ist rein illustrativ.

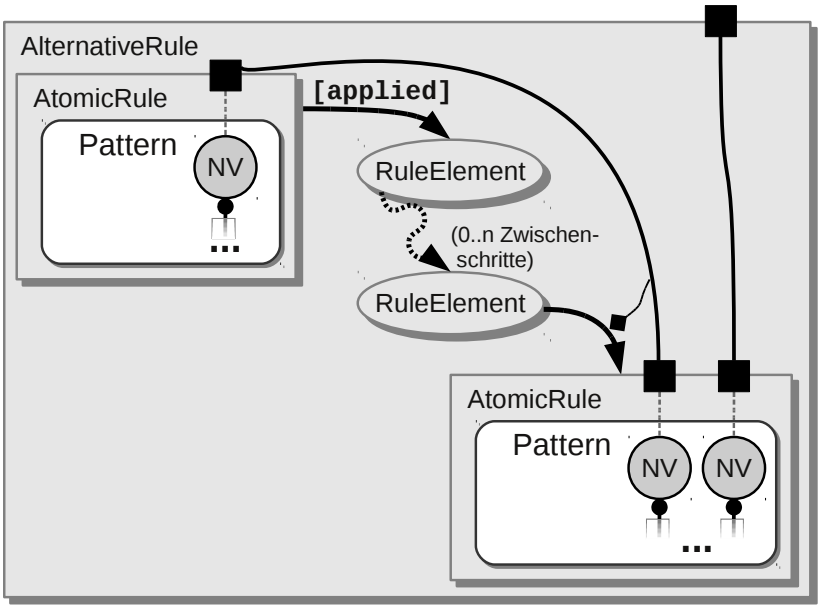


Abbildung 4.32: Zulässige Datenflussbeziehungen

terreferenzen ausgedrückt, sondern durch getrennte Regeln mit rekursivem Aufruf.

Als Eingabeparameter der gesamten Regelstruktur dient Schnittselle `1`, über die eine zu bearbeitende Person vorgegeben wird. Bei erfolgreicher Auswertung des Musters der oberen atomaren Regel wird mittels Übergangsverweis an die komponierte Regel verzweigt. Hierbei wird die ermittelte Aufgabe über die Schnittstellen `2` und `3` übertragen.

Die Ausführung der aufgerufenen komponierten Regel beginnt mit einer atomaren Regel zur Ermittlung von Unteraufgaben. Die übergeordnete Aufgabe wird durch die Schnittstelle `4` vorbelegt und entspricht initial dem Resultat der oberen atomaren Regel. Falls eine Teilaufgabe gefunden werden kann, wird dieser Wert an Schnittstelle `5` gebunden. Der Kontrollfluss verzweigt an die untere Success-Regel, die ihrerseits sowohl einen Aufruf- als auch einen Übergangsverweis als mögliche Nachfolgerbeziehungen besitzt. In diesem Fall wird zunächst der Aufrufverweis ausgewertet da der Kontrollfluss anschließend zum Aufrufer zurückkehrt.

Die untere atomare Regel dient zur Verknüpfung der gegebenen Person mit der ermittelten Teilaufgabe. Die notwendigen Eingabedaten werden durch die jeweiligen Schnittstellen `6` bzw. `7` aus den vorangegangenen Regelaufrufen übertragen. Anschließend kehrt der Kontrollfluss an die Success-Regel zurück, die ihrerseits an eine zweite leere Regel weiterschaltet.

Durch die zweite Success-Regel erfolgt schließlich der rekursive Aufruf der kompo-

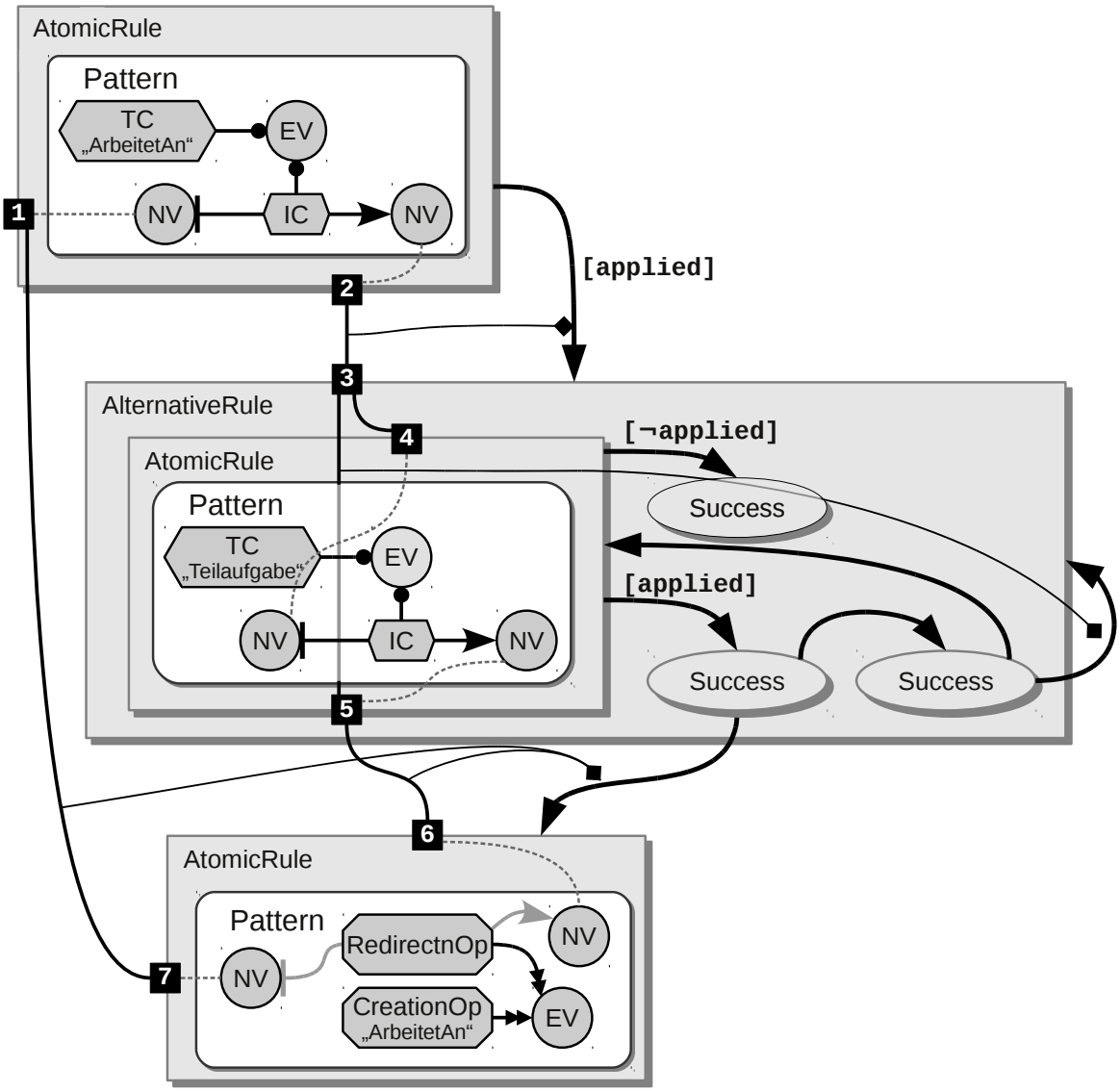


Abbildung 4.33: Rekursiver Kontrollfluss

nierten Regel. Hierzu wird diese per Aufrufverweis angesprungen, wobei die zuvor ermittelte Teilaufabe als übergeordnete Aufgabe des rekursiven Aufrufs übergeben wird. Nach Beendigung des rekursiven Aufrufs wird per Übergangsverweis erneut die atomare Regel angesprungen um weitere direkte Teilaufgaben auszuwerten.

Die Rekursion endet wenn zu einer gegebenen Aufgabe keine Teilaufgabe gefunden werden kann. Das Suchmuster wird dadurch nicht erfolgreich ausgewertet, wodurch zur oberen Success-Regel verzweigt wird. Diese besitzt keinen Nachfolger und beendet die Auswertung der komponierten Regel daher erfolgreich[8].

4.5.6 Semantische Definition

Die bislang lediglich informell eingeführten Sprachkonstrukte zur Ablaufsteuerung werden an dieser Stelle semantisch ausformuliert. Wie in den vorherigen Abschnitten wird dabei eine Familie von Funktionen SEM eingesetzt, für die pro Metamodellelement entsprechende Exemplare angegeben werden.

Zunächst wird eine Hilfsstruktur zur Darstellung von Ausführungszuständen definiert. Anschließend wird nach einer Top-Down Vorgehensweise mit der Definition komponierter Regeln begonnen, woran sich die Erläuterung atomarer Regeln anschließt. In beiden Fällen werden weitere semantische Relationen für Datenflüsse, Kontrollflüsse und Ableitungsschritte verwendet, die abschließend definiert werden.

Hilfsstrukturen zur Zustandsrepräsentation

Ein Ausführungszustand (`runtime-state`, kurz `RS`) beschreibt für den zugrundeliegenden Laufzeitgraphen eine Belegung von Schnittstellen mit dessen enthaltenen Entitäten. Mittels dieser Hilfsstruktur werden im Folgenden Datenflüsse zwischen Regeln beschrieben.

$$RS_{\mathcal{G}} : Port \times U_{\mathcal{G}}$$

Um darüber hinaus rekursive Regelaufrufe darstellen zu können, wird das Ausdrucksmittel eines Zustands*kellers* benötigt, welcher alle Schnittstellenbelegungen in Form von Aufrufrahmen aggregiert. Hierzu wird, in Anlehnung an algebraische Spezifikationsansätze wie etwa in [HL89], der Datentyp `STS` (`state-stack`) verwendet. Die Funktionen $push : STS_{\mathcal{G}} \times RS_{\mathcal{G}} \to STS_{\mathcal{G}}$ sowie $pop : STS_{\mathcal{G}} \to STS_{\mathcal{G}}$, $top : STS_{\mathcal{G}} \to RS_{\mathcal{G}}$, und $empty : STS_{\mathcal{G}} \to \{\mathbf{true}, \mathbf{false}\}$ sind wie üblich zu interpretieren – auf eine formale Definition wird an dieser Stelle verzichtet. Zusätzlich liefert die Funktion

[8] Ohne diese zweite Success-Regel würde der rekursive Aufruf als fehlgeschlagen gelten und die Transformation insgesamt abgebrochen werden.

$first : STS_{\mathcal{G}} \times Port \to U_{\mathcal{G}} \cup \{\bot\}$ die Belegung u einer Schnittstelle p im *obersten* Ausführungszustand des Zustandskellers s, in dem diese Schnittstelle referenziert wird. Definiert keiner dieser Zustände eine solche Belegung, ist das Ergebnis der Funktion das Symbol $\bot$. Dies lässt sich wie folgt algebraisch beschreiben:

$$first(s,p) \quad = \quad \begin{cases} u & \text{falls } \neg empty(s) \wedge (p,u) \in top(s) \\ first(pop(s),p) & \text{falls } \neg empty(s) \wedge (p,u) \notin top(s) \\ \bot & \text{sonst} \end{cases}$$

atomare Regeln

Eine atomare Regel (in Zeichen *ac*) setzt zwei Paare bestehend aus einem Laufzeitgraphen und einem Ausführungszustand in eine Vorher-Nachher-Relation:

$$SEM_{AtomicRule} : AtomicRule \to \left(LG(GS) \times STS_{\mathcal{G}}\right)^2$$

Die Ausführung einer atomaren Regel beinhaltet die Ermittlung einer geeigneten Auffindungsstelle für das referenzierte Muster sowie die entsprechende Transformation der Graphstruktur G. Zur Verarbeitung *geschachtelter* Muster wird eine *Folge* von Auffindungsstellen $m_{1..n}$ entsprechend der Musterhierarchie bestimmt. Diese müssen dabei konform zum Ausführungszustand sein, der durch die Kellerstruktur $sts_{\mathcal{G}}$ gegeben ist. Hierzu wird gefordert, dass für jede Schnittstellenbelegung in $sts_{\mathcal{G}}$ die zugehörige Variable v_{fix} in allen Auffindungsstellen m_i an den vorgegebenen Wert u gebunden ist. Durch Ausführung einer Transformationsregel wird der Ausführungszustand dahingehend verändert, dass bislang nicht belegte Schnittstellen der atomaren Regel an Werte aus m_i' gebunden werden. Auch werden durch Löschung von Graphelementen während der Transformationsausführung Schnittstellenbelegungen entfernt. Im diesem Fall erfolgt allerdings eine simultane Aktualisierung aller Schnittstellenbelegungen des Zustandskellers, ohne das eine Propagation entlang von Datenflussbeziehungen nötig wäre.

Die nachfolgende Bedingung formalisiert obige Beschreibung. Der erste Teilausdruck beschreibt die Vorbelegung von Variablen der geschachtelten Auffindungsstellen $m_{1..n}$ gemäß des aktuellen Ausführungszustands. Anschließend wird die Aktualisierung der Schnittstellenbelegungen beschrieben. Ferner gilt die Bedingung, dass alle während des Transformationsschritts gelöschten Graphelemente aus etwaigen

Schnittstellenbelegungen des resultierenden Ausführungszustands entfernt werden.

$$
\begin{aligned}
(G, sts_{\mathcal{G}}, G', sts'_{\mathcal{G}}) \in SEM[\![ac]\!] \quad \Leftrightarrow \quad & \exists m_{1..n}, m'_{1..n} \Big(\\
& \forall v_{fix}, p \Big(v_{fix} = binding(p) \wedge \bot \neq first(sts_{\mathcal{G}}, p) \rightarrow \\
& \quad \exists i, u \big((m_i, u) \in SEM[\![v_{fix}]\!] \wedge u = first(sts_{\mathcal{G}}, p) \big) \Big) \\
\wedge \quad & (m_{1..n}, G, G', m'_{1..n}) \in SEM_{\substack{\textbf{transform} \\ hrcl}}[\![pattern(ac)]\!] \\
\wedge \quad & \forall m'_i, v, p, u \big((m, u) \in SEM[\![v]\!] \wedge v = binding(p) \rightarrow \\
& \quad (p, u) \in top(sts'_{\mathcal{G}}) \big) \\
\wedge \quad & \forall \overline{u} \Big(\overline{u} \in G \setminus G' \rightarrow \nexists p \big(\overline{u} = first(sts'_{\mathcal{G}}, p) \big) \Big) \Big)
\end{aligned}
$$

Komponierte Regeln

Die semantische Definition komponierter Regeln setzt jeweils zwei Ausführungszustände und Graphstrukturen in eine Vorher-Nachher-Relation. Die beiden Auswertungsvarianten AlternativeRule und ParallelRule sind dabei größtenteils identisch, so dass nur die erste Variante (in Zeichen *ar*) explizit definiert wird. Auf Unterschiede zwischen beiden Varianten wird fallweise hingewiesen.

$$SEM_{AlternativeRule} : AlternativeRule \rightarrow \big(LG(GS) \times STS_{\mathcal{G}} \big)^2$$

Komponierte Regeln werden basierend auf den jeweils enthaltenen Teilregeln definiert. Hierbei wird von den jeweiligen Startelementen ι eine Ableitungssequenz bis zu einem Regelelement ohne auswertbare Nachfolgerregeln ω bestimmt. Für eine AlternativeRule wird gefordert, dass zu mindestens *einer* Startregel ι eine Ableitungssequenz zwischen Eingabe- zum Ausgabezustand ($G^{\mathbf{i}}$ bzw. $G^{\mathbf{o}}$) existiert (Existenzquantifizierung). Für eine ParallelRule müssen von allen Startregeln entsprechende Ableitungssequenzen vorhanden sein (Allquantifizierung).

Lokale Datenflüsse werden in der angegebenen Definition durch gesonderte „interne“ Ausführungszustände $\widetilde{sts}^{\mathbf{i}}_{\mathcal{G}}$ und $\widetilde{sts}^{\mathbf{o}}_{\mathcal{G}}$ berücksichtigt. Im Fall der Eingabedaten geht der interne Zustand aus der Auswertung derjenigen Datenflussbeziehungen DF_{ar} hervor, die eine Schnittstelle von *ar* mit einer Schnittstelle eines Regelelements in Beziehung setzen, das in *ar* enthalten ist[9].

$$DF_{ar} := \big\{ df | port_1(df) \in ports(ar) \wedge \exists r \big(port_2(df) \in ports(r) \wedge r \in elements(ar) \big) \big\}$$

[9] Hierdurch wird also bspw. die Übertragung von den Schnittstellen 3 zu 4 in Abbildung 4.33 beschrieben.

Die nachfolgende Bedingung fasst die bisherige Definition komponierter Regeln zusammen. Nach Auswertung der internen Eingabedatenflüsse wird eine beliebige Anzahl an Ableitungsschritten vorgenommen, die im Folgenden definiert werden. Von dem so erreichten Ausführungszustand und Laufzeitgraphen wird die letzte ausführbare Regel ω ausgewertet. Mit erneuter Auswertung der lokalen Datenflussbeziehungen werden zusätzliche Ausgabebelegungen an die übergeordnete komponierte Regel übertragen. Ferner verlangt die Bedingung, dass nach Auswertung der Regel ω keine weiteren Ableitungsschritte möglich sind.

$$\begin{aligned}(G^{\mathbf{i}}, sts^{\mathbf{i}}_{\mathcal{G}}, G^{\mathbf{o}}, sts^{\mathbf{o}}_{\mathcal{G}}) \in SEM[\![ar]\!] \quad \Leftrightarrow \quad & \exists \iota, \omega, G', \widetilde{sts}^{\mathbf{i}}_{\mathcal{G}}, \widetilde{sts}^{\mathbf{o}}_{\mathcal{G}}, sts'_{\mathcal{G}} \Big(\iota \in initial(ar) \\ \wedge \quad & \forall df \big(df \in DF_{ar} \rightarrow (sts^{\mathbf{i}}, \widetilde{sts}^{\mathbf{i}}) \in SEM[\![df]\!] \big) \\ \wedge \quad & (\iota, G^{\mathbf{i}}, \widetilde{sts}^{\mathbf{i}}_{\mathcal{G}}) \vdash^{*} (\omega, G', sts'_{\mathcal{G}}) \\ \wedge \quad & (G', sts'_{\mathcal{G}}, G^{\mathbf{o}}, \widetilde{sts}^{\mathbf{o}}_{\mathcal{G}}) \in SEM[\![\omega]\!] \\ \wedge \quad & \nexists r, G'', sts''_{\mathcal{G}} \big((\omega, G^{\mathbf{o}}, \widetilde{sts}^{\mathbf{o}}_{\mathcal{G}}) \vdash^{+} (r, G'', sts''_{\mathcal{G}}) \big) \\ \wedge \quad & \forall df \big(df \in DF_{ar} \rightarrow (\widetilde{sts}^{\mathbf{o}}, sts^{\mathbf{o}}) \in SEM[\![df]\!] \big) \Big)\end{aligned}$$

Leere Regeln

Leere Regeln zur Darstellung von Erfolg oder Fehlschlag einer Transformation werden analog zu komponierten und atomaren Regeln als Relationen $SEM_{Success}$ bzw. $SEM_{Failure}$ definiert.

Erfolgreiche leere Regeln verändern weder den Laufzeitgraphen noch den Ausführungszustand, weshalb die semantische Relation $SEM[\![sc]\!]$ alle identischen Tupel enthält:

$$SEM[\![sc]\!] = \big\{ (G, sts_{\mathcal{G}}, G, sts_{\mathcal{G}}) \big\}$$

Für fehlschlagende Regeln ist die semantische Relation $SEM[\![fl]\!]$ leer, so dass lediglich negative Kontrollflussbeziehungen weiter verfolgt werden.

$$SEM[\![fl]\!] = \emptyset$$

Kontrollflussbeziehungen

Eine Kontrollflussbeziehung modelliert semantisch den Übergang eines aktiven Regelelements auf ein nachfolgendes sowie eine Veränderung des Ausführungszustands anhand von Datenflussbeziehungen.

$$SEM_{ControlFlow} : ControlFlow \rightarrow \big(RuleElement \times STS_{\mathcal{G}} \big)^2$$

Ein Kontrollflussübergang beinhaltet die Auswertung derjenigen Datenflussbeziehungen DF_{cf}, die der Kontrollflussbeziehung direkt zugeordnet sind sowie bei Übergangsverweisen zusätzlich lokale Datenflussbeziehungen zwischen übergeordnetem und angesprungenem Regelelement.

$$\begin{aligned} DF_{cf} \;:=\; \Big\{ df \quad \| \quad & \big(df \in links(cf)\big) \\ \vee \quad & \big(port_1(df) = ports(to(cf)) \wedge port_2(df) \in ports(parent(to(cf)))\big) \Big\} \end{aligned}$$

Der modellierte Übergang betrifft also neben dem Wechsel der aktiven Regel r zu r' eine Umschreibung des Ausführungszustands gemäß aller relevanter Datenflussbeziehungen. Diese werden im folgenden Abschnitt präzisiert.

$$\begin{aligned} (r, sgs_{\mathcal{G}}, r', sgs'_{\mathcal{G}}) \in SEM[\![cf]\!] \;\Leftrightarrow\; & r = from(cf) \wedge r' = to(cf) \\ \wedge \quad & \forall df \big(df \in DF_{cf} \rightarrow \\ & \quad (sgs_{\mathcal{G}}, sgs'_{\mathcal{G}}) \in SEM[\![df]\!]\big) \end{aligned}$$

Datenflussbeziehungen

Die Auswertung von Datenflussbeziehungen verändert die aktuelle Belegung der zugehörigen Schnittstellen die im obersten Element von $STS_{\mathcal{G}}$ festgehalten wird. Semantisch setzt eine Datenflussbeziehung daher zwei dieser Zustandskeller zueinander in Beziehung.

$$SEM_{DataFlow} : DataFlow \rightarrow (STS_{\mathcal{G}})^2$$

Falls eine Schnittstelle nicht im aktuellen Ausführungszustand belegt wird, kann auch in früheren Zuständen innerhalb des Zustandskellers gesucht werden. Hierzu wird die oben definierte Funktion $first$ genutzt. Seien $p_1 = port_1(df)$ bzw. $p_2 = port_2(df)$ die durch die Datenflussbeziehung verbundenen Schnittstellen sowie $\tau' = top(sts'_{\mathcal{G}})$ das oberste Element des Zustandskellers $sts'_{\mathcal{G}}$. Existiert für eine der Schnittstellen ein Wert in $sts_{\mathcal{G}}$, so wird dieser als Wert der jeweils anderen im aktuellen Ausführungszustand festgehalten. Innerhalb eines Aufrufrahmens in $sts_{\mathcal{G}}$ sind Schnittstellenbelegungen nicht variabel, weshalb eine vorhandene Belegung in $top(sts_{\mathcal{G}})$ nicht verändert wird.

$$\begin{aligned} (sts_{\mathcal{G}}, sts'_{\mathcal{G}}) \in SEM[\![df]\!] \;\Leftrightarrow\; & \forall u \Big(u = first(sts_{\mathcal{G}}, p_1) \wedge \nexists u' \big((p_2, u') \in \tau\big) \rightarrow (p_2, u) \in \tau' \Big) \\ & \forall u \Big(u = first(sts_{\mathcal{G}}, p_2) \wedge \nexists u' \big((p_1, u') \in \tau\big) \rightarrow (p_1, u) \in \tau' \Big) \end{aligned}$$

Ableitungsschritt für Übergangsverweise

Mittels der beiden vorangegangenen Definitionen wird ein *Ableitungsschritt* zunächst für *Übergangsverweise* wie folgt definiert: Wird die aktive Regel r auf den Laufzeitgraphen G und den Ausführungszustand $sgs_{\mathcal{G}}$ erfolgreich angewendet, so kann eine positiv markierte oder unmarkierte Kontrollflussbeziehung traversiert werden. Dadurch wird r' zur aktiven Regel. G' repräsentiert den Laufzeitgraphen nach Ausführung von r, der zugleich als Ausgangspunkt von r' dient. Der Ausführungszustand $sgs'_{\mathcal{G}}$ bezieht sich auf das Ergebnis der Ausführung von r und der Auswertung der Datenflussbeziehungen, die der traversierten Kontrollflussbeziehung zugeordnet sind.

$$\begin{aligned}(r, G, sgs_{\mathcal{G}}) \vdash^{\mathbf{goto}}_{applied} (r', G', sgs'_{\mathcal{G}}) \quad \Leftrightarrow \quad & \exists cf, \widetilde{sgs}_{\mathcal{G}} \big(applied(cf) \in \{\mathbf{true}, \bot\} \\ \wedge \quad & (G, sgs_{\mathcal{G}}, G', \widetilde{sgs}_{\mathcal{G}}) \in SEM[\![r]\!] \\ \wedge \quad & (r, \widetilde{sgs}_{\mathcal{G}}, r', sgs'_{\mathcal{G}}) \in SEM[\![cf]\!]\big)\end{aligned}$$

Ist eine Regel r hingegen nicht anwendbar, so können Kontrollflussbeziehungen mit negativer oder keiner Markierung verfolgt werden. Dies wird als *negativer Ableitungsschritt* bezeichnet. Im Unterschied zum positiven Fall wird der Laufzeitgraph zwischen den Regeln r und r' nicht verändert, eine Manipulation des Ausführungszustands ist aufgrund von Datenflussbeziehungen jedoch möglich. Eine weitere Anwendungsmöglichkeit negativer Ableitungsschritte besteht, falls die Transformationsausführung im weiteren Verlauf aufgrund nachgelagerter Regeln scheitert[10]. Um eine insgesamt erfolgreiche Transformationsausführung zu erreichen, kann nach erfolgloser Prüfung aller alternativer Entscheidungen, wie etwa der Auffindungsstellen atomarer Regeln, auch eine negierte Kontrollflussbeziehung traversiert werden.

$$\begin{aligned}(r, G, sgs_{\mathcal{G}}) \vdash^{\mathbf{goto}}_{\neg applied} (r', G, sgs'_{\mathcal{G}}) \quad \Leftrightarrow \quad & \exists cf \big(applied(cf) \in \{\mathbf{false}, \bot\} \\ & \wedge (r, sgs_{\mathcal{G}}, r', sgs'_{\mathcal{G}}) \in SEM[\![cf]\!]\big) \\ \wedge \quad & \forall r^{\top}, G^{\top}, sgs^{\top}_{\mathcal{G}} \Big((r, G, sgs_{\mathcal{G}}) \vdash^{*} (r^{\top}, G^{\top}, sgs^{\top}_{\mathcal{G}}) \\ & \wedge \nexists r^{\top\prime}, G^{\top\prime}, sgs^{\top\prime}_{\mathcal{G}} \big((r^{\top}, G^{\top}, sgs^{\top}_{\mathcal{G}}) \vdash (r^{\top\prime}, G^{\top\prime}, sgs^{\top\prime}_{\mathcal{G}})\big) \\ & \quad \rightarrow \exists cf' \big(from(cf') = r^{\top}\big)\Big)\end{aligned}$$

Abbildung 4.34 stellt den Ausführungsablauf graphisch dar. Nach erfolgreicher Auswertung der Transformationsregel $\mathtt{R}_1$ und $\mathtt{R}_2$ wird für $\mathtt{R}_3$ keine gültige Auffindungsstelle gefunden. Da kein negativ markierter ausgehender Kontrollfluss vorliegt,

[10] Die zugehörige Formel drückt dies dadurch aus, dass alle terminalen Regeln ohne weitere Ableitbarkeit einen Kontrollflussnachfolger besitzen.

wird daher per Backtracking zu $\mathtt{R}_2$ zurückgesprungen und eine andere Auffindungsstelle gewählt. Angenommen auch diese letzte Alternative für $\mathtt{R}_2$ scheitert, so wird nach erneutem Backtracking per negativer Kontrollflussbeziehung zu $\mathtt{R}_4$ verzweigt. Sollte auch diese nicht erfolgreich angewendet werden können, so wird letztlich bis zu $\mathtt{R}_1$ zurückgesprungen, um von dort aus weitere Auffindungsstellen auszuwählen.

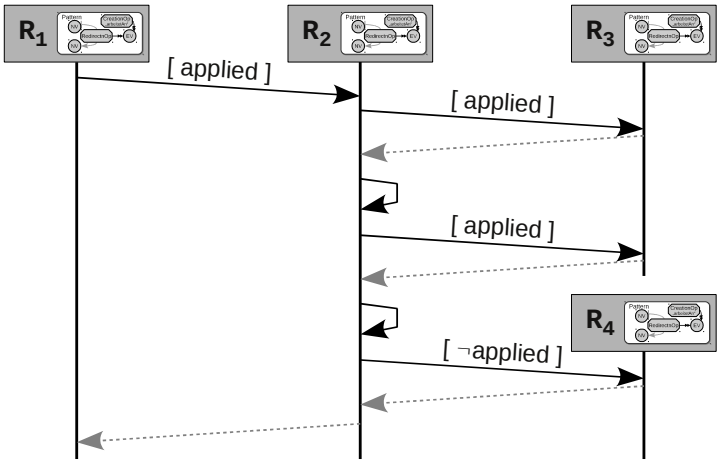


Abbildung 4.34: Backtracking mit positiven und negativen Nachfolgerbeziehungen

Ableitungsschritt für Aufrufverweise

Zusätzlich zu dem oben beschriebenen Ableitungsschritt für Übergangsverweise wird eine Definition für *Aufrufverweise* benötigt. Diese beschreibt einen Zugangsübergang, der bei der aufrufenden Regel *verbleibt*, in die Veränderung der Graphstruktur allerdings die Auswirkung der aufgerufenen Regel einbezieht. Eine spezielle Rolle kommt hier dem Ausführungszustand zu. Für diesen wird mit $\overline{sgs}_{\mathcal{G}}$ ein neuer Zustandsrahmen erstellt, der basierend auf dem Eingangszustand $sgs_{\mathcal{G}}$ und den Datenflussbeziehungen der traversierten Aufrufbeziehung initialisiert wird. Der Ausgangszustand $sgs'_{\mathcal{G}}$ des Ableitungsschritts entspricht dem Eingangszustand nach Rückübertragung der im neuen Zustandsrahmen $\overline{sgs}'_{\mathcal{G}}$ abgelegten Schnittstellenbelegungen. Zur leichteren Einbettung in die übrige Definition ist $\vdash^{\mathbf{call}}$ auch für Regelelemente definiert, die *keine* ausgehende Aufrufbeziehung besitzen. Die Auswertung ändert jedoch Lauf-

zeitgraph und Ausführungszustand nicht.

$$\begin{aligned}(r, G, sgs_{\mathcal{G}}) \vdash^{\textbf{call}} (r, G', sgs'_{\mathcal{G}}) \quad \Leftrightarrow \quad & \exists cf, \bar{r}, \overline{sgs}_{\mathcal{G}}, \overline{sgs}'_{\mathcal{G}} \Big(\\ & isCall(cf) \wedge \overline{sgs}_{\mathcal{G}} = push(sgs_{\mathcal{G}}, \{\}) \\ & \wedge (r, sgs_{\mathcal{G}}, \bar{r}, \overline{sgs}_{\mathcal{G}}) \in SEM[\![cf]\!] \\ & \wedge (G, \overline{sgs}_{\mathcal{G}}, G', \overline{sgs}'_{\mathcal{G}}) \in SEM[\![\bar{r}]\!] \\ & \wedge (r, \overline{sgs}'_{\mathcal{G}}, \bar{r}, sgs'_{\mathcal{G}}) \in SEM[\![cf]\!] \Big) \\ \vee \quad & \Big(\nexists cf \big(isCall(cf) \wedge r = from(cf) \big) \\ & \wedge G = G' \wedge sgs_{\mathcal{G}} = sgs'_{\mathcal{G}} \Big)\end{aligned}$$

Allgemeiner Ableitungsschritt

Eine Sequenz aus je einem Ableitungsschritt für Aufruf- und Übergangsverweise wird als *allgemeiner Ableitungsschritt* $\vdash$ bezeichnet. Der erste Teilschritt ist aufgrund obiger Definition auch für Regeln ohne ausgehenden Aufrufverweis möglich, so dass hier keine Fallunterscheidung zu erfolgen braucht.

$$\begin{aligned}(r, G, sgs_{\mathcal{G}}) \vdash (r', G', sgs'_{\mathcal{G}}) \quad \Leftrightarrow \quad & \exists G'', sgs''_{\mathcal{G}} (\\ & (r, G, sgs_{\mathcal{G}}) \vdash^{\textbf{call}} (r, G'', sgs''_{\mathcal{G}}) \vdash^{\textbf{goto}} (r', G', sgs'_{\mathcal{G}})) \end{aligned}$$

Als Wiederholung allgemeiner Ableitungsschritte mit beliebiger Häufigkeit wird die *transitiv-reflexive Hülle* $\vdash^*$ wie üblich definiert. Der nicht-reflexive Abschluss $\vdash^+$ entspricht dabei der um einen Ableitungsschritt verlängerten reflexiven Variante, wird hier allerdings nicht formal eingeführt.

$$\begin{aligned}(r, G, sgs_{\mathcal{G}}) \vdash^* (r', G', sgs'_{\mathcal{G}}) \quad \Leftrightarrow \quad & \big(r = r' \wedge G = G' \wedge sgs_{\mathcal{G}} = sgs'_{\mathcal{G}} \big) \\ \vee \quad & \exists r'', G'', sgs''_{\mathcal{G}} (\\ & (r, G, sgs_{\mathcal{G}}) \vdash (r'', G'', sgs''_{\mathcal{G}}) \vdash^* (r', G', sgs'_{\mathcal{G}})) \end{aligned}$$

Ausführungsverfolgung anhand eines Beispiels

Um die oben behandelte Definition der Sprachkonstrukte zur Ablaufsteuerung zu verdeutlichen, zeigt Abbildung 4.35 die hieraus ableitbaren Regelinterpretation. Als Beispiel dient die Ausführung von Abbildung 4.28 auf Seite 138. So erfolgt bei Auswertung der umgebenden komponierten Regel $\texttt{AR}_1$ zunächst eine Auswertung aller lokalen Datenflüsse `DF(...)`, wodurch der aktuelle Ausführungszustand entspre-

chend verändert wird. Dargestellt in der Abbildung ist lediglich ein einzelner Ausführungsrahmen, der anfangs eine von außen vorgegebene Schnittstellenbelegung[11] $(1, u_1)$ enthält. Der zugeordnete Wert wird durch Auswertung der Datenflussbeziehungen an die Schnittstellen 2 und 3 propagiert. Da während der gesamten Ausführung keinerlei Veränderungen am Laufzeitgraphen vorgenommen werden, ist in der Abbildung keine Graphstruktur dargestellt.

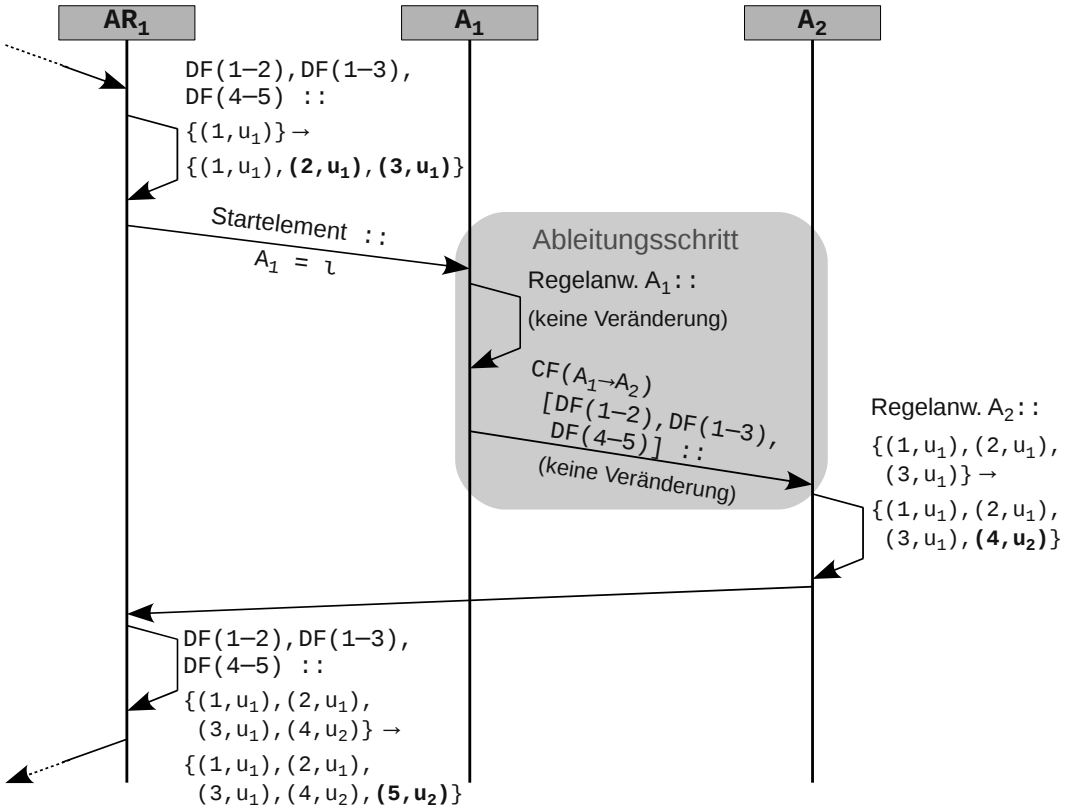


Abbildung 4.35: Verarbeitungsschritte zur Auswertung von Abbildung 4.28

Anschließend erfolgt die Auswertung der Startregel A_1, für die hier eine erfolgreiche Ermittlung von Auffindungsstellen angenommen wird. Daher wird anschließend die Kontrollflussbeziehung $CF(A_1–A_2)$ ausgewertet, welche wiederum die bereits oben behandelten Datenflussbeziehungen einbezieht. Auch hierbei wird der Ausführungszustand nicht verändert. Diese beiden Schritte, Ausführung des Regelelements und Auswertung der inzidenten Kontrollflussbeziehung, stellt einen Ableitungsschritt gemäß obiger Definition dar.

Die Anwendung der zweiten enthaltenen Regel A_2 erfolgt unter Berücksichtigung

[11] Die Nummerierung der Schnittstellen entspricht der Abbildung 4.28.

des Ausführungszustands für eine feste Belegung der *linken* Variable, wobei die rechte Variable aus den hierzu ermittelten Auffindungsstellen belegt wird. Dies führt zu einer Propagation dieses Wertes an die Schnittstelle `4`, und daher zu einem um das Tupel `(4,`u_2`)` ergänzten Ausführungszustand.

Da an dieser Stelle keine weitere Kontrollflussbeziehung vorliegt, terminiert die Ausführung der enthaltenden Regel. Deren Endergebnis ergibt sich durch eine letzte Auswertung der lokalen Datenflussbeziehungen von `AR`, wodurch die zuletzt ermittelte Schnittstellenbelegung zu `(4,`u_2`)` propagiert wird. Der so ermittelte Ausführungszustand stellt das Ergebnis der komponierten Regel dar.

Zwischenfazit: Kontrollflussbeschreibung erfolgt in DRAGULA explizit in graphischer Form und ist damit ähnlich zu UML Aktivitätsdiagrammen und vergleichbaren Ansätzen. Graphanfragen bzw. -transformationen werden direkt in die modellierten Regeln eingebettet. Per Komposition lassen sich Teilregeln zu komplexeren Anweisungsfolgen kombinieren. Neben der `goto`-artigen Weiterschaltung aktiver Regeln ist zudem der Aufruf abgeschlossener Regelblöcke möglich. Durch Datenflussbeziehungen werden Vorbelegungen von Mustersuchen und Transformationen

4.6 Spracherweiterungen

Im bisherigen Verlauf dieses Kapitels sind die grundlegenden Sprachkonstrukte DRAGULAs vorgestellt worden. Diese sind geeignet um Anfragen, Transformationen sowie die Komposition dieser Konstrukte modellieren zu können. Darüber hinaus bietet DRAGULA die Möglichkeit, weitere Sprachkonstrukte als *Erweiterung* des Sprachkerns zu definieren.

Eine Spracherweiterung für DRAGULA wird grundsätzlich als Erweiterung des bereitgestellte Metamodells sowie ggf. zusätzlicher Wohlgeformtheitsbedingungen definiert. Auf Basis dieser Informationen kann somit eine syntaktische Korrektheitsprüfung der modellierten Spezifikationen erfolgen. Die eigentliche Realisierung des gewünschten Verhaltens kann anschließend auf zwei verschiedene Weisen erfolgen, die in Abschnitt 5.5 eingehender behandelt werden. Im Rahmen dieses Kapitels wird daher nur auf die *äußere Sicht* zweier Erweiterungen als Ergänzung der Kernsprache eingegangen. Bei diesen handelt es sich zum einem um ein Sprachkonstrukt zur erleichterten Modellierung *transitiv abgeschlossener Verbindungsnavigation*, das bereits als Anwendungsszenario in Abschnitt 4.3 betrachtet worden ist. Zum anderen wird eine erweiterte Sprachsyntax zur Modellierung von Attributausdrücken zur Abfrage und Veränderung graphbasierter Laufzeitdaten angeboten. Letztere Erweiterung ergänzt

somit die bislang nicht unterstützten Elemente des DRAGOS Graphmodells.

4.6.1 Abschluss von Graphmustern

Als Teil der Kernsprache ist bereits in Abbildung 4.20 auf Seite 120 eine Möglichkeit vorgestellt worden, um mittels Musterreferenzen einen *transitiven Abschluss* von Graphmustern modellieren zu können. Da diese Spezifikationselemente umständlich anzuwenden sind, wird ein gesondertes Konstrukt in Form einer Spracherweiterung eingeführt. Diese wird im Rahmen dieses Unterabschnitts kurz vorgestellt und mit der Basisform verglichen.

Sprachbeschreibung

Das in Abbildung 4.36 angegebene Metamodellfragment zeigt die syntaktische Definition der beschriebenen Spracherweiterung. Zentraler Punkt ist die Abschlussbedingung (ClosureConstraint), die als einziges syntaktisches Element hinzugefügt wird. Die Art des Abschlusses wird als Wert des Attributs type festgelegt, dessen Auswirkung im folgenden Absatz erläutert wird. Des Weiteren referenziert die Bedingung ein Muster, das den auszuwertenden Teilausdruck bereitstellt. Über die allen Bedingungen zugeordnete Restricts-Beziehung werden durch Angabe entsprechender Rollenbezeichner, analog zur Inzidenzbedingung aus Abschnitt 4.2 auf Seite 86, die Start- und Endelemente des zu traversierenden Pfads festgelegt. Zusätzlich werden Variablen als Quell- und Zielfragmente (fragSource, fragTarget) referenziert, die Quelle und Ziel jedes Traversierungsschritts (fragment) definieren. Diese müssen gemäß unten angegebener Wohlgeformtheitsbedingung direkt dem referenzierten Muster entstammen.

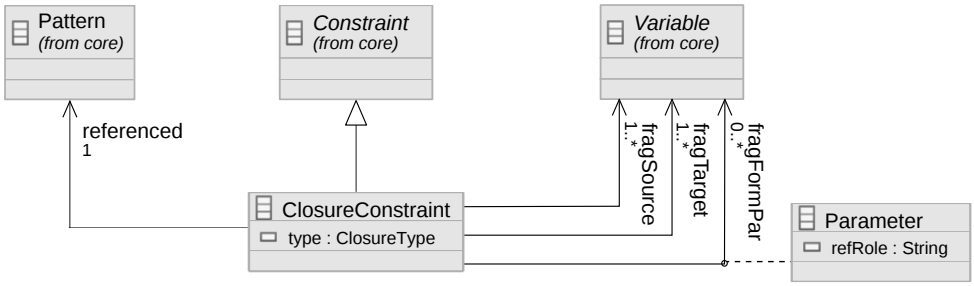


Abbildung 4.36: Definition abgeschlossene Muster

```
context ClosureConstraint
inv localFragment: self.referenced.variables→
      includesAll( self.fragSource→union( self.fragTarget ) )
```

Die *Anzahl* der referenzierten Quell- und Zielfragmente sowie der Quell- und Zielvariablen des referenzierenden Musters müssen jeweils identisch sein. Ist dieser Wert n größer als eins, so modelliert die Abschlussbedingung einen *mehrstelligen* Pfad der eine Relation zwischen $2 \cdot n$ anstelle von einem Variablenpaar beschreibt. Korrespondenzen zwischen Quell- und Zielvariablen sowie der Fragmente des referenzierten Musters werden in diesem Fall durch den Ordnungsindex der (geordneten) Verknüpfungen festgelegt.

```
context ClosureConstraint
inv fragCount: let cnt : Integer = self.fragSource→size() in
        self.fragTarget→size() = cnt and
        self.varByRole(„source“)→size() = cnt and
        self.varByRole(„target“)→size() = cnt and
```

Ein weiteres Modellierungskonstrukt der Abschlussbedingung stellt eine Möglichkeit zur *Parametrisierung* des referenzierten Musters dar. Durch den Einsatz von Parameter-Verknüpfungen werden Variablen des referenzierten Musters mit Werten vorbelegt, die der Abschlussbedingung zugeordneten Variablen entstammen. Hierbei wird analog zu oben eine Übereinstimmung der Rollenbezeichner zur Bindung von Aktualwerten verwendet. Anders als die Quell- und Zielangaben der gesamten Iterationsfolge (reservierte Rollenbezeichner source und target) werden Parameter an alle Auswertungen des Fragmentmusters übergeben.

```
context ClosureConstraint
inv uniqueParameters: self.parameter[fragActParam]→
      collect(p | p.refRole)→isUnique()

context Parameter
inv notReservedRole: not {„source“, „target“}→
      contains( self.refRole )
```

Abschlusstypen: Die konkrete Ausprägung einer Abschlussbedingung kann durch Angabe eines *Abschlusstypen* parametrisiert werden. Dabei besteht primär eine Wahl zwischen reflexiven und nicht-reflexiven Bedingungstypen, wie die folgende Auszählung zeigt.

reflexive: Das referenzierte Teilmuster soll beliebig oft traversiert werden, um eine transitiv-reflexive Verbindung zwischen Start- und Zielvariablen des referenzierenden Musters zu bestimmen. Hierbei ist die Länge null explizit eingeschlossen, so dass die Bedingung für identische Belegungen beider Variablen stets erfüllt ist. Dies entspricht der Bedeutung von Abbildung 4.20 auf Seite 120 sowie dem zur Beschreibung regulärer Sprachen genutzten *Kleenschen Stern* [Kle67].

non-reflexive: Der Ausschluss der Reflexivität führt zu einer Einschränkung der Verbindungssuche auf rein transitive Strukturen. Somit ist die Bedingung für identisch belegte Start- und Zielvariablen nicht standardmäßig erfüllt.

range: Als Verallgemeinerung obiger beider *klassischer* Abschlusstypen besteht zudem die Möglichkeit, die Anzahl der durchzuführenden Traversierungsschritte vorzugeben. Dies erfolgt in Form einer unteren und optional einer oberen Schranke.

terminal: Im Unterschied zu den vorgenannten Abschlusstypen setzt diese Variante voraus, dass von einem ermittelten Zielelement aus *keine* Fortsetzung des Pfades existiert, eine weitere Suche nach dem referenzierten Teilmusters also kein Ergebnis liefert. Hierdurch werden terminale Elemente eines Pfades bestimmt.

Anwendungsbeispiel

Abbildung 4.37 zeigt ein Anwendungsbeispiel für Abschlussbedingungen, in dem die geschachtelte Musterstruktur aus Abbildung 4.20 auf Seite 120 nachgebildet wird. Anstatt auf Musterreferenzen zurückzugreifen, wird an dieser Stelle eine Abschlussbedingung verwendet, der analog zu Inzidenzbedingungen Quell- und Zielvariablen des referenzierenden linken Musters zugeordnet sind. Durch grau gezeichnete Kanten verweist die Bedingung auf Variablen des referenzierten rechten Musters. Bei Prüfung dieser Bedingung wird, bspw. auf Basis einer vorgegebenen Belegung der Startvariablen, das referenzierte Muster einmalig ausgewertet. Die dabei ermittelten Auffindungsstellen entsprechen zudem denen des referenzierenden Musters, da dieses keine zusätzlichen Einschränkungen der Zielvariablen enthält.

Werden zu diesem weitere Auffindungsstellen angefragt, erfolgt ein weiterer Traversierungsschritt. Hierzu werden sukzessiv alle zuvor ermittelten Zielentitäten als Werte der Startvariablen vorbelegt und die jeweils erreichbaren Zielentitäten ermittelt. Jedes Element der so bestimmten Gesamtmenge bildet eine weitere Auffindungsstelle des referenzierenden Musters. Ausgeschlossen ist dabei, dass eine bereits zur Traversierung genutzte Belegung der Startvariable erneut herangezogen wird – zyklische Auswertungen werden somit vermieden. Sollten in Erweiterung dieses Beispiels *mehrere* Start- und Zielknoten angegeben sein, so erfolgt eine Auswertung falls das Tupel aller Variablenbelegungen nicht bereits zuvor ausgewertet worden ist.

Vergleich & Realisierungsansatz

Wie bereits eingangs beschrieben, handelt es sich bei der hier vorgestellten Spracherweiterung DRAGULAs lediglich um eine *syntaktische Abstraktion*, da sämtliche

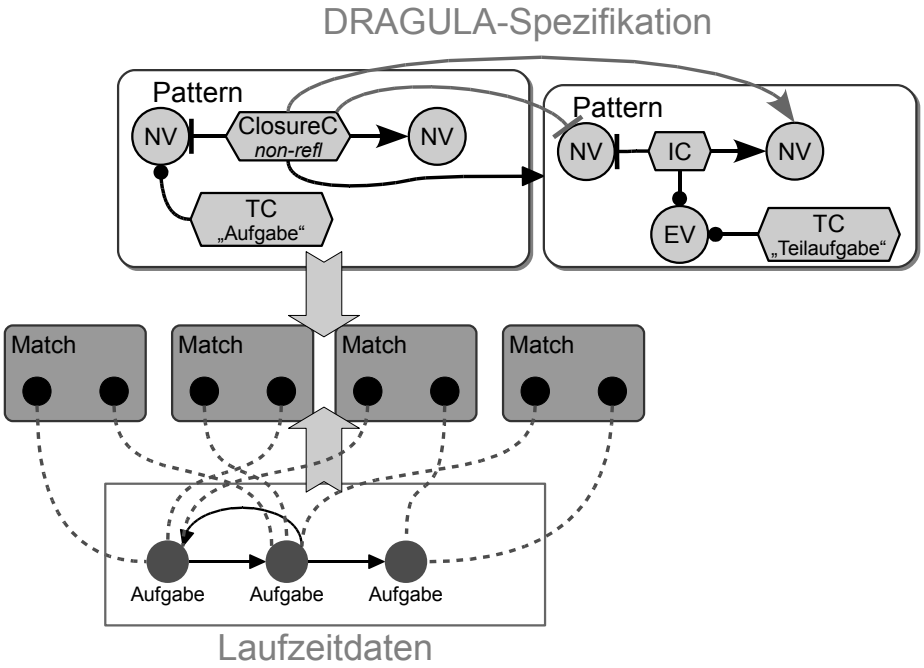


Abbildung 4.37: Muster zur Nutzung von Abschlussbedingungen

Funktionalität bereits durch die in Abschnitt 4.3 beschriebenen Musterreferenzen abgedeckt werden kann. Das hier vorgestellte Beispiel zeigt jedoch eine erhebliche Ersparnis beim Einsatz dieses Konstrukts, welche den Realisierungsaufwand auch bei Einsatz DRAGULAs als technische Ausführungsmaschine, rechtfertigt. So ist es nicht notwendig, die Übergabe von Werten zwischen Traversierungsschritten, vorzusehende Abbruchbedingungen in Form von alternativen Suchmustern oder den Zyklusausschluss explizit zu spezifizieren. Zugleich ist die Abschlussbedingung gegenüber den Musterreferenzen allerdings *eingeschränkt*, da lediglich iterative Navigation mittels referenzierter Graphmuster ausgedrückt werden kann. Hiermit nicht möglich ist bspw. die Bestimmung von *Kantenbüscheln*, die in Abbildung 4.19 auf Seite 117 mittels Musterreferenzen behandelt worden sind. Auch können die mittels einer Abschlussbedingung traversierten Graphentitäten nicht verändert werden, d.h. evtl. im referenzierten Muster enthaltene Änderungsoperatoren werden nicht ausgewertet.

Eine Parametrisierung der modellierten Pfadausdrücke ist in beiden Fällen auf ähnliche Weise möglich, wobei im Falle der Abschlussbedingung auf identische Rollenbezeichner anstelle einer expliziten Elementabbildung zurückgegriffen wird. Dieses Vorgehen bietet sich durch die Einbettung der Abbildungsbedingung in das übrige Sprachrahmenwerk, insbesondere durch Nutzung der Restricts-Beziehung an.

Wie im nachfolgenden Kapitel 5 erläutert werden wird, besteht für die Abschluss-

bedingung die Möglichkeit, eine *Rückführung* auf die Kernbestandteile DRAGULAs vorzunehmen. Dieser Ansatz sieht vor, im Vorfeld der Mustersuche eine *explizite Ersetzung* des zusätzlichen Sprachkonstrukts vorzunehmen, um an dessen Stelle auf die allgemeinere Notation der Musterreferenzen zurückzugreifen. Hierdurch ist also keine gesonderte Implementierung des zusätzlichen Sprachkonstrukts notwendig.

4.6.2 Attributausdrücke

Die bisher vorgestellten Sprachkonstrukte von DRAGULA umfassen keine Möglichkeit zur Auswertung und Veränderung von *Attributwerten*. Der Grund hierfür liegt darin, dass die erforderliche Funktionalität stark vom jeweiligen Anwendungsfall getrieben wird. So sind möglicherweise *textuelle* Attributausdrücke unter Nutzung einer externen Programmiersprache wünschenswert, wie dies bspw. von zahlreichen Graphtransformationswerkzeugen wie AGG, Fujaba, oder GReAT angeboten wird. Im Allgemeinen kann jedoch keine spezifische Programmiersprache und ein damit verbundener Generierungsprozess vorgegeben werden.

Ein grundsätzlich anderer Ansatz besteht in der Bereitstellung einer eigenen Attributsprache, die als Teil der DRAGULA-Sprachdefinition realisiert wird. Die hierfür bereitgestellte Teilsprache lehnt sich stark vereinfachend an die Object Constraint Language an. Zudem fügt sich die Attributsprache direkt in DRAGULA ein, insbesondere Redundanzen im Bezug auf die Funktionalität zwischen Kernsprache und Attributsprache werden vermieden. Bspw. bietet die im Weiteren vorgestellte Attributsprache im Gegensatz zu OCL keine Konstrukte zur Navigation entlang von Verbindungsstrukturen, da dies entsprechend in DRAGULA modelliert werden sollte.

Sprachbeschreibung der Attributsprache

Die syntaktisc Definition der Attributteilsprache DRAGULAs wird in Abbildung 4.38 angegeben. Auf eine vollständige formale Definition dieses Sprachbestandteils wird an dieser Stelle verzichtet, da seitens DRAGULAs keine Beschreibung des zugrundeliegenden Graphschemas und somit auch nicht der verarbeiteten Attributtypen erfolgt. Die Ausformulierung einer *Algebra* für Attributwerte würde daher zu einer stark eingeschränkten Ausdruckskraft der Sprache führen oder aber erheblichen Raum in Anspruch nehmen. Stattdessen wird als pragmatischer Ansatz die Verwendung beliebiger *serialisierbarer* Datentypen zugelassen, die seitens des Graphspeichers DRAGOS repräsentiert werden können. Arithmetische Operationen sind im Sinne ihres intuitiven Verständnisses zu interpretieren.

Zu beachten ist bei obiger Definition, dass die dargestellte Spracherweiterung keine Variablentypen für Attributwerte definiert, die anstelle von Graphentitäten skalare

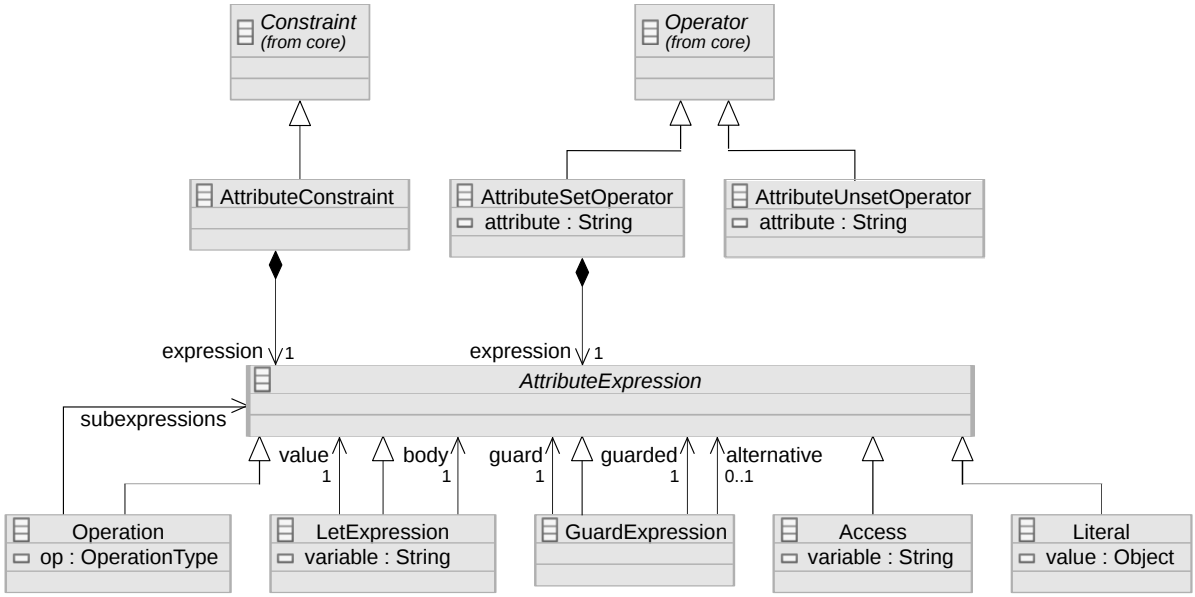


Abbildung 4.38: Definition Attributausdrücke

Werte aufnehmen. Stattdessen können Attributwerte nur innerhalb eines Attributausdrucks referenziert werden. Der Grund für diese Entscheidung liegt darin begründet, dass die Einführung skalarer Variablentypen einen Großteil der bestehenden Sprachdefinition, bspw. hinsichtlich Einschränkungen und Operationen beeinflussen und eine entsprechende Einschränkung auf Entitätsvariablen nötig machen würde. Zusätzlich würde die Fokussierung DRAGULAs auf Graphtransformationen insoweit verloren gehen, als dass stattdessen eine allgemeine Programmiersprache entstehen würde. Trotz der daher in Kauf genommenen Einschränkung zur Darstellung von skalaren Werten, die somit als Elemente „zweiter Klasse“ im Vergleich zu Graphentitäten behandelt werden, können skalare Werte zwischen Regelelementen propagiert und als Teil von Auffindungsstellen gehandhabt werden. Hierzu werden in Kapitel 6 sogen. Behälterknoten vom vordefinierten Typ ScalarHolder eingesetzt, die einen skalaren Wert als Attributwert beinhalten. Diese sind als temporär anzusehen und können nach Abarbeitung einer Transformationsfolge und etwaiger Ergebnisübertragung entfernt werden.

Schnittstelle zu DRAGULA. Die Schnittstelle zwischen DRAGULA und seiner Attributteilsprache bilden die spezialisierte Bedingung AttributeConstraint zum Prüfen von Attributwerten, sowie die beiden Operatoren AttributeSetOperator zum Setzen und AttributeUnsetOperator zum Löschen eines Attributwerts. Mit Ausnahme der Löschoperation wird dabei Bezug auf einen Attributausdruck genommen, der im Falle der Bedingung

zu einem booleschen Ergebnis ausgewertet werden muss. Im Falle des Setzoperation wird der ermittelte Wert dem zu setzenden Attribut zugewiesen. Um in beiden Fällen Bezug auf andere Graphelemente zu nehmen, werden diese durch Kanten des Typs restricts (vgl. Abschnitt 4.2.2 auf Seite 87) bzw. requires (vgl. Abschnitt 4.4.2 auf Seite 124) mit dem attributverarbeitenden Element verbunden. Der jeweils verwendete Rollenbezeichner legt zum einen den Attributnamen der referenzierten Graphentität fest. Zum anderen wird hierüber ein interner Bezeichner vergeben, auf den innerhalb des Attributausdrucks zugegriffen werden kann. Dieser interne Bezeichner muss pro AttributeConstraint bzw. AttributeSetOperator eindeutig sein.

Ausgestaltung von Attributausdrücken. Attributausdrücke werden in Form eines abstrakten Syntaxbaums angegeben, dessen Elemente jeweils einen der konkreten Untertypen von AttributeExpression instanziieren. Jedes Element dieses Baums liefert bei dessen Auswertung einen Wert zurück, der von übergeordneten Ausdrücken weiterverarbeitet oder als Endergebnis einer Prüfbedingung oder Setzoperation genutzt wird. Die Bedeutung der einzelnen Typen wird im Folgenden beschrieben:

Operation: Operationen verarbeiten eine beliebige Anzahl von Zwischenergebnissen, bspw. durch Anwendung mathematischer Funktionen. Die jeweils anzuwendende Funktion wird durch die Angabe eines OperationType präzisiert. Eine grobe Aufzählung der angebotenen Operationstypen gibt der nachfolgende Absatz.

LetExpression: Zuweisungsausdrücke ermöglichen die Wiederverwendung von Teilausdrücken, deren Wert der angegebene Bezeichner variable zugeordnet wird. Dieser kann durch einen Zugriff (s.u.) abgerufen werden.

GuardExpression: Mittels bedingter Ausdrücke können Fallunterscheidungen modelliert werden. Wird der als guard identifizierte Teilausdruck zu einem positiven booleschen oder numerischen Ergebnis ausgewertet, so entspricht das Ergebnis des gesamten Ausdrucks dem des guarded Unterausdrucks. Andernfalls wird, falls vorhanden, der Unterausdruck Alternative ausgewertet und dessen Ergebnis zurückgeliefert.

Access: Zugriffe liefern den Wert eines vorab definierten Bezeichners als Resultat des Teilausdrucks. Dieser kann sich auf Attributwerte von Graphelementen beziehen, die durch adjazente Variablen der zugehörigen Attributbedingung bzw. Setzoperation gegeben sind. Auch auf Variablen eines Zuweisungsausdrucks (s.o.) wird auf diese Weise zugegriffen.

Literal: Ein Literal stellt einen konstanten Wert als Teilausdruck bereit. Durch Verzicht auf eine formale Typisierung der Attributteilsprache können somit beliebige Objekte als Literal verwendet werden.

Operationstypen. Eine Operation beschreibt die Anwendung einer n-stelligen Funktion auf eine Folge von Unterausdrücken. Neben einer Reihe von Standardfunktionen, die im Folgenden aufgelistet werden, können zusätzliche Operationstypen durch den Verwender implementiert und bereitgestellt werden.

Boolesche Operationen repräsentieren binäre Funktionen zum Vergleich sowie zur logischen Verknüpfung von Teilergebnissen. Auch ein unärer Negationsoperator wird angeboten.

Arithmetische Operationen stellen mathematische Basisfunktionen, bspw. zur Addition numerischer Werte bereit.

Mengenoperationen ermöglichen den Vergleich mengenwertiger Attribute, die wie im Folgenden erläutert durch Variablen aggregierter Teilmuster ermittelt werden. Hierzu zählt bspw. der Test auf Inklusion zwischen zwei Mengen.

Anwendung der Attributteilsprache

Die Anwendung der DRAGULA Attributteilsprache auf ein einfaches Beispiel zeigt Abbildung 4.39. Das erwünschte Transformationsverhalten besteht darin, Personen, die zwei Aufgaben zeitgleich bearbeiten, nur noch derjenigen zuzuteilen die mit größerem Arbeitszeitanteil bearbeitet wird. Die dargestellte Attributbedingung und Setzoperation referenziert mit entsprechenden Rollenbezeichnern jeweils die anteil-Attribute[12] der ermittelten Verbindungen. Sowohl die Attributbedingung als auch die Setzoperation referenzieren jeweils einen abstrakten Syntaxgraph, der den als Kommentar angegebenen textuellen Ausdruck repräsentiert. Die beiden Ausdrücke führen einerseits einen Vergleich, andererseits eine Addition der referenzierten Attributwerte durch. Durch die Setzoperation wird wie angegeben das Attribut anteil der beeinflussten Kantenvariablen gesetzt.

Obiges Beispiel verdeutlicht die phasen- und stufenweisen Ausführung von Operatoren. Attributverarbeitende Operationen werden in einer früheren Phase als löschende, jedoch nach erstellenden Operationen abgearbeitet – andernfalls könnten referenzierte Attributwerte bereits gelöscht oder die Zielelemente einer Setzoperation noch

[12] Der Attributbezeichner wird technisch zusätzlich zum Rollenbezeichner angegeben, ist aus Übersichtlichkeitsgründen jedoch nicht dargestellt.

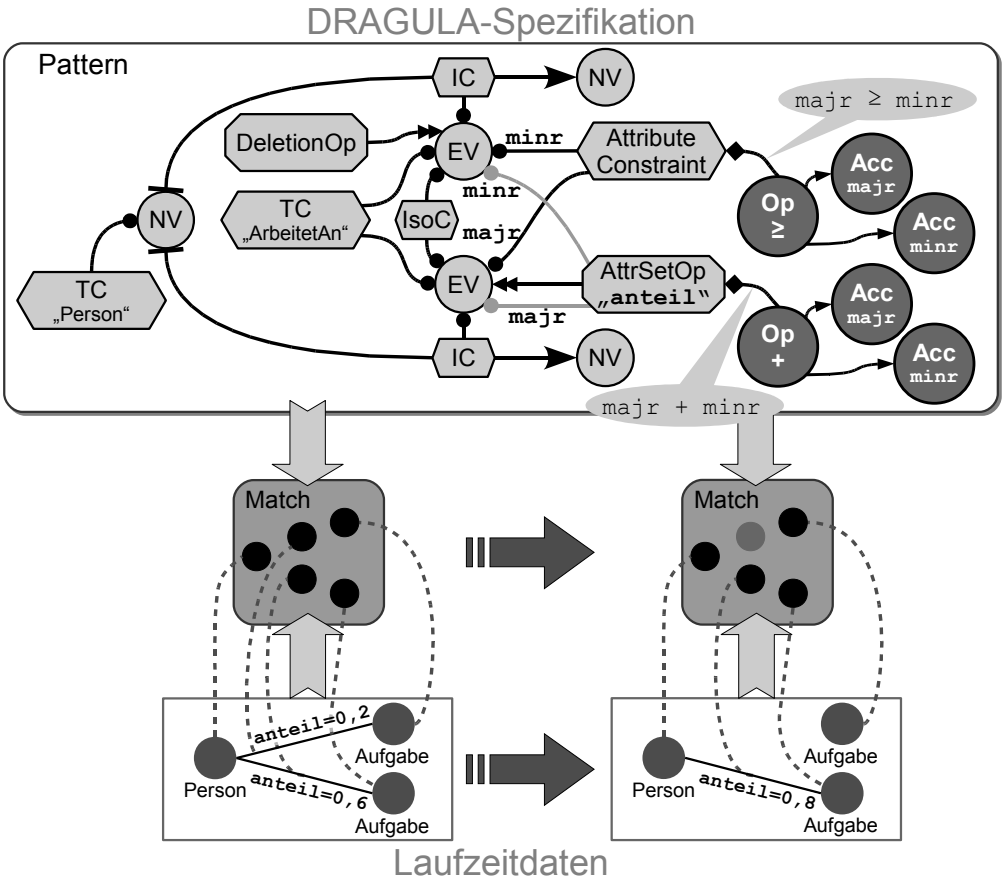


Abbildung 4.39: Transformationsregel mit Attributverarbeitung

nicht vorhanden sein. Ferner können Attributausdrücke auch auf das zu *ändernde* Attribut Bezug nehmen, weshalb die Berechnung des neuen Wertes in einer früheren Stufe als das eigentlich Setzen erfolgt.

Anwendung auf aggregierte Muster

Die DRAGULA Attributteilsprache erlaubt die Verarbeitung aggregierter Teilergebnisse, die durch Auffindungsstellen entsprechend geschachtelt und annotierter Muster ermittelt werden. Neben der in Abschnitt 4.4 beschriebenen Transformation unter Berücksichtigung *aller* Auffindungsstellen, stellen Attributausdrücke ein zweites Anwendungsgebiet dieses aggregierten Musters dar. Hierbei werden die Attributwerte der jeweiligen Auffindungsstellen als *Sequenz* aufgefasst und gemeinsam verarbeitet. Je nach darauf anzuwendender Funktion handelt es sich beim Ergebnis wiederum um eine Sequenz oder um einen einzelnen skalaren Wert. Der letztere Fall beschreibt die

Anwendung einer *Aggregationsfunktion*, bspw. zur Summierung der Elemente einer Wertesequenz.

Bei der Kombination zweier (oder je nach Stelligkeit der Funktion beliebig vieler) Variablenwerte durch eine Operation, entscheidet die Musterzugehörigkeit der jeweiligen Graphelementvariablen über deren weitere Verarbeitung.

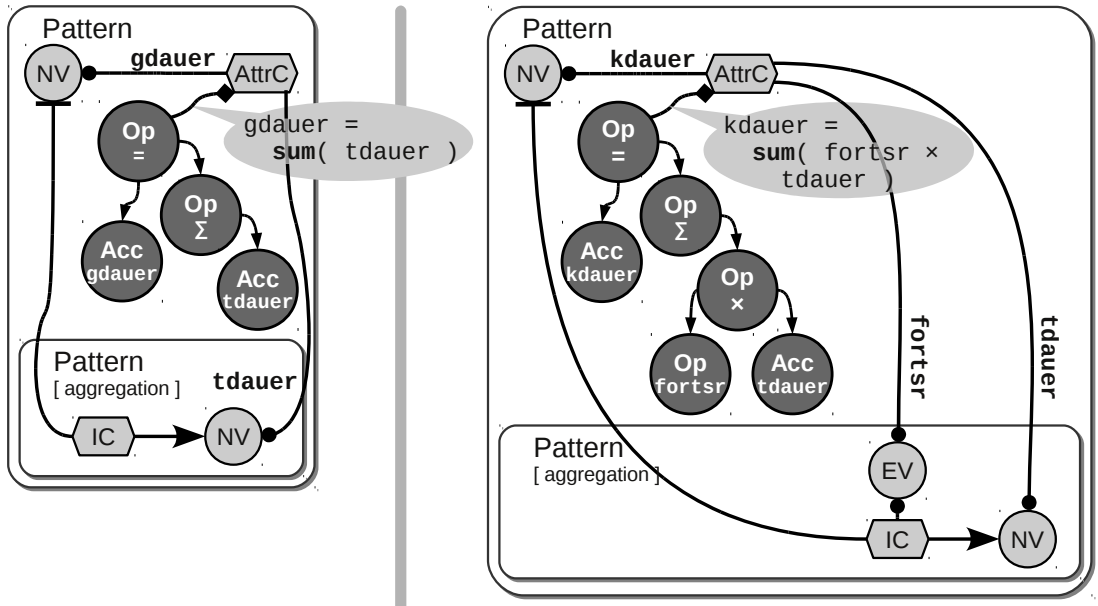
- Liegen zwei Graphelementvariablen in einem *gemeinsamen* aggregierten Muster und die Attributbedingung bzw. die Setzoperation in einem äußeren, so erfolgt eine *paarweise* Kombination der Werte durch die Operation. Die resultierende Sequenz besitzt somit so viele Elemente wie ihre als Parameter angegebenen zur verarbeitenden Sequenzen.

- Liegen zwei Graphelementvariablen in *unterschiedlichen* Mustern, von denen wenigstens eins aggregiert wird, so erfolgt die Bildung des *Kreuzprodukts*. Demzufolge umfasst die Ergebnissequenz $n_1 \times \ldots \times n_m$ viele Elemente, wobei n_i die Anzahl der Elemente einer verarbeiteten Sequenz angibt.

Eine Reihe einfacher Anwendungsbeispiele beschreibt die Verwendung aggregierter Muster in Bezug auf Attributausdrücke. Abbildung 4.40 bezieht sich auf die Gesamtdauer eines Projekts (`gdauer`) die als Summe aller enthaltenen Teilaufgaben (`tdauer`) definiert sei[13]. Im angegebenen Muster wird der Variable des äußeren Musters das Gesamtprojekt, und der Variablen des geschachtelten aggregierten Musters eine zu ermittelnde *Teilaufgabe* zugewiesen. Die Attributbedingung referenziert beide Variablen um deren jeweiligen Attributwerte im nachfolgenden Ausdruck bekannt zu machen. Dieser besteht als oberstes Element aus einer Vergleichsoperation, die den Attributwert `gdauer` der äußeren Variable mit einem zu berechnenden Teilausdruck vergleicht. Der Teilausdruck enthält einen Summierungsoperator, der alle aus dem geschachtelten Muster ermittelten Attributwerte verarbeitet. Der hierfür angegebene Zugriffsoperator greift auf die durch die Attributbedingung festgelegten Bezeichner `tdauer` zu, der aufgrund der Musterhierarchie eine Sequenz entsprechender Werte enthält.

Das zweite, ebenfalls in Abbildung 4.40 dargestellte Beispiel, variiert die vorherige Prüfbedingung derart, dass die *kumulierte Dauer* (`kdauer`) eines Projekts bestimmt werden soll, die sich aus der jeweiligen prognostizierten Dauer aller Teilaufgaben und ihres jeweiligen Fortschritts (`fortsr`) ergeben. Zu beachten ist in diesem Fall, dass der Multiplikationsoperator $\times$ die Sequenzeigenschaft beider Operanden beibehält. Zusätzlich wird, da beide Operanden *demselben* aggregierten Muster entstammen, eine *paarweise* Verarbeitung vorgenommen. So ergibt sich das *i*-te Ergebnis aus der

[13] Zum besseren Verständnis stellen diese und die folgende Abbildung anstelle der bisher aufgezeigten Auffindungsstellen lediglich eine tabellarische Erläuterung der Attributberechnung vor.



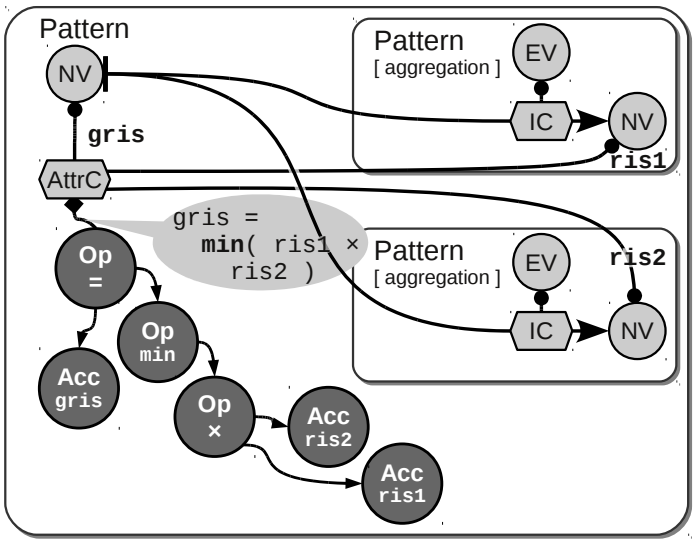
gdauer / Σ	tdauer
42	[10,15, 17]
[...]	[...]

kdauer / Σ	tdauer	fortsr	×
20.2	[10,15, 17]	[70%,20%, 60%]	[7,3, 10.2]
[...]	[...]	[...]	[...]

Abbildung 4.40: Muster zur Nutzung aggregierter Attributwerte

Multiplikation der *i*-ten Elemente der jeweiligen Operandensequenzen. Die Summe berechnet sich darauf basierend, wie oben beschrieben, als aggregierter Wert, der mit dem Attributwert des Projekts verglichen wird.

Ein drittes Beispiel in Abbildung 4.41 zeigt schließlich einen Anwendungsfall, bei dem Sequenzen von Attributwerten aus *verschiedenen* aggregierenden Mustern zusammengefasst werden. Im dargestellten Fall soll es sich um Erfolgswahrscheinlichkeiten einzelner Aufgaben handeln, zu denen die Gesamtwahrscheinlichkeit bei zweifacher Wiederholung sämtlicher Aufgaben errechnet werden soll. Zur Auswertung der Musterstruktur werden zunächst alle Auffindungsstellen beider Muster ermittelt, die hier jeweils zwei Werte ergeben. Bei der Multiplikation beider Faktoren `ris1` und `ris2` wird daher das *Kreuzprodukt* dieser Werte gebildet, wie die untenstehende Tabelle aufzeigt. Das durch die Aggregationsfunktion zu ermittelnde Minimum (hier `64%`) stellt das Ergebnis des Ausdrucks dar.



gris / min	ris1	ris2	×
64%	[90%, 80%]	[90%, 80%]	[81%,72%, 72%,64%]
[...]	[...]	[...]	[...]

Abbildung 4.41: Muster zur Nutzung aggregierter Attributwerte

4.6.3 Weitere Spracherweiterungen

Die hier diskutierten Erweiterungen sind lediglich zwei illustrative Beispiele. Darüber hinaus existieren verschiedene kleinere Erweiterungen:

- Durch ein Sprachkonstrukt zur *Strukturkopie* wird ein ausgewählter Teilgraph einschließlich aller enthaltenen Elemente kopiert. Analog zu der zur Expansion von Musterreferenzen eingeführten Funktionalität können auch hier Beziehungen zum Kontext des Teilgraphen, also den nicht mitkopierten Elementen, explizit angegeben werden. Wird keine Festlegung getroffen, so ist die erzeugte Kopie zu denselben Elementen adjazent wie der Originalteilgraph.

- Teilgraphen können zudem auf *Äquivalenz* geprüft werden. Hierbei werden die Typen, die Attributierung sowie die Verbindungsstrukturen aller enthaltener Elemente durch Isomorphieprüfung verglichen. Elemente außerhalb der zu vergleichenden Teilgraphen, die bspw. durch Verbindungsstrukturen mit diesen in Beziehung stehen, werden dabei ignoriert. Die Implementierung dieses Konstrukts ist jedoch rein experimentell, insbesondere sind bislang keine Optimie-

rungstechniken wie bspw. Prüfsummen für Graphstrukturen integriert. Ein solcher Ansatz findet sich in [Ren06] in Form sogen. *Graphsignaturen*.

- Schließlich lassen sich gleichartige Graphelemente über einen speziellen Operator *kombinieren*. Dabei werden inzidente Verbindungen auf ein einzelnes Element umgelenkt und alle Attributbelegungen mengenwertig vereinigt. Der Typ des resultierenden Graphelements bestimmt sich aus den zu kombinierenden Elementen: Falls diese sich unterscheiden, erwartet der Operator, dass diese in einer Typhierarchie zueinander stehen, ein Typ also eine transitive Generalisierung des anderen darstellt. Ist dies der Fall, so wird der speziellere Typ gewählt, andernfalls wird ein Laufzeitfehler ausgelöst.

- Nur indirekt eine Spracherweiterung von DRAGULA stellt der sogen. *Reflektive Typgraph* dar. Über einen speziellen übergeordneten Graphen wird das DRAGOS Graphschema als regulärer Instanzgraph eingeblendet, etwa in Form eines Knotens je vorhandener Graphentitätsklasse. Zusätzlich referenziert jeder Instanzknoten seinen zugeordneten Typ, genauer dessen reflektive Typknoten, über eine spezielle Kante. Auf diese Weise lässt sich das DRAGOS Graphschema durch reguläre DRAGULA Muster analysieren und reflektiv zur Anfrage von Instanzdaten verwenden. Auch eine Veränderung des Typgraphen ist zwecks Veränderung des Graphschemas zulässig.

4.7 Diskussion des Ansatzes

Zum Ende diesen Kapitels erfolgt eine Diskussion des Lösungsansatzes, bei dem Vor- und auch mögliche Nachteile der Sprache DRAGULA herausgestellt werden. Daran schließt sich ein detaillierter Vergleich mit verwandten Arbeiten an.

4.7.1 Vorteile

Vorteilhaft an der in DRAGULA verfolgten Sprachgestaltung ist die *klare Trennung* zwischen verschiedenartigen syntaktischen Elementen, wie etwa Variablen, Bedingungen, und Operatoren. Dies erleichtert einerseits die hier vorgestellte formale Definition, andererseits lassen sich auch Spracherweiterungen leichter einbinden. Anwender können das komplexe DRAGOS Graphmodell selektiv durch bestimmte syntaktische Elemente nutzen, erweiterte Funktionalität bspw. zur Darstellung hierarchischer Strukturen dagegen entfallen lassen. Zur programmatischen Nutzung, die das Hauptanwendungsgebiet von DRAGULA darstellt, können Spezifikationsdokumente

leichter inkrementell erstellt werden, da unterschiedliche Aspekte einer Variablenbelegung (Typisierung, Inzidenz, Enthaltenseinsbeziehungen, ...) durch unterschiedliche syntaktische Elemente formuliert werden.

DRAGULA umfasst im Kern einen beschränkten, jedoch nicht minimalen *Sprachumfang* einschließlich Rekursion und Kontrollflussbeziehungen. Durch diese im Vergleich zu minimalen Graphtransformationsansätzen erweiterte Funktionalität erlaubt die Kernsprache eine kompakte Darstellung komplexer Graphoperationen. Andernfalls müssten bspw. Pfadausdrücke aufwändig durch mehrere Transformationsregeln und Markierungen im Arbeitsgraphen simuliert werden. Spezifische höherwertige Sprachkonstrukte wie bspw. Pfadausdrücke werden von DRAGULA jedoch nicht direkt angeboten, sondern können in Form von Spracherweiterungen ergänzt werden.

Auch durch die Reduktion auf *graphorientierte Konzepte* bleibt der Sprachkern überschaubar. So verzichtet bspw. die Spracherweiterung für Attributausdrücke auf Variablentypen für skalare Datentypen. Hierdurch müssen skalare Ein- und Ausgabeparameter zwar ggf. durch Attributwerte von Graphentitäten simuliert werden. Allerdings kann die Behandlung von Attributwerten somit auf Ausdrucksbäume und einen minimalen Satz zusätzlicher Bedingungen und Operatoren beschränkt werden.

Vorteilhaft ist ferner die *graphbasierte* abstrakte Syntax der Kernsprache, da aufgrund der einheitlichen API Spezifikationsdokumente leicht programmatisch erstellt und verarbeitet werden können. Zudem lassen sich Spracherweiterungen leicht an den Sprachkern anschließen, wie das folgende Kapitel 5 zeigen wird. Durch Verwendung des Graphspeichers DRAGOS als technische Plattform ist insbesondere eine uniforme Darstellung von Sprachkonstrukten der Kernsprache und von Spracherweiterungen möglich.

4.7.2 Nachteile

Ein möglicher Nachteil DRAGULAs ist die ausführliche Darstellung und damit der Umfang komplexer Spezifikationen in praktischen Szenarien. Dieser ergibt sich z.T. direkt aus obigen Vorteilen wie der Trennung nach Strukturtypen und kann somit nicht grundsätzlich vermieden werden. Primärer Verwendungszweck der Kernsprache ist jedoch die Ausführung programmatisch aus Ausdrücken höherwertiger Sprachen generierter Spezifikationen. Da DRAGULA Spezifikationen daher idealerweise für den Entwickler nicht sichtbar sind, stehen ästhetische Aspekte zunächst im Hintergrund. Nichtsdestotrotz lassen sich geschachtelte Muster und Transformationsregeln auch zur Strukturierung von Spezifikationen nutzen. Für Muster können z.B. öffentlich zugreifbare Variablen in das äußerste Muster aufgenommen werden, wohingegen interne Variablen tiefer geschachtelt werden und somit nicht im Ergebnis sichtbar sind.

DRAGULA stellt keine Realisierung eines *Standards* wie etwa UML Aktivitätsdiagramme oder QVT dar. Vielmehr stellt DRAGULA eine allgemein verwendbare und an die jeweiligen Erfordernisse anpassbare Lösung bereit. Standards wie QVT verfolgen insoweit einen konträren Ansatz, als dass eine in sich abgeschlossene Lösung angestrebt wird die keine fallweise Anpassung erfordert. Notationen wie bspw. UML Aktivitätsdiagramme sind dagegen nicht zur direkten Ausführung der modellierten Sachverhalte konzipiert und dementsprechend weniger detailliert spezifiziert. Nicht-Standard Erweiterungen wie das Story-Driven Modeling [FNTZ98] bieten diesbezüglich ergänzende Funktionalität.

Als neu vorgeschlagene Sprache steht DRAGULA in vermeintlicher Konkurrenz zu einer Vielzahl vorhandener Ansätze, darunter auch in industriellen Szenarien erprobte Werkzeuge. Das hier angestrebte Einsatzszenario ist für diese jedoch insofern atypisch, als dass neu entwickelte Spezifikationssprachen zur leichteren Realisierung auf die vorgestellte Technologie abgebildet werden sollen. Neben der Verwendungsmöglichkeit als Kernsprache und Ausführungsmaschine spielt dabei die Erweiterbarkeit und Anpassbarkeit eine wesentliche Rolle. Dies wird in bestehenden Systemen nicht in vergleichbarem Maße wie in DRAGULA berücksichtigt, da bei diesem Ansatz sowohl Sprache als auch Ausführungsmaschine künftige Erweiterungen und Anpassungen unterstützen.

Die im Rahmen diese Kapitels gegebene formale Definition dient zur eindeutigen Interpretation von DRAGULA Spezifikationen, ist jedoch nicht geeignet um formale Schlüsse aus solchen zu ziehen – bspw. bezüglich Termination oder Konfluenz eines beschriebenen Graphtransformationssystems. Dies gilt jedoch auch für die Semantikbeschreibungen von PROGRES [Sch91] oder Fujaba [Zü01], da prinzipbedingt keine derartigen Aussagen in hinreichend allgemeinen Graphtransformationswerkzeugen getroffen werden können. Die Alternative wäre eine Beschränkung auf wenige formale Konzepte, die etwa in AGG [Tae99] oder GP [Plu09] erfolgt.

4.8 Verwandte Arbeiten

Dieser Abschnitt untersucht bestehende Ansätze im Bereich Graphtransformationen und zeigt Beziehungen zu DRAGULA auf. Dabei gliedert sich die Erläuterung anhand des Sprachaufbaus in die folgenden Bereiche: Einfache Graphmuster, Musterkomposition, Transformationen, und Kontrollflussmodellierung.

4.8.1 Einfache Graphmuster

Im Bezug auf einfache Graphmuster (vgl. Abschnitt 4.2 auf Seite 86) bieten gängige existierende Sprachen vergleichbare Funktionalität. Unterschiede bestehen hauptsächlich bezüglich des verarbeiteten Datenmodells, das bspw. bei PROGRES gerichtete attributierte Graphen umfasst, ohne jedoch dabei Kanten identifizieren oder attributieren zu können. Der flache, d.h. nicht hierarchisch aufgebaute Laufzeitgraph wird über ein eigenes zweistufiges Typsystem definiert. Diese Funktionalität wird von DRAGULA zur Mustersuche bzw. durch den Graphspeicher DRAGOS zur Typisierung voll abgedeckt.

Das an objektorientierte Konzepte angelehnte Fujaba bietet zu PROGRES vergleichbare Funktionalität, und ergänzt das Graphmodell um geordnete und qualifizierte Assoziationen. Durch die Qualifizierung wird indirekt das Identifizieren von Kanten im Instanzgraphen ermöglicht, eine Attributierung von Kanten ist allerdings auf diese Weise nicht möglich. Diese ergänzende Funktionalität wird derzeit nicht von DRAGULA bzw. DRAGOS direkt angeboten. Geordnete Kanten lassen sich jedoch durch Querbeziehungen zwischen Kanten, Qualifizierungen durch attributierte Kanten auf einfache Weise darstellen.

AGG als algebraisches Graphtransformationswerkzeug [Tae99] basierend auf dem Single-Pushout Ansatz ermöglicht das Spezifizieren einfacher Graphmuster, die durch Morphismensuche auf den Arbeitsgraphen abgebildet werden. Im Unterschied zu obigen Werkzeugen werden Kanten explizit identifiziert, wodurch wie in DRAGULA auch parallele Kanten zugelassen werden. Knoten, nicht jedoch Kanten können in AGG attributiert werden. Zur Schemadefinition werden in AGG Typgraphen [Tae03] spezifiziert, die ähnlich zu den Graphschemata von PROGRES aufgebaut sind.

Die bisher rein konzeptionell beschriebene Sprache DiaPlan [HM00] ermöglicht das Spezifizieren mittels Hypergraphgrammatiken. In diesem Ansatz sind hierarchische Graphen sowie n-äre Verbindungsstrukturen ein wesentlicher Bestandteil, da sogen. *nichtterminale* n-äre Kanten zu beliebigen Graphstrukturen umgeschrieben werden können. Da sowohl DRAGOS n-äre Verbindungsstrukturen darstellen als auch DRAGULA diese verarbeiten kann, lässt sich die Funktionalität von DiaPlan direkt abbilden.

4.8.2 Komposition von Graphmustern & rekursive Strukturen

Bei der Komposition von Graphmustern finden sich größere Unterschiede zwischen den einzelnen Graphtransformationswerkzeugen als im Fall der einfachen Mustersuche. PROGRES bietet insbesondere die Möglichkeit, zur Mustersuche Pfadausdrücke anzuwenden um komplexe Querbezüge zwischen zwei Variablenbelegungen zu prü-

fen. Diese können neben sequentiellen Kantentraversierungen auch Typprüfungen der erreichten Knoten, Alternativen sowie konjunktiv verknüpfte Teilausdrücke beinhalten. Auch Attributwerte von Knoten können im Zuge eines Pfadausdrucks überprüft werden. Pfadausdrücke sowie die einstellige und somit nur auf eine einzelne Variable bezogenen Restriktionen können explizit definiert und damit in verschiedenen Transformationsregeln referenziert werden. Fujaba bietet in diesem Kontext lediglich vereinfachte Pfadausdrücke, insbesondere ohne die Möglichkeit zur Attributprüfung an. Ergänzend hinzu kommt eine Möglichkeit zum Aufruf weiterer Regeln mittels sogen. *Kollaborationsmuster*.

Sowohl PROGRES als auch Fujaba bieten optionale und mengenwertige Knotenvariablen an, die abhängig von den übrigen Variablen des Musters nach „best effort“ belegt werden. Optionale Variablen können dabei ggf. unbelegt, mengenwertige Variablen mehrfach belegt werden. Durch negierte Bedingungen können in beiden Werkzeugen bestimmte Knoten- und Kantenbelegungen untersagt werden. AGG verallgemeinert diese Sprachkonstrukte einerseits zu amalgamierten Graphmustern[BFH87, TB93], bei denen Teilmuster in beliebiger Häufigkeit und mit wählbaren Querbezügen zum Regelkern angefragt werden können. Andererseits wird die Anwendbarkeit einer Transformationsregel durch negierte Graphmuster gesteuert, die eine höhere Ausdruckskraft besitzen als einzelne negierte Regelelemente.

Neuere Graphtransformationswerkzeuge bieten zusätzliche Ausdruckskraft einerseits durch Rekursion wie in VIATRA2 [CHM+02], wodurch zweistellige Pfadausdrücke auf n-äre Beziehungen verallgemeinert werden [VHV07]. Andererseits verfügt GrGen.net [GBG+06] über einen Mechanismus für zweistufige Grammatiken [HJG08]. Hierbei wird ein nichtterminaler Regelprototyp durch Anwendung von Graphgrammatikregeln passend zu einer terminalen Transformationsregel expandiert, die anschließend auf den Arbeitsgraphen angewendet wird. Auch auf diese Weise lassen sich n-äre transitive Verbindungen beschreiben.

Graphtransformationsregeln werden implizit existenzquantifiziert ausgewertet, d.h. das Ermitteln einer Auffindungsstelle genügt zur erfolgreichen Regelanwendung. Das Graphtransformationswerkzeug GROOVE [Ren03] bietet darüber hinaus auch allquantifizierte Teilmuster [Ren09], mittels derer Bedingungen für den gesamten Kontext eines Regelkerns beschrieben werden können. Dieser Ansatz geht über mengenwertige oder amalgamierte Teilregeln hinaus, da diese stets existenziell ausgewertet werden.

Darstellung in DRAGULA: Rekursive Suchmuster, zu denen auch Pfadausdrücke gehören, werden in DRAGULA durch Musterreferenzen dargestellt. Musterannotationen ermöglichen hierbei die konjunktive und disjunktive Kombination von Teilmus-

tern. Kardinalitätsbedingungen bilden zudem optionale und mengenwertige Strukturen ab, und verallgemeinern diese Sprachkonstrukte von einzelnen Knotenvariablen auf komplexe Teilmuster. Da Muster wie auch in VIATRA2 hierarchisch strukturiert werden können, übersteigt DRAGULA somit auch die Ausdruckskraft von AGG. Implizit lassen sich durch mehrfach negierte existenzielle Ausdrücke auch die von GROOVE angebotenen allquantifizierten Teilmuster realisieren.

4.8.3 Graphtransformationen

Der verändernde Anteil einer Graphtransformationsregel ist in verschiedenen existierenden Graphtransformationswerkzeugen sehr ähnlich aufgebaut. In den algebraischen Ansätzen DPO und SPO werden die modellierten Graphmuster durch Erzeugen und Löschen von Arbeitsgraphelementen hergestellt. Transformationswerkzeuge für attributierte Graphmodelle erlauben zudem die Veränderung von Attributwerten der erzeugten oder beibehalten Graphelemente. Aufgrund des komplexen Graphmodells benötigen Erstelloperationen in DRAGULA – anders als Graphtransformationswerkzeuge auf einfachen Graphmodellen – zusätzliche Angaben, etwa bzgl. des designierten enthaltenden Graphen.

PROGRES bietet darüber hinaus die Möglichkeit, mittels Einbettungsüberführungsoperationen bestehende Verbindungen umzuleiten, zu kopieren, oder zu entfernen. Hierdurch können Beziehungen zu Elementen des Arbeitsgraphen beeinflusst werden, die nicht Teil der Suchanfrage sind. Eine solche Funktion lässt sich in DRAGULA analog zu mengenwertigen Teilmustern realisieren. Durch die Möglichkeit zur Kantenumleitung wird auch eine eventuelle Attributierung beibehalten.

Das Werkzeug GROOVE unterstützt als selten angebotene Funktionalität das Kombinieren unterschiedlicher Knoten zu einem einzigen, der sämtliche Verbindungen und Attributwerte der Originalknoten vereinigt. Hierzu bietet DRAGULA keine direkte Unterstützung, allerdings wird entsprechende Funktionalität in Kapitel 6 benötigt und ist daher als Spracherweiterung realisiert worden. Nicht integriert ist ferner die in GROOVE verfügbare Funktionalität zur vollständigen Expansion von Transitionssystemen. Da dies eine hochgradig spezielle Eigenschaft darstellt, kann sie über entsprechende Erweiterungen bspw. mit Hilfe eines bereits verfügbaren Konstrukts zur Strukturkopie ergänzt werden.

Abbildung auf algebraische Graphgrammatiken

Algebraische Graphgrammatiken, insbesondere der für zahlreiche Werkzeugbeschreibungen verwendete Single-Pushout-Approach, lassen sich vollständig mit DRAGULA ausdrücken. Anders herum lässt sich zeigen, dass auch wesentliche Bestandteile DRA-

GULAs mit Hilfe des SPO dargestellt werden können, beide Ansätze also in Teilen einen äquivalenten Funktionsumfang abdecken. An dieser Stelle werden nur einfache DRAGULA Graphtransformationsregeln ohne geschachtelte Muster oder Musterreferenzen betrachtet. Komplexere Ausdrücke lassen sich durch Aufteilung in mehrere SPO-Regeln und graphische Hilfskonstrukte zur Kontrollflusssteuerung simulieren.

Um eine DRGAULA Transformationsregel algebraisch darzustellen, werden alle Variablen mit adjazenten Bedingungen als Knoten bzw. Kante der linken Regelseite hinzugefügt. Wie Abbildung 4.42 an einem einfachen Beispiel zeigt, werden neben Knoten- auch Graphvariablen als Regelknoten dargestellt, Kantenvariablen dagegen als Kanten. Die Verbindungsstruktur der linken Regelseite ergibt sich zum einen aus der Inzidenzbedingung des DRAGULA Musters, zum anderen auch aus den Enthaltenseinsbeziehungen.

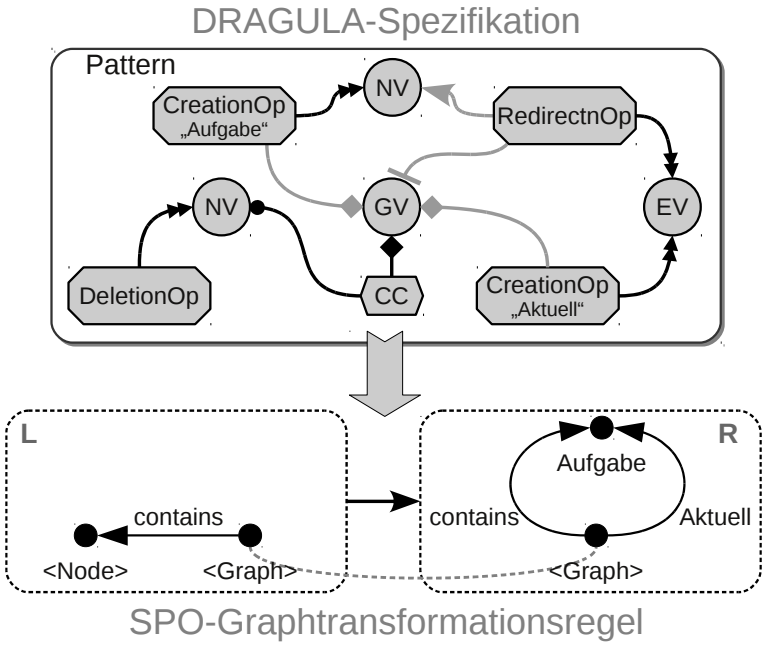


Abbildung 4.42: Abbildung einer DRAGULA Spezifikation auf den SPO

Eine derart konstruierte Regel ist anwendbar falls ein strukturerhaltender Morphismus zwischen der Regelseite L und dem jeweiligen Arbeitsgraphen vorliegt. Dieser muss die Typisierung der Elemente des Regelgraphen sowie des Arbeitsgraphen berücksichtigen. Falls Typbedingung mit gesetztem closure-Attribut, das auch Covariante Aktualtypen im Arbeitsgraphen zulässt, verwendet werden, ist eine spezielle Definition der Strukturerhaltung, etwa gemäß [LBE+07] nötig. Vererbungshierarchien von Kantentypen lassen sich so jedoch nicht darstellen.

Transformationsvorschriften werden in Form einer rechten Regelseite im SPO dadurch formuliert, dass alle Variablen, mit Ausnahme solcher mit adjazentem Löschoperator, einbezogen werden. Durch eine Transformation nicht veränderte Kanten werden mit isomorphen Bildern derjenigen linksseitigen Knoten verbunden mit denen die zugehörige Kante auf der linken Regelseite verbunden ist. Die Umleitung von Kanten wird dagegen durch unterschiedliche Konnektierung auf linker und rechter Regelseite ausgedrückt.

4.8.4 Ablaufsteuerung

In PROGRES werden einzelne Abfragen und Transformationsregeln durch explizite, textuell dargestellte Aufrufe miteinander verschaltet. Transformationen besitzen eine Schnittstelle mit Ein- und Ausgabeparametern zum typsicheren Datenaustausch. Ferner stellt PROGRES eine Ausdruckssprache sowie verschiedene Kontrollflusskonstrukte, u.a. Schleifen und Bedingungen, bereit. DRAGULA verfolgt dagegen einen vergleichsweise eingeschränkten Ansatz, da lediglich nach dem Erfolgsstatus einer Transformation verzweigt werden kann. Dennoch lassen sich PROGRES Sprachkonstrukte dadurch darstellen, dass Abbruch- oder Verzweigungsbedingungen als eigene Graphmuster bzw. atomare Regeln definiert werden.

Die Ausführung von PROGRES Transformationen erfolgt nichtdeterministisch und transaktional, d.h. dass bei Verletzung einer vorgegebenen Bedingung die zuletzt vorgenommen Transformationsschritte zurück genommen werden. Im Falle einer nichtdeterministisch erfolgten Auswahl kann so durch Backtracking nach alternativen Lösungswegen gesucht werden. Auch DRAGULA ist in der Lage bei nicht erfolgreicher Transformationsausführung Backtracking durchzuführen.

Fujaba, bzw. das hierfür eingesetzte Story-Driven Modeling (SDM) [FNTZ98], verwendet eine graphische Notation angelehnt an UML Aktivitätsdiagramme. Die gotoartigen Transitionen zwischen einzelnen Aktivitäten können durch Bedingungen, einschließlich dem Erfolg oder Misserfolg der vorgegangenen Transformationen, gesteuert werden. Zwischen den Aktivitäten besteht ein impliziter Datenfluss, d.h. alle Variablenbelegungen der Vorgängeraktivitäten können in Nachfolgeraktivitäten verwendet werden. Ferner können im Rahmen sogen. For-Each-Aktivitäten Transformationen für alle Auffindungsstellen ausgeführt werden.

VIATRA2 verwendet das Konzept Abstrakter Zustandsmaschinen (Abstract State Machines, ASM [BS03]) um, ähnlich zu obigen Systemen, Transformationsregeln zu einem Graphtransformationssystem zu kombinieren [VB07]. Insbesondere können die Parameter einer Transformationsregel je nach Aufrufart vorbelegt oder als freie Variablen verwendet werden ohne dass die Regeldefinition anzupassen wäre [BV06].

Das auf Modelltransformationen ausgerichtete Werkzeug GReAT [KASS03] ver-

wendet eine datenflussbasierte Sprache zur Regelkomposition. Transformationsregeln verfügen in GReAT über Ein- und Ausgabeports, die Eingabedaten in Form einzelner Elemente des Arbeitsgraphen liefern oder nachfolgenden Regeln bereitstellen. Die Abarbeitung erfolgt normalerweise getaktet, d.h. erst nach vollständiger Abarbeitung eines Pakets, wird das nächste in das System eingespeist.

Einen hybriden Ansatz von Kontroll- und Datenflussbeziehungen verwendet VMTS [LLMC04], das ähnlich wie Fujaba auf Aktivitätsdiagrammen zurück greift. Allerdings werden zur Parametrisierung von Transformationen gemeinsam genutzte Datenspeicher in Anlehnung an den UML2 Standard verwendet [LLMC06].

AGG bietet als algebraisch geprägtes Werkzeug keine explizite Kontrollflussunterstützung an, sondern steuert die Anwendbarkeit von Regeln allein durch entsprechende Anwendbarkeitsbedingungen. Durch Prioritäten kann zwischen zeitgleich anwendbaren Regeln deterministisch unterschieden werden.

Darstellung in DRAGULA: DRAGULA bietet eine zu PROGRES und Fujaba vergleichbare Unterstützung zur Ablaufsteuerung, die auf einer expliziten Kontrollflussmodellierung basiert.

- Datenflussbeziehungen sind pro Transition festzulegen, werden also nicht wie in PROGRES anhand lokaler Variablen und Ein- bzw. Ausgabeparametern ausgetauscht. Auch ein impliziter Datenfluss wie in Fujaba erfolgt nicht. Hierdurch reduziert DRAGULA den notwendigen Sprachumfang, da es keine zusätzlichen Konzepte wie regelübergreifende Variablen einführt, andererseits kann die Vorbelegung einer Variablen je nach gewähltem Kontrollflusspfad präzise gesteuert werden.

- Typwertige Parameter wie in VIATRA2 können durch den reflektiven Typgraphen in DRAGULA verwendet werden. Dieser blendet Typen als reguläre Graphelemente in den Laufzeitgraphen ein und erlaubt zudem Instanzprüfungen anhand von Inzidenzanalyse.

- Datenspeicher wie in VMTS sind zwar in DRAGULA nicht direkt vorgesehen, können aber bspw. durch gesonderte übergeordnete Graphen im DRAGOS Graphspeicher leicht simuliert werden.

4.8.5 Anpassbarkeit & Erweiterbarkeit

Im Kontext dieser Arbeit besteht ein besonderer Bedarf an die Anpassbarkeit und Erweiterbarkeit einer Graphtransformationssprache. DRAGULA ist strukturell und

architektonisch hierfür ausgelegt, allerdings bieten auch einige weitere Graphtransformationssprachen entsprechende Funktionalität an.

Fujaba bietet einen Mechanismus, um Spracherweiterungen in Form von Plugins dynamisch, d.h. ohne Veränderung des Grundsystems, zu ergänzen. Hierbei können Sprachkonstrukte einerseits syntaktisch sowohl in abstrakter Form als auch durch visuelle Diagrammelemente definiert werden. Andererseits wird durch entsprechende Ergänzungen des Codegenerators auch eine semantische Abbildung auf Quelltext ermöglicht. Ferner kann durch Austausch der verwendeten Quelltextvorlagen auch die gesamte Codegenerierung eigener Vorgaben entsprechend angepasst werden.

In PROGRES besteht zur verbesserten Abstraktion zwischen Sprachsyntax einerseits, und Codegenerator bzw. dem interpretativen Debugger andererseits, mit dem sogen. PROGRES Graph Code (PGC, [SWZ95]) eine explizite Zwischenschicht. Hierdurch kann PROGRES prinzipiell durch Verwendung des IPSEN Systems zur Erzeugung syntaxgesteuerter Editoren leicht um zusätzliche Sprachkonstrukte ergänzt werden. Durch eine Abbildung auf die bestehende Graph Code Syntax können auch semantische Abbildungen auf höherer Abstraktionsebene erfolgen. Nachteilig an dem gewählten Ansatz ist jedoch das verhältnismäßig niedrige Abstraktionsniveau des Graphcodes, der bspw. einzelne Graphzugriffe und Mengenoperationen auf den Ergebnissen beschreibt. Hierdurch bleibt die Abbildung neuer Sprachkonstrukte entsprechend aufwändig, zudem müssen Spracherweiterungen durch Codegenerierung und Kompilation fest in das PROGRES System integriert werden.

Das auf Codegenerierung basierende VMTS bietet ebenfalls Anpassungsmöglichkeiten hinsichtlich der Generierung an. VMTS bietet jedoch lediglich die Abbildung hochsprachlicher Konstrukte auf eine abstrakte Syntax (Document Object Model – DOM) an. Von diesem DOM ausgehend wird durch die zugrundeliegende .NET-Technologie Quellcode erzeugt. Praktisch zeigt sich dieser Ansatz weniger übersichtlich als bei der Verwendung von Quelltextvorlagen, wie etwa in Fujaba. Insbesondere eine Anpassung der bestehenden Implementierung ist so nur schwer möglich.

4.9 Zusammenfassung

Dieses Kapitel hat mit der Kernsprache für Graphtransformationen DRAGULA einen wesentlichen Bestandteil des in Kapitel 3 umrissenen Gesamtkonzepts vorgestellt. DRAGULA stellt als Kernsprache eine *Grundlage* für Graphtransformationen bereit, die hierdurch in eine ausführbare Form übertragen werden können. Zusätzlich wird ein umfangreicher Satz an Sprachkonstrukten aktueller Graphtransformationswerkzeuge angeboten, der zudem fallweise durch Erweiterungen ergänzt werden kann. Die wesentlichen Eigenschaften des Ansatzes werden im Folgenden den Anforderun-

gen aus Kapitel 1 gegenübergestellt.

Abstraktionsniveau: Die durch den vorgeschlagenen Ansatz realisierbaren Spezifikationssprachen liegen auf einem hohen Abstraktionsniveau, das vornehmlich deklarativ spezifizierte Operationen auf Graphen umfasst. DRAGULA greift dieses Abstraktionsniveau auf, in dem die Kernsprache grundlegende Graphoperationen wie etwa Mustersuchen und Veränderungsoperationen anbietet. Dabei bietet die Kernsprache keine stark verfeinerte Funktionalität wie etwa die in PROGRES verfügbaren komplexen Attribut- und Pfadausdrücke. Stattdessen beschränkt sich DRAGULA auf einen überschaubaren Satz grundlegender und zu komplexeren Strukturen kombinierbarer Sprachkonstrukte. Hierdurch wird trotz des hohen Abstraktionsniveaus eine allgemeine Nutzbarkeit sichergestellt.

Anpassbar- und Erweiterbarkeit: Aufgrund der Divergenz graphbasierter Spezifikationssprachen entstehen hohe Ansprüche an die Anpassbarkeit und Erweiterbarkeit einer allgemeinen Realisierungsgrundlage. DRAGULA kann aufgrund seiner Modularisierung leicht sprachseitig erweitert werden, insbesondere in dem neue Sprachkonstrukte definiert und mit der bestehenden Kernsprache in Bezug gesetzt werden. Wie das folgende Kapitel 5 können auch operationale Realisierungen dieser Sprachkonstrukte leicht bereitgestellt oder bestehende Konstrukte redefiniert werden.

Unterstützung komplexer Graphmodelle: Eine Vielzahl von Anwendungsszenarien basieren auf komplexen graphbasierten Datenstrukturen, die sich nicht durch einfache Graphmodelle wie den gakk-Graphen (vgl. Kapitel 2) ausdrücken lassen. Um benötigte Eigenschaften des Graphmodells nicht aufwändig simulieren zu müssen, basiert DRAGULA auf dem reichhaltigen Modell des DRAGOS Graphspeichers. Hierdurch wird eine direkte Abbildbarkeit auch komplexer Datenstrukturen sichergestellt.

Im weiteren Verlauf dieser Arbeit wird zunächst in Kapitel 5 die DRAGULA *Ausführungsmaschine* vorgestellt. Der Schwerpunkt liegt hier auf den architektonischen Besonderheiten des Systems. In diesem Zusammenhang wird auch der modellgetriebene Erweiterungsansatz der Maschine behandelt. Zur Erleichterung der Anwendung des DRAGULA Systems zeigt Kapitel 6 eine bereitgestellte Transformationssprache zur Beschreibung von Übersetzern, mittels derer DRAGULA-Spezifikationen aus vorhandenen syntaktischen Spezifikationsdokumenten anderer Modellierungssprachen abgeleitet werden können.

5 Die DRAGULA Ausführungsmaschine

Dieses Kapitel stellt die Ausführungsmaschine der Kernsprache DRAGULA vor, was in Abbildung 5.1 graphisch in den Kontext der Arbeit eingeordnet wird. Neben der Implementierung der im vorherigen Kapitel vorgestellten Sprachkonstrukte bietet diese Ausführungsmaschine zwei wesentliche Parametrisierungsmöglichkeiten: Erstens ist die jeweilige Sprachimplementierung aufgrund der komponentenbasierten Architektur flexibel austauschbar. Diese Eigenschaft ist wünschenswert um zwecks Effizienzsteigerung auf die Spezifika des gewählten Hintergrundspeichers Bezug nehmen zu können. Zweitens können Erweiterungen der Kernsprache leicht in die Ausführungsmaschine integriert werden. Hierfür wird eine syntaktische Spezifikationsmöglichkeit bereitgestellt, die eine Definition von Spracherweiterungen auf hohem Abstraktionsniveau ermöglicht. Zudem kann durch Rückführung einer Spracherweiterung auf die Kernsprache eine modellseitige Beschreibung des Ausführungsverhaltens erfolgen.

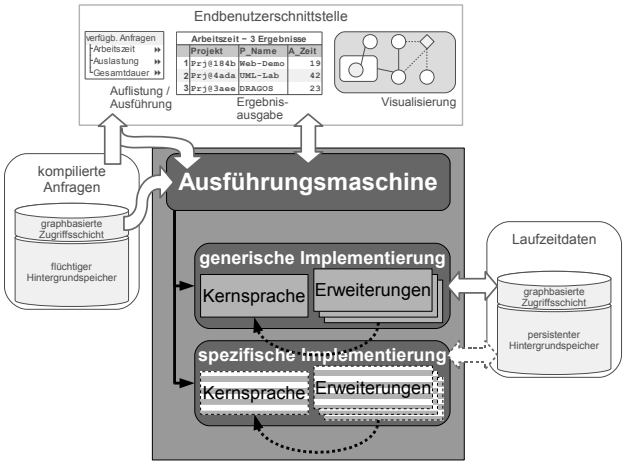


Abbildung 5.1: Einordnung der Ausführungsmaschine (analog zu Abbildung 1.6)

5.1 Anforderungen & Vorbemerkungen

Die effiziente Nutzung der verfügbaren Infrastruktur ist in Verbindung mit der Adaption bekannter Ansätze zur Realisierung von Graphtransformationen eine wichtige Voraussetzung, um eine effizient nutzbare Ausführungsmaschine bereitzustellen. Im Folgenden wird auf unterschiedliche Aspekte dieses Grundsatzes eingegangen, aus denen sich die letztlich bereitgestellte Werkzeuglösung ergibt. Zur Infrastruktur zählt hier insbesondere der in Abschnitt 2.4 vorgestellte Graphspeicher DRAGOS, der als graphbasierte Zugriffsschicht auf verschiedene Datenpersistenzlösungen aufsetzen kann.

5.1.1 Realisierung von Graphtransformationen

Ausführbare Realisierungen von Graphtransformationssystemen gehen zeitlich bis in die 1970er Jahre zurück, wie in Abschnitt 5.7 gezeigt wird. Insbesondere zur *Effizienzsteigerung* der erstellten Lösung ist seitdem erheblicher Aufwand betrieben worden um der zugrundeliegenden theoretischen Problematik entgegenzuwirken: So ist die Suche nach Auffindungsstellen eines Graphmusters in einer Datenbasis direkt rückführbar auf das *Teilgraphisomorphieproblem*, dessen NP-Vollständigkeit nachgewiesen ist [Epp99]. Effiziente Realisierungen von Graphtransformationssystemen müssen daher insbesondere die Skalierbarkeit der *Mustersuche* auch für große Anfragen gewährleisten, wohingegen andere Aspekte wie Transformationen oder auch die Ablaufsteuerung weniger effizienzkritisch sind.

Zahlreiche existierende Graphtransformationswerkzeuge sehen miteinander vergleichbare Maßnahmen zu Effizienzsteigerung vor. Dabei ist es unerheblich ob die Werkzeuge *generativ* arbeiten, also bspw. aus einem graphischen Suchmuster Quellcode zur Auswertung der Anfrage generieren, oder aber interpretativ mittels einer Auswertungsmaschine vorgehen. Der Ansatz besteht häufig darin, Elemente einer zu suchenden Struktur als Variablen aufzufassen, die schrittweise geeignet mit Werten des Laufzeitgraphen belegt werden. Gewählte Werte dürfen dabei Einschränkungen bspw. bezüglich des angefragten Typs oder der Verbindung zu Werten bereits belegter Variablen nicht widersprechen. Die Reihenfolge dieser Belegung beeinflusst dabei wesentlich das Laufzeitverhalten, da die jeweiligen Entscheidungen bspw. geforderte Inzidenzeigenschaften miteinander in Beziehung stehen.

Speziell bei der *interpretativen* Vorgehensweise zur Auswertung von Graphmustern und -transformationsregeln besteht die Möglichkeit, Laufzeitinformationen in die Optimierung der Auswertung einfließen zu lassen. Im generativen Fall stehen i.A. lediglich vergleichsweise grobe Schemainformationen über *potenzielle* Graphstrukturen zur Verfügung. Dem gegenüber kann bei der Interpretation die Anzahl *tatsächlich* vorhan-

dener Graphelemente verwendet werden um die Belegungsreihenfolge von Mustervariablen festzulegen.

Da die hauptsächlich genutzte DRAGULA-Realisierung dem interpretativen Ansatz folgt, lassen sich bestehende und verbreitet eingesetzte Optimierungsansätze relativ leicht wiederverwenden. Eine gewisse Unterscheidung zu üblichen Graphtransformationswerkzeugen besteht in der streng nach Variablen und zu erfüllenden Bedingungen unterscheidenden DARGULA-Sprachstruktur. Hierdurch wird einerseits die Implementierung individueller Sprachkonstrukte erleichtert, andererseits lassen sich Zusammenhänge eines Graphmusters schwieriger und nur unter Kenntnis der eingesetzten Sprachkonstrukte ermitteln. Wesentliche Eigenschaften des DRAGULA-Interpreters werden in Abschnitt 5.3 behandelt.

5.1.2 Nutzung eines Graphspeichers

Obige Effizienzgesichtspunkte beziehen sich auf die algorithmische Realisierung von Graphtransformationssystemen. Bei der Verwendung von DRAGOS ist zudem der jeweils eingesetzte Hintergrundspeicher zu berücksichtigen. Insbesondere erfordert DRAGOS den Einsatz *relationaler Datenbanken* um transaktionale Eigenschaften wie der atomaren Ausführung sowie Isolation in Bezug auf Nebenläufigkeit bereitzustellen. Die seitens DRAGOS angebotene API-Funktionalität gemäß dem DRAGOS-Graphmodell (vgl. Abschnitt 2.4) ist jedoch nicht mit der wünschenswerten Effizienz auf entsprechende SQL-Kommandos dieser Datenbanken abbildbar. So werden zur Anfrage selbst eines einfachen Graphmusters zahlreiche API-Aufrufe durchgeführt, um bspw. zu einem gegebenen Graphelement adjazente Elemente zu ermitteln oder auf bestehende Verbindungen zu prüfen. Da SQL-Anfragen mit vergleichsweise hohen Laufzeitkosten verbunden sind, ergibt sich – trotz Optimierung des Anfrageverhaltens – ein erheblicher Performanzeinbruch.

Die geschilderte Problematik, die in Abschnitt 5.4.3 quantitativ untersucht wird, ist in Abbildung 5.2 im Überblick skizziert. Hierbei wird der Effekt deutlich, den eine naiv realisierte Graphmustersuche für den verwendeten Hintergrundspeicher hätte. Sowohl für die jeweilige Auswahl von Kandidaten aus der Datenbasis, als auch für die Prüfung auf Typkonformität und der gewünschten Konnektierung wären hierfür einzelne Datenbankanfragen nötig. Auch ist bei der Anfrage von Kanten in diesem Fall eine Verknüpfung der dargestellten Tabellen nötig. Trotz einer in der Praxis deutlich reduzierteren Zugriffsform bleibt die grundsätzliche Problematik bestehen, so dass sich rein auf die graphbasierte DRAGOS-API stützende Anfrage- und Transformationsregeln nicht für komplexe Musterstrukturen einsetzen lassen.

Ein alternativer Ansatz zur Ausführung von Graphtransformationen besteht darin, den jeweils verwendeten Hintergrundspeicher direkt zu nutzen. Für SQL-basierte

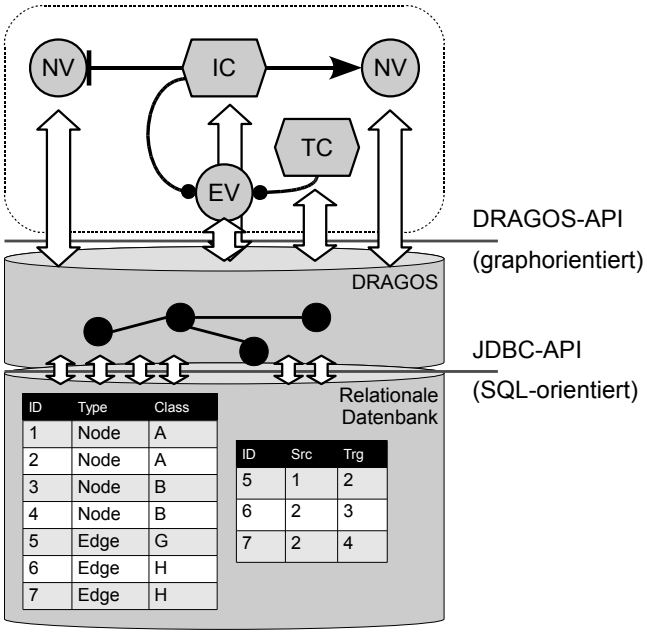


Abbildung 5.2: Problematische Mustersuche auf Relationalen Datenbanken

Datenbanken können bspw. entsprechende Anfragen eingesetzt werden. Hierbei werden Graphmuster gemäß der internen Tabellenstruktur in SQL-Anfragen übersetzt und vom zugrundeliegenden Datenbanksystem direkt ausgewertet. Hierdurch entfällt obige schrittweise Belegung von Mustervariablen und damit die jeweils verbundenen Anfragen an den Hintergrundspeicher. Stattdessen wird eine zuvor erstellte Gesamtanfrage vom Hintergrundspeicher in einem einzelnen Verarbeitungsschritt ausgewertet. Hierdurch entfallen wesentliche Kostenfaktoren bei der Synchronisation des DRAGULA-Systems mit der Datenbank. Allerdings ist es Aufgabe des Hintergrundspeichers eine geeignete Optimierung des SQL-Code vorzunehmen, soweit dies bei der Anfrageübersetzung nicht bereits erfolgt. Einen geeigneten Ansatz zur Realisierung eines solch spezifischen Hintergrundspeichers zu ermitteln ist Gegenstand von Abschnitt 5.4.

5.1.3 Erweiterbarkeit der Ausführungsmaschine

Orthogonal zu obigem Effizienzgesichtspunkt steht die Möglichkeit, die DRAGULA-Ausführungsmaschine durch weitere eigenimplementierte Sprachkonstrukte zu erweitern. Dies ist nötig, wenn die in Kapitel 4 beschriebene Reduktion auf bereits verfügbare Sprachkonstrukte nicht möglich ist, bspw. falls es sich, wie bei der Erweiterung um Attributausdrücke, um eine echte Erweiterung der Sprache handelt. Auch

aus Effizienzgesichtspunkten kann eine Eigenimplementierung sinnvoll sein da hierdurch deutlich mehr Einfluss auf die operationale Auswertung einzelner Sprachkonstrukte möglich ist. Die Reduktion auf bestehende Konstrukte kann in diesen Fällen zusätzlich als Referenzsystem bspw. zu Testzwecken genutzt werden.

Der Ansatz der DRAGULA Ausführungsmaschine besteht darin, die Entwicklung syntaxbezogener Anteile einer Spracherweiterung effektiv zu unterstützen. Die dynamische Realisierung kann durch Reduktion oder durch manuelle Implementierung erfolgen. Softwarearchitektonische Details zur Erweiterbarkeit der Ausführungsmaschine sowie eine integrierte Entwicklerunterstützung stellt Abschnitt 5.5 vor.

5.2 Allgemeiner Architekturüberblick

Zunächst wird in diesem Abschnitt ein Überblick der DRAGULA Architektur gegeben, bevor die nachfolgenden beiden Abschnitte auf Details der jeweiligen Realisierungsvariante eingehen. Den jeweils verfügbaren Sprachrealisierungen gemeinsam ist eine Reihe von Programmierschnittstellen, über die eine Ansteuerung der Mustersuche bzw. der Transformationsausführung erfolgt. Des Weiteren greifen alle Varianten gleichermaßen unter Nutzung der DRAGOS-API auf die auszuwertenden DRAGULA-Spezifikationen zu.

5.2.1 Graphbasierte Darstellung von DRAGULA-Spezifikationen

DRAGULA-Spezifikationen, die unter Verwendung der hier vorgestellten Ausführungsmaschine ausgewertet werden sollen, sind in Form ihrer abstrakten Syntaxdarstellung im DRAGOS Graphspeicher abzulegen. Auf diese Weise stellt DRAGOS nicht nur eine allgemeine Schnittstelle zum Zugriff auf die zu verarbeitenden Laufzeitdaten, sondern auch auf die darauf anzuwendenden Spezifikationsdokumente bereit.

Die hierzu nötige Abbildung der DRAGULA-Syntax aus Kapitel 4 auf das DRAGOS Graphmodell (vgl. Abschnitt 2.4) erfolgt durch Deklaration einer Graphentitätsklasse pro Element der Syntaxdefinition. Dabei werden eigenständige Entitätstypen wie bspw. Variablen, Bedingungen und Operatoren als Knotenklassen dargestellt. Die jeweils inzident definierten Verbindertypen, wie bspw. die Restricts-Beziehung werden durch Kantenklassen repräsentiert. Entitätstypen, deren Instanzen als Behälter weiterer Entitäten verwendet werden können, sind als Graphklassen formuliert – hierzu zählen demnach die Syntaxelemente für Muster und Regeln. Abbildung 5.3 zeigt beispielhaft das DRAGOS Graphschema zu dem in Abbildung 4.5 auf Seite 88 gezeigten Metamodellfragment. Im Unterschied zur fachlichen Darstellung enthält das Graphschema – aufgrund des DRAGOS Graphmodells – keinerlei Enthaltenseinsbeziehung

zwischen der Graphklasse Pattern und den darin enthaltenen Elementen. Stattdessen kann ein Graph *beliebig* typisierte Elemente aufnehmen. Zusätzlich ist die Kantenklasse Restricts nicht wie im Metamodell endseitig beschriftet, stattdessen wird die Beziehung der verbundenen Elemente durch die Kantenrichtung angegeben.

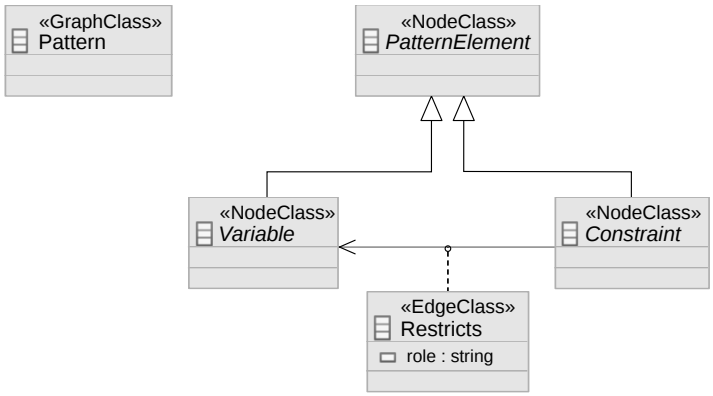


Abbildung 5.3: Graphschema der DRAGULA-Definition aus Abbildung 4.5

5.2.2 Technische Realisierungsarchitektur

Abbildung 5.4 zeigt die Realisierungsarchitektur der DRAGULA Ausführungsmaschine anhand des Sprachkonstrukts Variable. Hierbei ist zwischen dem für alle Sprachrealisierungen gleichermaßen verwendeten Anteil sowie Klassen spezifisch für die jeweilige Realisierung zu unterscheiden.

Der realisierungsunabhängige Anteil umfasst in Abbildung 5.4 eine Schnittstelle *Variable* sowie eine entsprechende Implementierung BaseVariable. Durch diese Implementierungsartefakte wird syntaktische Zugriffsfunktionalität bereitgestellt, etwa die exemplarisch dargestellte Methode zur Auflistung aller adjazenten Bedingungen getConstraints. Die Basisimplementierung verweist zudem auf eine DRAGOS Graphentität, die zur persistenten Ablage des Spezifikationsdokuments verwendet wird. Zugriffe auf die Synatxstruktur, bspw. durch obige Methode, werden auf Anfragen dieses Speicherobjekts auf Basis der DRAGOS-API realisiert. Im konkreten Beispiel würden also alle adjazenten Knoten der Knotenklasse Constraint ermittelt, die durch eine Kante des Typs Restricts erreicht werden.

Die fachliche Sprachrealisierung, die insbesondere unabhängig von der syntaktischen Analyse erfolgt, wird in gesonderten Klassen GenericVariable bzw. RelDBVariable bereitgestellt. Diese stellen ebenfalls die Basisschnittstelle Variable und die darin definierte syntaktische Funktionalität bereit, delegieren entsprechende Aufrufe jedoch

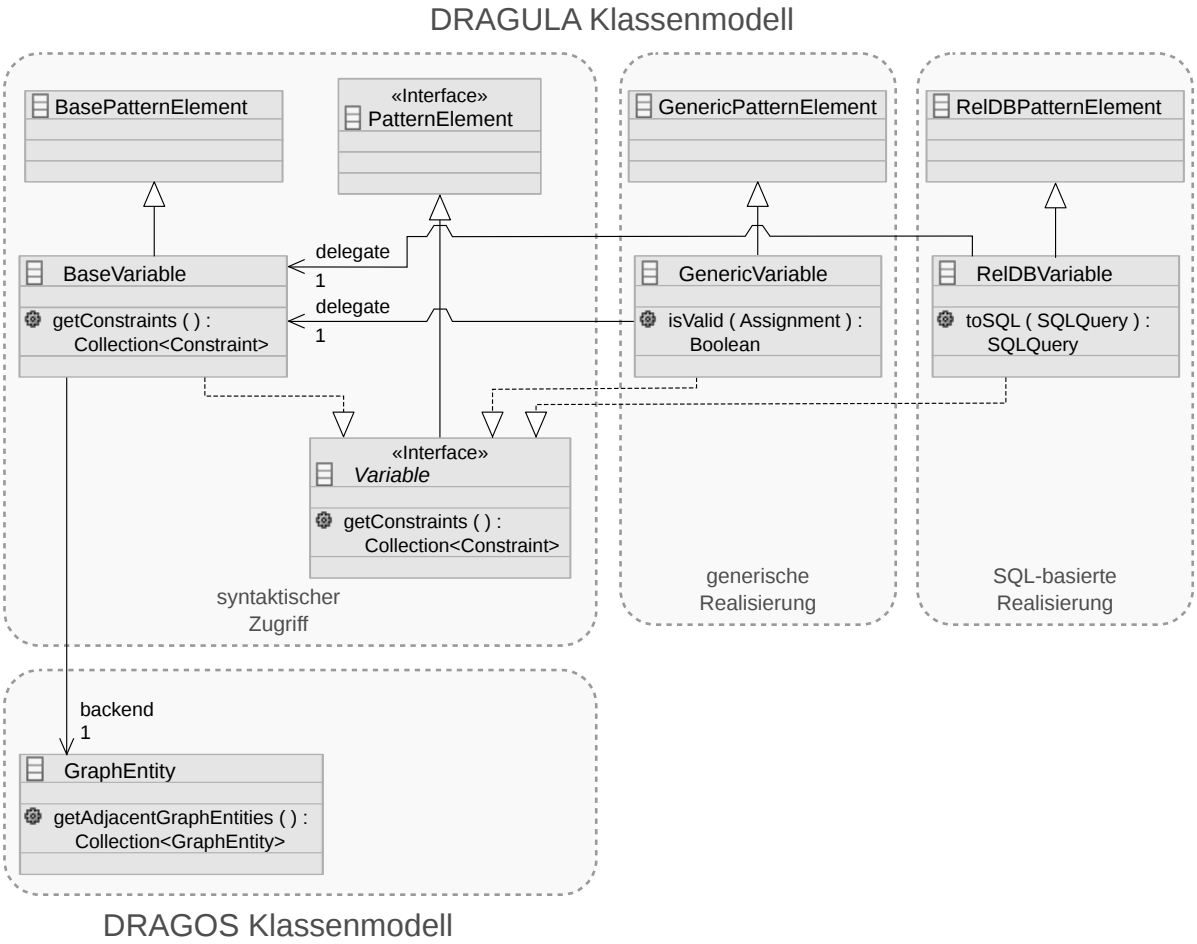


Abbildung 5.4: Beispiel zur Hierarchie der Implementierungsklassen

direkt an eine Instanz der Basisklasse BaseVariable. Zusätzlich definieren realisierungsspezifische Klassen zusätzliche Methoden, etwa isValid der Generischen Realisierung zur Gültigkeitsprüfung einer ausgewählten Variablenbelegung. Die SQL-basierte Implementierung stellt dagegen eine Methode zur Generierung von SQL Textfragmenten bereit.

5.3 Generische Realisierung

Als erste Sprachrealisierung stellt dieser Abschnitt die sogenannte *Generische Realisierung* vor. Diese greift auf die zu verarbeitenden Laufzeitdaten ausschließlich durch Nutzung der DRAGOS-API zu, so dass diese Realisierungsvariante keinerlei Abhängigkeiten zu spezifischen Hintergrundspeichern besitzt. Dem gegenüber stellt der fol-

gende Abschnitt eine spezifische Lösung für SQL-kompatible Datenbanken als Hintergrundspeicher vor.

Bei der Generischen Realisierung handelt es sich um einen Interpreter, der mittels DRAGOS zugreifbare Laufzeitdaten gemäß einer DRAGULA-Spezifikation auswertet. Als Rückgabewert eines Interpreteraufrufs entstehen bei Mustersuchen und Transformationen entweder einzelne oder Mengen von *Auffindungsstellen*, je nach Auswertungsart. Endergebnisse eines Regelaufrufs sind dagegen einzelne Schnittstellenbelegungen.

Die nachfolgenden Abschnitte stellen verschiedene Aspekte näher vor, die sich auf die jeweiligen Sprachbestandteile DRAGULAs beziehen. Dabei werden neben der angebotenen Funktionalität auch einige zur Optimierung des Interpreters verfolgte Ansätze diskutiert.

5.3.1 Einfache Mustersuche

Zur Auswertung einfacher Graphmuster, d.h. ohne Musterschachtelungen, wird eine schrittweise Zuweisung von Graphelementen zu dessen Variablen vorgenommen. Das Grundprinzip besteht darin, zu jeder Variablen zunächst eine Menge geeigneter *Kandidaten* zu bestimmen, welche die direkt inzidenten Bedingungen nicht verletzen. Aus dieser Menge wird ein Element nichtdeterministisch entnommen und der zu belegenden Variablen vorläufig zugewiesen. Sollten weitere Variablen des Musters nicht geeignet belegt werden können, wird die getroffene Entscheidung zurückgenommen und stattdessen ein weiteres Element der Kandidatenmenge ausgewählt – dies wird als *internes Backtracking* bezeichnet[1].

Die Bestimmung geeigneter Kandidaten einer Variablen erfolgt auf Basis ihrer inzidenten Bedingungen und der bereits im Vorfeld zugewiesenen Variablen. Zur Ermittlung einer Kandidatenmenge wird für jede Bedingung die möglichen Belegungen der fraglichen Variablen bestimmt, welche die Bedingung nicht verletzen würden. Die Gesamtmenge aller gültigen Variablenbelegungen ergibt sich somit aus dem mengenwertigen Schnitt der pro Bedingung ermittelten Kandidatenmenge. Insbesondere kann im Fall eines leeren Schnitts die Variable *nicht* belegt werden, was zu internem Backtracking und der Neuzuweisung bereits belegter Variablen führt. Ebenfalls zulässig ist es, dass zu einer Bedingung *keine* Kandidatenmenge zu einer Variablen bestimmt, etwa weil die Bestimmung einer hinreichend kompakten Kandidatenmenge nicht effizient möglich wäre. Dies ist insbesondere dann der Fall, falls eine Bedingung die Belegung weiterer Variablen benötigt. Liefert keine inzidente Bedingungen

[1] Der Begriff „intern“ bezieht sich auf die Lokalität des Backtrackings innerhalb eines DRAGULA-Musters, im Unterschied zum externen Backtracking zwischen Regelaufrufen (vgl. Abschnitt 4.5).

einer Variable ein Kandidatenergebnis, wird die Variable in der Mustersuche *verschoben*, also zunächst nach Belegungen anderer Variablen gesucht. Ist auch dies nicht erfolgreich wird schließlich eine *globale Suche* vorgenommen, bei der alle gespeicherten Graphelemente sukzessiv auf Verletzung jeder inzidenten Bedingung geprüft werden. Dies ist i.d.R. mit hohem Berechnungsaufwand verbunden und daher zu vermeiden.

Abbildung 5.5 zeigt die Mustersuche der Generischen Realisierung am Beispiel der mittigen, hervorgehobenen Knotenvariablen. Hierbei wird von einer bereits vorhandenen Belegung der Graphvariablen (oben, `g1`) und der rechten Knotenvariablen (`n2`) ausgegangen. Neben der gewählten Belegung zeigt das abgerundete Kommentarfeld außerdem mögliche weitere Kandidaten in geschweiften Klammern. Die Kantenvariable sowie die durch eine Isomorphiebedingung angeschlossene Knotenvariable (oben rechts) sind bisher nicht belegt.

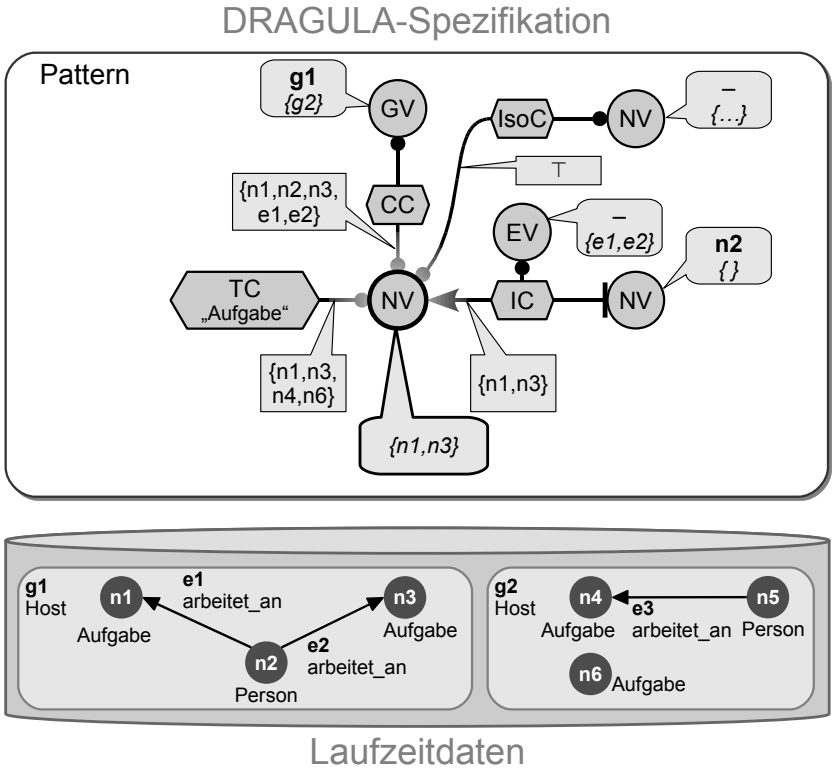


Abbildung 5.5: Kandidatenermittlung anhand inzidenter Bedingungen

Da die zu belegende Variable Ziel einer Inzidenzbedingung ist, kommen alle Graphelemente als Kandidaten in Frage, die mit dem Wert der Startvariablen durch eine Kante verbunden sind – hier also {`n1`,`n3`}. Analog stellt die Typbedingung alle entsprechend typisierten Knoten sowie die Enthaltenseinsbedingung alle enthaltenen Graphelemente bereit. Für die Isomorphiebedingung kann an dieser Stelle keine Kan-

didatenmenge ermittelt werden, da die ebenfalls angeschlossene Variable noch nicht belegt ist. Dies wird in der Abbildung durch das Symbol $\top$ gekennzeichnet. Da die resultierende Kandidatenmenge *allen* angeschlossenen Bedingungen genügen muss, ergibt sich diese aus dem *Schnitt* aller bislang ermittelten Teilmengen. Demzufolge können Belegungen aus der Menge {n1,n3} ausgewählt werden.

Zur Effizienzsteigerung bietet die Generische Realisierung eine Möglichkeit, Anfrageergebnisse inkrementell zu liefern. So kann zunächst eine bestimmte Höchstanzahl an Auffindungsstellen ermittelt werden, woraufhin die verbleibenden Kandidatenmengen der Mustersuche zwischengespeichert werden. Anschließend können so weitere Ergebnisse der Anfrage bestimmt werden, sollte dies im Nachhinein – bspw. aufgrund externen Backtrackings – nötig werden.

5.3.2 Komponierte Mustersuche & Gültigkeitsprüfung

Die Komposition einfacher Muster wird durch die DRAGULA-Ausführungsmaschine direkt durch rekursive Aufrufe der Mustersuche realisiert. Im Hinblick auf eine effiziente Realisierung ist dabei zu unterscheiden, ob Untermuster *erschöpfend*, d.h. durch Ermittlung aller möglichen Auffindungsstellen ausgewertet werden oder nicht. Hierbei können die in Abschnitt 4.3 auf Seite 97 vorgestellten Musterannotationen herangezogen werden: Bei Angabe einer Kardinalitätsbedingung mit festgelegter oberer Schranke genügt es somit, maximal *eine* weitere Auffindungsstelle zu ermitteln, da diese insgesamt zu einer Ungültigkeit der Mustersuche für das jeweilige Untermuster führen würde. Ferner stellen Muster mit unbeschränkter Kardinalität erfahrungsgemäß lediglich boolesche Bedingungen dar, die zum Nachweis des Vorhandenseins bestimmter Teilgraphen dienen. Zu solchen Untermustern wird standardgemäß zunächst nur *eine* Auffindungsstelle ermittelt. Obige Technik zur Zwischenspeicherung und Fortsetzung von Suchanfragen kann eingesetzt werden um die vollständige Auswertung fallweise anzufordern. Ist durch eine Musterannotation eine *Mindestanzahl* verschiedener Auffindungsstellen gefordert, so ist zudem mindestens diese Anzahl an Ergebnissen zu bestimmen. Ein als Aggregation gekennzeichnetes Muster führt zu einer erschöpfenden Anfrage um alle möglichen Auffindungsstellen in die weitere Auswertung einfließen zu lassen.

Eine zweite Optimierungsmöglichkeit besteht in einer passenden Auswahl bei der Auswertung von Untermustern. So genügt bei Angabe einer Oder-Bedingung die Auswertung *eines* Untermusters um die Gültigkeit der Auffindungsstelle des umgebenden Musters nachzuweisen. Analog hierzu kann im Falle einer Und- sowie einer Exklusiv-Oder-Bedingung ein frühzeitiger Abbruch der Mustersuche erfolgen, falls ein ungültiges bzw. ein weiteres gültiges Untermuster ermittelt werden kann. Die *Reihenfolge* der Auswertung wird nicht weiter festgelegt, kann allerdings heuristisch auf Basis

der jeweils zu ermittelnden Auffindungsstellen und der jeweiligen Variablenanzahl erfolgen. Erstere Kenngröße ergibt sich dabei wie oben beschrieben aus der Annotation des Untermusters, letztere aus dessen Struktur. Muster mit kleinen Werten sind vorzuziehen, da zu diesen mit möglichst wenig Rechenschritten eine gültige Auffindungsstelle ermittelt werden kann, bspw. in dem wenige Variablen belegt werden müssen.

5.3.3 Transformationen

Transformationen werden in DRAGULA durch Operatoren in aufeinanderfolgenden Phasen sowie jede einzelne Phase zweistufig ausgeführt. Die Phaseneinteilung erfolgt statisch auf Basis sogen. *Meta-Attribute*, die Elemente des DRAGULA-Graphschemas entsprechend markieren. Gegenwärtig wird hierbei eine einfache numerische Priorisierung vorgenommen.

Die zweistufige Ausführung von Operatoren einer Stufe wird durch entsprechende Schnittstellenmethoden ermöglicht. Eine Operatorimplementierung darf in der ersten Phase lediglich Änderungen vorbereiten, d.h. notwendige Informationen aus dem Laufzeitgraphen ermitteln ohne diesen zu verändern. In der zweiten Phase sind dann lediglich Änderungen, aber keine Analysen des mittlerweile durch andere Operatoren veränderten Laufzeitgraphen zulässig.

5.3.4 Ablaufsteuerung

Die Ablaufsteuerung ist in zwei Komponenten gegliedert und zwar der Verwaltung von Kontrollflüssen zwischen Regelaufrufen sowie den damit verbundenen Datenflüssen. Die Sprachrealisierung verwaltet hierbei die notwendigen Laufzeitdaten wie Aufruf- und Datenkeller und steuert die Auswertung der jeweils aktuellen Regel an. Zur nichtdeterministischen Transformationsausführung werden zudem mögliche alternative Anwendungsstellen einer Atomaren Regel im Kontext des Aufrufkellers abgelegt und zudem bei jeder Regelanwendung entsprechende Versionspunkte im Graphspeicher gesetzt. Sollte Backtracking eintreten wird der Zustand des Laufzeitgraphen durch den Graphspeicher zurückgesetzt und die nächste verfügbare Anwendungsstelle verwendet.

Anzumerken ist, dass die Unterstützung der nichtdeterministischen Regelausführung derzeit zwar die hier beschriebene Funktionalität bereitstellt, jedoch nicht auf Effizienz hin ausgerichtet ist. Insbesondere existiert, anders als in logikorientierten Programmiersprachen oder auch in PROGRES, keine Möglichkeit zum Abschneiden von Suchbäumen, dem sogen. *Cut* [Llo84]. Hierdurch wird zwar die praktische Anwendbarkeit nichtdeterministischer Kontrollstrukturen reduziert, allerdings ergeben

sich aus praktischen Erfahrungen im graphbasierten Werkzeugbau ohnehin wenige Anwendungsfelder für diese Spracheigenschaft.

5.3.5 Effizienzsteigerung der Generischen Realisierung

Insbesondere die Graphmustersuche stellt aufgrund ihrer Berechnungskomplexität einen zeitlich kritischen Aspekt bei der Ausführung von Graphtransformationssystemen dar. Um eine entsprechend effiziente, dennoch aber von der zugrundeliegenden Graphspeicherrealisierung unabhängige Implementierung bereitstellen zu können, sind entsprechende Optimierungsmöglichkeiten zu verfolgen. Hierbei kann auf bereits vorhandene Ansätze aus dem Bereich angewandter Graphgrammatiken zurückgegriffen werden.

Allgemein profitieren sowohl die Reihenfolgeoptimierung als auch die Kandidatenermittlung von *Laufzeitinformationen*, die in der Generischen Realisierung als Interpreter durch Auswertung des Laufzeitgraphen ermittelt werden können. Gängige Ansätze zur Codegenerierung können dagegen nur auf statische Information zurückgreifen, die sich bspw. aus dem zugrundeliegenden Graphschema ergeben. Diese sind insofern unscharf, als dass sie nur potenzielle Laufzeitzustände beschreiben, jedoch keine tatsächlichen Instanzdaten auswerten können.

Reihenfolgeoptimierung der Variablenzuweisung

Ein bei der effizienten Realisierung von Graphtransformationssystemen häufig eingesetzter Ansatz besteht darin, die Reihenfolge sukzessiver Variablenzuweisungen möglichst günstig zu wählen. Im einfachsten Fall werden Schemainformationen des Laufzeitgraphen genutzt, um eindeutige Beziehungen zwischen den zu suchenden Elemente bevorzugt auszuwerten. Eine solche Beziehung ist bspw. durch gerichtete Kantenklassen der Kardinalität „*höchstens eins*“ gegeben. Somit wird bei der Mustersuche ein Iterieren über mehrere mögliche Zielknoten durch eine Kantenklasse der Kardinalität „*beliebig viele*“ vermieden. Stattdessen wird lediglich ein *Test* vorgenommen, ob die aufgefundenen Elemente in einer Beziehung dieses Typs stehen.

Implementierungen zur Ermittlung effizient suchbarer Variablenreihenfolgen basieren häufig auf einem *Minimalen Spannbaum*, was sowohl in PROGRES als auch in Fujaba eingesetzt wird. Ziel ist es, alle Variablen mit Suchoperationen zu möglichst geringen Kosten zu belegen, wobei primär Kardinalitäten von Kantenklassen einberechnet werden. Wie eingangs beschrieben, werden Traversierungen entlang von Kanten der Kardinalität „*höchstens eins*“ als günstig angesehen, höhere Kardinalitäten entsprechend als teuer, da diese eine Iteration über erreichbare Elemente erfordern. Der Spannbaum legt ausgehend von vorgegebenen Variablenbelegungen fest durch

welche Kantentraversierung alle weiteren Variablen des Musters an Werte gebunden werden. Die Berechnung eines solchen Baums kann auf Basis der durch die Traversierung erwarteten Ergebnisgrößen als Kostenfunktion erfolgen. Da diese zudem von der Traversierungs*richtung* abhängt, bietet sich bspw. der Algorithmus nach Edmond [Edm67] zur Berechnung minimaler Spannbäume an. Nach Zuweisung aller Variablen werden alle nicht im Spannbaum enthaltenen Traversierungen als *Prüfoperation* durchgeführt, die im Gegensatz zur Suche lediglich Enthaltenseinstests zwischen Mengen erfordern.

Abbildung 5.6 zeigt zwei mögliche Varianten von Such- und Prüfoperationen an einem Beispielmuster. Im linken Beispielablauf wird ausgehend von einer Belegung des Projekts `p1` über verfügbare Aufgaben (`a1`) iteriert, von denen laut Graphschema eine bis beliebig viele vorhanden sein können. Von einer so ermittelten Aufgabe aus wird die Variable `p2` gebunden, indem erneut über eine Menge von möglichen Ergebnissen iteriert wird. Abschließend wird überprüft, ob die Belegung von `p1` mit der ermittelten Belegung von `p2` verbunden ist. Nur unter dieser Bedingung wird die aktuelle Variablenbelegung als Ergebnis festgehalten, andernfalls wird die Suche fortgesetzt. In der zweiten, rechts dargestellten Variante wird ausgehend von einer Aufgabe, das zugehörige Projekt und die damit verbundene Person ermittelt. Abschließend wird wiederum eine Verbindungsprüfung zwischen den Werten von `p2` und `a1` durchgeführt. Da hier die Suche ausschließlich über Kantenklassen erfolgt die in Traversierungsrichtung die Maximalkardinalität *eins* aufweisen, ist diese Variante günstiger als die vorangegange. Insgesamt werden zwei Iterationen über mengenwertige Verbindungen und damit verbunden potenzielles Backtracking eingespart.

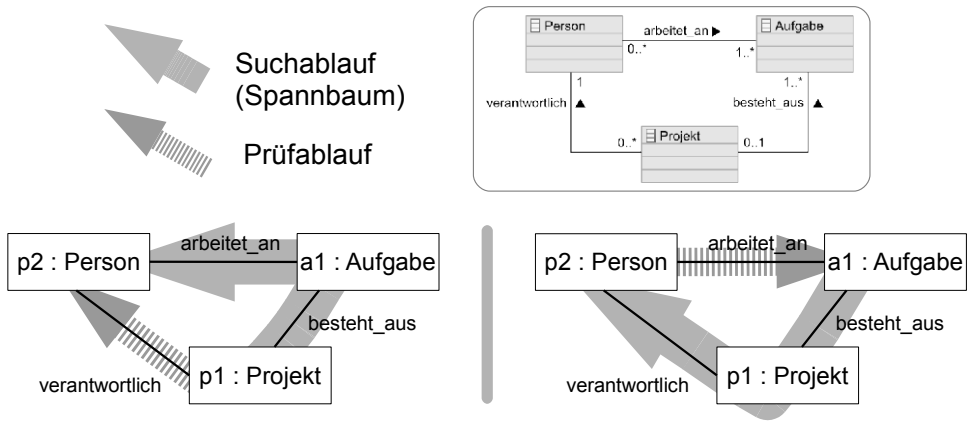


Abbildung 5.6: Alternative spannbaumbasierte Such- und Prüfoperationen

Die Generische Realisierung DRAGULAs verfolgt einen zu gängigen Implementierungen vergleichbaren Ansatz, der allerdings allgemein auf Bedingungen anstelle

von Kantentraversierungen arbeitet. Durch die Analyse eines DRAGULA-Musters im Vorfeld der Mustersuche werden interne Querbeziehungen bestimmt. Diese ergeben sich aus paarweise durch Bedingungen verbundene Variablen, die in Abbildung 5.7 in Form von Kanten eingetragen sind. Diese sind als wechselseitige gerichtete Kanten zu verstehen, die gemäß der jeweiligen Kandidatenanzahl bei Ableitung einer Belegung entstehen. So ist bspw. bei einer vorhandenen Belegung der dargestellten Kantenvariablen eine Belegung der Start- und Zielvariablen eindeutig bestimmt, weshalb das entsprechende Pfeilende die Beschriftung `1` erhält[2]. In Fällen bei denen mutmaßlich mehrere Kandidaten ermittelt werden können, sind geeignete Kostenschätzungen schwieriger abzugeben. Eine grundlegende Methode ist der Einbezug des Graphschemas, etwa bei Traversierung einer Kante deren Typ durch eine Typbedingung eingeschränkt wird. Aus diesen beiden Bedingungen und dem zur Laufzeit verwendeten Graphschema kann somit auf die Auswertungskosten geschlossen werden.

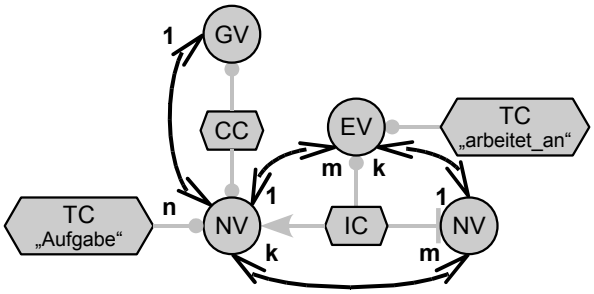


Abbildung 5.7: Kosteninformationen eines DRAGULA-Musters

Alternativ kann bei einer Inzidenzprüfung auch die durchschnittliche oder maximale Anzahl inzidenter aus- bzw. einlaufender Kanten bestimmter Knoten ermittelt werden. Im dargestellten Beispiel könnte also `m` die Anzahl einlaufender Kanten von Knoten des Typs `Aufgabe` beschreiben. Hierzu ist zusätzlich eine Einschränkung des begrenzenden Start- bzw. Zielknotens auf einen bestimmten Typ nötig. DRAGULA verwendet aktuell Laufzeitinformationen für Typbedingungen um eine statistische Kenngröße für die Menge der Instanzen ermitteln zu können.

Spezifische Optimierungsstrategien können flexibel in die Ausführungsmaschine eingebunden werden. Somit bleibt die Maschinerie leicht erweiterbar, auch wenn Spracherweiterungen zunächst mangels Optimierungshinweisen evtl. nicht effizient ausgewertet werden. Zudem sind Optimierungsstrategien von einzelnen Variablen- oder Bedingungsimplementierungen entkoppelt, so dass komplexere Zusammenhänge wie in obigem Fall bezüglich mehrerer Bedingungen einer Variablen ausgewertet

[2]Die so ableitbaren Variablenbelegungen sind allerdings nicht gegen weitere Bedingungen der zuzuweisenden Variablen geprüft, stellen also lediglich eine Obermenge gültiger Belegungen dar.

werden können.

Ausgeführt wird die Optimierung eines Musters allein auf statischen oder statistischen Informationen die zur Laufzeit aus dem verarbeiteten Graphschema oder der Struktur des Laufzeitgraphen bekannt sind. Zu Beginn der Mustersuche ist die Reihenfolge von Variablen fixiert und wird nicht von während der Mustersuche gewonnenen Informationen beeinflusst. Hintergrund dabei ist, dass andernfalls eine Greedy-artige Vorgehensweise realisiert werden würde, die zu effizienztechnischen Fehlentscheidungen führen könnte. So könnte ausgehend von vorhandenen Variablenbelegungen bspw. entlang Inzidenzbedingungen traversiert werden, zu denen am vorgewählten Startknoten möglichst wenige auslaufende Kanten vorhanden sind. Eine solche Vorgehensweise wird bspw. in [GHS09] eingesetzt. Allerdings sind Fälle konstruierbar, bei denen die Iteration größerer Kantenbüschel effizienter verläuft, falls hierdurch in Folgeschritten *eindeutige* Beziehungen traversiert werden können. Eine solche „vorausschauende“ Optimierung ist jedoch nur *vor* Belegung der entsprechenden Variablen, und damit nur eingeschränkt unter Nutzung von Laufzeitinformationen möglich. Auch wenn sich die Unterscheidung i.A. wenig auf reale Laufzeiten auswirkt, können im Fall des Greedy-Ansatzes durchaus Skalierungsprobleme bei großen Probleminstanzen entstehen.

Backtracking & Backjumping

Internes Backtracking beschreibt die Rücknahme vorgenommener Variablenbelegungen um verbleibende Kandidaten als Wert einer Variablen zuzuordnen. Backtracking wird insbesondere dann angewendet, falls die Variablen eines Musters nicht mit gültigen Werten belegt werden können. Dabei wird stets die letzte erfolgreiche Variablenbelegung zurückgenommen. Dies kann eine nicht optimale Entscheidung sein, wenn die zurückgenommene mit der vergeblich belegten Variablen nicht in Verbindung steht. In diesem Fall würde nach Neuzuweisung der zurückgenommenen Variablen die vorher nicht zuweisbare Variable nach wie vor nicht belegt werden können.

Da aufgrund der syntaktischen Trennung zwischen Variablen und Bedingungen mögliche Querbeziehungen zwischen Variablenwerten *explizit* vorliegen, lässt sich diese Problematik relativ einfach umgehen. Hierzu wird sogenanntes Back*jumping* [TB93] eingesetzt, um eine Variablenbelegung zu ändern, die sich auf die vorher nicht belegbare Variable auswirkt. Ein Rücksprung erfolgt immer auf die *zuletzt* belegte Variable, die mit der nicht zuweisbaren durch eine Bedingung *verbunden* ist. Die Auswirkung dieses Ansatzes auf obiges Beispiel zeigt Abbildung 5.8. Im linken Beispiel wird reguläres Backtracking eingesetzt, wobei nach erfolgloser Belegung der Knotenvariablen zunächst die zuletzt belegte Kantenvariable zurückgenommen wird. Im Gegensatz dazu wird beim Backjumping auf der rechten Seite direkt zur Knotenvariablen

gesprungen, die mit der nicht belegbaren Variablen durch eine Isomorphiebedingung verbunden ist. Anschließend erfolgen eine Neubelegung der angesprungenen Variablen und die Fortführung der Mustersuche.

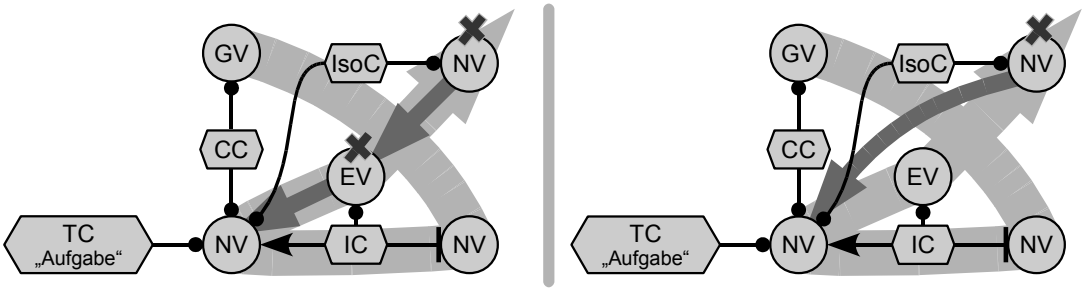


Abbildung 5.8: Sukzessive Variablenbelegung sowie Backtracking & Backjumping

Da auch nach Durchführung eines Rücksprungs die festgelegte Variablenreihenfolge beibehalten wird, müssen demnach auch „übersprungene“ Variablen neu belegt werden. Dies ist jedoch aufwandsmäßig nicht vergleichbar mit der erfolglosen Belegung dieser Variablen mit allen möglichen Kandidaten, die bei einfachem internen Backtracking durchgeführt werden würde.

Eingeschränkte Kandidatenermittlung

Die Ermittlung von Kandidatenmengen einer Variablen auf Basis ihrer inzidenten Bedingungen kann ggf. mit hohem Berechnungsaufwand verbunden sein. Hauptsächlich ist dies bei der Ermittlung sehr großer Ergebnismengen der Fall, bspw. also bei der Bestimmung aller Instanzen eines Entitätstyps durch eine Typbedingung. Auch die Ermittlung aller zu einer gegebenen Menge *unterschiedlichen* Graphelemente, was bei der Auswertung von Isomorphiebedingungen erfolgt, resultiert i.d.R. in einer sehr großen Ergebnismenge.

Die DRAGULA-Ausführungsmaschine setzt insgesamt drei Techniken ein, um die Ermittlung großer Ergebnismengen zu vermeiden:

1. Die Implementierung einer Bedingung ist nicht gezwungen eine Ergebnismenge zu liefern, und kann somit auch einen `null`-Wert zurückgeben. Im Vertrag der Programmierschnittstelle festgehalten ist jedoch, dass falls eine Bedingung eine Kandidatenmenge zu einer Variablen liefert, diese Menge *vollständig* sein muss und somit keine für die Bedingung gültigen Belegungen auslässt. Diese Möglichkeit wird für Bedingungen genutzt, die mangels Belegungen anderer Variablen keine sinnvollen Kandidaten ermitteln können. Bspw. sind Kandidaten der Verbindungsvariablen einer Inzidenzbedingung nur dann bestimmbar, wenn zumindest ein begrenzendes Element, also Quelle oder Ziel, bekannt ist.

2. Die Reihenfolge, in der Bedingungen angefragt werden, wird nach aufsteigender Anzahl erwarteter Kandidaten sortiert. Nur für die jeweils günstigste Kandidatenquelle werden diese direkt angefragt, darüber hinaus erfolgt ein sukzessiver Gültigkeitstest aller vorab ermittelten Werte. Auch die Reihenfolge dieser Gültigkeitsprüfung wird kostenbasiert ermittelt. Bezogen auf Abbildung 5.5 wird die Kandidatenmenge aus der Inzidenzbedingung ermittelt, und anschließend gegen die weiteren Bedingungen geprüft.

3. Zusätzlich definiert die DRAGULA-API eine Funktion, *Gegenkandidaten* zu bestimmen, die eine Bedingung definitiv verletzen würden. Im Fall der Isomorphiebedingung werden hierfür die Werte der bereits belegten Variablen verwendet, die sich effizient bestimmen lassen. Wiederrum besteht für Implementierungen die Möglichkeit durch `null`-Werte die Ermittlung einer Gegenkandidatenmenge abzulehnen. Vertraglich festgehalten ist auch hier die Vollständigkeit der Auskunft, d.h. dass alle nicht erlaubten Variablenbelegungen enthalten sind.

Obige Kostenfunktionen basieren auf Statistikdaten des Graphspeichers und sind somit nicht auf statische Informationen wie dem Graphschema der Laufzeitdaten angewiesen. So wird bspw. bei der Bestimmung von Kandidaten durch eine Typbedingung die Anzahl der zuletzt ermittelten Instanzen gespeichert und als Grundlage für die Kostenfunktion der Typbedingung verwendet. Diese Statistikinformationen sind global, können als auch für Typbedingungen in anderen Mustern verwendet werden. Bei wesentlichen Änderungen im Graphspeicher, bspw. der Entfernung großer Graphbereiche, ist das manuelle Anstoßen einer Reorganisation sinnvoll.

Zwischenfazit: Die Generische Realisierung der DRAGULA-Ausführungsmaschine stellt quasi eine Referenzimplementierung der Kernsprache für Graphtransformationen bereit. Dabei wird nicht nur die notwendige angebotene Funktionalität zu diesem Zweck angeboten, die neben der reinen Anfrage- und Transformationsausführung auch Persistenzmechanismen insbes. zur inkrementellen Mustersuche anbietet. Darüber hinaus wendet die Implementierung verschiedene Ansätze zur Effizienzsteigerung an, die erprobte Vorgehensweisen bspw. in Form von Suchplänen auf die Spezifika von DRAGULA übertragen.

5.4 Spezifische Realisierung für relationale Datenbanken

Im vorangegangenen Abschnitt ist eine Implementierung der Kernsprache DRAGULA vorgestellt worden, die ausschließlich auf Basis der API des Graphspeichers DRAGOS arbeitet. Das eingangs in Abbildung 5.2 dargestellte Problem bei Verwendung relationaler Datenbanken als Hintergrundspeicher, sind die zahlreichen resultierenden Zugriffe auf den Graphspeicher. Ziel der hier behandelten Realisierung ist es daher, SQL-Anfragen direkt aus einer DRAGULA-Spezifikation abzuleiten und somit in größeren Fragmenten an den Hintergrundspeicher zu senden. Auf diese Weise kann eine komplexe Anfrage innerhalb des Datenbanksystems optimiert werden, was bei vergleichsweise einfachen Anfragen im Fall der Generischen Realisierung kaum möglich ist. Stattdessen können mehrere Variablen eines DRAGULA-Musters belegt werden ohne das zwischenzeitlich eine Prozesskommunikation zwischen Ausführungsmaschine und Datenbanksystem notwendig wäre. In oben behandeltem Fall würde eine solche Kommunikation für jeden Berechnungsschritt – darunter Ermittlung von Kandidaten und Gültigkeitsprüfung – anfallen.

5.4.1 Abbildung anfragender Sprachkonstrukte

Eine Abbildung von DRAGULA-Spezifikationen auf den verwendeten SQL-basierten Hintergrundspeicher ist spezifisch für die jeweilige Tabellenstruktur vorzunehmen. Die DRAGOS-Realisierung für das Datenbanksystem PostgreSQL[Pos10], auf dessen Basis die weitere Erläuterung basiert, verwendet hierzu *Tabellenvererbung* um die Gemeinsamkeiten verschiedener Graphentitätstypen – insbes. Identifizierung und Typisierung – in einer zentralen Tabelle beschreiben zu können.

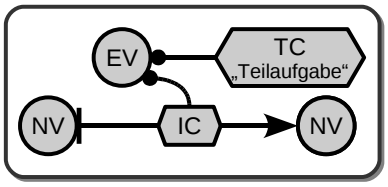
Abbildung 5.2 auf Seite 184 zeigt das für allgemeine Graphelemente verwendete Tabellenschema . Die Spalte `ID` definiert den eindeutigen Bezeichner der Graphentität, `Type` ihre Sorte (Knoten, Kante, ...), und `Class` die instanziierte Graphentitätsklasse als Fremdschlüssel. Die Tabelle `Edge` ergänzt dies um Fremdschlüssel für Quell- und Zielelemente. Durch die angebotene Tabellenvererbung stellt `Edge` implizit auch die Spalten der übergeordneten Tabelle bereit – eine eigene Spalte `ID` ist in PostgreSQL somit nicht nötig. Andere SQL-basierte Datenbanksysteme bieten das Nichtstandardkonstrukt der Tabellenvererbung nicht an. So muss die DRAGOS-Realisierung für Apache Derby [Apa10] auf Fremdschlüssel zur Vererbungsmodellierung zurückgreifen [EN94, Kap. 21], wodurch bei der Anfrage von Kanten ein zusätzlicher Join über die verwendeten Tabellen notwendig wird.

Für nicht-hierarchische DRAGULA Muster setzt sich eine korrespondierende SQL-

Anfrage aus insgesamt drei Bestandteilen zusammen. Hierzu zählen die Entitätsbezeichner der ermittelten Auffindungsstelle (`#entities`), die Tabellen aus denen diese entnommen werden (`#entity_tables`) sowie Bedingungen, die eine Belegung gemäß der DRAGULA-Anfrage erfüllen muss (`#conditions`).

```
select #entities        // Zu ermittelnde Graphelemente
  from #entity_tables   // Tabellen, die diese Elemente enthalten
  where #conditions     // Zu erfuellende Bedingungen
;
```

Folgendes Beispiel in Abbildung 5.9 erläutert die Ableitung obiger SQL-Fragmente aus der dargestellten Anfrage. Für jede Variable wird hierbei zunächst ein eindeutiger Bezeichner vergeben, wobei die Variablensorte[3] als Prefix verwendet wird. Zu jedem dieser Bezeichner wird ein entsprechender Tabellenbezeichner abhängig von der Variablensorte erstellt. So werden Knoten direkt der Tabelle `GraphEntity` entnommen, wohingegen Kanten in der davon abgeleiteten Tabelle `Edge` enthalten sind. Bei der Anfrageauswertung werden die angegebenen Tabellenbezeichner in einer *Join*-Operation miteinander verknüpft. Als Bedingungen sind zunächst für jede Variable der allgemeinen Tabelle `GraphEntity` Sortenüberprüfungen einzufügen, hier also für `n1` und `n2`. Bezüglich der Typbedingung wird geprüft, ob der Typ der ermittelten Kante in einer statisch bestimmten Menge – hier `{ 4 }` – enthalten ist. Diese Menge ergibt sich aus dem referenzierten Typ `Teilaufgabe` und dessen *Obertypen* um die Verträglichkeit der Zuweisung mit einer Typhierarchie zu prüfen. Zur Umsetzung der Inzidenzbedingung wird ausgehend von der Verbindervariablen `e1` geprüft, ob diese per Fremdschlüssel auf die Belegungen der beiden Knotenvariablen verweist.



Entities
n1
n2
e1

Entity_tables
GraphEnity as n1
GraphEntity as n2
Edge as e1

Conditions
n1.sort = Node
n2.sort = Node
e1.type in { 4 }
e1.source = n1.id
e1.target = n2.id

Abbildung 5.9: DRAGULA-Anfrage und zugehörige SQL-Fragmente

[3] D.h. Knotenvariable, Kantenvariable, etc.

Neben dieser beispielhaften Umsetzung anfragender Sprachkonstrukte existieren einige weitere, analog zu realisierende Variablen- und Bedingungsarten. So prüft die Enthaltenseinsbedingung die `parent`-Eigenschaft des enthaltenen Graphelements. Für Isomorphiebedingungen mit n angeschlossene Variablen werden als Test auf paarweise Verschiedenheit $\binom{n}{2}$ Bedingungen der Form `v`$_i$`.id != v`$_{i+j}$`.id` mit $i, j \in \{1..n-1\}, i+j \leq n$ generiert.

Attributbedingungen können dagegen nicht sinnvoll in SQL überprüft werden, da der DRAGOS-Graphspeicher Attributwerte stets als binäres Objekt in serialisierter Form ablegt. Von daher würde die Auswertung von Attributbedingungen zunächst die Deserialisierung der abgelegten Java-Objekte erfordern, was innerhalb des Datenbanksystems nicht sinnvoll realisiert werden kann. Attributbedingungen werden von daher nach wie vor durch die Generische Realisierung ausgewertet.

5.4.2 Behandlung weiterer Sprachkonstrukte

Bislang unterstützt die SQL-basierte Realisierung lediglich DRAGULA-*Anfragen* auf Basis einer SQL Datenbank, jedoch keine weiteren Sprachaspekte wie Transformation oder Ablaufsteuerung. Dennoch kann diese Realisierung für beliebige DRAGULA-Spezifikationen verwendet werden. Zur Ausführung nicht unterstützter Sprachkonstrukte wird auf die oben vorgestellte Generische Realisierung zurückgegriffen, wobei die Spezifische Realisierung direkte SQL-Zugriffe geeignet verkapselt.

Der Grund für die Einschränkung auf anfragende Sprachanteile besteht einerseits in dem ansonsten geringen Effizienzgewinn gegenüber der Generischen Realisierung, andererseits in der durch DRAGOS-Erweiterungen erschwerten Abbildung bspw. von Änderungsoperationen auf den Hintergrundspeicher. So sieht etwa die Versionierungserweiterung vor, alle Graphänderungen intern zu protokollieren um darauf basierend Undo-/Redo-Funktionalität bereitstellen zu können. Bei der Abbildung von DRAGULA-Operatoren auf SQL-Konstrukte müsste daher ggf. auch diese Protokollierung analog zur Versionierungserweiterung erfolgen.

Eine Ablaufsteuerung von DRAGULA-Regeln lässt sich in Standard-SQL zudem nicht realisieren, da dieses keine Kontrollflussstrukturen bereitstellt. Lediglich durch Nutzung herstellerspezifischer Skriptsprachen wie bspw. Pg/PLSQL der PostgreSQL-Datenbank könnten einzelne Operationen zusammengeschaltet werden. Der Vorteil einer solchen skriptbasierten Lösung bestünde in ihrer serverseitigen Ausführung, die von der clientseitigen Anwendung entkoppelt werden kann. Da eine serverseitige Ablaufsteuerung allerdings nur bei vollständiger Abdeckung sonstiger DRAGULA-Konstrukte einschließlich der Transformation von Graphstrukturen möglich ist, wird dieser Ansatz im Kontext der vorliegenden Arbeit nicht weiter verfolgt.

5.4.3 Evaluation

Da Performanzgründe die hauptsächliche Motivation darstellen, Spezifische Realisierungen der DRAGULA-Ausführungsmaschine zu nutzen, wird an dieser Stelle ein kurzer Vergleich der bislang messbaren Ergebnisse durchgeführt. In dem verwendeten Szenario wird nach *Zyklen* von Knoten gesucht, die durch Kanten miteinander verbunden sind. Der Laufzeitgraph ist ein n-vollständiger Graph ohne Typinformationen, alle Knoten und Kanten sind also identisch typisiert. In dieser synthetischen Problemstellung ist häufiges internes Backtracking zu erwarten, da die Anzahl vorhandener Zyklen im Vergleich zu navigierbaren Wegen in der Graphstruktur klein ist. Der Test betont also bewusst die Reihenfolgeoptimierung der Ausführungsmaschine.

Die Ergebnisse des durchgeführten Vergleichs zeigt Abbildung 5.10 in logarithmischer Skalierung im Bereich von Mikrosekunden pro Anfrage. Dargestellt sind Anfragen der Zyklenlänge drei und zehn auf verschieden großen Laufzeitgraphen. Zum Einsatz kommt ein PostgreSQL Datenbanksystem der Version 8.3 in Verbindung mit dem DRAGOS Graphspeicher. Zu Ansteuerung der API wird optimierter Quellcode eingesetzt, der mittels PROGRES aus einer entsprechenden Spezifikation generiert worden ist – auf diese Weise sollen mögliche Ineffizienzen der Generischen Realisierung ausgeblendet werden. Zum Vergleich ist im Übrigen noch die frühere Version des Graphspeichers GRAS angegeben, die durch PROGRES-generierten C-Code angesteuert wird. Alle Messungen wurden wiederholt auf einem einzelnen PC durchgeführt, dargestellt sind Mittelwerte.

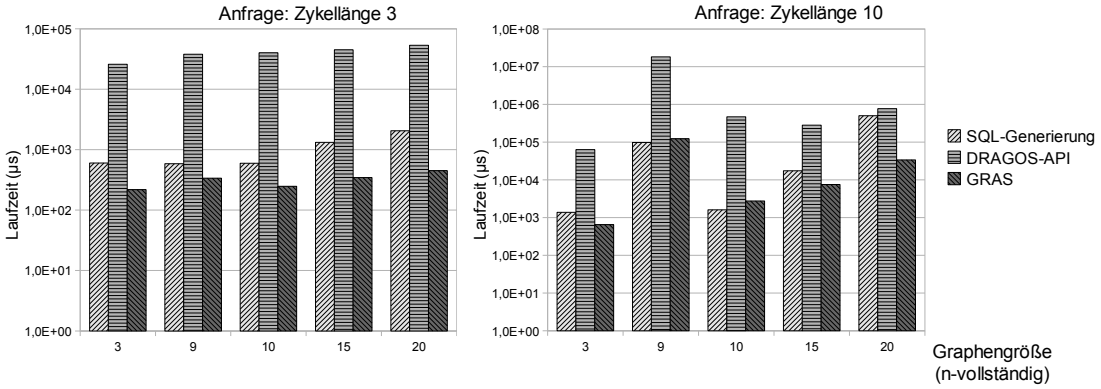


Abbildung 5.10: Laufzeitvergleich von SQL- und DRAGOS-basierten Anfragen

Für *kleine* Anfragen der Größe drei zeigt sich ein erheblicher Vorteil des direkten Zugriffs über SQL-Anfragen gegenüber einzelnen DRAGOS-Aufrufen, der sich in einem zweistelligen Faktor bewegt. Dennoch ist die hochgradig optimierte GRAS-Datenbank mit Ansteuerung durch C-Code deutlich effizienter als die DRAGOS-Lösung. Alle drei Systeme zeigen wiederum vergleichbares Verhalten bei wachsender Größe des

Laufzeitgraphen, was auch zu einer Verlängerung der Anfragezeit führt. Interessant ist hieran, dass sich bereits für kleine Anfragen ein erheblicher Performanzvorteil der SQL-basierten Lösung gegenüber der reinen DRAGOS-Lösung ergibt.

Größere Anfragen, hier der Zykluslänge zehn, verhalten sich bereits deutlich anders als die kleineren Varianten. Zwar weisen alle Systeme bei Laufzeitgraphen der Größe neun einen Spitzenwert auf, da im Unterschied zu größeren Graphen keine Auffindungsstelle gefunden werden kann und daher die vollständige Datenbank durchsucht werden muss. Bei größeren Graphen zeigt die Realisierungsvariante unter Nutzung der DRAGOS-API vergleichsweise stabiles Verhalten. Abhängig von der Laufzeitgraphgröße steigt die Bearbeitungsdauer der generierten SQL-Anfrage jedoch erheblich an. Der Grund hierfür scheint in einer ineffizienten Behandlung der *Join*-Operation der angefragten Graphelemente zu liegen. Untersuchungen der intern durchgeführten Anfrageoptimierung haben zumindest für das PostgreSQL-System gezeigt, dass die verwendeten Zyklusanfragen nicht hinreichend bezüglich der navigierten Verbindungsstrukturen optimiert werden. Stattdessen werden teilweise Joins über kleine Tabellen – etwa zur Ermittlung zweier Knoten – bevorzugt, deren Ergebnisse aber nicht durch entsprechende Bedingungen eingeschränkt werden können.

Ein weiteres Problem der SQL-basierten Anfrageauswertung besteht darin, dass serverseitige Optimierungen häufig erst nach Erreichen einer gewissen Aufrufanzahl einer Anfrage erfolgen. Je nach Konfiguration des Systems werden die ersten Anfrageaufrufe schlichtweg in keiner Weise optimiert, was schlimmstenfalls zu naiven Joins aller angefragten Tabellen, und damit zu sehr hohem Speicheraufwand auch für die hier behandelten kleinen Probleminstanzen führt. Aus diesem Grund sollten Anfrageoptimierungen immer explizit durchgeführt werden, auch wenn dies bei selten verwendeten Anfragen mit hohen Laufzeitkosten versehen ist.

Konsequenz: Als Konsequenz aus der durchgeführten Untersuchung sollte vorgesehen werden die Join-Reihenfolge der erzeugten Anfrage explizit zu steuern. Dies könnte anhand einer spezifischen Optimierungsstrategie erfolgen, die pro Datenbanksystem gesondert erfolgen muss. Ein solcher Ansatz für Graphtransformationen ist bereits in [VFV06b, Var08] verfolgt worden, allerdings sind lediglich vorläufige Ergebnisse der spezifischen Anfrageoptimierung veröffentlicht worden. Möglich wäre auch, eine Zerlegung von Graphanfragen in einzelne SQL-Fragmente sowie die Kombination dieser Fragmente mittels geschachtelter SQL-Anfragen vorzunehmen. Hierdurch würde die interne Anfrageoptimierung implizit durch die Strukturierung der geschachtelten Anfragen beeinflusst, und eine datenbankspezifische Lösung könnte entfallen. Zwar haben initiale Versuche bereits eine Effizienzsteigerung bei der Zerlegung von Zyklen der Länge zehn in zwei „Halbzyklen“ gezeigt, eine eingehendere

Untersuchung ist jedoch noch nicht erfolgt. Die Anwendung praxisorientierter Benchmarks wie [VSV05] würde hierzu zusätzliche Erkenntnisgewinne bringen.

5.5 Erweiterbare Systemarchitektur

Das vorangegange Kapitel stellt die *Erweiterbarkeit* der Kernsprache DRAGULA als wesentlichen Aspekt einer technischen Grundlage für ausführbare Spezifikationssprachen vor, da nur auf diese Weise auf anwendungsspezifische Anforderungen angemessen eingegangen werden kann. Neben der dahingehend angepassten Sprachgestaltung ist auch eine entsprechende Erweiterbarkeit der Ausführungsmaschine vorzusehen um zusätzliche Sprachkonstrukte leicht realisieren und in die bestehende Maschinerie integrieren zu können.

5.5.1 Problemstellung

Die in Abschnitt 5.2 beschriebene Bereitstellung einer Sprachimplementierung ist in einigen Punkten mit einem hohem Implementierungsaufwand verbunden. Im Einzelnen sind für jede Spracherweiterung folgende Implementierungsfragmente bereitzustellen:

1. Eine *syntaktische* Abbildung der Sprachkonstrukte auf Elemente eines DRAGOS-Graphschemas. Dieses Fragment umfasst Programmcode der durch Nutzung der DRAGOS-API ein entsprechendes Schema anlegt.

2. Ein implementierungsseitiges *Klassenmodell* zur programmatischen Navigation der abstrakten Syntax, bspw. um Syntaxprüfungen realisieren zu können. Hierzu zählt in Abbildung 5.4 die Methode `getConstraints` der Implementierungsklasse `Variable`.

3. Ein *Aktivator*, der die bereitgestellte Implementierung im System bekannt macht und den DRAGOS-Schemaelementen zuweist.

4. Fachliche Logik zur Realisierung der Spracherweiterung, entweder als Ergänzung zur Generischen und optional weiterer Realisierungen, oder in Form einer manuell implementieren Rückführung auf vorhandene Sprachkonstrukte.

Insgesamt betrachtet ist insbesondere der zweite Aspekt mit erheblichem, jedoch rein schematischem, Implementierungsaufwand verbunden. Im Kontext dieser Arbeit wird daher eine gesonderte *Werkzeugunterstützung* bereitgestellt um die Realisierung

von Spracherweiterungen effektiv zu unterstützen. Dabei wird das Entwicklungsmodell der *Architekturzentrierten Modellgetriebenen Entwicklung* (vgl. Abschnitt 1.1) angewandt. Demzufolge werden aus einer kompakten Spezifikation Quellcodefragmente für die ersten drei Punkte obiger Aufzählung generiert um die weitere manuelle Implementierung – insbes. im Hinblick auf die fachliche Logik – zu vereinfachen. Falls neu definierte Sprachkonstrukte durch Rückführung realisiert werden sollen, kann auch der fachliche Anteil modellgetrieben beschrieben werden.

5.5.2 Lösungsansatz

Der verfolgte Lösungsansatz zur syntaktischen Beschreibung von Spracherweiterungen sieht eine Modularisierung einzelner Sprachbestandteile vor. Auf diese Weise können Spracherweiterungen sowohl Elemente der Kernsprache referenzieren, als auch Sprachkonstrukte bereits bestehender Spracherweiterungen. Abbildung 5.11 gibt hierzu eine graphische Übersicht und zeigt zudem die jeweils generierten Fragmente.

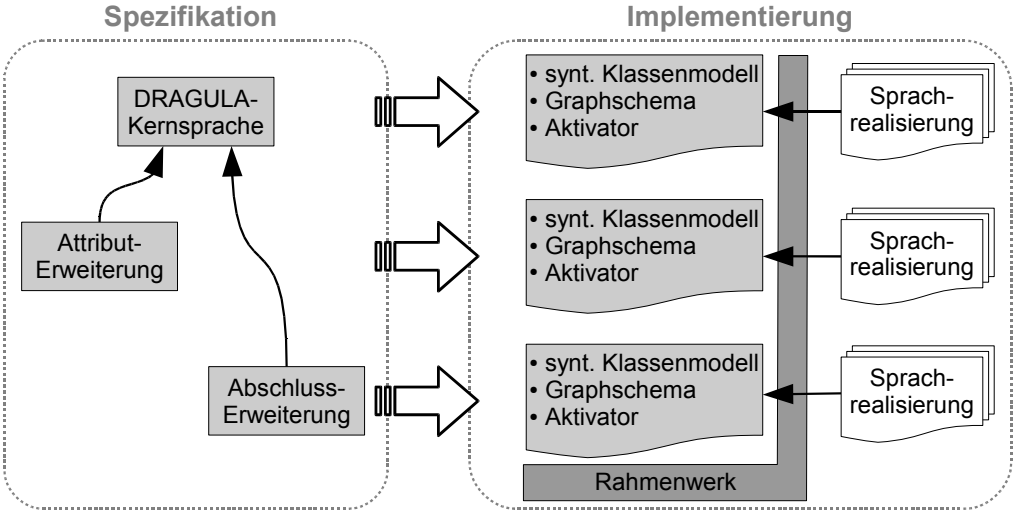


Abbildung 5.11: Modularisierte Spezifikation von DRAGULA-Sprachfragmenten

Aus den physisch getrennt vorliegenden Spezifikationsdokumenten werden auf Anforderung des Entwicklers Implementierungsfragmente generiert. Diese umfassen die oben erläuterten stereotypen Fragmente, wie etwa ein sprachspezifisches Klassenmodell sowie ein zugehöriges Graphschema für den Graphspeicher DRAGOS und entsprechende Aktivierungsfunktionalität. Diese generierten Anteile werden in ein Grundgerüst eingebettet das Basisfunktionalität, wie den Zugriff auf den Graphspeicher bereitstellt.

Da auch der Sprachkern mit einer zur leichteren Erweiterbarkeit der Implementierung bereitgestellten Lösung spezifiziert wird, kann DRAGULA insgesamt als Erweiterung einer äußerlich funktionslosen Basisbibliothek aufgefasst werden. Ferner bleiben alle Implementierungsanteile flexibel austauschbar, so dass auch die im vorherigen Kapitel vorgestellten Sprachkonzepte nicht unveränderlich in die bereitgestellte Realisierung eingebaut sind. Vielmehr definieren Referenzierungen zwischen syntaktischen Spezifikationsdokumenten eine Familie zusammengehöriger Sprachmodule, die unter Nutzung der Basismaschine realisiert ist.

5.5.3 Syntaxspezifikation

Die folgende Beschreibung stellt die entwickelte Spezifikationssprache anhand einer EBNF-Spezifikation [EBN96] vor. Im Unterschied zu der in Abschnitt 3.3 gegebenen *metamodellbasierten* Definition kann durch eine EBNF-Grammatik leichter der Bezug zur konkreten textuellen Syntax hergestellt werden. Da die Details der weiteren Verarbeitung, zu der wiederum das Abstrakte Syntaxmodell verwendet wird, aufgrund der stark technischen Natur von geringem Interesse sind, wird hier der Fokus auf ein leichteres Verständnis der Sprachdefinition gelegt.

Um auch im Rahmen einer kontextfreien Grammatik auf Bezeichnerbindungen hinweisen zu können, wird im Folgenden die Notation des IPSEN-Buchs [Nag96] eingesetzt. Darin wird zwischen der Art des referenzierten Elements und einem deklarierenden bzw. angewendeten Auftreten unterschieden. Auf diese Weise wird aus der Grammatik ersichtlich, welche Deklarationen durch angewendete Bezeichner referenziert werden dürfen. So stellt im unten stehenden Fall das Nichtterminal ExtApplId einen auftretenden Erweiterungsnamen dar, der durch das Nichtterminal ExtDeclId an anderer Stelle definiert worden ist.

Spezifikationsrahmen

Das nachstehende Grammatikfragment zeigt zunächst die Gesamtstruktur eines Spezifikationsdokuments, die jeweils eine Spracherweiterung[4] definiert. Eine Erweiterungsspezifikation umfasst jeweils eine Menge von Entitäten als Spezifikationsrumpf. Hierbei ist die Referenzierung anderer Erweiterungsmodule möglich, wozu deren Namen und eine Adressierungsmöglichkeit für das referenzierte Dokument benötigt werden.

```
Document = "extension" ExtDeclId { "uses" ExtApplId "from" Url }
           "{" { Entity } "}" (* Spezifikationsrumpf *)
```

[4] im weiteren Sinne einschließlich der Kernsprache

```
        { Implementation };
```

Im Folgenden wird als Beispiel die *Abschlussbedingung* auf Abschnitt 4.6.1 syntaktisch spezifiziert. Wie unten angegeben referenziert diese Erweiterung die Kernsprache, da die dort definierten Konzepte der Variablen und Bedingungen verwendet werden.

```
extension dragula.closure
  uses dragula.core from "platform:/resource/dragula.core/core.des"
{
  [...] // Syntaktische Spezifikation
}
```

Spezifikation von Entitäten

Das folgende EBNF-Fragment zeigt die Definition einer Entität, womit hauptsächlich *Sprachkonstrukte* einer DRGAULA-Erweiterung beschrieben werden. Wie die Definition festlegt, besteht eine Entitätsspezifikation aus einem entsprechenden Schlüsselwort sowie einer im Bereich des Spezifikationselements eindeutigen Bezeichnung. Entitätsspezifikationen können als *abstrakt*, d.h. nicht-instanziierbar deklariert werden und dienen dann nur zur Bildung einer Typhierarchie. Ferner ist eine Vererbung zwischen Entitätsspezifikationen möglich, die im Sinne der Objektorientierung behandelt wird. Ein Entitätsrumpf, sofern angegeben, besteht aus einer Folge von Referenzen, Attributen sowie Meta-Attributen die im Folgenden erläutert werden. Zur Unterstützung einer *semantischen* Spezifikation ist die Angabe von Reduktionsregeln auf in referenzierten Sprachmodulen definierten Konstrukten möglich.

```
Entity = [ "abstract" ] "entity" DeclEntityId
         [ "extends" ApplEntityId ]
         ( "{" { BodyElement } "}" { EntityReduction }
           | [ EntityReduction ] ";" );
BodyElement = Link | Attribute | Meta
         (* Referenzen & Attributierung, s.u. *);
```

Da die Abschlusserweiterung lediglich ein einziges Sprachkonstrukt, die Abschlussbedingung, definiert, kann die syntaktische Spezifikation entsprechend einfach gehalten werden.

```
entity ClosureConstraint extends Constraint { [...] }
```

Aus dem dargestellten Spezifikationsfragment können bereits verschiedene Codefragmente erzeugt werden. Zum einen handelt es sich um Klassen des DRAGULA-Syntaxmodells, die gemäß der Realisierungsarchitektur aus Abbildung 5.4 erzeugt

werden. Neben einer Schnittstelle ClosureConstraint wird also auch ein Klassenmodell zur syntaktischen Analyse einer DRAGULA-Spezifikation erzeugt, das hier aus der Klasse BaseClosureConstraint besteht.

Zum anderen wird Quellcode erzeugt, durch den ein DRAGOS-Graphschema analog zur Klassenhierarchie aufgebaut wird. Der nachfolgende Quelltext, der aufgrund der komplexen DRAGOS-API zugegebenermaßen unübersichtlich wird, muss somit nicht manuell durch Entwickler erstellt werden. Festgelegt wird in diesem Fragment bislang lediglich, dass eine Knotenklasse ClosureConstraint erzeugt wird, die von der Klasse Constraint aus dem Sprachmodul dragula.core erbt.

```
public class ClosureSchemaModel extends AbstractSchemaModel {
  /* verschiedene Zeichenkettenkonstanten */
  public static final String CLOSURE_CONSTRAINT = [...];

  /** Erweitert das Graphschema um das Modul dragula.closure. */
  public void initSchema(Schema s  ) {
    NodeClass cc = s.createNodeClass(CLOSURE_CONSTRAINT);
    NodeClass c = s.getNodeClassByName(CoreSchemaModel.CONSTRAINT)
    cc.addSuperClass(c);
    [...]
}}
```

Spezifikation von Referenzen

Referenzen erlauben die Definition von gerichteten Beziehungen zwischen Entitäten. Eine Referenz kann einlaufend oder ausgehend sein, was durch die Richtung des Pfeils spezifiziert wird. Primär betrifft dies die Abbildung der Spezifikation auf ein korrespondierendes DRAGOS-Graphschema. Allein für ausgehende Referenzen kann mittels der Markierung $*->$ festgelegt werden, dass es sich um eine Kompositionsbeziehung handelt, was insbesondere ein kaskadiertes Löschen bewirkt. Um zwischen der referenzierenden und der referenzierten Entität bidirektional navigieren zu können, ist die Möglichkeit vorgesehen durch das Schlüsselwort opposite auf *gegenüberliegende* Referenzen zu verweisen. Dieses Konstrukt bewirkt referenzielle Integrität zwischen den jeweiligen Entitäten. Ferner besteht die gesondert behandelte Möglichkeit *abgeleitete* Referenzen zu spezifizieren.

```
Link = DeclLinkId ( [ "*" ] "->" | "<-" ) ApplEntityId
         [ Cardinality ]
         [ LinkSubsets (* Abgeleitete Referenzen, s.u. *) ]
         [ "opposite" ApplLinkId ]
         ( "{" { Attribute } "}" | ";" );
```

```
Cardinality = "[0:1]" | "[1:1]" | "[0:n]" | "[1:n]";
```

Einfache Referenzen eines ClosureConstraint umfassen bspw. das referenzierte Muster von dem genau eins anzugeben ist (referenced). Ferner können in diesem Muster *eine oder mehrere* Variablen als Start- bzw. Zielfragmente identifiziert werden, was durch die Referenzen fragSource und fragTarget angegeben wird[5].

```
entity ClosureConstraint [...] {
  referenced -> Pattern [1:1];
  fragSource -> Variable [1:n];
  fragTarget -> Variable [1:n];
  [...]
}
```

Zu dieser Spezifikation werden der Modellschnittstelle ClosureConstraint entsprechende Zugriffsmethoden hinzugefügt, die je nach Kardinalität den Typ der referenzierten Entität oder einen entsprechend parametrisierten Listentyp als Rückgabe- bzw. Parametertyp verwenden. So werden im dargestellten Beispiel u.a. die Methoden Pattern getReferenced() sowie List<Variable> getFragSource() generiert. Die automatisch generierte Basisimplementierung implementiert diese Methoden, indem der Syntaxgraph traversiert und gefiltert wird. Somit kann ein Verwender von einem ClosureConstraint-Objekt direkt auf alle referenzierten Variablen zugreifen ohne zunächst adjazente Elemente des Knotens im Graphspeicher zu ermitteln und zu diesem entsprechenden Variable-Objekt zu bestimmen. Zusätzlich zur Schreibersparnis bewirkt dies eine höhere Typsicherheit und damit verbunden eine geringere Fehleranfälligkeit.

Das für Referenzen erzeugte Graphschema wird insbesondere durch die Enthaltenseinsmarkierung beeinflusst. Ist diese Eigenschaft nicht gesetzt, werden Kantenklassen zur Darstellung der Referenz verwendet, wie das folgende Fragment zeigt. Falls die Referenz nur in eine Richtung definiert ist, wird als Quellkardinalität die Angabe [0:∗] (OPT_SET) verwendet.

```
public void initSchema(Schema s) {
    NodeClass cc = s.createNodeClass(CLOSURE_CONSTRAINT);
    [...]
    NodeClass p = s.getNodeClassByName(CoreSchemaModel.Pattern);
    s.declareDirectedEdgeClass(CLOSURE_CONSTRAINT_REFERENCED, cc,
        Cardinality.OPT_SET, p, Cardinality.OBL_UNIQUE);
    [...]
```

[5]Die kontextsensitive Forderung, dass die referenzierten Variablen im angegeben Muster enthalten sind, lässt sich so allerdings nicht formulieren.

Handelt es sich bei der Referenz um eine Enthaltenseinsbeziehung, so wird die definierende Entität anstelle auf einen Knotenklasse auf eine *Graphklasse* abgebildet. Graphen können laut Graphschemamodell beliebige Elemente enthalten, wobei eine Löschpropagation durch den Graphspeicher vorgesehen ist. Es erfolgt somit keine Deklaration einer Kantenklasse, stattdessen wird die Enthaltenseinsrelation von Graphen genutzt.

Attributierung von Entitäten und Referenzen

Sowohl Entitäten als auch Referenzen können attributiert werden, was mittels einer Reihe vordefinierter skalarer Datentypen als auch mit beliebigen extern bereitgestellten Java-Typen erfolgen kann. Attribute von *Referenzen* können ferner als *Index*, und außerdem für die bereits erwähnten *abgeleiteten Referenzen* verwendet werden.

```
Attribute = DeclAttributeId ":" ( JavaType | ScalarType )
              [ IndexdAttribute ] ";";
```

Obiges Beispiel verwendet eine attributierte Referenz um die Formalparameter des zu traversierenden Musters anhand eines Rollenbezeichners zu identifizieren. Ferner wird ein Entitätsattribut zur Festlegung der Traversierungsart anhand eines extern festgelegten Aufzählungstyps definiert.

```
entity ClosureConstraint [...] {
  [...]
  parameter -> Variable [0:n] {
    refRole : string; // Referenzattributierung
  }
  type : dragula.ext.closure.ClosureType;
}
```

Während für Entitätsattribute reguläre Zugriffsmethoden im generierten Code hinzugefügt werden, müssen Referenzattribute komplexer verarbeitet werden. Da Referenzen als Methoden ihrer enthaltenden Entität realisiert sind, müssen auch Zugriffe auf deren Attribute dort platziert werden. Hierzu werden die Namen der Referenz und ihres Attributs konkateniert. Handelt es sich um eine mengenwertige Referenz, so weisen die Liste der referenzierten Elemente und die Liste der Attributwerte identische Längen und Wertereihenfolgen auf. Die zugehörige Java-Schnittstelle für obige Spezifikation zeigt der nachfolgende Codeabschnitt:

```
public interface ClosureConstraint [...] {
  [...]
  dragula.ext.closure.ClosureType getType();
  /* Referenzvariable */
```

```
  List<Variable> getParameter();
  /* Referenzattribut refRole von parameter */
  List<String> getParameterRefRole();
}
```

Index-Attribute

Bei der syntaktischen Analyse von DRAGULA-Spezifikation ist häufig neben einer Unterscheidung nach Referenz*typen*, auch eine Differenzierung nach *Werten* bestimmter Referenzattribute gewünscht. Durch das Schlüsselwort **index** wird ein Attribut als Index definiert, der zur wertbasierten Adressierung der referenzierten Elemente dient. Als Resultat wird eine zusätzliche Zugriffsmethode erzeugt die referenzierte DRAGULA-Elemente anhand des Indexwertes zurück liefert. Wird der Index mittels vorangestelltem **unique** als *eindeutig* bezüglich *einer Instanz* des enthaltenden Entitätstyps gekennzeichnet, wird diese Zugriffsmethode auf singuläre Ergebniswerte beschränkt. Hierdurch wird die generierte API entsprechend leichter nutzbar.

```
IndexedAttribute = [ "unique" ] "index";
```

In obigem Beispiel kann der Rollenbezeichner des Verweises auf Formalparameter als Index eingesetzt werden um die direkte Adressierung dieser Variablen zu ermöglichen. Dabei wird davon ausgegangen, dass eine Abschlussbedingung jeden Parameterbezeichner refRole nur einmalig verwendet.

```
[...]
parameter -> Variable [0:n] {
  refRole : unique index string;
}
```

Hieraus wird wiederum eine zusätzliche Methode in der API erzeugt, die Parametervariablen bezüglich der Referenzattributierung ausgibt. Würde das Schlüsselwort **unique** in obigem Beispiel ausgelassen, so würde List<Variable> als Rückgabetyp der Methode verwendet. In einem solchen Fall ist sichergestellt, dass Elemente der Gesamtliste (getParameter) und der gefilterten Liste (getParameterByRefRole) identisch sortiert sind.

```
public interface ClosureConstraint [...] {
  [...]
  Variable getParameterByRefRole(String refRole);
}
```

Abgeleitete Referenzen

Index-Attribute vereinfachen den Zugriff auf referenzierte Entitäten bei vorab *unbekannten* Attributwerten, da die möglichen Werte erst zur Laufzeit bestimmt werden müssen. Sind mögliche Werte jedoch bereits statisch, etwa durch informelle Vereinbarung bekannt, bietet eine virtuelle *abgeleitete* Referenz einen bequemeren Zugriffsweg durch gesonderte Methoden in der API. Üblicher Anwendungsfall dieses Mechanismus ist die *Filterung* von referenzierten Entitäten auf Basis der Referenzattributierung.

```
LinkSubsets = "subsets" ApplLinkId "where"
                ApplAttributeId ApplOpId Expression;
```

Im Beispiel der Abschlusserweiterung bietet es sich an, Start- und Zielvariablen, die als Aktualparameter der Prüfung verwendet werden, vereinfacht zugreifbar zu machen. Ohne die Nutzung abgeleiteter Referenzen werden Variablen einer Bedingung lediglich über die allgemein verwendete restricts -Referenz angebunden, wobei das Rollenattribut implizit zwischen den jeweils referenzierten Elementen unterscheidet. Folgende Ergänzung der Spezifikation sorgt dafür, dass stattdessen gesonderte *Zugriffsmethoden* für die Referenzen source (getSource) und target (getTarget) erzeugt werden. Diese werden jedoch nicht aus dem Graphspeicher entnommen oder im Graphschema definiert, sondern ergeben sich aus den über die Referenz restricts erreichbaren Entitäten. Wie angegeben wird eine Attributprüfung vorgenommen bei deren Erfüllung eine Entität in die Ergebnismenge der abgeleiteten Referenz aufgenommen wird. Als zu prüfender Ausdruck ist an dieser Stelle lediglich ein Vergleich des role-Attributs mit konstanten Literalen vorgesehen. Möglich sind auch relationale numerische Ausdrücke, wobei der durch das EBNF-Element Expression aufgefundene Text *direkt* in den Java-Quelltext der Basisimplementierung übertragen wird.

```
source -> Variable [1:n] subsets restricts where role = ''SOURCE'';
target -> Variable [1:n] subsets restricts where role = ''TARGET'';
```

Meta-Attribute

Meta-Attribute ermöglichen die Attributierung von *Meta-Objekten*, also dem Graphschema einer DRAGULA-Spezifikation. Auf diese Weise können Verarbeitungsinformationen an die Ausführungsmaschine übermittelt werden. Als Wertausdruck sind dabei alle gängigen Java-Ausdrücke erlaubt, die ein *serialisierbares* Ergebnis liefern.

```
Meta = "meta" DeclMetaId "=" Expression;
```

Eine typische Anwendung von Meta-Attributen zeigt folgendes Beispiel aus der DRAGULA Kernsprache. Hier wird ein Meta-Attribut _priority des Erstellungsoperators auf einen in der Implementierung festgelegten konstanten Wert gesetzt. Anhand

dieser Priorisierung nimmt die Ausführungsmaschine die in Abschnitt 4.4 beschriebene Phaseneinteilung der Operatoren vor. In der gegenwärtigen Implementierung repräsentiert die Konstante PRIORITY einen Zahlwert.

```
entity CreationOperator {
  [...]
  meta _priority = CreationOperator.PRIORITY;
  [...]
}
```

5.5.4 Verhaltensorientierte Realisierung von Spracherweiterungen

Im bisherigen Verlauf dieses Abschnitts sind ausschließlich syntaktische Aspekte von Spracherweiterungen modelliert worden. Zwar ermöglichen die verfügbaren Sprachkonstrukte hierbei eine angemessene Abstraktion von der eigentlichen Sprachrealisierung, dennoch wird für das Verhalten eines Sprachkonstrukts keine vergleichsweise mächtige Spezifikationsmöglichkeit angeboten. Der Grund hierfür liegt darin, dass die semantische Beschreibung einer Spracherweiterung einen deutlich feingranulareren Detaillierungsgrad benötigt, um bspw. den Erfülltheitstest der Abschlussbedingung algorithmisch zu beschreiben. Hierbei ist letztlich fraglich, in wie weit ein hinreichend allgemeiner und mächtiger Spezifikationsansatz nennenswert von gängigen Programmiersprachen abstrahiert. Stattdessen wird einerseits eine Möglichkeit zur *Gerüstspezifikation* angeboten mit dem die manuelle Realisierung vereinfacht wird. Andererseits kann eine automatisierte *Reduktion* ergänzter Sprachkonstrukte auf Elemente anderer Sprachmodule kompakt durch Angabe eines korrespondierenden DRAGULA-Musters angegeben werden.

Gerüstgenerierung

Die Generierung eines Implementierungsrahmens je Sprachkonstrukt wird durch entsprechende **implementation** Direktiven in der Erweiterungsspezifikation beschrieben. Einzelne Entitätsdefinitionen oder alternativ alle Definitionen einer Erweiterungsspezifikation können die Methoden zur fachlichen Realisierung durch Angabe einer Java-Schnittstelle festlegen. Für die Generische Realisierung ist eine solche Schnittstelle in GenericVariable definiert, wie in Abbildung 5.4 auf Seite 187 zu sehen ist.

```
Implementation = "implementation" ImplId
                   ( "{" { ImplSpecifier } "}" | ";" );

ImplSpecifier = ( "*" | ApplEntityId ) ":" JavaInterface ";";
```

Die Angabe einer Implementierungsklausel in obigem Beispiel bewirkt für die Entität ClosureConstraint das Anlegen einer entsprechend benannten Realisierungsklasse GenericClosureConstraint. Durch den Entwickler muss in dieser Klasse eine händische Realisierung des hinzugefügten Sprachkonstrukts erfolgen, bspw. um die Erfülltheit der neu erstellten Bedingung bezüglich einer Variablenbelegung zu prüfen. Da die hierfür vorgesehene Methode isFullfilled durch die Kernsprache innerhalb der Generischen Realisierungsklasse GenericConstraint deklariert wird, muss diese nicht gesondert in der Erweiterungsspezifikation angegeben werden. Das zweite dargestellte Beispiel bewirkt, das alle Klassen der SQL-basierten Realisierung die Schnittstelle SQLFragmentProvider implementieren.

```
extension dragula.closure {
    [...]
}
implementation generic;
```

```
extension dragula.closure {
    [...]
}
implementation sql {
    *: SQLFragmentProvider;
}
```

Zusätzlich zur Klassenstruktur der spezifizierten Entitäten wird für eine Sprachimplementierung ein Aktivator generiert, der die generierten Klassen im Laufzeitsystem registriert. Außerdem wird über den Aktivator sichergestellt, dass alle verwendeten Sprachmodule die gewünschte Sprachrealisierung unterstützen.

Reduktion auf DRAGULA

Der zweite Ansatz zur Spezifikation von Sprachkonstrukten besteht in ihrer *Reduktion* auf Elemente der Kernsprache bzw. anderer Sprachmodule. Hierdurch kann offensichtlich keine *echte* Erweiterung der Sprache beschrieben werden, da lediglich bereits bestehende Funktionalität zusammengefasst und als eigenes Sprachkonstrukt angeboten werden kann. Dennoch fördert diese funktionale Abstraktion, wie bereits in Abschnitt 4.6.1 diskutiert, eine kompaktere Anwendung DRAGULAs und reduziert zugleich Risiken bspw. in Bezug auf Fehleranfälligkeit der erstellten Spezifikationen. Reduktion entfernt wiederum die vorgenommene funktionale Abstraktion um eine Spezifikation mit der verfügbaren Sprachrealisierung ausführen zu können.

Erfahrungsgemäß ist es aus Effizienzgründen i.d.R. sinnvoll neben der Reduktion zusätzlich eine Generische Realisierung bzw. eine spezifische Variante ausgerichtet auf einen speziellen Graphspeicher bereitzustellen. Primär bietet sich die Sprachrückführung daher zur prototypischen oder testweisen Realisierung von Erweiterungsmodulen an. Insbesondere beim Testen unterschiedlicher Sprachrealisierungen kann

diese als „Test-Orakel“ Soll-Werte bspw. bei der Ermittlung von Auffindungsstellen liefern.

Reduktionsregeln beschreiben die Übertragung von erweiterten Sprachkonstrukten auf solche der Kernsprache oder anderer Spracherweiterungen. Zu jeder Reduktionsregel wird eine extern bereitgestellte DRAGULA-Spezifikation angegeben, die das einzusetzende Spezifikationsfragment beinhaltet. Innerhalb dieser Spezifikation ist ein gesondert gekennzeichnetes DRAGULA-Element (DragulaElementMarker) anzugeben, auf das reduziert wird. Zusätzlich besteht mittels AttributeRule-Ausdrücken die Möglichkeit, Attribute des zu reduzierenden Sprachkonstrukts auf markierte Elemente der DRAGULA-Spezifikation zu übertragen. Die über inzidente Verknüpfungen direkt erreichbare Entitäten können ihrerseits durch EntityRule-Ausdrücke auf Elemente der DRAGULA-Spezifikation abgebildet werden. Hierdurch wird das referenzierte Spezifikationsfragment in die Umgebung des zu ersetzenden Konstrukts eingebettet. Ferner kann eine Reduktionsregel *bedingt* bezüglich den Attributwerten einer zu reduzierenden Entität spezifiziert werden. Diese so spezifizierten *Wächter* sollten sich durch die gewählten Bedingungen gegenseitig ausschließen um Mehrdeutigkeiten zu vermeiden.

```
Reduction = "reduces to" DragulaElementMarker "from" Url
    [ "with" { AttributeRule | EntityRule } ]
    [ "when" Condition ] ";";

AttributeRule: ApplAttributeId "="
                DragulaElementMarker "." ApplDragulaAttributeId;

EntityRule: ApplLinkId "as" DragulaElementMarker;

Condition: ApplAttributeId ApplOpId Expression;
```

Das folgende Spezifikationsfragment zeigt eine bedingte Reduktionsregel, die für Abschlussbedingungen des Typs STAR – d.h. transitiv-reflexive Abschlüsse – anwendbar ist. Diese Spezifikation referenziert eine externe DRAGULA-Spezifikation, welche die nötige Funktionalität zur Realisierung des Sprachkonstrukts auf Basis der Kernsprache schablonenartig bereitstellt. Die Reduktionsregel legt konkret fest, dass Abschlussbedingungen auf ein markiertes DRAGULA-Element PatternRef_Star reduziert werden. Inzidente Elemente, die durch Verknüpfungen der Entitätsspezifikation bzw. ihrer Obertypen geben sind, werden im **with**-Block auf entsprechend markierte DRAGULA-Elemente abgebildet.

```
entity ClosureConstraint [...]
    reduces to PatternRef_Star from
      "platform:/resource/dragula.closure/red.dragula"
```

```
with source as Variable_Source
     target as Variable_Target
     referenced as Pattern_Referenced
     fragSource as Variable_FragSource
     [...]
when type = ClosureType.STAR;
```

Abbildung 5.12 zeigt das zur Reduktion einer transitiv-reflexiven Abschlussbedingung eingesetzte DRAGULA-Muster sowie die damit einhergehende Übersetzung. Die einzusetzende Musterstruktur entspricht weitestgehend der aus Abbildung 4.20 auf Seite 120, wo bereits rekursive Muster zur Modellierung des transitiven Abschlusses einer Verbindungsnavigation verwendet worden sind. Der hier behandelte Fall abstrahiert jedoch von spezifischen Traversierungsschritten, in dem das entsprechende Teilmuster ausgelagert und durch eine Musterreferenz eingebunden wird.

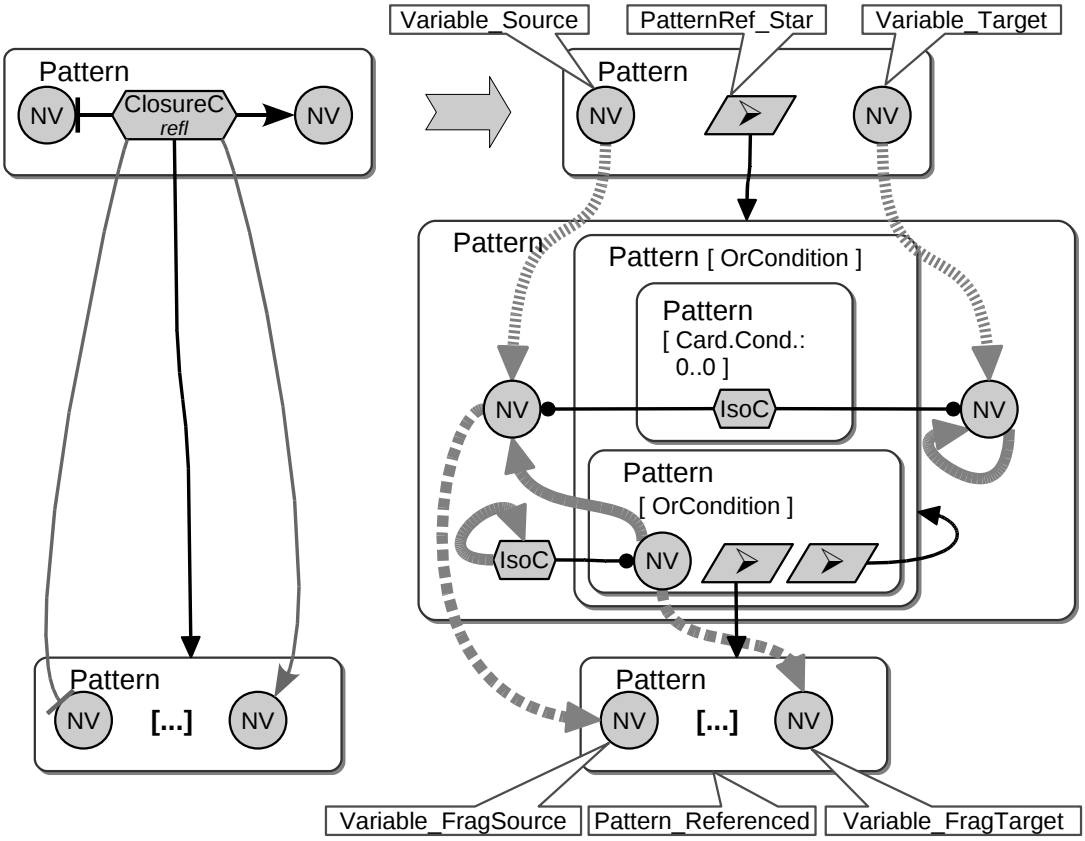


Abbildung 5.12: Reduktion der reflexiven Abschlussbedingung auf die Kernsprache

Die Reduktion der Abschlussbedingung auf die dargestellte Struktur erfolgt informell auf folgende Weise: Eine Musterreferenz *ersetzt* die zu reduzierende Bedingung

einschließlich aller inzidenten Verbindungen zum Ursprungsmuster. Zuvor als source bzw. target angeschlossene Variablen werden auf entsprechend markierte Variablen des referenzierten Musters abgebildet, die sich aus der Reduktionsspezifikation ergeben.

Das referenzierte Muster wird erfolgreich ausgewertet, falls die beiden äußeren Variablen an identische Graphelement gebunden werden, wie die negierte Isomorphiebedingung fordert. Alternativ kann eine Variablenbelegung des zweiten inneren Musters ermittelt werden, was eine Expansion der beiden dargestellten Musterreferenzen erfordert. Die linke Referenz verweist auf die eigentlich Traversierungsoperation, die – hier als Platzhalter angedeutet – beliebige Musterelemente enthalten kann. Hierbei werden die gestrichelt dargestellten Abbildungsrelationen verwendet um die jeweiligen Start- und Zielvariablen des referenzierten Musters an die Traversierungsvariablen des mittleren Musters zu binden. Diese dienen somit als Start- bzw. Endpunkt sukzessiver Traversierungsschritte. Die zweite Musterreferenz dient wie in Abbildung 4.20 dazu, weitere Traversierungsschritte rekursiv auszuführen. Zur Bindung des referenzierten Musters an die referenzierende Umgebung werden hier die durchgezogenen Abbildungsrelationen verwendet. Insbesondere wird die Ausgangsvariable eines Traversierungsschritt um eine Stufe „weitergeschaltet“, also auf die zuvor als Ziel verwendete Variable abgebildet. Die Zielvariable des Gesamtmusters sowie die Isomorphiebedingung zum Ausschluss von Zyklen, werden in allen Schrittinstanzen gemeinsam verwendet.

Das DRAGULA-Muster zur allgemeinen Beschreibung dieser Reduktion wird zusätzlich zur textuellen Entitätsspezifikation angegeben und entspricht größtenteils der in Abbildung 5.12 gezeigten Reduktionsanwendung. Lediglich das untere Muster, welches die Traversierung spezifiziert wird als minimales Fragment mit lediglich zwei Variablen spezifiziert. Das obere Muster ist dagegen nicht Teil der Spezifikation, da die Musterreferenz bei Anwendung der Reduktion die Abschlussbedingung an gleicher Stelle ersetzt. Sowohl das untere Muster als auch die beiden darin enthaltenen Variablen, werden durch obige Entitätsspezifikation an mit der Abschlussbedingung verbundene Elemente gebunden. Das in der Vorlage als Pattern_Referenced markierte Muster wird bspw. an das Muster gebunden, welches durch die Abschlussbedingung mittels der Referenz referenced angegeben ist.

Technische Vorgehensweise: Die technische Vorgehensweise zur Spezifikation von Reduktionsregeln zeigt Abbildung 5.13 im Überblick. Links dargestellt ist der oben beschriebene generative Ansatz zur Sprachrealisierung. Dies ergänzend verweist die Spezifikation der Reduktionsregeln auf ein DRAGULA-Fragment, das zwecks Referenzierbarkeit als *Datei* vorgehalten und mit der graphbasierten Darstellung über

einen Adapter abgeglichen wird. Die Verwendung einer Datei zur Referenzierung ist primär technisch bedingt um eine Integration mit der Werkzeuginfrastruktur zur Erweiterungsspezifikation zu ermöglichen. Die dateibasierte Spezifikation wird um Markierungselemente ergänzt, die als Bezeichner in der Erweiterungsspezifikation verwendet werden können. Zur Laufzeit der spezifizierten Anwendung wird mittels bereitgestellter Infrastrukturfunktionalität die Reduktion von Sprachkonstrukten bei Bedarf[6] durchgeführt. Hierdurch entsteht die angedeutete reduzierte Fassung, die anschließend mittels der verfügbaren Sprachrealisierungen ausgewertet werden kann.

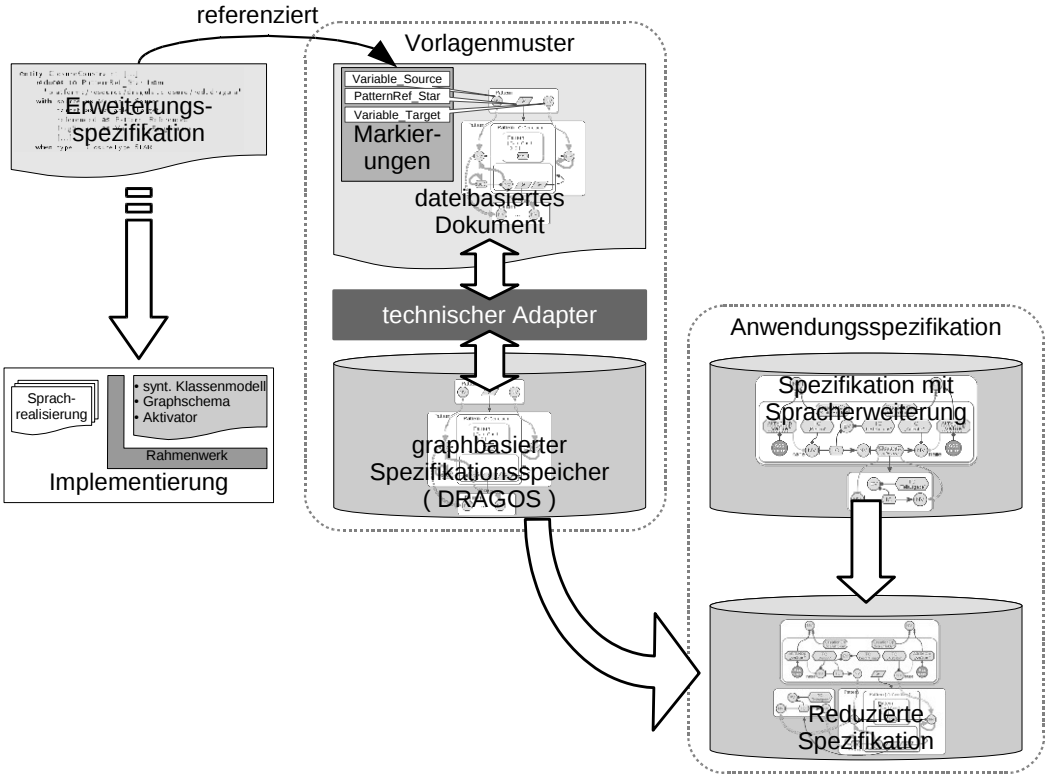


Abbildung 5.13: Spezifikation und Anwendung einer Reduktion

Erweiterte Abschlussbedingungen: Die syntaktische Definition der Abschlussbedingung aus Abbildung 4.36 auf Seite 158 erlaubt des Weiteren auch den Abschluss über *n*-äre Pfade, die bspw. auch in [LLP07] vorgestellt werden. Zusätzlich ist eine Parametrisierung in Form von Variablenbelegungen vorgesehen, die in allen Traversierungsschritten in Form von festgelegten Variablenbindungen eingebunden werden.

[6] Im Fall der Abschlussbedingung bei Auswertung des enthaltenden Musters.

Beide Ergänzungen „einfacher“ Abschlüsse sind in obiger Erläuterung nicht enthalten, aber grundsätzlich von der bereitgestellten Lösung vorgesehen. So werden bei Angabe mehrerer Start- und Zielelemente entsprechende Vorlagenfragmente je nach Kardinalität *vervielfältigt*, bzw. bei Auslassung eines optionalen Elements, wie den Formalparametern, entfernt. In einem Pfad mit zwei Traversierungsvariablen würden bspw. *zwei* als Variable_Source markierte Variablen erzeugt werden. Durch gesonderte Markierungen kann diese Duplizierung auf weitere Elemente propagiert werden, insbesondere würden im dargestellten Beispiel auch Variablen des mittleren Musters sowie die Gleichheitsprüfung zwischen Start- und Zielvariablen vervielfältigt werden müssen. Ferner ist der Zyklustest dahingehend zu erweitern, dass zwischen je zwei *Gruppen* von Variablen mindestens *eine* unterschiedliche Belegung vorliegt.

5.5.5 Einsparpotenzial

Um das Einsparpotenzial des hier vorgestellten generativen Ansatzes zur Realisierung von Spracherweiterungen quantitativ zu bewerten, werden die jeweiligen Realisierungsvolumen verglichen. Die Angaben beziehen sich dabei auf die Spracherweiterung zur Attributbehandlung aus Abschnitt 4.6.2, da diese deutlich mehr Sprachkonstrukte und damit eine komplexere Realisierung aufweist, als die hier erläuterte Abschlussbedingung.

Aus einer Spezifikation von etwa 100 Zeilen – einschließlich Kommentaren – werden insgesamt 1850 Zeilen Java-Quelltext generiert. Dies entspricht einem Vervielfältigungsfaktor von 18, der im Rahmen üblicher generativer Softwareentwicklung liegt. Zusätzlich werden etwa 750 Zeilen händisch erstellten Programmcodes zur eigentlichen Implementierung der Attributverarbeitung dienen. Wird hier die Syntaxspezifikation hinzugerechnet, so kann etwa die Hälfte des zur Realisierung einer Spracherweiterung notwendigen Programmcodes automatisch erzeugt werden. Dies berücksichtigt noch nicht die mögliche dynamische Realisierung eines Sprachkonstrukts durch Reduktion. Dieser Ansatz ist im Bereich von Attributausdrücken einerseits nicht sinnvoll einsetzbar, andererseits ist er – aufgrund der graphischen Notationsform DRAGULAs – mit schwer messbarem Spezifikationsaufwand verbunden.

Zwischenfazit: Die Möglichkeit Sprachmodule syntaktisch zu spezifizieren erlaubt eine durchgängig modellgetriebene Entwicklung der DRAGULA Ausführungsmaschine. Insbesondere erleichtert dies durch Werkzeugunterstützung in Form eines Eclipse-basierten Editors die syntaktische Definition auf hohem Abstraktionsniveau. Technische Details bzgl. der Abstützung einzelner Sprachkonstrukte auf das DRAGOS-Graphmodell sowie die Verwendung der DRAGULA Basisbibliothek werden somit größtenteils vor dem Entwickler verborgen.

5.6 Diskussion

Dieser Abschnitt diskutiert Vor- und Nachteile die aus dem Realisierungsansatz der DRAGULA Ausführungsmaschine entstehen.

5.6.1 Vorteile

Die wesentliche Neuerung der DRAGULA-Ausführungsmaschine besteht in der Adaption der jeweils verfügbaren Basisfunktionalität. Hierzu zählt bei Verwendung relationaler Datenbanken insbesondere die Nutzung komplexer SQL Anfragen, wodurch eine erhebliche Effizienzsteigerung entsteht. Je nach Kontext kann jedoch auf den Einsatz relationaler Datenbanken verzichtet werden, etwa um den damit üblicherweise verbundenen Administrationsaufwand zu umgehen. In diesem Fall wird durch die Generische Realisierung lediglich die Schnittstelle des DRAGOS Graphspeichers genutzt.

Aufgrund der modularen Struktur sowie der bereitgestellten Werkzeuge können Spracherweiterungen mit geringem Aufwand realisiert und in die bestehende Ausführungsmaschine eingebunden werden. Dies unterstützt den Anspruch DRAGULAs, als Kernsprache fallweise mittels Erweiterungen angepasst werden zu können. Sowohl die syntaktische als auch – durch Rückführung realisierte – semantische Darstellung erlauben somit eine effiziente Beschreibung der zu ergänzenden Funktionalität. Eine vergleichbare Fokussierung auf die Erweiterbarkeit der Ausführungsmaschine findet sich in den verwandten Arbeiten ebenfalls nicht.

5.6.2 Nachteile

Der Gesamtumfang der DRAGULA Sprache wird derzeit stets unter Zuhilfenahme der Generischen Realisierung ausgewertet, da die SQL-basierte Lösung lediglich effizient in Datenbanken abbildbare Teilfunktionalität abdeckt. Die Generische Reali-

sierung verfolgt dabei einen interpretativen Ansatz, der erfahrungsgemäß deutlich ineffizienter arbeitet als die Generierung von Quelltext oder maschinenausführbaren Bytecodes. Neben einer höheren Effizienz ist generierter Code jedoch immer mit einer weniger flexiblen Laufzeitverwaltung verbunden. Bspw. können kompilierte Java Klassen zur Laufzeit nur bedingt verändert werden, was mit einem Neustart des Laufzeitsystems und damit ggf. längeren Wartezeiten verbunden ist. Insbesondere im Kontext prototypischer Entwicklungen ist jedoch ein flexibleres Laufzeitsystem häufig einer effizienteren Transformationsverarbeitung vorzuziehen.

Derzeit sieht die DRAGULA Ausführungsmaschine keine *inkrementelle* Auswertung der Graphmustersuche vor, was bei bestimmten Anwendungsfällen ebenfalls zu Effizienznachteilen im Vergleich zu Systemen wie VIATRA2 führen kann. In der SQL-basierten Realisierung ließen sich materialisierte Sichten nutzen um Zwischenergebnisse einer Mustersuche zwischen Suchanfragen zu speichern und bedarfsgetrieben zu aktualisieren. Derartige Funktionalität findet sich allerdings derzeit nur in kommerziellen Datenbankmanagementsystemen. Aufgrund der dadurch entstehenden engen Bindung an hochpreisige Basissysteme wurde dieser Ansatz nicht weiter verfolgt.

Ein prinzipieller Nachteil graphbasierter Spezifikationstechniken ist die problematische Einbindung händisch erstellter Programmanteile. In DRAGULA besteht dieses Problem in eingeschränkter Form, da das Typsystem durch den DRAGOS Graphspeicher bereitgestellt wird und somit von externen Anwendungen genutzt werden kann. Um zusätzliche Programmteile in DRAGULA nutzen zu können bieten sich Bedingungen und Operatoren als Erweiterungspunkte der Kernsprache an. Durch leichtgewichtige Spracherweiterungen und der hier vorgestellten Gerüstgenerierung kann somit bestehende Bibliotheksfunktionalität leicht nutzbar gemacht werden.

5.7 Verwandte Arbeiten

In der Literatur existieren zahlreiche Realisierungen von Graphtransformationswerkzeugen und vergleichbaren Lösungen. Dieser Abschnitt setzt die hier vorgestellte Ausführungsmaschine zu diesen in Relation.

5.7.1 Realisierung von Graphtransformationen

Ausgehend von ersten praktischen Realisierungen von Graphtransformationen, die zuerst in [Nag78] vorgestellt worden sind, liegt mittlerweile das Hauptaugenmerk auf der *effizienten* Lösung des zugrundeliegenden Teilgraph-Isomorphieproblems. Da dieses aufgrund seiner nachgewiesenen NP-Vollständigkeit [Epp99] nicht im allge-

meinen Fall effizient zu lösen ist, wird daher nach entsprechend optimiertem Verhalten für übliche Anwendungsfälle gesucht.

Während Lösungsansätze auf *beschränkten* Graphstrukturen dies teils in linearer Laufzeit bewerkstelligen wie bspw. [Dö95], verwenden allgemeine Ansätze *Heuristiken* um die Teilgraphensuche für übliche Anwendungsfälle effizient zu realisieren. Neben den üblichen *suchplanbasierten* Varianten kann auch auf Lösungen der Künstlichen Intelligenz, insbesondere dem Bereich der Constraint-Satisfaction-Probleme zurückgegriffen werden. In jüngster Zeit werden zudem inkrementelle Ansätze im Bereich der Graphmustersuche verfolgt.

Suchplanbasierte Verfahren

Übliche Herangehensweisen zur Graphmustersuche ermitteln einen *Suchplan*, mit dessen Hilfe eine effiziente Auswertung von Graphmustern bezüglich der zugrundeliegenden Laufzeitgraphen angestrebt wird. Eine erste Realisierung dieses Ansatzes ist in PROGRES zu finden [Zün94, Zü96, Zün93]. Die hierbei eingeführte Kostenermittlung für unterschiedliche Navigationsoperationen ist später in das Fujaba-Projekt eingeflossen [GSR05] und findet auch im Kontext der Generischen Realisierung in dieser Arbeit Verwendung.

Sowohl in PROGRES als auch in Fujaba erfolgt eine *statische* Codegenerierung, wodurch Optimierungsentscheidungen naturgemäß nicht vom jeweils aktuellen Laufzeitgraphen abhängig gemacht werden können. Die in dieser Arbeit realisierte Ausführungsmaschine arbeitet *interpretativ* und hat daher die Möglichkeit Laufzeitdaten zur Optimierung heranzuziehen. Dies erfolgt auf Basis statistischer Informationen, nicht jedoch ad-hoc während der Mustersuche, wie im Fujaba-Interpreter aus [GHS09]. Dagegen schlagen [HVV07] und [VFV06a] vor, verschiedene *Varianten* unter unterschiedlichen Optimierungsgesichtspunkten zu generieren und zur Auswertung eine der Struktur des Laufzeitgraphen möglichst angepasste zu verwenden. Bedenkt man den Performanzvorteil *generativer* Lösungen, der bspw. in [Nag96] auf einen Faktor von zehn geschätzt wird, scheint dies ein gangbarer Ansatz zu sein. Problematisch wird dabei ggf. die kombinatorische Explosion möglicher Alternativen, die das Generieren angepasster Codevarianten stark einschränken kann.

Behandlung als Constraint-Satisfaction-Problem

Die Realisierung des Algebraischen Graphgrammatikwerkzeugs AGG [Tae99] basiert im Unterschied zu oben genannten Werkzeugen nicht auf der Erzeugung von Suchplänen. Stattdessen wird auch hier interpretativ vorgegangen, wobei Graphmuster auf sogen. Constraint-Satisfaction-Probleme (CSPs) zurückgeführt werden [Rud98].

Eine weitere Untersuchung zur Anwendbarkeit dieses Ansatzes zur Realisierung von Graphtransformationen findet sich im Übrigen in [LV02].

Obwohl die Generische Realisierung DRAGULAs primär der suchplanbasierten Variante zuzuordnen ist, können dennoch einige Konzepte der CSP-Realisierung übernommen werden, was aufgrund der syntaktischen Strukturierung DRAGULAs nahe liegt. So findet sich hier ebenso wie in AGG die Möglichkeit des Backjumping über mehrere zu belegende Variablen hinweg. Die direkte Anwendbarkeit obiger CSP-Algorithmen zur Realisierung der Ausführungsmaschine ist jedoch noch nicht untersucht worden.

Inkrementelle Graphtransformationsansätze

Im Unterschied zu üblichen Verfahren zur Graphmustersuche, bei der jede Suchanfrage *einzeln* ausgewertet wird, ermöglichen inkrementelle Ansätze die Wiederverwendung von Zwischenergebnissen, insbesondere einzelner Teilanfragen. Die im Kontext des VIATRA-Projekts verfolgte Variante [VVS06, BHRV08] greift auf die sogen. *RETE*-Netzwerke [For79] zurück, um bei Änderungen des Laufzeitgraphen Auswirkungen auf zwischengespeicherte Anfrageresultate zu ermitteln. Auf diese Weise werden unnötige Neuberechnungen bei lokal beschränkten Graphänderungen vermieden, allerdings erhöht dies den Aufwand zur Graphänderung deutlich.

Derzeit finden inkrementelle Ansätze noch keine Verwendung in der DRAGULA-Ausführungsmaschine. Der Einsatz bestehender Infrastruktur insbes. der im Graphspeicher vorhanden Unterstützung für *abgeleitete Attribute* [Bö06, Kap. 6] und der Ereignisverwaltung könnte jedoch einen Ansatzpunkt für zukünftige Arbeiten darstellen.

5.7.2 Graphtransformationen auf relationalen Datenbanken

Die im Zuge der austauschbaren Sprachrealisierung verfolgte Grundidee, bestehende relationale Datenbanken als Basis für Graphtransformationen zu nutzen, ist zuerst durch [VFV06b] vorgeschlagen worden. Im Unterschied zu DRAGOS, das Graphelemente nach *Sorten* in Tabellen einordnet, führt Varró diese Einteilung auf Basis von *Typen* durch. Somit werden anstelle der von DRAGOS eingesetzten Tabellen `Node`, `Edge`, etc. jeweils schemaspezifische Tabellen pro Entitätsklasse definiert. Kanten werden durch je nach Kardinalität durch Fremdschlüssel oder als gesonderte Tabellen repräsentiert.

Zur Beschreibung von Graphtransformationen verwendet [VFV06b] einfache Graphtransformationsregeln gemäß dem Single-Pushout-Approach einschließlich negativer Anwendbarkeitsbedingungen. Sowohl Suchmuster der linken Regelseiten als auch

negative Bedingungen werden als Datenbankanfrage abgelegt und bei der Ausführung von Graphtransformationen ausgewertet. Transformationen werden wiederum als Folge von Einfüge- und Aktualisierungskommandos an den Datenspeicher realisiert.

Im Vergleich zum hier vorgestellten Ansatz kann in den Arbeiten von Varró besser auf das jeweils bearbeitete Modell Bezug genommen werden, da eine typbasierte Abbildung von Modellen auf Datenbanktabellen erfolgt. Durch die direkte Ablage von Attributwerten in diesen Tabellen kann hierbei insbesondere von einer höheren Performanz ausgegangen werden, wohingegen DRAGULA Joins der Entitäts- und Attributwertetabellen erfordert. In dieser Hinsicht limitiert das vorgegebene Graphspeichersystem DRAGOS den Ansatz der vorliegenden Arbeit. Dagegen bietet DRAGULA flexiblere Einbindungsmöglichkeiten für zusätzliche Sprachkonstrukte an, die auf Basis eines eingesetzten Hintergrundspeichers als Generische Realisierung oder durch Reduktion erfolgen kann.

Weitere Arbeiten der Varró-Gruppe schlagen einen expliziten Mechanismus zur persistenten Speicherung von Zwischenergebnissen vor [VV04]. Diese umgehen das Problem unveränderlicher Datenbanksichten gängiger SQL-Datenbanksysteme und gehen in das Forschungsfeld inkrementeller Graphtransformationen (s.o.) über.

5.7.3 Statische Spezifikation von Graphmodellen

Der in [Bö06] vorgeschlagene Ansatz zur Verwendung des DRAGOS Graphen sieht vor, je nach Anwendung ein geeignetes *Graphmodell* zu definieren. Hierunter sind grundsätzliche Eigenschaften wie die Beschriftung von Knoten und Kanten, hierarchische Graphen, etc. zu verstehen. Hintergrund dieses Vorschlags ist es, die komplexe API des DRAGOS-eigenen Graphmodells zu verbergen und nur notwendige Funktionalität nach außen bereitzustellen. Hierzu beschreibt [Bö06] ein Werkzeug zur *Spezifikation* dieser Graphmodelle auf Basis von UML-Klassendiagrammen, die mit beliebigen UML-Editoren erstellt werden können um anschließend in lauffähigen Quellcode übersetzt zu werden. Durch Stereotypen erfolgt dabei die Abbildung der spezifizierten Klassen auf entsprechende Elemente des DRAGOS-Graphmodells. Zur Beschreibung der bereitzustellenden API stehen zudem verschiedene textuelle Makros zur Verfügung, mit denen die Funktionalität des zugrundeliegenden DRAGOS-Graphmodells vereinfacht genutzt werden kann. Ein Beispiel hierfür zeigt Abbildung 5.14, wo zum einen ein spezifiziertes Klassendiagramm, und zum anderen eine exemplarische Methodenspezifikation dargestellt sind.

Der DRAGOS-eigene Ansatz ist vergleichbar mit der hier beschriebenen syntaktischen Spezifikation von Sprachmodulen, da diese ebenfalls ein Klassenmodell erzeugt, welches sich auf den DRAGOS Graphspeicher abstützt. Der hier beschriebe-

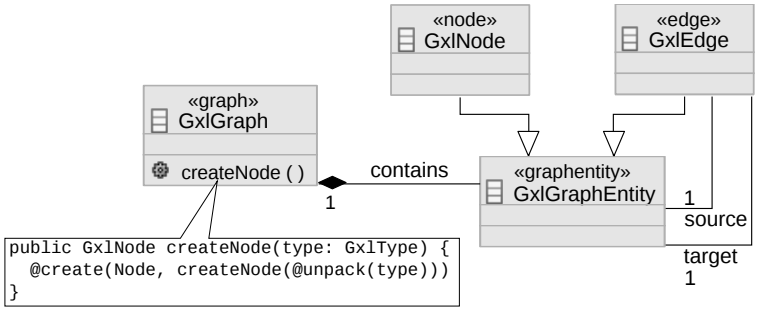


Abbildung 5.14: Spezifikation eines Graphmodells, reproduziert aus [Bö06]

ne Ansatz ist jedoch spezifisch zur Beschreibung bzw. Erweiterung der DRAGULA-Kernsprache ausgelegt, in dem Sinne, dass eine Standardarchitektur und entsprechende Konventionen zur Codegenerierung vorgegeben werden. Von daher ist die Beschreibung einzelner Zugriffsmethoden im Unterschied zum DRAGOS-eigenen System nicht nötig, allerdings besteht diese Eingriffsmöglichkeit auch nicht. Mit Hilfe der DRAGOS-eigenen Werkzeugunterstützung wäre aufgrund der fehlenden Unterstützung der DRAGULA Standardarchitektur die Spezifikation einer Spracherweiterung mit erheblichem Mehraufwand verbunden. Die hier vorgestellte Spezifikationssprache ist somit auf einem höheren Abstraktionsniveau angesiedelt und für ein engeres Anwendungsfeld konzipiert.

5.7.4 Dynamische Spezifikation von Sprachkonstrukten

Die Spezifikation dynamischen Verhaltens eines Sprachkonstrukts durch Rückführung kann als Anwendung eines *mehrstufigen Grammatikansatzes* aufgefasst werden. Im Kontext der Beschreibung von Programmiersprachen ist dies bereits durch [Wij74] eingeführt worden. Aus dem Bereich der Graphgrammatiken könnte auch bspw. der in [DHJ+07] vorgeschlagene Ansatz, mittels *Platzhaltern* Teile einer Graphersetzungsregel zunächst offen zu lassen, angewendet werden. Hierbei würde die Reduktionsregel des nichtterminalen Sprachkonstrukts die einzig anwendbare Ersetzungsregel darstellen, die eine ausführbare terminale Ersetzungsregel erzeugt.

Ein allgemeinerer Ansatz als die bislang verwendeten kontext*freien* Grammatiken stellen die in [van08] behandelten Transformationen höherer Ordnung dar. Diese fordern keine Kontextfreiheit der reduzierenden Transformationsregel und können daher allgemeiner formuliert werden. Einen ähnlichen Ansatz verfolgen [VP04] für reflektive Transformationen in unbekannten Graphschemata sowie [Ran08] zur Realisierung von Graphsichten. Im Kontext der vorliegenden Arbeit ist bewusst auf die Einführung kontextsensitiver Reduktionsregeln verzichtet worden, da dies einen un-

erwünschten Komplexitätsgrad sowohl in die Regelspezifikation als auch in die Ausführung der Reduktion einbringen würde. Zudem müssten geeignete Analysen, wie etwa eine Critical-Pair-Analysis [HKT02], das Überlappen verschiedener Reduktionsregeln ausschließen, um ein deterministisches Ergebnis der Sprachkonstruktreduktion zu erhalten. Im kontextfreien Fall ist dies vergleichsweise einfach durch Ausschluss mehrerer Regeln für ein nichtterminales Konstrukt möglich.

5.8 Zusammenfassung

Dieses Kapitel hat einen Einblick in die Implementierung der DRAGULA Ausführungsmaschine vermittelt. Hierzu zählt sowohl die Generische Realisierung als auch eine SQL-basierte Variante, wobei letztere aus Gründen der Pragmatik lediglich die Muster*suche* realisiert. Bezogen auf die in Kapitel 1 aufgestellten Anforderungen wird somit Anforderung 4 auf Seite 12 bedient. Durch die Anpassbarkeit des Hintergrundspeichers an konkrete Speicherlösungen sowie die Möglichkeit spezifische Speicher effizient zu nutzen wird eine Abstraktion von technischen Plattformen erreicht. Die mittels DRAGULA beschriebenen Graphtransformationen sind somit unabhängig davon einsetzbar.

Den ungefähren Realisierungsaufwand der Ausführungsmaschine und damit zusammenhängender Komponenten zeigt Tabelle 5.1. Die einzelnen Angaben beziehen sich auf Quelltextzeilen und umfassen sowohl Implementierungs- als auch Testcode. Zu sehen ist dabei, dass die Realisierung der *Kernsprache* den wesentlichen Umfang einnimmt. Zusätzlich ist der *Technische Adapter*, mit dessen Hilfe dateibasierte DRAGULA-Spezifikationen mit dem Graphspeicher ausgetauscht werden, mit etwa 2200 Quelltextzeilen realisiert worden. Dieser ist jedoch nicht spezifisch für die Reduktion von Sprachkonstrukten wie oben angegeben, sondern wird auch durch einen *visuellen Editor* für DRAGULA-Spezifikationen genutzt. Allgemeine Infrastruktur sowie Java-basierte Realisierungen der Spracherweiterungen nehmen einen vergleichsweise geringen Umfang ein.

Komponente	Java-Quelltext	Spezifikation
Kernsprache	13.440	140
Erw. Attributausdrücke	1.050	100
Erw. Abschluss	180	20
Infrastruktur	980	–
Techn. Adapter	2.210	–
Entwicklungsumgebung f. Spracherw.	–	1200
Visueller DRAGULA Editor	–	1.900
Gesamt	**17.860**	**3.360**
einschließlich Generat	50.640	

Tabelle 5.1: Umfang der Ausführungsmaschine & zugehöriger Komponenten

Eine semantische Spezifikation von Sprachkonstrukten wird bisher primär durch Reduktion unterstützt. Die Mächtigkeit dieses Ansatzes ist vergleichbar mit der kontextfreier Grammatiken, da jeweils nur direkt adjazente Spezifikationselemente verarbeitet werden können. Diese Ersetzung könnte somit mittels adaptiver Sterngrammatiken [DHJ+06] beschrieben werden. Nicht verfolgt – da bisher nicht notwendig – wurde bisher eine Integration des im folgenden Kapitel vorgestellten Übersetzungsansatzes, um höherwertige Sprachkonstrukte in korrespondierende Konstrukte der Kernsprache zu übersetzen. Auch der Einsatz DRAGULAs zur Ersetzung in Form sogenannter *Higher-Order-Transformations* ist leicht denkbar, wird aber aufgrund der damit einhergehenden Komplexität für den Entwickler nicht verfolgt.

6 SViTL - Sprachrealisierung durch Transformation

Im bisherigen Verlauf dieser Arbeit ist DRAGULA als Kernsprache für Graphtransformationen und die zugehörige Ausführungsmaschine vorgestellt worden. Dieses Kapitel vervollständigt den in Abbildung 6.1 gezeigten Gesamtüberblick um eine Möglichkeit, eigene Spezifikationssprachen durch einen regelbasierten Übersetzer an diese Maschinerie anzuschließen. Somit wird eine leichtere Nutzung der bisher vorgestellten Basistechnologien ermöglicht, so dass der Realisierungsaufwand verhaltensorientierter Spezifikationssprachen auf insgesamt zweierlei Weise reduziert wird: Einerseits besteht mit DRAGULA eine Realisierungsplattform auf hohem Abstraktionsniveau, andererseits unterstützt die in diesem Kapitel vorgestellte Transformationssprache Entwickler bei der technischen Abbildung einer zu realisierenden Sprache auf DRAGULA. Auch diese Transformationsbeschreibung ist durch eine deklarative und graphbasierte Ausgestaltung auf einem hohen Abstraktionsniveau angeordnet, und fügt sich somit in das übrige Konzept dieser Arbeit ein.

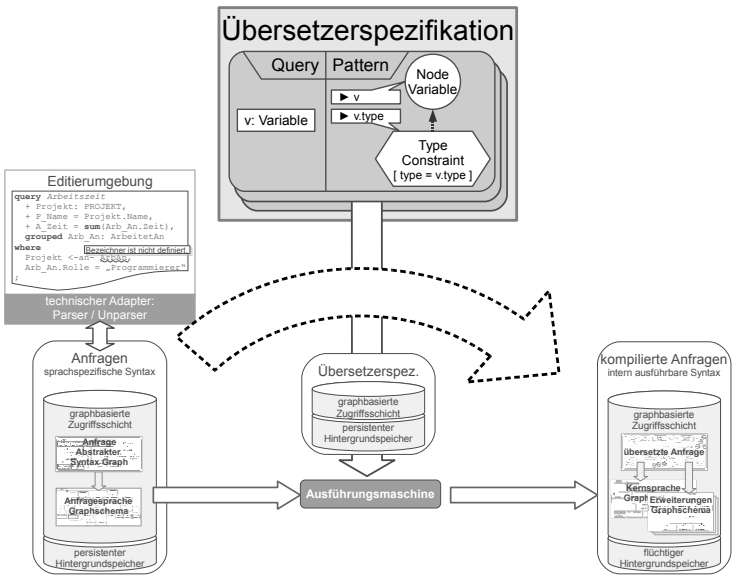


Abbildung 6.1: Einordnung der Übersetzersprache (analog zu Abbildung 1.5)

Im Folgenden wird zunächst das in Abschnitt 3.3 vorgestellte Beispiel der Anfragesprache EMAA aufgegriffen und mit DRAGULA als Realisierungsgrundlage verglichen. Hieraus leitet sich die notwendige Sprachabbildung ab, die werkzeugunterstützt ermöglicht werden soll. Auch werden hieran spezielle Anforderungen deutlich, die durch Nutzung DRAGULAs als Zielplattform entstehen. Diese können durch Bereitstellung einer *spezialisierten* Lösung in Form einer Transformationssprache besser abgedeckt werden, als dies in bestehenden anwendungsfallunabhängigen Ansätzen möglich ist. Beschreibung und Demonstration dieser als SViTL – *Simple Visual Transformation Language* – bezeichneten Transformationssprache, unter Nutzung des Anfragemechanismus EMAA als durchgängiges Beispiel, ist Gegenstand des weiteren Kapitels.

6.1 Einleitendes Beispiel

Um den Bedarf und die Anforderungen nach einer geeigneten Transformationssprache zu klären, analysiert dieser Abschnitt zunächst ein einleitendes Beispiel. Zunächst erfolgt ein exemplarischer Vergleich einer EMAA-Anfrage in der zugehörigen abstrakten Syntax mit einer entsprechenden DRAGULA Musterstruktur. Hieran können Anforderungen identifiziert werden, die zur Ausgestaltung von SViTL geführt haben.

6.1.1 Umsetzung einer Beispielanfrage

Die eingangs in Abbildung 3.5 auf Seite 65 behandelte, und in Abbildung 6.2a erneut dargestellte Anfrage, dient zur Ermittlung aller Projekte einer Projektmanagementumgebung sowie deren transitiv zugeordneten Teilaufgaben. Letztere sind durch direkt zugeordnete Aufgaben gegeben die durch Verbindungen des Typs `besteht_aus` erreichbar sind sowie deren (möglicherweise transitiven) `Teilaufgabe`n.

Dieses Beispiel vermittelt einen Eindruck unter Nutzung weniger Sprachkonstrukte, setzt jedoch auch bereits nicht-trivial zu realisierende Sprachkonstrukte, wie den transitiven Pfadabschluss ein. Für dieses Beispiel sollen nun sowohl die Darstellung der Anfrage in abstrakter Syntaxform, basierend auf dem Metamodell von EMAA (vgl. Abbildung 3.3 auf Seite 60), als auch eine äquivalente ausführbare Form unter Nutzung von DRAGULA verglichen werden.

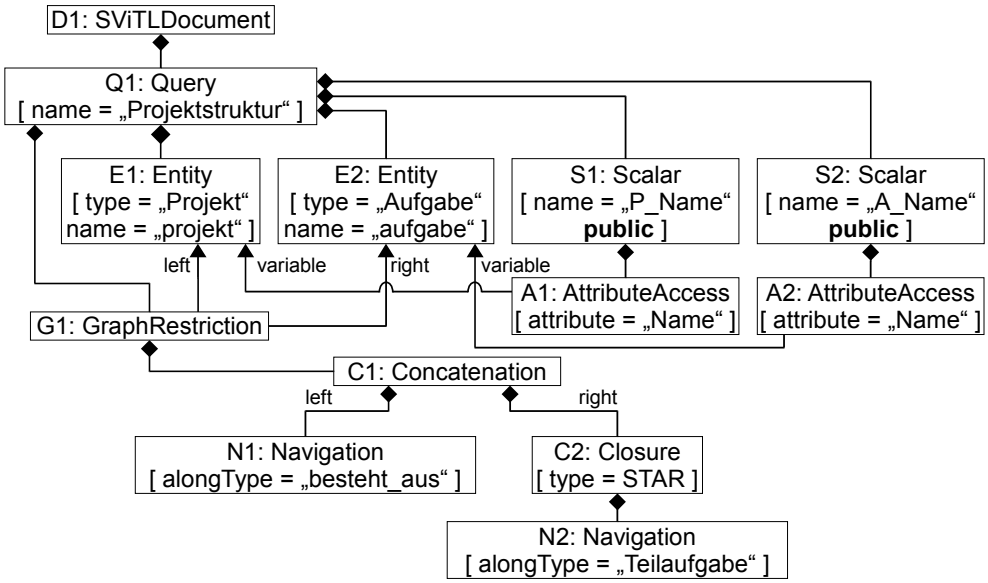
EMAA-Darstellung

Der erste Schritt zur Verarbeitung der Anfrage besteht, wie das Übersichtsbild in Abbildung 6.1 auf der linken Seite andeutet, in der Analyse der konkret-syntaktischen

Darstellung bspw. durch einen Parser. Abbildung 6.2b zeigt einen hierzu ermittelten abstrakten Syntaxgraphen, der das Metamodell aus Abbildung 3.3 instanziiert. Bezogen auf die konkrete Syntax sind angewandte Auftreten von Bezeichnern bereits durch Querbezüge im syntaktischen Modell aufgelöst worden, es handelt sich daher nicht um eine sonst übliche Baumstruktur. Beispielsweise verweist die dargestellte Graphrestriction als Behälter des Pfadausdrucks auf die zugeordneten Start- und Zielvariablen, anstatt dies durch implizite Bezeichnerbindung zu modellieren.

```
query Projektstruktur
  + P_Name = projekt.Name,
  + A_Name = aufgabe.Name,
  projekt : Projekt,
  aufgabe : Aufgabe
where
  projekt -/besteht_aus/->
    ( -/Teilaufgabe/-> )* aufgabe
;
```

(a) Konkrete Syntax (aus Abbildung 3.5)



(b) Abstrakte Syntax gemäß EMAA-Metamodell

Abbildung 6.2: Beispielanfrage und zugehöriger Abstrakter Syntaxgraph

DRAGULA-Darstellung

Abbildung 6.3 zeigt obige Anfrage in Form einer DRAGULA Musterstruktur. Dargestellt sind insgesamt drei Muster mit verschiedenen Zuständigkeiten, wie die jeweiligen Kommentarreiter andeuten.

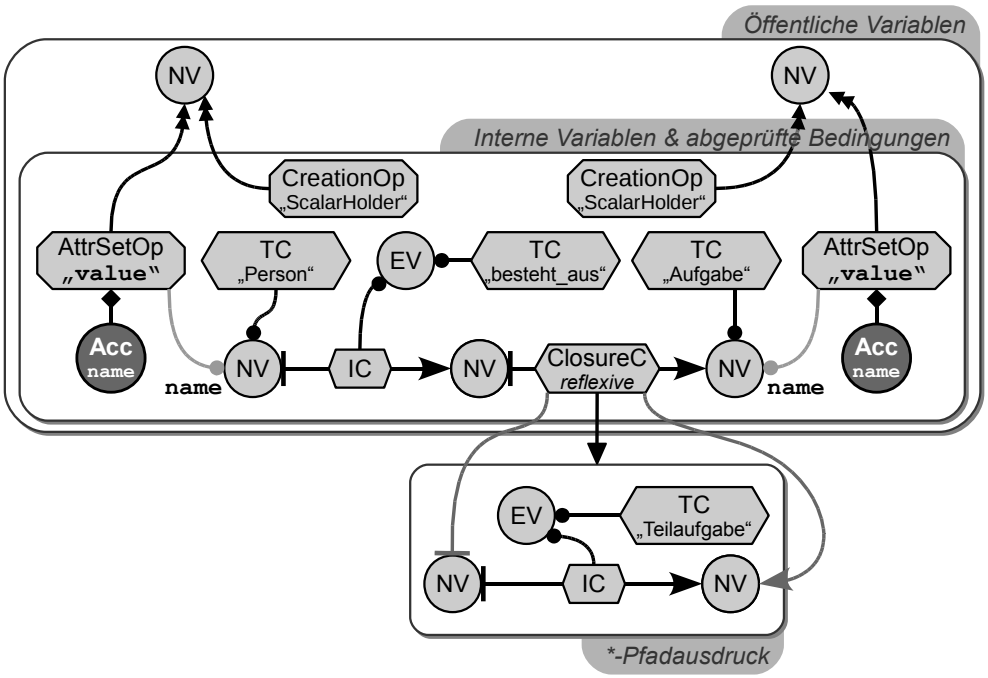


Abbildung 6.3: Ausführbare Form in DRAGULA

- Das äußere Muster oben enthält *öffentliche Variablen*, die das Ergebnis der Anfrage repräsentieren. Wie in Abschnitt 4.6.2 beschrieben, wird hierzu ein Behälterknoten vom Typ ScalarHolder eingesetzt, der zunächst mittels einer Erstelloperation erzeugt werden muss.
- Im inneren Muster finden sich nicht-öffentliche Variablen der Anfrage, hier zur Ermittlung der Graphentitäten für das zu suchende Projekt und die zugehörige Aufgabe. Diese sind durch Typ- und Inzidenzbedingungen gemäß der angefragten Entitätstypen sowie der GraphRestriction aus Abbildung 6.2b eingeschränkt. Zur Ergebnisermittlung sind zudem Operationen zur Wertezuweisung der Behälterknoten definiert. Diese ermitteln die Werte der name-Attribute der Entitätsvariablen, und weisen diese dem Attribut value des Behälters zu.
- Das untere Muster dient zur Definition des transitiv abgeschlossenen Teilausdrucks Closure der GraphRestriction, und liegt außerhalb obiger Musterstruktur.

Referenziert wird dies durch ein ClosureConstraint das in Abschnitt 4.6.1 vorgestellt worden ist.

Bei Ausführung dieses Graphmusters, d.h. der Bestimmung aller gültigen Auffindungsstellen, wird als Ergebnis eine Struktur analog zur Tabelle in Abbildung 3.5 auf Seite 65 ermittelt. Hierbei entspricht eine Tabellenzeile den Attributwerten der Behälterknoten einer Auffindungsstelle des äußeren Musters von Abbildung 6.2b.

6.1.2 Ergebnis des Vergleichs

Die vorgestellte Abbildung von EMAA auf DRAGULA zeigt, dass in verschiedenen Fällen keine direkte Korrespondenz der jeweiligen Elemente besteht. Stattdessen bestehen verschiedene *übergreifende* Zusammenhänge, etwa in Hinblick auf die Darstellung eines Pfadausdrucks in beiden Formen. Dieser wird im EMAA-Metamodell baumartig (zzgl. Querbeziehungen) aufgetragen, mittels DRAGULA dagegen als flache Musterstruktur modelliert. Gleichzeitig sind Spezifika der Zielsprache DRAGULA zu berücksichtigen, bspw. die Auslagerung transitiv abgeschlossener Teilmuster.

Dieses Beispiel zeigt den Bedarf nach einer Unterstützung bei der Übersetzung von der zu realisierenden Modellierungssprache in die Kernsprache DRAGULA. Ziel dieser Arbeit ist eine angemessene Werkzeugunterstützung auf Spezifikationsebene, so dass die Übersetzung einzelner Dokumente, im Falle EMAAs also Anfragen, automatisiert erfolgen kann. Die Erstellung eines solchen Übersetzers kann aufgrund der bereits hier aufgezeigten konzeptionellen Unterschiede nicht automatisiert bspw. unter alleiniger Betrachtung beider Metamodell erzeugt werden. Von daher ist eine entsprechende Spezifikation durch einen Entwickler notwendig.

Ferner kann dem Beispiel entnommen werden, dass bei der zu spezifizierenden Übersetzung verschiedene Eigenschaften der Zielsprache zu berücksichtigen sind. So müssen neben der reinen Abbildung von EMAA-Sprachkonstrukten auf DRAGULA-Elemente letztere einem passenden DRAGULA-Muster zugeordnet werden, da auf diese Weise eine Unterscheidung bspw. zwischen öffentlichen und privaten Variablen erfolgt. Die Zuordnung zum dritten, ausgelagerten Muster bestimmt die Ermittlung des transitiven Abschlusses der modellierten Verbindungsnavigation. Somit ist also die Musterzuordnung der zielseitig erstellenden Ergebnisse gezielt zu berücksichtigen. Um den hierfür notwendigen Spezifikationsaufwand gering zu halten, sollte die Spezifikation einzelner Übersetzungsaspekte unabhängig von dem zu verwendenden Behälter erfolgen können, wodurch nur an wesentlichen Punkten eine entsprechende Festlegung erfolgt. Hierfür benötigt der Transformationsmechanismus einerseits allgemeine Kenntnis über das zugrundeliegende hierarchische Datenmodell, andererseits auch über die spezifische Verwendung hierarchischer Strukturen in DRAGULA.

Somit kann der Spezifikationsaufwand für den Übersetzer gesenkt werden, in dem Behälterinformationen durch den anzuwendenden Mechanismus zumindest teilweise automatisiert behandelt werden.

6.2 Übersetzeransatz

Ausgehend vom initialen Beispiel des vorangegangenen Abschnitts stellt dieser einige grundlegende Eigenschaft des Lösungsansatzes der vorliegenden Arbeit vor. Unterschieden wird hierbei zunächst zwischen den Entwurfsentscheidungen der Transformationssprache sowie der grundsätzlichen Spezifikationsweise und dem damit modellierten Übersetzerablauf. Im Anschluss an diesen exemplarischen Überblick wird in den nachfolgenden Abschnitten die angebotene Funktionalität detailliert erläutert sowie die Übersetzung des Eingangsbeispiels ausspezifiziert.

6.2.1 Rational

Bei der Auswahl bzw. der Entwicklung eines geeigneten Transformationsansatzes sind verschiedene Randbedingungen zu berücksichtigen, die im Folgenden aufgezeigt werden. Diese ergeben sich aus der Zielsetzung, die Nutzung der in den vorangegangenen Kapiteln vorgestellte Basistechnologie zu erleichtern. Von daher finden sich hier die bereits im Vorfeld identifizierten Problemstellungen wieder, die u.a. die Nutzung von DRAGULA als Zielsprache der Sprachübersetzung hervorbringt. Diese entsprechen in vielen Aspekten dem gängigen Stand der Technik im Kontext von Modelltransformationssprachen und -werkzeugen, so dass diese entsprechend kurz behandelt werden können. Darüber hinaus sind die oben diskutierten Spezifika des von DRAGULA syntaxseitig eingesetzten Datenmodells und der Spezifika der zugeordneten Interpretation bspw. geschachtelter Musterstrukturen zu berücksichtigen.

Deklarative Spezifikation. Durch eine deklarative Übersetzerspezifikation wird das Potential einer hochsprachlichen Ausführungsplattform wie DRAGULA konsequent zur Realisierung von Sprachabbildungen fortgeführt. Von daher sollten Übersetzer nicht *imperativ*, d.h. unter Angabe einzelner Analyse- und Syntheseschritte erstellt werden, wie dies für klassische Programmiersprachen üblicherweise erfolgt. Eine *deklarative* und insbesondere regelbasierte Herangehensweise ermöglicht dagegen eine verständlichere Beschreibung und erleichtert zudem die spätere Anpassbarkeit. Ein solcher Ansatz wird von gängigen Arbeiten im Bereich Modelltransformationen verfolgt und ist in zahlreichen Werkzeugen anzutreffen [TEG+05].

In SViTL wird diese Eigenschaft erreicht indem Transformationen als *Gegenüberstellung* korrespondierender Modellfragmente der Quell- und Zielseite beschrieben werden. Insbesondere werden nur minimal zusätzliche Konstrukte eingeführt um die jeweiligen Fragmente zueinander in Beziehung setzen zu können. Ein einführendes Beispiel hierzu liefert der folgende Abschnitt 6.2.2.

Konzeptzentrische Modellierung. Durch Fokussierung auf einzelne Sprachkonzepte kann die Übersetzerspezifikation vereinfacht an eine (bspw. durch Evolution) veränderte Quellsprache angepasst werden, da Querbeziehungen innerhalb der Spezifikation reduziert werden.

SViTL erreicht diese Eigenschaft durch den Verzicht auf explizite Zusammenhänge zwischen spezifizierten Regeln, wie dies in anderen Ansätzen etwa durch Ein- und Ausgabeparameter erreicht wird. Zudem können Übersetzungsregeln nicht auf Ergebnisse vorheriger Regelanwendungen zugreifen, so dass eine Reihenfolgeunabhängigkeit bei der Übersetzerausführung gewährleistet ist. Dies unterstützt gleichzeitig den gewählten deklarativen Spezifikationsansatz.

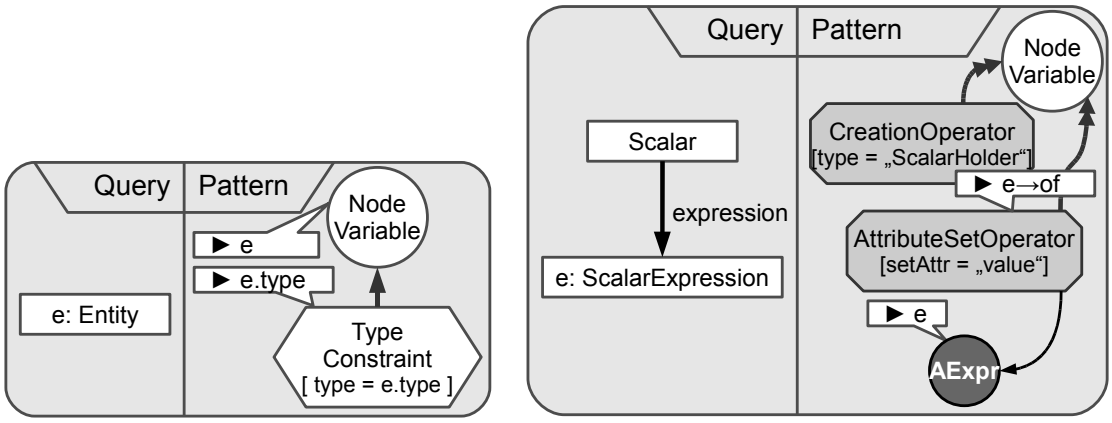
Unabhängigkeit vom Quelldatenmodell. Um eine allgemeine Plattform für verhaltensorientierte Spezifikationssprachen anbieten zu können, sollte auch die als Hilfswerkzeug angebotene Transformationssprache keine Beschränkungen der zu verarbeitenden Datenmodelle herstellen. SViTL erreicht dies durch die Nutzung des Graphspeichers DRAGOS zur Darstellung der Quelldaten, wodurch ein entsprechend mächtiges Graphmodell zur Verfügung steht.

Unterstützung des Zieldatenmodells. Im selben Kontext wie oben ist die zielseitige Darstellung von Datenstrukturen zu sehen. Obwohl seitens DRAGULAs verschiedene nicht-standard Konstrukte wie etwa hierarchische Graphstrukturen und attributierte Kanten eingesetzt werden, kann das Datenmodell im Übrigen auf diese beschränkt werden. Eine zielseitige Unterstützung von *Hyperkanten* muss seitens SViTL daher bspw. nicht erfolgen.

Unterstützung der Zielsprache. Durch eine Berücksichtigung der Zielsprache kann der Spezifikationsaufwand für Entwickler wesentlich reduziert werden. Wie bereits erläutert können so bspw. Enthaltenseinshierarchien teilweise implizit behandelt werden, da diese Eigenschaft quer zu anderen Aspekten der Sprache – wie etwa Verbindungsstrukturen – liegt. Diesbezüglich verfolgt SViTL einen *heuristikbasierten* Ansatz der entsprechende Informationen von benachbarten Elementen propagiert. Ermöglicht wird dies durch eine Fokussierung auf DRAGULA als Zielsprache der Übersetzung, dieser letzte Punkt ist also nicht direkt auf andere Zielplattformen übertragbar.

6.2.2 Modellierungsseitiger Überblick

Dieser Abschnitt gibt einen kurzen Überblick zur Nutzung von SViTL als Spezifikationssprache. Abbildung 6.4 zeigt hierzu zwei Abbildungsregeln der abstrakten Syntax von EMAA auf entsprechende Elemente der DRAGULA Kernsprache. Als Beispiel dienen hierbei die Darstellung von Entitätsvariablen in Abbildung 6.4a sowie von skalaren Rückgabewerten in Abbildung 6.4b.



(a) Übersetzung des Konstrukts Entity (b) Übersetzung des Konstrukts Scalar

Abbildung 6.4: Übersetzung von Entitäts- und Skalarvariablen

Das erste Abbildungsfragment in Abbildung 6.4a ermittelt gemäß der linken Seite eine Entität in der zu transformierenden Anfrage. Als Behälter dieses Elements ist ein Query-Objekt zu ermitteln, was durch die Textangabe im umgebenden Rahmen festgehalten wird. Auf Basis einer derartigen Auffindungsstelle soll die rechts dargestellte Struktur erzeugt werden, die hier aus einer Knotenvariablen und einer Typbedingung sowie einem enthaltenden Muster besteht. Zur Darstellung von skalaren Variablen wird in Abbildung 6.4b eine Variable für Behälterknoten erzeugt. Dieser Variablen soll im Zuge der Anfrageverarbeitung ein neuer Knoten zugeordnet und dessen value Attribut entsprechend gesetzt werden. Der zu setzende Wert wird dabei noch nicht festgelegt, sondern nur durch Angabe eines Ausdrucksknotens (Expression) als „Platzhalter“ bereitgestellt.

Neben der Gegenüberstellung von Sprachkonstrukten werden in den beiden obigen Beispielen zielseitige Elemente an entsprechenden Gegenstücken der Quellseite *verankert*, was durch die Kästen mit Pfeilbeschriftung angegeben wird. Diese verweisen auf ein Element der Quellseite zu dem das zielseitige Element zugeordnet werden soll. Eine solche Zuordnung muss eindeutig sein, d.h. pro quellseitigem Element kann nur eine Verankerung auf der Zielseite auftreten. Möglich ist jedoch die Angabe er-

gänzender *Beschriftungen* zur weiteren Unterscheidung, die durch den dargestellten Pfeil → vom Elementnamen getrennt sind. In Abbildung 6.4b verweisen beide Verankerungen auf den quellseitigen Berechnungsausdruck e, durch die Verwendung der Beschriftung of sind jedoch beide Angaben unterscheidbar.

Neben quellseitigen Elementen der abstrakten Syntax können Verankerungen auch Bezug auf deren Attribute nehmen. In der Typbedingung von Abbildung 6.4a wird dies benötigt, da diese spezifisch für einen anzufragenden Entitätstypen (Attribut type) gelten soll.

Da eine SViTL-Spezifikation primär Paare von Modellfragmenten in Relation setzt, wird hierfür im Folgenden der Begriff *SViTL-Paar* verwendet. Dieser vermeidet zugleich Mehrdeutigkeiten mit dem sonst üblichen Begriff *Regel*, da diese auch ziel- und ggf. quellseitig auftreten können.

6.2.3 Überblick des Transformationsablaufs

Das Ergebnis der oben beschriebenen Übersetzung zeigt Abbildung 6.5. Wie dargestellt werden quellseitig zunächst *alle Auffindungsstellen* beider Paare *unabhängig* voneinander ermittelt, und entsprechende DRAGULA-Fragmente zielseitig erstellt. Das Resultat hierzu zeigt die rechte Seite der Abbildung. Ankerdefinitionen werden in Form von *Verankerungen*[1] der DRAGULA-Elemente umgesetzt, die auf das jeweilige Element der Auffindungsstelle im abstrakten Syntaxgraphen (vgl. Abbildung 6.2b) verweisen. Bei Nutzung eines Attributs wie in obiger Typbedingung wird anstelle eines Elements des Syntaxgraphen der konkrete Attributwert verankert, hier also die Zeichenketten „Projekt“ bzw. „Aufgabe“.

Im späteren Verlauf der Übersetzung werden die bislang einzelnen Fragmente miteinander *verklebt* und zu einer vollständigen DRAGULA-Spezifikation zusammengeführt. Dies erfolgt auf Basis identischer Verankerungen der Elemente, wie die nachfolgenden Beispiele zeigen werden. Insofern synchronisieren Anker die Anwendung von SViTL-Paaren *nachträglich*, da sie zusammenhängende Modellfragmente zusammenführen. Dies erfordert insbesondere keine explizite Festlegung von Transformationsreihenfolgen durch den Entwickler.

Insgesamt ergeben sich drei Verarbeitungsschritte, die im Folgenden aufgezählt werden. Im weiteren Verlauf dieses Kapitels werden die entsprechenden Punkte jeweils eingehend diskutiert.

1. Ermittlung aller quellseitigen Auffindungsstellen und synchrone zielseitige Erzeugung der zugeordneten DRAGULA-Fragmente. Dieser Aspekt betrifft haupt-

[1] ebenfalls dargestellt in Form von legendenförmigen Beschriftungen

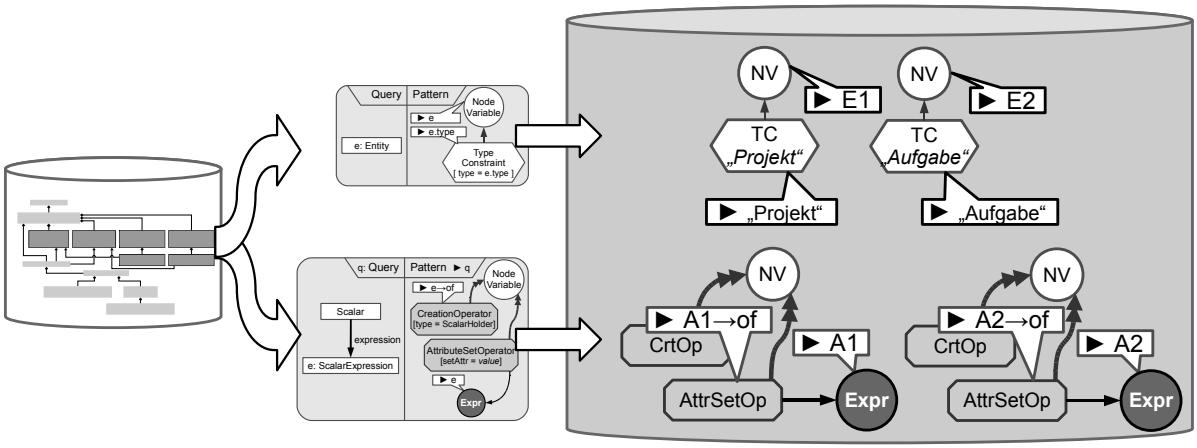


Abbildung 6.5: Ergebnis der Übersetzung

sächlich die syntaktische Definition von SViTL und ist Gegenstand von Abschnitt 6.3.

2. Verklebung der erstellten Elemente gemäß gemeinsamer Verankerungen sowie Berücksichtigung unverankerter Elemente. Dies betrifft die Transformationsausführung und wird eingehend in Abschnitt 6.4 behandelt.

3. Bestimmung geeigneter Behälter für die erstellten Elemente sowie Zuordnung zu diesen. Dieser letzte Teil der SViTL-Sprachbeschreibung wird in Abschnitt 6.5 behandelt.

6.2.4 Nutzung einer übersetzten Spezifikation

Um die übersetzte EMAA-Anfrage nutzen zu können, ist eine entsprechende Ansteuerung der DRAGULA-Ausführungsmaschine erforderlich. Hierzu greift ein Anwender durch eine geeignete Programmierschnittstelle auf den DRAGOS Graphspeicher zu, um die als DRAGULA-Muster vorliegende übersetzte Anfrage auszuwählen und an die Ausführungsmaschine zu übergeben. Ferner ist seitens der Programmierschnittstelle eine Auswertung der resultierenden Auffindungsstellen vorzunehmen, um bspw. Behälterentitäten für skalare Werte auszulesen und entsprechend darzustellen. Obwohl die hierfür benötigte Funktionalität aufgrund der hohen Anwendungsfallspezifizität derzeit nicht automatisiert bereitgestellt wird, ist der manuelle Implementierungsaufwand vergleichsweise gering. Aufgrund der technischen Eigenschaften dieses Aspekts wird er in diesem Kapitel nicht weiter betrachtet.

6.3 Modellierung und Übersetzung einfacher Spezifikationsdokumente

Dieser Abschnitt stellt die grundlegenden Sprachkonstrukte der Transformationssprache SViTL vor. Ferner werden das Laufzeitverhalten und hierbei insbesondere die bereits erwähnte Verklebung der bei der Transformationsdurchführung erstellten Elemente eingehend behandelt. Die Erläuterung erfolgt in diesem Abschnitt zunächst unabhängig vom verwendeten hierarchischen Datenmodell und einer möglichen Berücksichtigung der damit verbundenen DRAGULA-seitigen Bedeutungszuordnung – insofern können bislang nur einfache Spezifikationssprachen als Eingabe verarbeitet werden.

6.3.1 Syntaktische Sprachdefinition

Das syntaktische Metamodell der Sprache SViTL wird in Abbildung 6.6 dargestellt. Im Unterschied zur Definition DRAGULAs kann die Sprachbeschreibung hier stark verdichtet erfolgen, da SViTL größtenteils auf bereits vorhandenen Syntaxstrukturen basiert. So referenziert die linke Seite eines SViTL-Paars das DRAGOS Graphmodell (vgl. Abschnitt 2.4), die rechte Seite aufgrund der spezifischen Ausrichtung der Transformationssprache bezieht sich dagegen auf die DRAGULA Definition (vgl. Abschnitt 4.2).

Allgemeine Syntaxelemente:

Lediglich zwei Elemente des dargestellten Metamodells beziehen sich auf die allgemeine Beschreibung einer SViTL-Spezifikation. Hierzu zählen SViTLDocument zur Darstellung einer Gesamtspezifikation, die wiederum eine (nicht-leere) Menge von Paaren (Pair) enthält. Paare setzen sich aus quell- und zielseitigen Elementen zusammen, deren Querbezüge mittels weiterer Sprachkonstrukte ausspezifiziert werden können.

Quellseitige Syntaxelemente

Quellseitig erlaubt SViTL die Spezifikation beliebiger Musterstrukturen, für die passende Auffindungsstellen im zugrundeliegenden Graphspeicher ermittelt werden sollen. Ein einfaches Sprachkonstrukt, das sich direkt auf ein solches Graphelement bezieht, wird durch EntityLHS bereitgestellt. Hierdurch können bspw. die in Abbildung 6.4b dargestellten quellseitigen Knoten und die verbindende Kante modelliert werden.

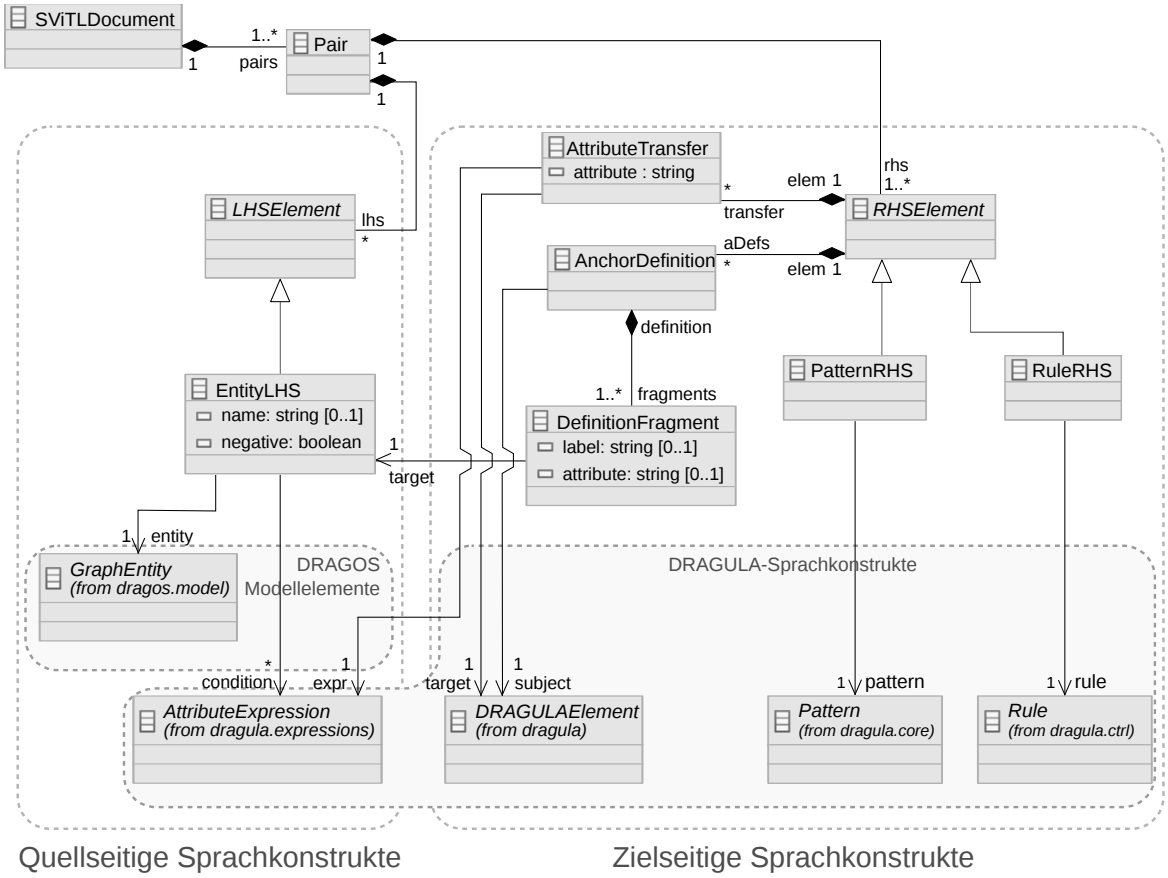


Abbildung 6.6: SViTL Metamodell

Durch die Einführung eines dedizierten Konstrukts EntityLHS, anstatt direkt Bezug auf das Graphmodell zu nehmen, kann eine optionale *Benennung* der Graphelemente erfolgen, die nicht im DRAGOS-Graphmodell vorgesehen ist. Benannte Entitäten können bei der Auswertung von Attributausdrücken und bei der Korrespondenzmodellierung referenziert werden.

Attributwerte der quellseitig gesuchten Elemente können überprüft werden, in dem ein Objekt des Typs AttributeExpression der Attributerweiterung aus Abschnitt 4.6.2 als Bedingung angegeben wird. Diese Ausdrücke können auf Attributwerte quellseitig benannter Elemente des gleichen Paars zugreifen. Hierbei wird die aus gängigen Programmiersprachen bekannte *Punktnotation* verwendet, um den Namen des graphischen Elements vom zu lesenden Attribut zu unterscheiden[2]. Die folgende Wohlgeformtheitsbedingung beschreibt den Zugriff von Attributausdrücken auf Elemente

[2]Dies ist in DRAGULA nicht notwendig, da referenzierte Entitätsvariablen und Attributbedingungen bzw. -operatoren *explizit* durch entsprechende Kanten verbunden werden.

der linken Seite. Dabei wird eine Hilfsfunktion allElements für AttributeExpresssion vorausgesetzt, die alle transitiv enthaltenen Teilbedingungen liefert – diese Funktion wird zur kompakteren Darstellung nicht ausspezifiziert.

```
context Access
def accessed : String = self.variable.split('.').at(0)
      -- Segment vor erstem Punkt --
def attribute : String = self.variable.split('.').at(1)
      -- Segment nach erstem Punkt --

context EntityLHS
inv properAccess :
      let accessible : Collection(EntityLHS) = self.pair.lhs→
        select( l | not l.name.oclIsUndefined() ) in
          -- alle zugreifbare Entitaeten der Quellseite --
      let valueAccess : Collection(Access) = self.condition→
        collect( c | c.allElements() )→
          select( a | a.isOclTypeOf(Access) ) in
          -- alle Wertzugriffe des referenzierten Ausdrucks --
      valueAccess→forAll(v | v.accessed() in
        accessible→collect( a | a.name ) )
```

Durch die *Negation* einzelner Sprachkonstrukte wird festgelegt, dass *keine* Belegung bei der Mustersuche gefunden werden darf – diese stellt eine eingeschränkte Form der negierten Musterstrukturen aus Abschnitt 4.3 dar. Negierte quellseitige Elemente können nur dann in Attributbedingungen auftreten, falls diese selbst einem negierten Element zugeordnet sind.

Des Weiteren ist die Beschreibung von *Pfadausdrücken* zwischen quellseitigen Elementen vorgesehen, die syntaktisch als konkrete Unterklasse von LHSElement beschrieben werden. Diese Möglichkeit ist jedoch noch nicht in die formale Sprachbeschreibung aufgenommen worden, und wird daher im Folgenden nicht genutzt.

Zielseitige Syntaxelemente

Zielseitig können grundsätzlich zwei verschiedene Konstrukte eingesetzt werden um einen Behälter für die jeweiligen DRAGULA-Elemente bereitzustellen. PatternRHS bezeichnet einen Behälter für Musterstrukturen, der sich auf ein DRAGULA-*Muster* (vgl. 4.2) bezieht. Hierüber werden einzelne Such- und Transformationsvorschriften erzeugt. Dem gegenüber beschreibt RuleRHS eine *Regel*struktur, mit deren Hilfe die Zusammenschaltung einzelner Muster beschrieben werden kann. Demzufolge referenziert dieses Konstrukt ein Rule-Element aus DRAGULA (vgl. 4.5).

Das Syntaxelement AttributeTransfer beschreibt schließlich die Zuweisung eines Attributwerts an das als subject referenzierte DRAGULA-Element. Auch hier wird zur Wertermittlung auf die Attributerweiterung aus DRAGULA insoweit zurückgegriffen, als dass die vordefinierten syntaktischen Elemente wiederverwendet werden können. Das zu setzende Attribut des referenzierten Elements wird durch seinen Namen attribute identifiziert. Ein Beispiel hierfür zeigt die bereits behandelte Abbildung 6.4a, wo das Attribut type des zielseitigen Elements TypeConstraint gesetzt wird.

Korrespondenzmodellierung

Wesentlicher syntaktischer Aspekt der Transformationssprache SViTL ist es, Ausgangsdatenstrukturen mit den zielseitig erzeugten Elementen in Beziehung zu setzen. SViTL definiert hierfür das Konstrukt AnchorDefinition, das zu einem zielseitigen Element (subject) eine Reihe von Verweisen auf die Quellseite enthält. Diese werden durch eine Menge von DefinitionFragments dargestellt, die auf jeweils ein Quellelement (target) verweisen. Ferner ist die Angabe eines Attributs dieses quellseitigen Elements möglich, dessen Wert anstelle der referenzierten Entität zur Verankerung bei der Übersetzung genutzt werden soll. Auch die bereits erwähnte Beschriftung zur Unterscheidung verschiedener referenzierender Zielelemente ist Teil eines Ankerfragments. Durch Ankerfragmente dürfen nur nicht-negative quellseitige Elemente referenziert werden, die zudem aus demselben SViTL-Paar der Ankerdefinition stammen müssen.

```
context DefinitionFragment
inv properTarget : self.target.pair =
        self.definition.rhselement.pair
inv nonNegativeTarget : not self.target.negative
```

Eine Ankerdefinition beschreibt informell eine eindeutige Zuordnung eines zielseitigen zu einem quellseitigen Element. Nicht sinnvoll ist daher die mehrfache Angabe mehrerer Ankerdefinitionen innerhalb eines SViTL-Paars, und wird daher durch die folgenden Wohlgeformtheitsbedingungen ausgeschlossen. Zunächst werden Ankerdefinitionen auf Basis der Eigenschaften target, label und attribute ihrer Fragmente verglichen, die zusammen betrachtet eine Definition eindeutig bestimmen müssen. Der dargestellte Ausdruck schließt somit zwei Ankerdefinition dann aus, falls deren Fragmentmengen wechselseitig in der jeweils anderen enthalten sind. Hierbei wird die Gleichheit der *Eigenschaften* der jeweiligen Fragmente durch die Hilfsfunktion equals geprüft. Zusätzlich zu dieser globalen Eindeutigkeitsbedingung sollte eine Ankerdefinition sinnvollerweise keine zwei identischen Fragmente aufweisen.

```
context DefinitionFragment:
def equals( other : DefinitionFragment ) : Boolean =
```

```
        self.target = other.target and self.label = other.label and
          self.attribute = other.attribute

context AnchorDefinition:
inv globalUniqueness :
        let otherDefs : Collection(AnchorDefinition) =
          self.elem.pair.rhs→collect( r | r.aDefs )
            →reject( ad | ad = self) in
          -- andere Ankerdefinitionen --
        not otherDefs→exists( o |
          self.fragments→forAll( f1 | o.fragments→
            exists( f2 | f1.equals(f2) ) )
          and o.fragments→forAll( f1 | self.fragments→
            exists( f2 | f1.equals(f2) ) ) )

inv localUniqueness :
        self.fragments→forAll( f1 | self.fragments→
          exists( f2 | f1.equals(f2) ) implies f1 = f2 )
```

6.3.2 Varianten der Verankerungsspezifikation

Mit Abschluss der syntaktischen Beschreibung wird im Folgenden die Interpretation von *Ankerdefinitionen* näher beschrieben. Hierfür stellt Abbildung 6.7 verschiedene Spezifikationsvarianten gegenüber, deren Auswirkung auf die Transformationsausführung jeweils aufgezeigt wird. Die Beispiele beziehen sich jeweils auf eine Quellseite mit zwei zu suchenden Elementen deren Belegungen nicht wechselseitig – bspw. durch Verbindungsprüfungen – eingeschränkt sind. Zielseitig wird jeweils eine Knotenvariable erstellt. In der zu verarbeitenden Datenquelle sollen zu jedem dieser Musterelemente zwei Belegungen gefunden werden können. Insgesamt existieren also vier Auffindungsstellen des quellseitigen Musters aufgrund der freien Kombinierbarkeit der Belegungen. Die Abbildung stellt neben der konkreten SViTL-Syntax ihr abstraktes Gegenstück dar, um den Bezug zur Sprachdefinition zu erleichtern. Ferner ist die Verarbeitung der Transformationsvorschrift anhand einer Laufzeitdatenstruktur festgehalten. Dabei wird die Ausgangsstruktur, die erzeugte Zielstruktur sowie die Auswirkung des *Verklebens* identisch verankerter Elemente beschrieben.

Eindeutige Verankerung

Der erste Fall der Abbildung zeigt eine einzelne Verankerung des zielseitigen Elements. Hierfür werden insgesamt für jedes der vier Auffindungsstellen Knotenva-

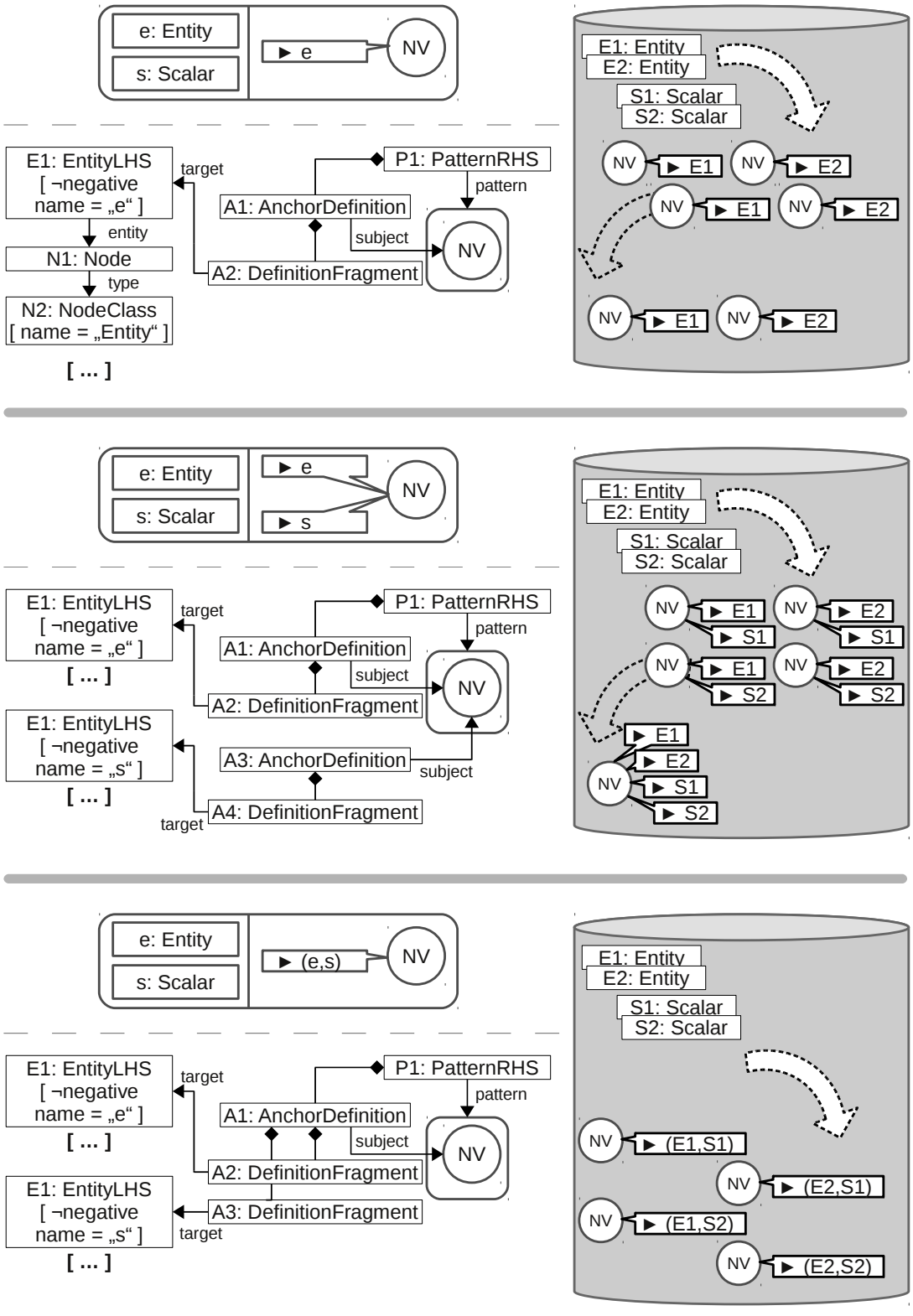


Abbildung 6.7: Verschiedene Varianten der Ankerdefinition

riablen erzeugt, die jeweils zur Belegung der Variablen `e` verankert werden. Nach erfolgter Verklebung fallen die mit `E1` bzw. `E2` beschrifteten Elemente zusammen. Das Ergebnis der Transformationsausführung besagt also eine Erstellung der Knotenvariablen *für jede Belegung von* `e`.

Mehrfache Verankerung

In der Mitte von Abbildung 6.7 ist die Möglichkeit dargestellt, ein einzelnes zielseitiges Element mehrfach zu verankern. Hierzu werden unterschiedliche Ankerdefinitionen verwendet, die auf ein einzelnes Subjekt verweisen. Bei der Transformationsausführung werden mehrere Verankerungen pro zielseitigem Element erstellt, die sich aus der jeweiligen Variablenbelegung der unterschiedlichen Auffindungsstellen ergeben. Zu beachten ist hier der Effekt der Verklebung: Da *alle* Auffindungsstellen mit jeweils zwei weiteren Stellen überlappen, verursacht die Verklebung von Entitäten mit gemeinsamen Verankerungen ein *Zusammenfallen* der dargestellten Struktur auf ein *einzelnes* Element, das mit allen anzutreffenden Ankern versehen ist. Diese Modellierungsvariante erstellt somit *ein* zielseitiges Element für alle *verträglichen* quellseitigen Belegungen – wobei *Verträglichkeit* die Übereinstimmung mit Belegungsbeschränkungen ausdrückt. In der Praxis entstehen diese Beschränkungen bspw. durch Verbindungsprüfungen zwischen den dargestellten Variablen. Hierbei würde jeweils ein zielseitiges Element pro verbundenem Element entstehen.

Verankerung mit mehreren Fragmenten

Eine dritte Möglichkeit zur Ankerdefinition zeigt der letzte Abschnitt von Abbildung 6.7. Hierbei werden für eine *einzelne* Ankerdefinition *zwei* Fragmente definiert, die auf unterschiedliche Variablen verweisen. Das Ergebnis der Transformation wird nicht durch die Verklebung von Elementen beeinflusst, da diese nur über alle Fragmente identische Verankerungen berücksichtigt. Somit wird ein zielseitiges Element *pro Kombination der referenzierten Variablen* erstellt, was hier vier unterschiedliche Paarungen aus vier Auffindungsstellen hervorbringt.

6.4 Übersetzerausführung

Dieser Abschnitt behandelt die Ausführung einer SViTL-Transformation, wohingegen im vorangegangenen Abschnitt die Syntax der Sprachkonstrukte und eine grobe semantische Zuordnung vorgestellt worden ist. Der Schwerpunkt liegt dabei auf der Verklebung der jeweils erstellten Elemente, da die vorgelagerten Schritte wie Mus-

tersuche und das zielseitige Erzeugen von Elementen bereits vorab erläutert worden sind. Diese Punkte werden daher nur kurz beschrieben.

6.4.1 Ermittlung von Auffindungsstellen

Der Ermittlung von Auffindungsstellen erfolgt in SViTL unter Nutzung derselben Maschinerie wie bei der Auswertung von DRAGULA-Mustern. Tatsächlich wird zur Auswertung eines SViTL-Paares eine *Übersetzung* nach DRAGULA vorgenommen. Dementsprechend kann die im vorangegangenen Kapitel vorgestellte Ausführungsmaschine direkt wiederverwendet werden.

6.4.2 Erzeugung zielseitiger Elemente

Auch der zweite Aspekt dieses Abarbeitungsschritts, die zielseitige Erstellung von DRAGULA-Elementen, kann auf die bestehende Technologie zurückgeführt werden. Dies erfolgt konzeptionell pro ermittelter Auffindungsstelle getrennt, kann jedoch implementierungsseitig auch zwecks Optimierung mit dem nachfolgenden Schritt, der Verklebung erstellter Elemente, kombiniert werden. Die hier gegebene Erläuterung geht vom ersteren und leichter darstellbaren Fall aus.

Grundsätzlich werden alle zielseitig dargestellten Elemente auf Erstellungsoperatoren in DRAGULA zurückgeführt, die bei ihrer Ausführung eine entsprechende Graphstruktur erzeugen. Eine Ausnahme bildet dabei der Einsatz von *abstrakten* Typen, von denen mittels DRAGULA-Sprachkonstrukten keine Instanzen erzeugt werden können. In SViTL ist dies jedoch aufgrund der nachgelagerten Verklebung von Elementen, die eingeschränkt auch über verschiedene Typen erfolgen kann, sinnvoll durchführbar. Implementierungsseitig ist somit ein synthetischer konkreter Typ einzufügen um die Verwendung eines abstrakten Typen als DRAGULA-Element eines SViTL-Paars zu ermöglichen. Die im weiteren Verlauf gezeigten Beispiele vereinfachen die Darstellung jedoch dahingehend, dass stets der zielseitig angegebene Typ instanziiert wird, auch wenn dies technisch eigentlich nicht zulässig ist.

6.4.3 Verankerung

Die Verankerung der erstellten Elemente zu quellseitig aufgefundenen erfolgt durch eine Hilfsstruktur die in Abbildung 6.8 syntaktisch definiert wird. Ein Anchor besteht, analog zu seiner Definition in Abbildung 6.6, aus einer nicht-leeren Menge von Fragmenten (AnchorFragments), die jeweils optional beschriftet sein können. Mittels der zwei dargestellten konkreten Typen kann einerseits ein skalarer *Wert* referenziert werden

(ValueFragment) oder auf ein Graphelement verwiesen werden (EntityFragment in Verbindung mit der Assoziation target). Diese *Nachverfolgungsinformationen* werden ebenfalls durch Erstellungsoperatoren aus DRAGULA erzeugt.

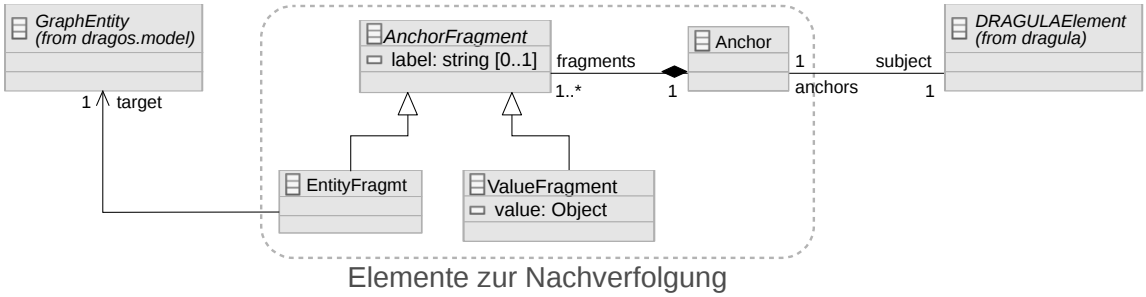


Abbildung 6.8: Syntaktische Definition von Verankerungen

6.4.4 Verklebung

Die nachgelagerte Verklebung der erstellten Elemente dient dazu die bislang einzeln erzeugten DRAGULA-Fragmente zu einer Gesamtspezifikation zu kombinieren. Hierbei werden Behälterinformationen zunächst vernachlässigt, da diese erst im folgenden Abschnitt eingeführt werden. Im Gegensatz zu verschiedenen anderen Transformationsansätzen wie beispielsweise Tripel-Graph-Grammatiken, kann die Gesamtübersetzung in jeweils einzelne abgeschlossene Teilschritte zerlegt und getrennt ausgeführt werden. Konzeptionell wird erst nach Erzeugung aller Zielstrukturen eine Kombination überlappender Strukturen vorgenommen. Somit greifen die einzelnen Transformationsschritte nicht auf Ergebnisse anderer Schritte zurück, und es entstehen keine Abhängigkeiten im Sinne der Ausführbarkeit von Transformationsregeln.

Zulässigkeit der Verklebung

Die Verklebung zweier erzeugter Elemente erfolgt auf Basis ihrer gemeinsamen Verankerung. Für zwei zu verklebende Elemente genügt das Vorhandensein *eines* Paars von Ankern, von denen *alle* Fragmente identisch beschriftet sind[3] und die identische Skalarwerte bzw. quellseitige Elemente referenzieren. Die folgenden Ausdrücke formulieren diese Beschreibung mit OCL-Hilfsfunktionen. Wie bereits bei der Ankerdefinition wird auch hier eine wechselseitige Enthaltenseinsbeziehung äquivalenter

[3]Unbeschriftete Anker sind nur zu anderen unbeschrifteten vergleichbar und dienen nicht etwa als *Wildchar*.

Ankerfragmente gefordert. Somit sind zwei zielseitige Elemente mit den Ankern a1 bzw. a2 genau dann verklebbar, wenn zu jedem *Ankerfragment* in a1 ein äquivalentes in a2 existiert und zu jedem Fragment in a2 ebenfalls ein äquivalentes in a1.

```
context AnchorFragment
def equals (other : AnchorFragment) : boolean =
      (self.label = other.label or (self.label.oclIsUndefined() and
        other.label.oclIsUndefined())) and
      if self.oclOfType(EntityFragment) and
          other.oclOfType(EntityFragment) then
        self.oclAsType(EntityFragment).target =
          other.oclAsType(EntityFragment).target
      else if self.oclOfType(ValueFragment) and
              other.oclOfType(ValueFragment) then
        self.oclAsType(ValueFragment).label =
          other.oclAsType(ValueFragment).label
      else
        false
      endif endif

context DRAGULAElement
def glueable (other : DRAGULAElement) : boolean =
      self.anchors.exists( a1 | other.anchors→exists( a2 |
        a1.fragments→forAll( f1 | a2.fragments→exists( f2 |
          f1.equals( f2 ) ) ) and
        a2.fragments→forAll( f1 | a1.fragments→exists( f2 |
          f1.equals( f2 ) ) ) ) )
```

Beispielhaft werden gültige Paare von DRAGULA-Elementen, bei denen eine Verklebung zulässig ist, in Abbildung 6.9 gezeigt. Die linke Seite stellt dabei den einfachen Fall vor, bei dem zwei Elemente mit identischen Ankermengen kombiniert werden können. In der Mitte wird gezeigt, dass auch eine Übereinstimmung einer Teilmenge der vorhandenen Anker genügt, eine Verklebung also möglich ist obwohl lediglich das untere Element die Verankerung auf E2→alt beinhaltet. Zu diesem Beispiel ist auch die abstrakte Syntaxnotation aufgezeigt, um die sonst angewendete konkrete Darstellung mit obigen Definitionen in Bezug setzen zu können. Die rechte Seite zeigt schließlich ein *Gegenbeispiel*: Obwohl beide Elemente die gleichen Anker*fragmente* umfassen, verhindert die jeweilige Strukturierung – im oberen Fall als Tupel, im unteren dagegen als unterschiedliche Anker – die Verklebung.

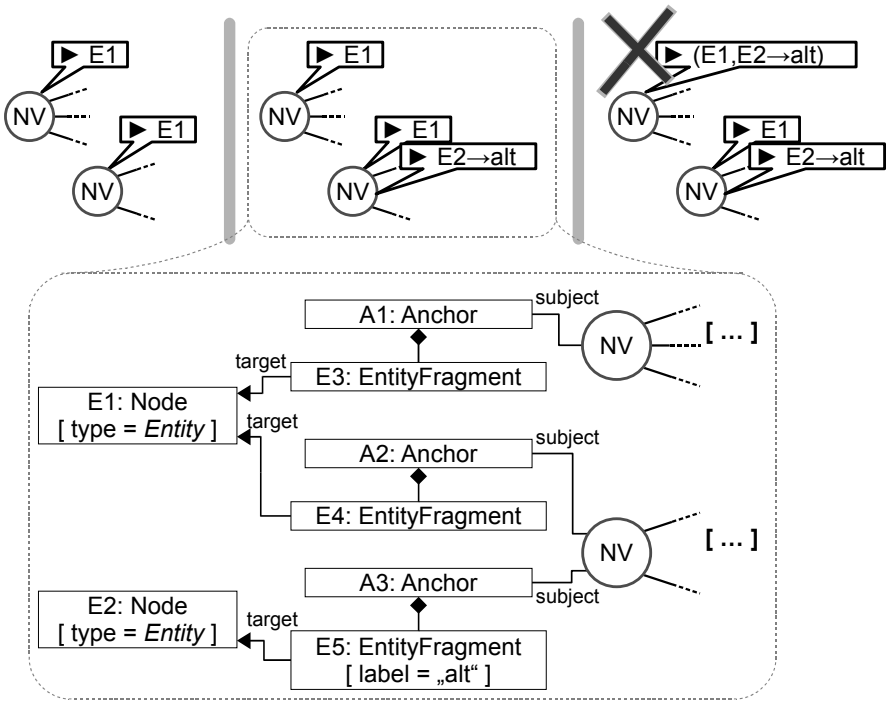


Abbildung 6.9: Vergleich verklebbarer und nicht-verklebbarer Elemente

Ablauf der Verklebung

Die Verklebung von Elementen ergibt stets ein *einzelnes* resultierendes Element, dessen Eigenschaften gemäß der im Weiteren vorgestellten Vorschriften ermittelt werden. Hierbei sind dessen Typisierung, Attributierung und Inzidenzeigenschaften zu berücksichtigen. *Anker* von verklebten Elementen werden mengenwertig vereint, so dass das resultierende Element alle Verankerungen der Ausgangselemente enthält.

Die Verklebung von Elementen erfolgt *iterativ*, wodurch auch bereits verklebte Elemente herangezogen werden können. Die Auswirkung hiervon ist in Abbildung 6.10 gezeigt, wo drei Elemente zu einem einzigen zusammengeführt werden. Das obere und das untere Element auf der linken Seite sind zwar *nicht* gemäß obiger Definition verklebbar, dies kann jedoch über ein Zwischenergebnis mit jeweils gemeinsamen Verankerungen erfolgen. Möglich wäre hier auch zuerst die Verklebung der unteren beiden Elemente vorzunehmen was zum gleichen Endergebnis führen würde. Diese formale Eigenschaft, über verschiedene Verklebungsschritte ein eindeutiges Endergebnis zu erzeugen, wird in Abschnitt 6.7 zum Thema Konfluenz weiter diskutiert. Festzuhalten ist an dieser Stelle jedoch das die Verklebung von Elementen ohne Beschränkung der Allgemeinheit jeweils in *Paaren* erfolgen kann. Die weitere Erläuterung der Verklebungssemantik greift auf diese Eigenschaft zurück.

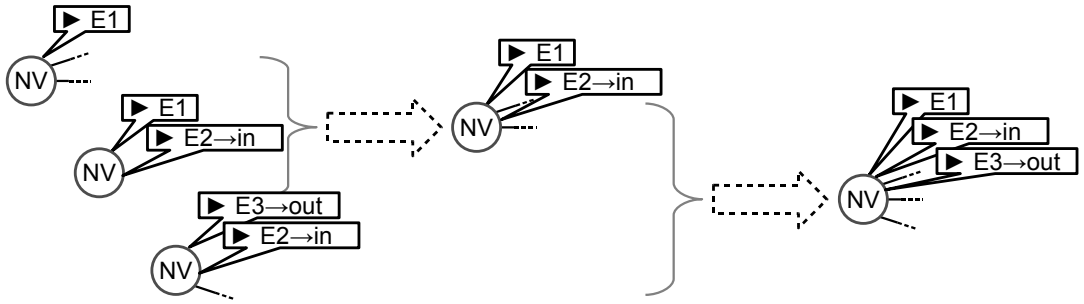


Abbildung 6.10: Mehrstufige Verklebung von Verankerungen

Behandlung von Typinformationen

Sollten zwei zu verklebende DRAGULA-Elemente unterschiedliche Aktualtypen aufweisen, so verlangt SViTL, dass diese in einer Hierarchierelation stehen. Der Typ eines Elements muss somit direkter oder indirekter Ober- bzw. Untertyp des anderen sein. Bei Erfüllung dieser Bedingung wird der speziellere der beiden Typen für das resultierende Element verwendet. Liegt bei einem der Elemente ein *synthetischer* Typ vor, da bspw. in der zielseitigen Spezifikation eines SViTL-Paars ein abstrakter Typ verwendet worden ist, wird dieser zur Ermittlung der Hierarchieeigenschaft wie der abstrakte Typ behandelt. Sind die jeweiligen Typen nicht in einer hierarchischen Relation, wird ein Laufzeitfehler ausgelöst, da kein eindeutiges Ergebnis erzeugt werden kann.

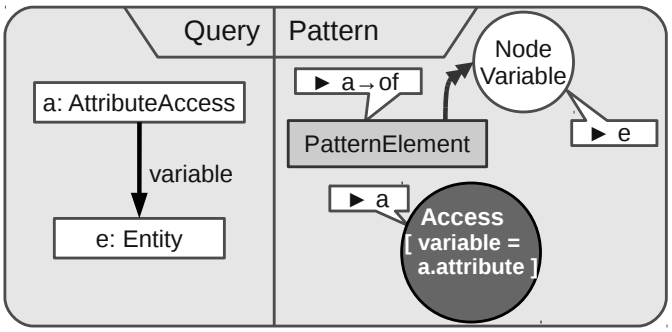
Der Zweck dieser *Typverfeinerung* bei der Verklebeoperation besteht darin, Übersetzungsmuster allgemein darstellen zu können, wie dies bspw. in Abbildung 6.4b auf Seite 234 für den Attributausdruck Expr erfolgt ist. Dieses SViTL-Paar ist somit für alle möglichen Attributausdrücke anwendbar. Weitere Paare beschreiben die Abbildung spezieller Ausdrücke, bspw. zum Zugriff auf Attributwerte, wie in Abbildung 6.11a dargestellt. Durch Verwendung identischer Ankerdefinitionen werden die in beiden Paaren erstellten Ausdruckselemente miteinander verklebt, wobei der speziellere (insbes. nicht abstrakte) Typ AttributeAccess das Endergebnis stellt. Diese Vorgehensweise wird in Abbildung 6.11b aufgezeigt[4].

Behandlung von Attributwerten

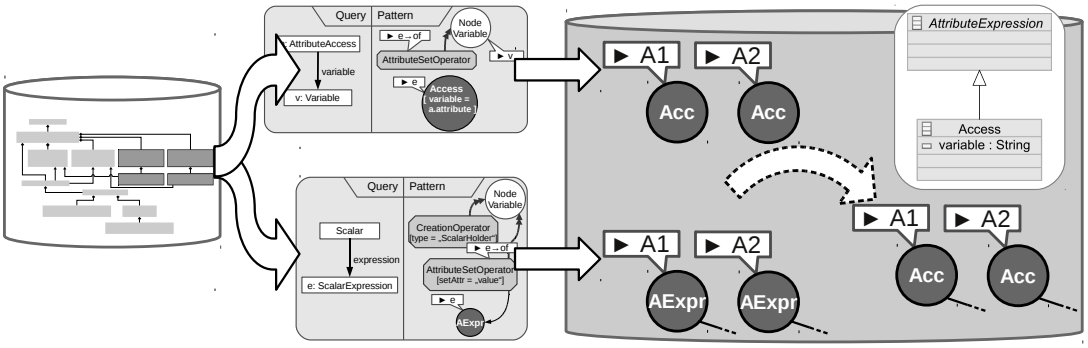
Bei der Verklebung zweier zielseitiger Elemente werden deren jeweilige Attributwerte auf Basis folgender Maßgaben zusammengeführt:

- *Identische* Attributwerte in zwei zu verklebenden Elementen werden als redun-

[4]Die inhaltliche Erläuterung der dargestellten SViTL-Paare erfolgt in Abschnitt 6.6.



(a) SViTL-Paar zur Abbildung von Attributzugriffen



(b) Verklebung unterschiedlich getypter Elemente

Abbildung 6.11: Typbehandlung bei der Verarbeitung von Attributzugriffen

dante Information aufgefasst und dementsprechend nur einmal in das resultierende verklebte Element aufgenommen.

- Liegt ein Attributwert lediglich in *einem* der zu verklebenden Elemente vor, wird dieser in das Ergebniselement aufgenommen. Somit dominiert eine vorliegende Attributierung über nicht gesetzte Werte.

- Liegen zwei *unterschiedliche* Attributwerte in den zu verklebenden Elementen vor, entscheidet der jeweilige Typ der beinhaltenden Elemente über das weitere Vorgehen. Liegt eine Hierarchierelation gemäß obiger Definition vor, so dominiert der Wert des zu verklebenden Elements mit dem *spezielleren* Typen. Sind diese jedoch nicht in einer Typhierarchie geordnet, wird wiederum ein Laufzeitfehler ausgelöst.

Auf diese Weise ist sichergestellt, dass ein im resultierenden Element eingetragener Attributwert seiner Attributdeklaration entspricht. Dies wird durch obige Behandlung von Typinformationen erreicht, da bei der Verklebung lediglich eine Verfeinerung

hin zu spezielleren Typen möglich ist. Diese definieren jedoch *mindestens* auch die Attribute der allgemeineren Typen.

Behandlung adjazenter Elemente

Die Verklebeoperation wirkt sich grundsätzlich auch auf adjazente Elemente der zu verklebenden Elemente aus, wobei zwischen verbindenden Elementen (i.d.R. Kantenelemente) und begrenzenden Elementen (Knoten- und Graphelemente) unterschieden wird. Abbildung 6.12 zeigt die verschiedenen Varianten im Überblick.

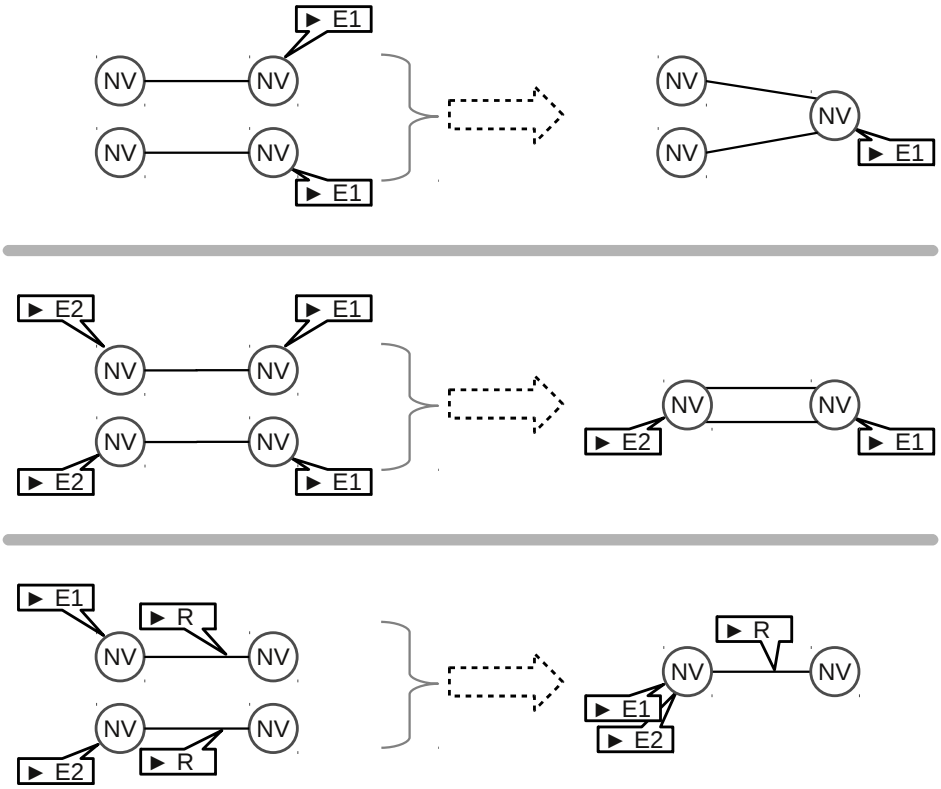


Abbildung 6.12: Verklebung von Verbindungsstrukturen

In den ersten beiden Fällen beiden Fällen erfolgt eine Verklebung aufgrund gemeinsamer Verankerungen der begrenzenden Elemente. Im ersten Fall wird ein begrenzender Knoten mit einem weiteren entsprechenden verankerten Knoten verklebt. Beide adjazenten Beziehungen bleiben zu den jeweiligen anderen begrenzenden, hier jedoch unverankerten Knoten, bestehen.

Der zweite dargestellte Fall zeigt die Verankerung sowohl der Quell- als auch der Zielseite einer Beziehung. Hierbei bleiben ebenfalls beide unverankerte Beziehungen

bestehen, was zu zwei parallelen Kanten im resultierenden Graph führt. Sollte dieses Verhalten unerwünscht sein kann, wie im Folgenden erläutert, eine Verankerung der Beziehungen vorgenommen werden.

Der dritte Fall demonstriert schließlich die Verklebung entsprechend verankerter Beziehungen. Dabei werden zusätzlich verankerte und unverankerte Quell- und Zielknoten miteinander verklebt, was zum dargestellten Ergebnisgraphen führt. Aufgrund der Zusammenführung der unterschiedlich verankerten Knoten können zudem weitere Elemente des Ergebnisgraphen miteinander verklebt werden – hier solche mit den Verankerung `E1` und `E2`, die ansonsten keine gemeinsame Verankerung besitzen. Bei der Spezifikation verankerter Beziehungen ist dies ggf. zu berücksichtigen.

Die Umlenkung von Verbindungsstrukturen bei der Verklebung adjazenter Elemente kann ggf. zu Verletzungen des Graphschemas führen, etwa in dem das verklebte Element mehr Kanten eines Typs, als durch eine obere Kardinalitätsangabe zugelassen wird, erhält. In diesem Fall wird wiederum ein Laufzeitfehler ausgelöst, da eine mögliche Behebung dieses Fehlers nicht automatisch gewählt werden kann. Neben der vollständigen Entfernung einer oder mehrerer Kanten wäre auch deren Verklebung unter Beibehaltung ihrer Attributwerte und jeweils inzidenter Verbindungsstrukturen denkbar.

6.5 Spezifikation von Enthaltenseinsbeziehungen

Im bisherigen Verlauf dieses Kapitels sind SViTL-Konstrukte für einfache Sprachübersetzer vorgestellt worden. Verglichen mit Abbildung 6.2 auf Seite 229 sind dabei Enthaltenseinsinformationen außen vor gelassen worden. Da diese Informationen jedoch wesentlichen Einfluss auf die Bedeutung der erstellten DRAGULA-Struktur besitzen, wird die Übersetzerspezifikation im Folgenden entsprechend ergänzt. SViTL bietet hierfür einen eingebauten Mechanismus, der die Spezifikation solcher Schachtelungsstrukturen erleichtert.

6.5.1 Motivation & Ansatz

Gängige Transformationssprachen, von denen einige Vertreter in Abschnitt 6.9 betrachtet werden, beziehen Enthaltenseinsinformationen nicht explizit in die Transformationssemantik ein, sondern behandeln diese als rein *syntaktische* Struktur. Hierbei entstehen zweierlei hauptsächliche Problematiken:

- Enthaltenseinsinformationen müssen einerseits *explizit* und andererseits für *jeden* Transformationsaspekt gesondert spezifiziert werden. Dies ist zwar prinzipiell möglich, beinhaltet jedoch zusätzlichen Spezifikationsaufwand. Ferner be-

steht das Risiko einer inkonsistenten Behälterzuordnung durch den Entwickler und damit einer Fehlerquelle.

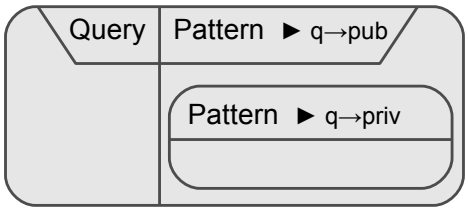
- Der zielseitige Behälter des Transformationsergebnisses ist oftmals nicht trivial durch eine Transformationsregel zu ermitteln. So kann im eingangs dargestellten Beispiel für einen quellseitigen Attributausdruck nicht ohne Weiteres entschieden werden, ob die zugehörigen DRAGULA-Elemente dem äußeren oder dem inneren Muster zuzuordnen wären. Entsprechendes Kontextwissen müsste in Form zusätzlicher SViTL-Paare oder durch Pfadausdrücke ergänzt werden. Auf diese Weise entsteht zusätzlicher Spezifikationsaufwand falls Enthaltenseinsbeziehungen rein syntaktisch aufgefasst werden.

Als Konsequenz aus diesen beiden Beobachtungen stellt SViTL einen Mechanismus bereit, der Behälterzuordnungen *implizit* vornimmt und der zugleich *explizit* gesteuert bzw. überschrieben werden kann. Somit bleibt die explizite Spezifikationsweise möglich, wobei in zahlreichen Fällen darauf verzichtet werden kann. Hierzu setzt SViTL eine Reihe von *Heuristiken* ein, die gemäß der üblichen Zuordnung von Muster- bzw. Regelelementen zu ihren jeweiligen Behältern ausgelegt sind.

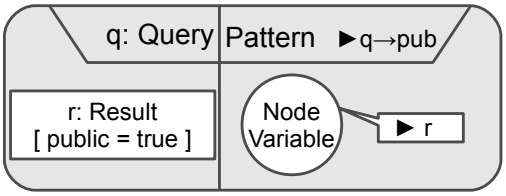
6.5.2 Explizite Modellierung von Enthaltenseinsinformationen

Da als Eingabedaten für SViTL-Transformationen rein syntaktische Informationen bereitstehen, kann die Zuordnung passender Behälter nicht vollständig automatisiert erfolgen. SViTL bietet daher die Möglichkeit, Behälterzuordnungen explizit mittels *Ankerspezifikationen* vorzunehmen. Bleibt dies aus, erfolgt die Zuordnung der enthaltenen Elemente zu Behältern heuristikbasiert. Simultan durch ein SViTL-Paar erzeugte Elemente können dabei letztendlich auch unterschiedlichen Behältern zugeordnet werden.

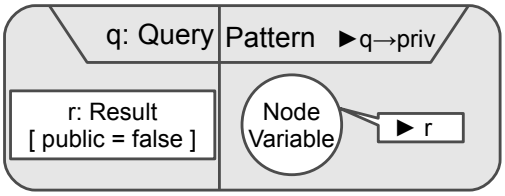
Im Detail behandeln die in Abbildung 6.13 dargestellten SViTL-Paare drei Anwendungsfälle der Behälterspezifikation: Zunächst wird in Abbildung 6.13a eine Struktur aus zwei geschachtelten DRAGULA-Mustern erzeugt, die jeweils auf die zugeordnete EMAA-Anfrage `q` verankert werden. Hierbei wird die Beschriftung `pub` für das umgebende, `priv` für das innere Muster verwendet. Diese dienen jeweils zur Aufnahme öffentlicher bzw. privater Variablen. In zwei weiteren Paaren erfolgt die Zuordnung von Variablen zu den jeweiligen Mustern. Öffentliche Variablen werden gemäß Abbildung 6.13b in einem Muster abgelegt, das wie oben mit der Beschriftung `pub` versehen ist. Analog dazu verweist Abbildung 6.13c auf die Beschriftung `priv`. Auf Basis dieser SViTL-Paare können die erzeugten DRAGULA-Variablen getrennt von der tatsächlichen Musterhierarchie einem Behälter zugeordnet werden.



(a) Hierarchie zwischen den verschiedenen DRAGULA-Mustern



(b) Öffentliche Variablen



(c) Private Variablen

Abbildung 6.13: Explizite Behälterspezifikation

Syntaktisch eingeschränkt werden SViTL-Spezifikationen dahingehend, dass die Angabe einer Ankerdefinition lediglich für den Wurzelbehälter der Zielseite optional ist. Geschachtelte Behälter müssen dagegen stets verankert sein.

```
context RHSElement
inv nestedContainerIsAnchored :
      self.pair.rhs→exists(r | self.content.parent = r.content)
        implies not self.aDefs→isEmpty()
        -- content verweist auf self.pattern bzw. self.rule,
        -- abhaengig vom Aktualtyp
```

6.5.3 Heuristikbasierte Behälterzuordnung

Zusätzlich zu oben behandelten Sprachkonstrukten zur *expliziten* Behälterspezifikation wendet SViTL verschiedene DRAGULA-spezifische Heuristiken an, um die Modellierung der Enthaltenseinsbeziehungen zu vereinfachen. Diese Heuristiken greifen in verschiedene Teile der bisher vorgestellten Sprachbeschreibung. So sind zunächst implizite Enthaltenseinsinformationen zu ermitteln, die sich aus dem Zusammenhang simultan, d.h. durch einen einzelnen Transformationsschritt erzeugter DRAGULA-Elemente ergeben. Des Weiteren ist eine auf Basis der gewonnen Informationen basierende *Zuordnung* von DRAGULA-Elementen zu Behältern vorzunehmen.

Ermittlung von Enthaltenseinsinformationen

Um Enthaltenseinsinformationen von explizit zugewiesenen Elementen zu den übrigen propagieren zu können, sind verschiedene Ansätze denkbar. Naheliegenderweise können die zielseitig erzeugten *Verbindungsstrukturen* herangezogen werden, etwa zwischen einer Typbedingung und der dadurch eingeschränkten Variablen. Die Typbedingung würde demnach aufgrund dieser Verbindung dem gleichen Behälter zugeordnet wie die zugeordnete Variable. Dieser Ansatz ist jedoch nur sehr bedingt geeignet, da Verbindungen Behältergrenzen passieren können (vgl. Abbildung 6.3 auf Seite 230). Eine implizite Propagation von Behälterzuordnungen entlang dieser übergreifenden Beziehungen könnte daher unerwartete Ergebnisse bewirken.

Anstelle auf Verbindungsstrukturen zurückzugreifen, berücksichtigt SViTL jeweils *simultan*, d.h. durch Anwendung eines SViTL-Paares auf eine einzelne Anwendungsstelle, erzeugte Elemente. Unabhängig von der erzeugten Verbindungsstruktur werden simultan erzeugte Elemente in eine *Nachbarschaft* zusammengefasst. Diese umfasst sowohl die verankerten als auch nicht-verankerten DRAGULA-Elemente, geht jedoch nicht über Behältergrenzen hinaus. Ein *verankerter* Behälter bildet somit eine Nachbarschaft mit seinen enthaltenden Elementen, jedoch nicht mit seinem eigenen Behälter. *Unverankerte* Behälter werden nicht in die Nachbarschaft ihrer enthaltenen Elemente aufgenommen da sie lediglich als beliebige Platzhalter eines SViTL-Paares dienen, wie etwa Abbildung 6.4 auf Seite 234 zeigt.

Abbildung 6.14 zeigt beispielhaft Nachbarschaften, die durch Anwendung der in Abbildung 6.13 dargestellten Paare entstehen[5]. Durch Anwendung des Paares aus Abbildung 6.13a entsteht die links dargestellte Musterhierarchie. Die erzeugten Muster bilden aufgrund obiger Definition keine gemeinsame Nachbarschaft, die somit entstehenden einelementigen Nachbarschaften können bei der weiteren Verarbeitung ignoriert werden. Die Ergebnisse der expliziten Zuordnung von Variablen zu Mustern gemäß 6.13b und 6.13c wird in den folgenden beiden Ausschnitten dargestellt. Hier bilden die verankerten Muster jeweils eine Nachbarschaft mit ihren enthaltenen Variablen. Als vierter Ausschnitt wird das Ergebnis aus Abbildung 6.4a dargestellt, wobei eine Nachbarschaft bestehend aus der Variablen und ihrer Bedingung entsteht. Gemäß obiger Definition wird das unverankerte umgebende Muster des letzten SViTL-Paares ignoriert.

Nachbarschaften bilden die Grundlage zur Propagation von Enthaltenseinsinformationen, die im weiteren Verlauf dieses Abschnitts behandelt wird. Aufgrund der hier getroffenen Einschränkungen wird eine Nachbarschaft höchstens ein Behälterelement, also ein DRAGULA-Muster oder eine -Regel enthalten. Zusätzlich werden direkte Enthaltenseinsbeziehungen zwischen simultan erzeugten und veranker-

[5] Dargestellt ist lediglich das Ergebnis jeweils *einer* quellseitigen Auffindungsstelle.

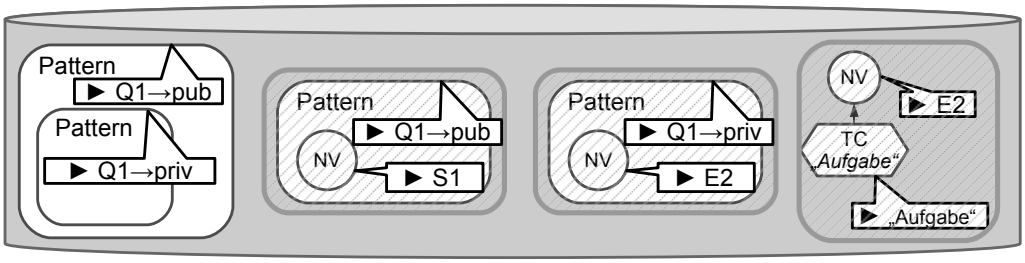


Abbildung 6.14: Nachbarschaften durch Anwendung der Paare aus Abbildung 6.13

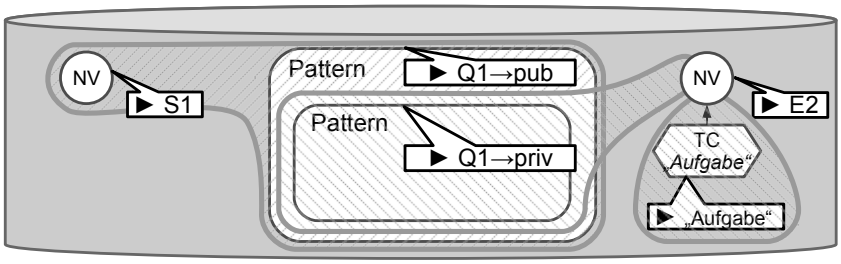


Abbildung 6.15: Verklebung der Nachbarschaften aus Abbildung 6.14

ten DRAGUA-Behältern festgehalten. Diese Informationen zusammengenommen erlauben eine semi-automatische Zuordnung der unverankerten Elemente zu geeigneten Behältern.

Verklebung unter Berücksichtigung von Enthaltenseinsinformationen

Die Verklebung von DRAGULA-Elementen, die bereits in Abschnitt 6.4.4 beschrieben worden ist, behandelt Nachbarschaftszugehörigkeiten ebenso wie reguläre Verbindungstrukturen. Werden zwei in jeweils einer Nachbarschaft befindliche Elemente miteinander verklebt, so ist das resultierende Element *zwei* Nachbarschaften zugeordnet. Dabei können Nachbarschaften mit identischen Elementen zu einer einzelnen reduziert werden, nicht jedoch Nachbarschaften mit Unter- bzw. Obermengen einer anderen. Das Ergebnis der Verklebung der Teilstrukturen aus Abbildung 6.14 wird in Abbildung 6.15 dargestellt. Die zu E2 verankerte Variable ist somit zwei Nachbarschaften zugeordnet, von denen eine zugleich einen Behälter umfasst. Diese Zusammenhänge dienen im Folgenden zur *Propagation* der Behälterinformationen.

Implizit propagierte Enthaltenseinsinformationen dürfen der explizit vorgenommenen Zuordnung nicht widersprechen. Somit können zwei identisch verankerte Behälter nur dann verklebt werden, falls ihre übergeordneten und enthaltenen Behälter ebenfalls miteinander verklebt werden können, ohne widersprüchliche Hierarchien zu erzeugen. Andernfalls wird ein Laufzeitfehler ausgelöst. Ein Beispiel und Gegen-

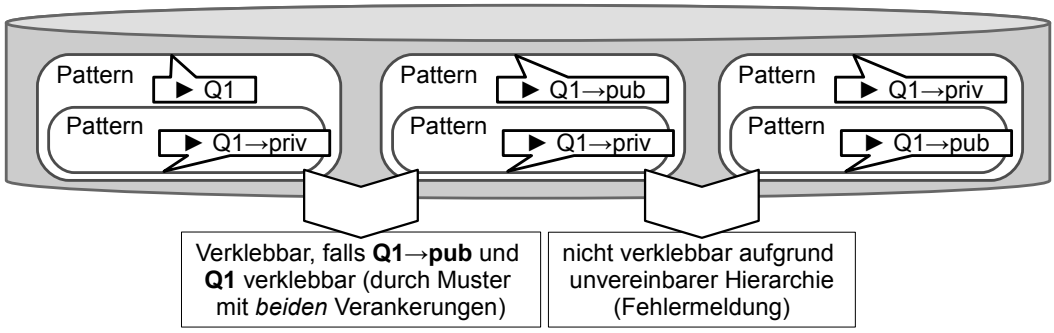


Abbildung 6.16: Zulässige und unzulässige Verklebung von Behältern

beispiel hierzu zeigt Abbildung 6.16: Die linken beiden Muster können unter der Bedingung verklebt werden, dass `Q1` und `Q1→priv` verklebbar sind; die rechten beiden Muster können nicht verklebt werden.

Nach vollständiger Übersetzung des Eingangsbeispiels aus Abbildung 6.2 auf Seite 229 ergibt sich eine Reihe von Nachbarschaften, die in Abbildung 6.17 dargestellt werden[6]. Zu Beibehaltung der ursprünglichen Anordnung der DRAGULA-Elemente sind die Nachbarschaften entsprechend verzerrt dargestellt. Auf die hier getroffene Nummerierung bezogen sind in Abbildung 6.15 die Nachbarschaften `1`, `7`, und `10` verarbeitet worden.

Heuristische Behälterzuordnung von Nachbarschaften

Behälterzuordnungen werden zunächst zwischen *Nachbarschaften* propagiert, bevor im letzten Abarbeitungsschritt schließlich die den Nachbarschaften zugehörigen Elemente ihrem jeweiligen Behälter zugewiesen werden. Der durch SViTL verfolgte Ansatz besteht darin, zunächst *zusammenhängende Nachbarschaften* zu ermitteln, die zumindest ein gemeinsames Element umfassen. Diese induzieren einen ungerichteten, knotenbeschrifteten *Nachbarschaftsgraphen*, dessen Knoten durch die Menge der Nachbarschaften gegeben sind und dessen Kanten die jeweiligen Zusammenhangsbeziehungen modellieren. Abbildung 6.18 zeigt den Nachbarschaftsgraph zugehörig zu obigem Beispiel.

Anschließend werden diejenigen Nachbarschaftsknoten ermittelt die einen verankerten DRAGULA-Behälter umfassen. Diese sind in der Abbildung durch hervorgehobene Umrandungen markiert, zusätzlich zeigt die Abbildung die Verankerungsbeschriftung des Behälters. Die Menge der insgesamt vorhandenen Behälter wird als Beschriftungsalphabet für die Knoten des Nachbarschaftsgraphen verwendet, was

[6]Nicht alle hierzu benötigten SViTL-Paare sind bereits vorgestellt worden, sondern werden erst in der Gesamtübersicht in Abschnitt 6.6 eingeführt.

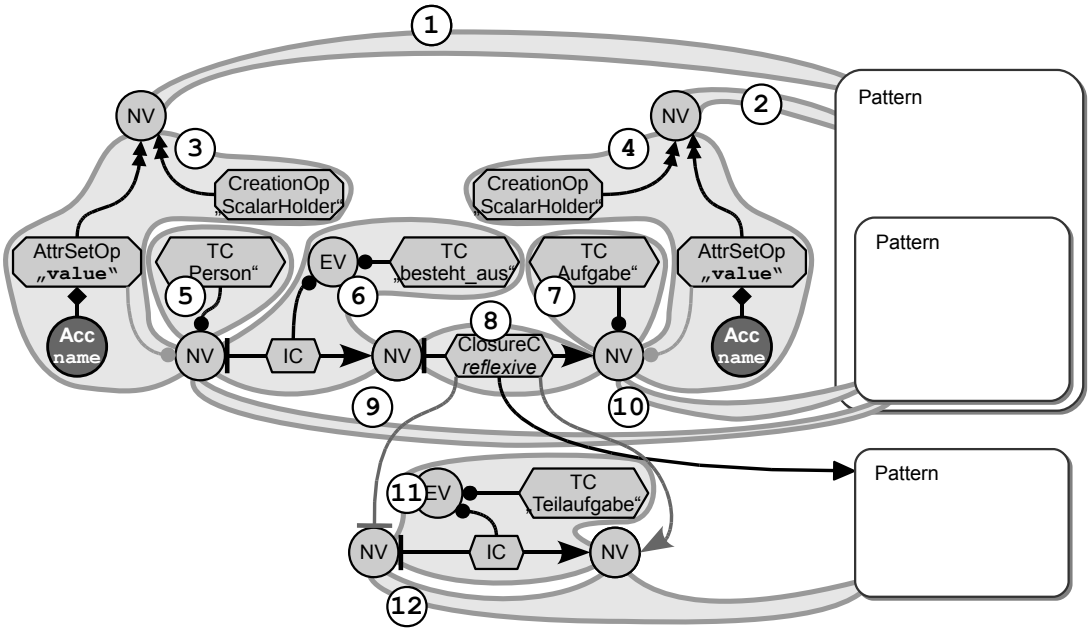


Abbildung 6.17: Aus Abbildung 6.3 resultierende Nachbarschaften

durch die farbliche Kennzeichnung ausgedrückt wird. Zunächst ist jeder hervorgehobene Knoten mit der Beschriftung seines Behälters versehen, die übrigen Knoten sind unmarkiert.

Anschließend wird für jeden unmarkierten Knoten die Menge der adjazenten und bereits beschrifteten Knoten geprüft. Falls alle diese benachbarten Knoten identisch beschriftet sind, wird diese Beschriftung auch für den bislang unbeschrifteten Knoten übernommen. Sind unterschiedliche Beschriftungen in unmittelbarer Nachbarschaft anzutreffen, dann dominiert die Beschriftung, deren zugeordneter DRAGULA-Behälter *am tiefsten* geschachtelt ist. Dies gilt falls alle auf diese Weise konkurrierenden Behälter in einer Hierarchiebeziehung stehen, andernfalls wird ein Laufzeitfehler ausgelöst. Dieser Schritt wird wiederholt solange unbeschriftete Knoten existieren.

Der Sinn dieses Verarbeitungsschritts besteht in der Zuweisung von Enthaltenseinsinformationen an Nachbarschaften. Diese Beschriftungspropagation bewirkt, dass sich Nachbarschaften ohne explizite Behälterzugehörigkeit nach graphisch nahegelegenen Festlegungen richten. Der Begriff „nahegelegen“ bezieht sich dabei bewusst nicht auf Verbindungsstrukturen in DRAGULA, sondern auf Nachbarschaften mit gemeinsamen Elementen. Im Konfliktfall dominiert in dieser heuristischen Regel der jeweils *tiefer* geschachtelte Behälter. Begründet wird dies darin, dass tiefer geschachtelte Teilstrukturen in DRAGULA eher explizit auf Elemente höherer Behälter zugreifen als umgekehrt. Zusammenhängende Strukturen sind daher eher dem untergeordneten

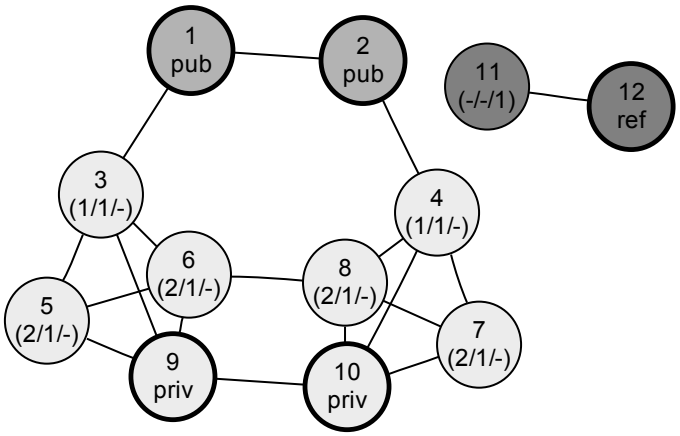


Abbildung 6.18: Aus Abbildung 6.17 gegebener Nachbarschaftsgraph und Behälterinformationen

Behälter zuzuordnen.

In Abbildung 6.18 werden alle ursprünglich unmarkierten Knoten des linken zusammenhängenden Teilgraphen durch entsprechende Beschriftung dem inneren Muster zugeordnet. Die Textbeschriftung der Knoten zeigt zudem die jeweilige minimale Entfernung zu den drei hervorgehobenen Knoten, in der Reihenfolge (pub/priv/ref). Die gewählte Beschriftung ergibt sich auch aus dieser Entfernungsangabe, wobei priv bei gleichem Wert gegenüber pub dominiert. Nicht erreichbare hervorgehobene Knoten sind durch ein „-“ markiert.

Heuristische Behälterzuordnung von Elementen

Nachdem für alle Nachbarschaften passende Behälter ermittelt worden sind, werden die darin enthaltenen DRAGULA-Elemente entsprechend zugeordnet. Durch die Verklebung von DRAGULA-Elementen können diese im Endeffekt Bestandteil mehrerer Nachbarschaften sein die *unterschiedlichen* Behältern zugeordnet sind. Auch bei der Zuweisung von DRAGULA-Elementen an Behälter ist somit eine geeignete Heuristik anzuwenden. Hierbei dominiert derjenige Behälter der am weitesten *außerhalb* der Schachtelungshierarchie liegt, in dem also *alle anderen* fraglichen Behälter transitiv enthalten sind. Bei Vorhandensein eines eindeutigen Behälters wird das DRAGULA-Element diesem Behälter zugeordnet. Andernfalls erfolgt auch hier eine Fehlermeldung zur Laufzeit.

Das in Abbildung 6.17 beschriebene Beispiel wird wie folgt vervollständigt: Da die Nachbarschaften 3 bis 10 dem inneren Muster zugeordnet worden sind, werden auch alle ausschließlich in diesem enthaltene DRAGULA-Elemente in das innere Muster

platziert. Ebenso werden alle Elemente der Nachbarschaften `11` und `12` dem durch die Abschlussbedingung referenzierten Muster zugewiesen. Fraglich bleibt hier lediglich der Verbleib der oberen DRAGULA-Variablen. Diese sind auch in den Nachbarschaften `1` bzw. `2` enthalten die dem äußeren Muster zugeordnet sind. Aufgrund obiger Vorzugsregel werden die Variablen also dem äußeren Muster zugeordnet. Dies ist eine natürliche Entscheidung, da diese Enthaltenseinsinformation in Abbildung 6.13b *explizit* spezifiziert worden ist.

Berücksichtigung manueller Enthaltenseinsangaben durch Heuristiken

Das vorgestellte Verfahren zur impliziten Behälterzuordnung berücksichtigt stets explizite Festlegungen. So ist die explizite Zuordnung zu *inneren* Mustern (vgl. Abbildung 6.13c) gesichert, da alle adjazenten Nachbarschaften *ohne eigene* Behälterinformationen aufgrund der minimalen Entfernung ebenfalls dem inneren Muster zugeordnet werden. Somit ist ein DRAGULA-Element mindestens in einer Nachbarschaft enthalten, die den explizit spezifizierten Behälter umfasst. Gleichzeitig kann das Element aufgrund obiger Propagationsregel lediglich in weiteren Nachbarschaften liegen, die mit einem tiefer geschachtelten Behälter beschriftet sind. Das Element selbst wird jedoch dem *äußersten* der in Frage kommenden Behälter zugewiesen, der somit durch die explizite Festlegung bestimmt wird.

Lediglich falls *mehrere* explizite Enthaltenseinsspezifikationen für ein einzelnes Element, etwa durch Verklebung, angegeben werden kann die Zuordnung nicht eindeutig erfolgen. Hier wird bei Vorhandensein einer Enthaltenseinshierarchie zwischen den fraglichen Behältern der *äußere* gewählt, andernfalls wird ein Laufzeitfehler ausgelöst. Dieser Fall ist jedoch grundsätzlich als Spezifikationsfehler zu werten und sollte vermieden werden.

Zwischenfazit: Mit der Bereitstellung grundlegender Sprachkonstrukte zur Übersetzerspezifikation im vorangegangenen sowie spezieller Konstrukte zur Enthaltenseinsspezifikation in diesem Abschnitt ist die bislang verfügbare SViTL-Funktionalität vollständig erläutert worden. Insbesondere die Möglichkeit, geeignete Behälter der zielseitig erstellten Elemente auf Basis DRAGULA-spezifischer Heuristiken zuzuordnen schafft einen erheblichen Mehrwert gegenüber allgemeinen Transformationsansätzen. So wird einerseits der Entwicklungsaufwand reduziert, andererseits entfällt die fehleranfällige Einplanung der Enthaltenseinsmodellierung in die Gesamtspezifikation. Notwendig ist die explizite Spezifikation weiterhin bei *Übergängen* zwischen verschiedenen Behältern, insbesondere wenn die gewünschte Bedeutung eines übersetzten Konstrukts von der Wahl des zielseitigen Behälters abhängt.

6.6 Vervollständigung der Beispielspezifikation

Während der Fokus des vorherigen Abschnitts auf einer Erläuterung der Transformationssprache SViTL liegt, ergänzt der folgende die auszugsweise behandelte Spezifikation zur Vervollständigung des durchgängigen Beispiels. Ziel ist dabei die Anwendung des Transformationsansatzes zu vertiefen und eine Richtlinie zur Erarbeitung eines Übersetzers zu skizzieren.

Beim Entwurf einer Übersetzerspezifikation wird vorgeschlagen, zunächst anhand der quellseitigen *Metamodellelemente* und deren jeweiligen Beziehungen vorzugehen. I.d.R. genügt es dabei jeweils ein zentrales Metamodellelement mit entsprechenden Kontextbeziehungen aufzuführen und korrespondierende DRAGULA-Fragmente anzugeben. SViTL schränkt die Spezifikation jedoch nicht darauf ein, sondern erlaubt die Angabe beliebig komplexer Graphausschnitte. Hierbei sind ggf. Fallunterscheidungen anhand quellseitiger Attribute vorzunehmen wenn das Übersetzungsergebnis, wie im Beispiel bei public-Variablen der Fall, dadurch strukturell beeinflusst wird. Optionale und mengenwertige Verknüpfungen im Quellmetamodell werden durch jeweils eigene Paare spezifiziert falls, pro zusätzlicher Auffindungsstelle zielseitige Elemente *ergänzt* werden sollen. Durch negative Anwendbarkeitsbedingungen können jedoch auch strukturelle Informationen zur Fallunterscheidung der Transformation genutzt werden.

Bezogen auf die Anfragesprache EMAA werden bislang nicht behandelte Übersetzungsregeln im Folgenden schrittweise vorgestellt. Das jeweils zugrundeliegende Metamodell ist in Abbildung 3.3 auf Seite 60 dargestellt. Zusammengefasst sind also die Elemente Document, Query, Scalar, Entity, ScalarRestriction, GraphRestriction sowie

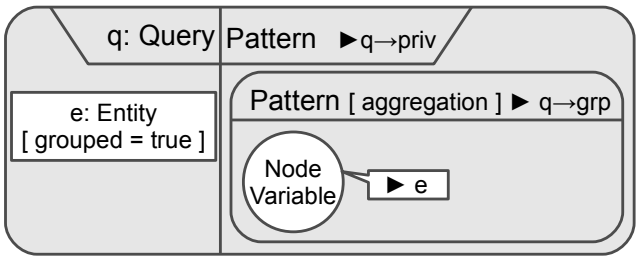


Abbildung 6.19: Enthaltenseinsspezifikation für gruppierte Variablen

die Ausprägungen der Attributs- und Navigationsausdrücke zu beschreiben. Letztere Teilsprachen werden einschließlich ihrer Restriction-Varianten in den gesonderten Unterabschnitten 6.6.2 bzw. 6.6.3 behandelt.

Das Strukturierungselement Document kann bei der Übersetzung ignoriert werden da alle EMAA-Anfragen ungegliedert als DRAGULA-Anfragen abgelegt werden sollen. Der Behälter Query wird ebenfalls nicht direkt übersetzt, sondern in verschiedenen Paaren zur Enthaltenseinsspezifikation referenziert. Für die Unterscheidung zwischen öffentlichen und privaten Variablen sowie der zu erzeugenden Musterhierarchie, wird dies in Abbildung 6.13 gezeigt. Vorab ist außerdem die Übersetzung von Variablen in Abbildung 6.4 erläutert worden, so dass diese hier ausgelassen werden können.

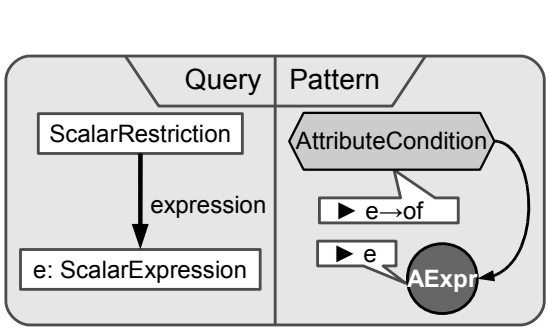
6.6.1 Übersetzung gruppierter Ergebnisvariablen

Gruppierte Ergebnisvariablen stellen eine Möglichkeit bereit um Attributwerte mengenwertig zu erfassen. Hierfür wird das DRAGULA-Sprachelement für *aggregierte Muster* eingesetzt. Da in EMAA keine mehrfachen Gruppierungen vorgesehen sind, genügt die Angabe *eines* solchen Suchmusters zur Aufnahme aller gruppierten Variablen einer Anfrage. Dieses wird innerhalb des Musters für private Ergebnisvariablen abgelegt um auch zu deren Berechnungen aggregierte Werte nutzen zu können. Das entsprechende SViTL-Paar zeigt Abbildung 6.19. Ferner ist durch eine ergänzende Attributbedingung [¬grouped] in Abbildung 6.13 sicherzustellen, dass die explizite Behälterzuordnung nur für *nicht-gruppierte* Entitätsvariablen angewendet wird. Andernfalls würde aufgrund einer mehrfachen Behälterzuweisung die zum inneren Behälter unberücksichtigt bleiben.

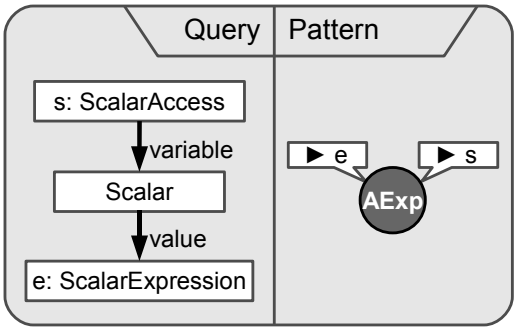
6.6.2 Übersetzung von Attributausdrücken

Mit den Sprachkonstrukten ScalarRestriction und Scalar bietet EMAA zwei Anwendungsfälle für Attributausdrücke, einerseits zur Beschränkung der Ergebnismenge einer

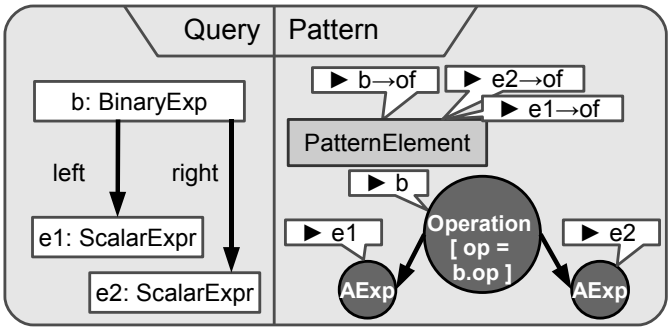
Anfrage und andererseits zur Ausgabe eines skalaren Wertes. Letzterer Fall ist bereits in Abbildung 6.4b behandelt worden, so dass lediglich die skalare Restriktion zu erläutern bleibt.



(a) Restriktionen auf skalaren Wertausdrücken



(b) Zugriff auf skalare Ergebnisse



(c) Werteverknüpfung mittels Binärausdrücken

Abbildung 6.20: Verarbeitung skalarer Ausdrücke

Auf Attributwerte wird in der DRAGULA Teilsprache für Ausdrücke entweder im Rahmen von Attributbedingungen oder -setzoperationen zugegriffen. In beiden Fällen werden die Graphentitätsvariablen durch diese Sprachkonstrukte durch Restricts- bzw. Requires-Beziehungen referenziert und mit einem Bezeichner versehen. Dieser Bezeichner wird in einem Access-Ausdrucksknoten referenziert. In EMAA wird auf Attributwerte dagegen durch das Sprachkonstrukt AttributeAccess direkt unter Angabe der Entitätsvariablen und des Attributnamens zugegriffen. Daher ist bei der Übersetzung zu berücksichtigen, dass bei Erzeugung eines DRAGULA-Access Konstrukts die entsprechenden Entitätsvariablen durch die zugehörige Bedingung oder Setzoperation referenziert werden.

Die in SViTL modellierte Abbildung skalarer Zugriffe besteht aus insgesamt drei Paaren. Abbildung 6.20a bildet zunächst eine Restriktion auf eine Attributbedingung

ab und verankert diese mit Beschriftung of an den referenzierten Ausdruck. Zusätzlich wird ein zielseitiger Ausdrucksknoten ohne konkrete Typinformation angelegt und ebenfalls auf den EMAA-Ausdruck verankert. Ein Attributzugriff kann dadurch wie in Abbildung 6.11a auf Seite 249 abgebildet werden: Durch die Verankerung des zielseitigen Access-Knotens auf den skalaren Zugriff wird dieser in den Syntaxbaum des Attributausdrucks aufgenommen, bspw. indem dadurch die unspezifische AttributeExpression in obigem Paar durch Typverfeinerung konkretisiert wird. Die Verankerung des PatternElement mit Beschriftung of sorgt für eine Verklebung entweder mit dem zugeordneten Attributausdruck oder der -setzoperation. Unabhängig von der tatsächlichen Positionierung des Attributzugriffs im Syntaxbaum kann somit auf dieses Musterelement zugegriffen und eine Verknüpfung zur referenzierten Graphentitätsvariablen eingefügt werden.

Das zweite Paar in Abbildung 6.20b repräsentiert den Zugriff auf skalare Werte anstelle von Attributwerten eines Graphelements. Da skalare Ergebnisse in DRAGULA durch Setzoperationen zugewiesen werden, stehen sie nicht direkt zur Anfragebearbeitung zur Verfügung. Wie dargestellt wird daher lediglich die AttributeExpression sowohl an den berechnenden Ausdruck der skalaren Variable, als auch an den Zugriff innerhalb eines anderen Attributausdrucks verankert. Auf diese Weise verweisen sowohl die skalare Variable als auch ihre zugeordnete Abfrage auf einen gemeinsamen Syntaxbaum.

Zur Verknüpfung von Teilausdrücken werden in EMAA binäre Ausdruckselemente (BinaryExp) verwendet, die jeweils zwei Teilausdrücke miteinander kombinieren. Als mögliche Operatoren werden dabei die üblichen und von DRAGULA bereits angebotenen Vertreter direkt übernommen, weshalb in Abbildung 6.20c der Attributwert von op direkt übertragen werden kann. Wie im ersten hier behandelten Paar wird die Verankerungsbeschriftung of verwendet um das zielseitige Musterelement zu verankern. Durch die identische Verankerung sowohl auf den kombinierenden Ausdruck als auch auf die referenzierten Teilausdrücke verwenden alle Elemente des Syntaxbaums ein gemeinsames Musterelement. Ein ähnliches Sprachkonstrukt stellt der unäre Aggregationsoperator (Aggregation) bereit, der ebenfalls direkt auf Elemente der DRAGULA-Attributerweiterung abgestützt werden kann und daher hier ausgelassen wird. Das ebenfalls ausgelassene Sprachkonstrukt der Skalarwerte (Literal) wird durch eine direkte Übersetzung von EMAA- zu DRAGULA-Elementen abgebildet.

6.6.3 Übersetzung der Verbindungsnavigation

Zur Einschränkung von Anfrageergebnissen auf Basis von Verbindungsstrukturen stellt EMAA das Sprachkonstrukt GraphRestriction bereit. Der angewandte Strukturierungsansatz ist dabei nicht direkt auf DRAGULA übertragbar, sondern benötigt ei-

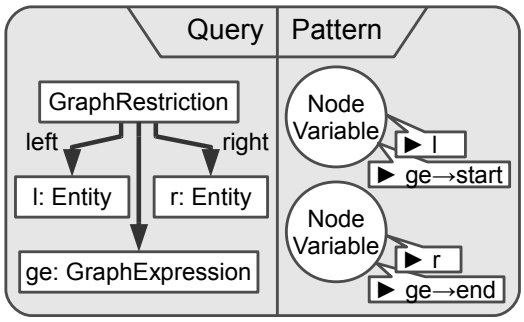
ne etwas aufwändigere Umsetzung. So spezifiziert EMAA graphische Restriktionen in Form von Traversierungsschritten die baumartig durch Konkatenation und Abschlussoperationen kombiniert werden können. In DRAGULA ist diese baumartige Struktur nicht vorhanden, vielmehr wird hier eine direkte Verknüpfung der angefragten Variablen genutzt. Die mittels SViTL zu spezifizierende Übersetzung sollte also die in EMAA modellierte hierarchische Baumstruktur auf eine flache DRAGULA-Anfrage reduzieren. Dies wird im Folgenden durch Nutzung von Verankerungen erreicht die eine DRAGULA-Variable als Start- bzw. Endknoten eines einzelnen quellseitigen Graphausdrucks markieren.

Die zur oben geschilderten Übersetzung erforderlichen EMAA-Paare sind in Abbildung 6.21 dargestellt. Durch eine GraphRestriction werden zwei Entitätsvariablen referenziert die den Start- bzw. Endpunkt des gesamten Ausdrucks angeben. Zielseitig wird, wie in Abbildung 6.21a dargestellt, lediglich eine Verankerung dieser Variablen spezifiziert aber keine anderweitigen Konstrukte hinzugefügt.

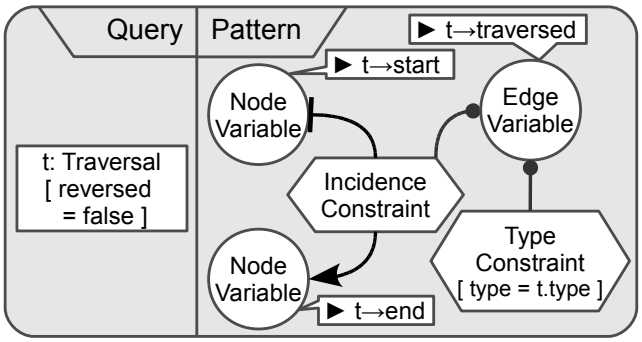
Die Traversierung einer einzelnen Verbindung wird in EMAA mittels einer Traversal-Operation spezifiziert. Diese wird in Abbildung 6.21b auf eine Inzidenzbedingung sowie deren benachbarten Variablen abgebildet. Hierbei erfolgt eine Verankerung der Start- und Zielvariablen gemäß obiger Beschriftungen, so dass eine entsprechende Verklebung vorgenommen werden kann. Stützt sich bspw. eine GraphRestriction lediglich auf eine einzelne Traversal-Operation ab, dann werden die DRAGULA-seitig für die referenzierten Entitätsvariablen erstellten Elemente mit den entsprechenden Knotenvariablen aus Abbildung 6.21b verklebt. Die Übersetzung der Traversierung muss ihre gewünschte Richtung beachten, wobei im dargestellten Fall vom „Vorwärts“-Fall ausgegangen wird[7]. Eine unter Vertauschung der zielseitigen Variablen ableitbare Regel zur umgekehrten Navigation wird hier ausgelassen.

Zur Verknüpfung von Teilausdrücken, die in Abbildung 6.21c dargestellt ist, wird analog zu obigen GraphRestrictions vorgegangen. Im Fall der Concatenation sind zwei Paare vorhanden, wobei im ersten Fall zunächst Start- bzw. Endpunkt der konkatenierten Navigationen festlegt und eine zu traversierende Zwischenvariable ohne weitere Bedingungen etc. erzeugt wird. Die optionale Möglichkeit, diese Zwischenvariable explizit zu spezifizieren und ggf. weiter einzuschränken, wird durch Abbildung 6.21d beschrieben. Dieses Paar verankert die Zwischenvariable auf die explizit referenzierte Entität. An dieser Stelle wird ein Vorteil der Arbeitsweise SViTLs deutlich: Optionale Sprachkonstrukte lassen sich in ihrem „default“-Verhalten übersetzen, hier also die Traversierung über eine unbenannte Zwischenvariable (Abbildung 6.21c). Dieses Verhalten lässt sich durch Angabe eines weiteren Paares variieren, in dem die Zwischenvariable mit einer festgelegten Variablen verankert wird (Abbildung 6.21d).

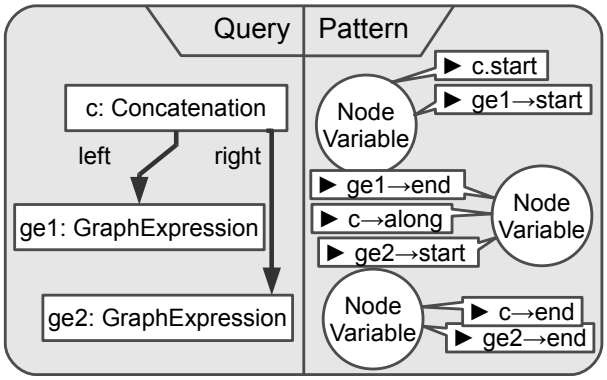
[7] Erreicht durch die Attributbedingung reversed=false.



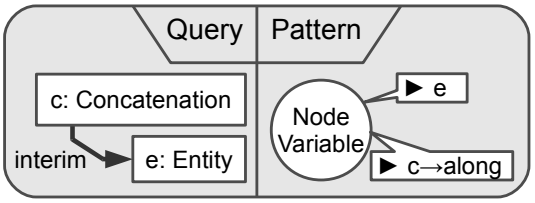
(a) Restriktionen auf Verbindungsstrukturen



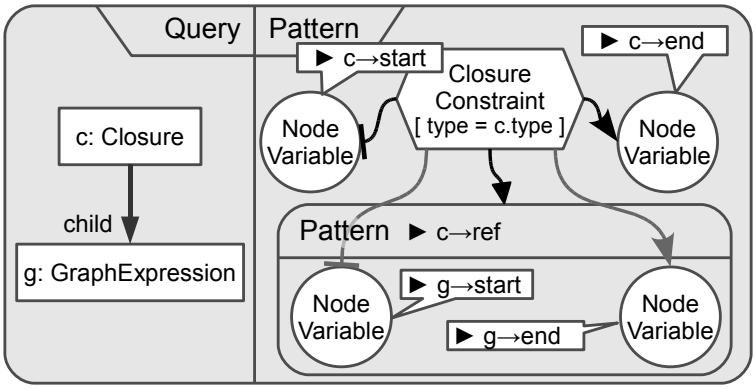
(b) Typisierte Verbindungstraversierung



(c) Konkatenation von Graphausdrücken



(d) Konkatenation mit explizitem Zwischenelement



(e) Abschluss von Graphausdrücken

Abbildung 6.21: Verarbeitung von Graphausdrücken

Es ist also im ersten genannten Paar keine Entscheidung notwendig ob eine anonyme oder eine benannte Variable verwendet wird.

Schlussendlich zeigt Abbildung 6.21e den transitiven oder reflexiven Abschluss von Teilausdrücken. Hierzu wird die in Abschnitt 4.6.1 vorgestellte Spracherweiterung verwendet und demzufolge das hierin definierte ClosureConstraint genutzt. Dieses bezieht sich auf die Variablen des referenzierenden Teilausdrucks (oben) sowie das zu iterierende Muster und die darin als Start- und Endpunkt einer Iteration zu nutzenden Variablen (unten). Bezüglich der Enthaltenseinsspezifikation ist anzumerken, dass lediglich das zur Iteration benötigte Muster explizit verankert wird. Da für das umgebende Muster keinerlei Festlegung erfolgt, wird auch keine Hierarchie zwischen dem referenzierten Muster und den übrigen DRAGULA-Elementen wie bspw. der Abschlussbedingung modelliert. Demzufolge bildet das referenzierte Muster eine von dem Anfrage-Muster unabhängige Struktur, wie in Abbildung 6.3 auf Seite 230 zu sehen ist.

Zwischenfazit: Insgesamt werden zur Abdeckung der vollständigen EMAA-Syntax 18 SViTL-Paare benötigt, von denen bis auf drei Paare mit hohem Wiederholungsanteil alle erläutert worden sind. Somit kann das ebenfalls 18 Klassen und Beziehungen umfassende Metamodell von EMAA – einschließlich optionalen und mengenwertigen Konstrukten – knapp und übersichtlich dargestellt werden. Lediglich fünf dieser SViTL-Paare legen dabei explizite Enthaltenseinsinformationen fest, was durch implizites Behandeln dieser Information im Sinne der DRAGULA-Sprache erreicht wird.

6.7 Formale Eigenschaften des Übersetzerverhaltens

Dieser Abschnitt untersucht verschiedene formale Aspekte der Übersetzersprache SViTL. Zunächst stellt sich die Frage, ob mit SViTL beschriebene Übersetzer *eindeutige Ergebnisse* unabhängig von der Reihenfolge der internen Übersetzerausführung liefern. Diese Eigenschaft wird anhand eines Terminations- und Konfluenznachweises gezeigt, woraus in Termersetzungssystemen die eindeutige Ableitbarkeit von Normalformen [BN99] und damit die erwünschte Eindeutigkeit folgt. Der zweite betrachtete Aspekt betrifft die Ausdruckskraft der Übersetzung, wodurch sich die Anwendbarkeit auf mögliche Quellsprachen ergibt.

Obwohl eine formale Herleitung dieser Eigenschaften wünschenswert und relevant ist, kann diese dennoch nicht im Rahmen dieser Arbeit hergestellt werden. Dies liegt an dem grundsätzlich allgemeinen Ansatz der zur Ausführung verwendeten Spezifi-

kationssprache DRAGULA, die insbesondere keine formale Konfluenzanalyse ermöglicht. Hierzu wäre ein grundsätzlich enger umfasster Ansatz, wie bspw. der Double-Pushout-Approach für Graphtransformationen, anzuwenden. Dieser ist jedoch aufgrund seiner starken modellierungsseitigen Einschränkung nicht mit dem Anspruch auf allgemeine Nutz- und Anpassbarkeit dieser Arbeit vereinbar. Die folgenden Untersuchungen werden daher lediglich auf informellem Niveau durchgeführt.

6.7.1 Termination

Der Nachweis der Terminationseigenschaft ergibt sich für SViTL aus einer Untersuchung der einzelnen Übersetzungsphasen, die in Abschnitt 6.2.3 aufgezählt sind. Im ersten Schritt erfolgt die Ermittlung der Auffindungsstellen aller SViTL-Paare eines auszuführenden SViTL-Dokuments. Zugleich werden für jede dieser Stellen zielseitig DRAGULA-Elemente entsprechend der Spezifikation erzeugt.

Zur Terminationsanalyse wird angenommen, dass sowohl die Menge der Paare als auch die Menge der verschiedenen Elemente im Eingabemodell *endlich* ist. Unter dieser Voraussetzung können nur *endlich viele* Auffindungsstellen ermittelt und dementsprechend endlich viele Transformationsschritte ausgeführt werden:

- Quellseitig stellt die SViTL-Sprachdefinition gemäß Abschnitt 6.7 lediglich *einfache* Konstrukte zur Verfügung, die direkt auf Elemente des Eingabemodells abgebildet werden. Nicht definiert ist dagegen die Verwendung von DRAGULA-ähnlichen *Musterreferenzen*, die ggf. zu einer unendlichen Auswertung auch auf endlichen Eingabemodellen führen könnten. Anzumerken ist dazu, dass die Verwendung der in Abschnitt 4.6.1 vorgestellten Abschlusserweiterung für DRAGULA die Terminationseigenschaft nicht abschwächen würde.

- Bei der Ermittlung von Auffindungsstellen kann nur auf *quellseitige* Modellelemente zugegriffen werden. Somit können durch die Ausführung einzelner Teiltransformationen *keine neuen* Auffindungsstellen entstehen, da neue Elemente lediglich im separierten Zielmodell erzeugt werden. Eine stärkere Einschränkung, bspw. die Forderung alle Auffindungsstellen *vor* der ersten Transformation zu bestimmen, kann somit entfallen.

Im zweiten Schritt der Ausführung einer SViTL-Spezifikation werden zuvor erstellte Elemente miteinander verklebt, was auch über mehrere Stufen hinweg erfolgen kann. Wird die Verklebung jeweils paarweise betrachtet, resultiert aus *zwei* zu verklebenden Elementen genau *ein* verklebtes, wobei die Eingangselemente für weitere Verklebungen nicht mehr zur Verfügung stehen. Die Anzahl der verklebbaren Elemente nimmt also mit jedem Ausführungsschritt *streng monoton ab* und terminiert

somit nach endlich vielen Schritten sobald keine zwei verklebbaren Elemente mehr verbleiben.

Der dritte Schritt betrifft die Behälterzuordnung der zuvor verklebten Elemente. Diese erfolgt auf Basis der Nachbarschaften, deren Anzahl sich aus der Anzahl der angewendeten Transformationen im ersten Schritt ergibt. Auf Basis dieser Elemente erfolgt eine kürzeste-Wege Berechnung, die bspw. mit Dijkstra's Algorithmus [Dij59] erfolgen kann. Zu Dijkstra's Algorithmus existieren Realisierungen die nachweislich terminieren [Mis01].

Die weitere Verarbeitung quantifiziert einerseits über die endliche Anzahl an Nachbarschaften, andererseits im zweiten Schritt über die Anzahl der zielseitig erstellten Elemente. Diese ist ebenfalls endlich, da lediglich endlich viele Transformationsschritte mit endlicher Größe der zielseitigen Paar-Seite hierzu beigetragen haben. Insgesamt terminiert die Übersetzung somit.

6.7.2 Konfluenz

Die zweite relevante Eigenschaft betrifft die Konfluenz der Übersetzung, letztlich also die Fragestellung, ob nichtdeterministisch getroffene Entscheidungen bei der Übersetzung das Endergebnis beeinflussen. Hierbei sind Nichtdeterminismen an verschiedenen Stellen des Transformationsablaufs zu untersuchen.

Im ersten Schritt wird nichtdeterministisch entschieden in welcher Reihenfolge Transformationen zur Erzeugung der Zielstrukturen ausgeführt werden. Hierbei werden lediglich *zielseitige* Elemente erzeugt die keinen Einfluss auf die Anwendbarkeit weiterer Transformationsregeln besitzen, da diese sich ausschließlich aus *quellseitigen* Elementen ergibt. Somit entstehen *keinerlei* kritische Paare von SViTL-Paaren im Sinne der Kritischen-Paar-Analyse [HKT02] und die Transformation im ersten Schritt erfüllt die Konfluenzeigenschaft. Anders ausgedrückt sind somit Transformationsanwendungen sowohl bezüglich eines einzelnen SViTL-Paars als auch einer Gesamtspezifikation in ihrer Reihenfolge austauschbar.

Der zweite Übersetzungsschritt betrifft die Verklebung der erzeugten Elemente. Ist ein Element α mit einem weiteren β verklebbar, das wiederum mit einem Element γ verklebt werden kann, so ist zunächst sicherzustellen, dass im Endeffekt alle diese Elemente tatsächlich zusammengeführt werden können. Da sich die Verklebbarkeit aus dem Vorhandensein mindestens *einer* gemeinsamen Verankerung ergibt, ist diese Eigenschaft gegeben: Das Resultat der ersten Verklebung $\delta_{\alpha,\beta}$ enthält die Vereinigung beider Verankerungen und teilt demnach auch eine Verankerung mit γ. Daher ist es mit $\delta_{\alpha,\beta}$ verklebbar – die Verklebbarkeitsrelation ist somit transitiv. Zusätzlich ist sicherzustellen, dass die Eigenschaften des Ergebniselements unabhängig von der Reihenfolge der angewendeten Teiloperationen sind. Hierbei ist anzumerken, dass Prio-

ritäten bei der Attributierung oder Typisierung lediglich auf Basis der *Typhierarchie* der zu verklebenden Elemente erfolgen. Diese ist unabhängig von der Transformationsreihenfolge gegeben. Ein Attributwert von α überschreibt somit die Werte von β und γ wenn ersteres Element den spezialisiertesten Typ besitzt, unabhängig von der Reihenfolge der Verklebung.

Im letzten Verarbeitungsschritt treten keine Nichtdeterminismen auf: So erfolgt die Zuordnungen von Nachbarschaften zu Behältern deterministisch, wenn auch durch Heuristiken bestimmt. Die Zuordnung der darin enthaltenen zielseitigen Elemente wird ebenfalls nicht extern beeinflusst.

Experimentelle Überprüfung: Zur Überprüfung der lediglich informell nachgewiesenen Konfluenzeigenschaft ist unter Einsatz der Graphtransformations- und Modellprüfungswerkzeugs GROOVE [Ren03] die hier vorgestellte Beispielspezifikation analysiert worden. Mittels GROOVE können Graphtransformationssysteme durch Ermittlung ihres vollständigen *Zustandsraums* untersucht werden, so dass bei auftretenden Nichtdeterminismen alle möglichen Auffindungsstellen simultan verarbeitet werden. Hieraus ergibt sich ein *beschriftetes Transitionssystem*, das jeden bei der Transformationsverarbeitung erreichbaren Zwischenzustand in Form eines Laufzeitgraphen enthält und deren Übergänge festhält. Durch Einsatz von GROOVE konnte im vorliegenden Fall nachgewiesen werden, dass sowohl nach Schritt 1 (Mustersuche & Transformation), als auch nach Schritt 2 (Verklebung) ein *eindeutiger* Zustand erreicht wird, die Transformation also stets konvergiert. Zusätzlich wird ein eindeutiger *Endzustand* erreicht, so dass insbesondere die Termination der Transformation sichergestellt ist. Diese Untersuchung validiert lediglich die Anwendung *einer spezifischen* SViTL-Spezifikation auf *ein* Eingabemodell. Allgemeinere Aussagen können aufgrund des unendlichen Modellraums – sowohl des Eingabemodells als auch der Transformationsbeschreibung – nicht gewonnen werden. Hierzu könnte in zukünftigen Arbeiten eine Übertragung auf höherwertige Logikkalküle erfolgen, um bspw. Verifizierer wie Isabelle/HOL [NPW02] einsetzen zu können.

6.7.3 Ausdrucksmächtigkeit der Transformationssprache

Grundsätzlich eignet sich SViTL zur Spezifikation von Übersetzern, die syntaktische Spezifikationsmodelle als Eingabe und Elemente der DRAGULA-Syntax als Ausgabe verwenden. Es handelt sich hierbei gewissermaßen um eine *niveauerhaltene Transformation*, da in beiden Fällen ein ausführbares Dokument verwendet wird.

Anders dagegen verhält es sich in Anwendungsfällen, bei der die Zielseite eine *reduzierte Struktur* darstellt, bspw. die Zieldomäne einer Wertanfrage, wie etwa den

booleschen Werten `true` und `false`. Hierbei würde die Transformationsspezifikation als Interpreter fungieren und die Eingabe in ihr Ergebnis „transformieren“. Eine solche Transformation kann mit SViTL *nicht* spezifiziert werden, da das auszugebende Zielmodell pro Übersetzungsschritt streng monoton erweitert, jedoch nicht beliebig verändert werden kann. Somit kann eine getroffene Aussage `true` – bspw. bzgl. des Erreichens eines akzeptierenden Zustands einer Turingmaschine – nur dann zu einem Ergebnis `false` umgeschrieben werden, falls dies durch eine entsprechende Typverfeinerung bei der Verklebungsoperation begründet wird.

Ebenso verhält es sich, wenn als Ausgabemodell eine zeitliche Abfolge des Zielzustands einer Maschine inkrementell ergänzt wird – in diesem Fall ist jedoch eine individuelle Verankerung der Quellseite zu jedem der Quellseite hinzugefügten Knoten erforderlich. Da Ankerspezifikationen aus Tupeln von Bezeichnern der Quellseite bestehen – ausgehend von der Größe des Eingabegraphen – können höchstens polynominell viele Endzustände erzeugt werden. Exponentielle Algorithmen sind auf diese Weise nicht beschreibbar. Insgesamt ist festzuhalten, dass somit die Simulation allgemeiner Turingmaschinen mit SViTL nicht möglich ist – die Übersetzungssprache ist daher nicht Turing-vollständig.

6.8 Diskussion

Wie in den vorangegangenen Kapiteln diskutiert dieser Abschnitt Vor- und Nachteile die sich aus dem SViTL-Ansatz ergeben, bevor im Folgenden Beziehungen zu konkreten verwandten Arbeiten hergestellt werden.

6.8.1 Vorteile

Mit Hilfe von SViTL können Übersetzer zwischen spezifischen operationalen Modellierungssprachen und DRAGULA relational beschrieben werden. Der wesentliche Vorteil liegt hierbei in der *deklarativen* Spezifikationsweise, die sich nah an fachlichen Aspekten der Übersetzung orientiert. Hingegen werden technische Aspekte, wie die Realisierung der quellseitigen Mustersuche, Kontroll- oder Datenfluss zwischen Regeln etc., weitestgehend ausgeblendet.

Damit einher geht die Möglichkeit zur *konzeptorientierten* Spezifikation. Konzepte der Quellsprache werden in zusammenhängenden Beispielszenarien beschrieben und auf entsprechende Ausschnitte einer DRAGULA-Spezifikation abgebildet. Insbesondere können in SViTL beschriebene Übersetzer eng an einer konzeptionell ausgearbeiteten Sprachabbildung realisiert werden. Aufwändige Transferleistungen bspw. in eine herkömmliche Programmiersprache entfallen somit.

Durch Berücksichtigung der Zielsprache DRAGULA werden einige Aspekte der Transformation implizit abgebildet, speziell Enthaltenseinsbeziehungen der Zielseite. Hierdurch wird eine häufig ausführliche Beschreibung dieses Sachverhalts vermieden, was im Vergleich kleinere und weniger SViTL-Paare erfordert. Dieser Aspekt unterstützt somit die konzeptorientierte Spezifikation, da insbesondere technisch motivierte SViTL-Paare wie eben zur Festlegung von Behälterinformationen reduziert werden.

6.8.2 Nachteile

Die Ausdruckskraft von SViTL ist bewusst auf nicht-turingvollständige Übersetzer beschränkt. Somit können strukturelle Übersetzungen zwischen Sprachen beschrieben werden, komplexe Operationen die in möglicherweise unendlichen Übersetzerläufen resultieren würden sind jedoch ausgeschlossen. Entsprechende Einschränkungen, die auch den Zugriff der Mustersuche auf zielseitig erzeugte Elemente untersagen, sind bewusst getroffen worden um möglichst bedarfsgerechte Funktionalität bereitstellen zu können.

SViTL unterstützt konzeptionell bedingt lediglich unidirektionale Transformationen von der jeweiligen Quellsprache nach DRAGULA. Änderungen an der DRAGULA-Spezifikation können daher mit dem erzeugenden Satz SViTL-Regeln nicht in das ursprüngliche Quelldokument zurück überführt werden. Der Grund für diese Einschränkung liegt zum einen in der Auftragung von Elementverankerungen allein auf der Zielseite. Zum anderen handelt es sich bei der Elementverklebung als Teil der SViTL-Regelausführung um eine i.A. nicht-umkehrbare Operation. Im spezifischen Kontext der Sprachübersetzung ist jedoch die Rückübersetzung der erzeugten DRAGULA-Spezifikationen nicht von praktischer Relevanz.

Als weiterer Nachteil ist anzumerken, dass der Entwickler derzeit nicht *inhaltlich* in der Übersetzerbeschreibung unterstützt wird, etwa indem toolunterstützt Vorschläge zur Abbildung auf die Kernsprache angeboten werden. Dies könnte zukünftig bspw. durch Heuristiken und Rückgriffe auf Wissensdatenbanken mit bestehenden Übersetzerspezifikationen erfolgen, indem Sprachkonzepte unterschiedlicher Quellsprachen als ähnlich identifiziert werden.

6.9 Verwandte Arbeiten

Der Forschungsbereich Modelltransformationen, der letztlich eine Obermenge der hier betrachteten Übersetzersprachen bildet, wird durch eine nahezu unüberschaubare Menge an Werkzeugen bedient. An dieser Stelle kann daher nur auf eine begrenzte

Teilmenge eingegangen werden. Die hier vorgestellten Lösungen orientieren sich zum einen daher stark an Graphtransformationen, zum anderen verwenden sie spezielle Ansätze SViTLs, wie etwa die Verklebungsoperation.

6.9.1 Tripel-Graph-Grammatiken

Tripel-Graph-Grammatiken (TGGs) [Sch94] stellen einen grammatikbasierten Ansatz zur Übersetzung zwischen formalen Sprachen dar. Eine TGG verbindet die erzeugenden Grammatiken der Quell- und Zielsprache mit der einer dritten Struktur, dem sogen. Korrespondenzgraphen. Konzeptionell wird zur Übersetzung eines gegebenen Satzes – also einer Graph- oder Objektstruktur – in der Quellsprache zunächst der Ableitungsbaum der zugehörigen Grammatik bestimmt. Hieraus ergeben sich die entsprechend anzuwendenden Grammatikregeln der Ziel- und Korrespondenzgrammatik und damit die zu erzeugenden Strukturen. Dabei sind die Rollen der Quell- und Zielsprachen nicht vorab festgelegt, Transformationen können somit grundsätzlich in beide Richtungen durchgeführt werden. Zur Konsistenzprüfung können bereits vorhandene Graphstrukturen miteinander verglichen und darüber ein Korrespondenzgraph aufgebaut werden.

Abbildung 6.22 zeigt eine TGG-Regel um das EMAA-Sprachkonstrukt Entity in eine DRAGULA NodeVariable zu übersetzen. Die Grammatikregel besagt, dass bei *Auffinden* des Kontextes die Nicht-Kontextelemente – angezeigt durch die +++-Markierung – in allen drei Graphen simultan *erstellt* werden. Da hierbei eine Entity genau einer NodeVariable zugeordnet werden soll, stellen diese den Nicht-Kontext Teil der Regel dar. Beide Elemente sind jeweils eindeutigen Behältern (Query bzw. Pattern) zugeordnet, die ihrerseits miteinander korrespondieren müssen. Dies stellt den Kontext der Regel dar, der zur Anwendbarkeit der TGG-Regel zunächst durch andere Regeln erzeugt worden sein muss.

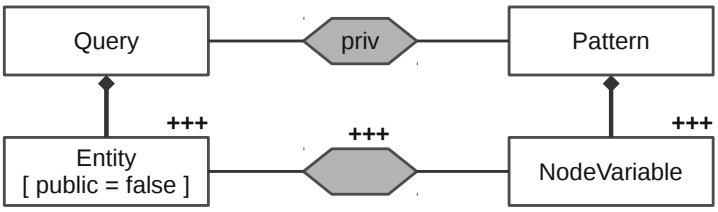


Abbildung 6.22: Abbildung privater Variablen als TGG-Regel

Zur Ausführung dieser Regel in eine bestimmte Transformationsrichtung werden alle Kontextelemente sowie die quellseitigen Nicht-Kontextelemente gesucht. Für alle so gefundenen Auffindungsstellen werden entsprechende Nicht-Kontextelemente der

Ziel- und Korrespondenzgraphen erzeugt. SViTL-Paare unterscheiden nicht zwischen Kontext- und Nicht-Kontextelementen auf der Zielseite, sondern stellen lediglich deklarativ den erwarteten Zustand dar.

Die Übersetzung mengenwertiger, sogen. *eins–zu–n*-Beziehungen, die im gegebenen Fall zwischen einer Query und mehreren Entity besteht, erfordert stets zwei getrennte TGG-Regeln. In einer ersten Regel tritt die Query mit dem zielseitig zugeordneten Pattern als Nicht-Kontextelement auf. Die zweite Regel verwendet eine identische Struktur als Kontext und fügt die Elemente der *zu–n*-Seite als Nicht-Kontextelemente hinzu. SViTL erfordert dagegen keine derartige Auftrennung der Transformationsregeln anhand mengenwertiger Strukturbeziehungen. Bei Verankerung der zielseitig erzeugten Elements auf die quellseitige Query würde durch die nachgelagerte Verklebung wie erwünscht nur ein einzelnes Muster verbleiben. SViTL-Paare können daher *eins–zu–n*-Beziehung zusammen abbilden, anstatt eine Auftrennung in mehrere TGG-Regeln zu erfordern. Dadurch wird die Anzahl der benötigten Regeln reduziert.

Da der TGG-Ansatz keine spezielle Zielsprache festlegt und somit deren Semantik einbeziehen kann, müssen *Behälterzuordnungen* stets explizit spezifiziert werden. Jede Transformationsregel setzt neu erzeugte oder aus dem Kontext bekannte Behälter mit den erzeugten Elementen in Beziehung. In SViTL werden Hierarchien der erzeugten DRAGULA Elemente insoweit berücksichtigt, als dass Behälterinformationen implizit entsprechend einfacher Heuristiken propagiert werden. Bei der Modellierung von SViTL-Paaren kann der zu wählende Behälter also oftmals ignoriert werden, wodurch SViTL-Paare im Vergleich zu TGG-Regeln kompakter und einfacher ausfallen.

Wie [KS06a] feststellt lassen sich Ansätze wie SViTL nicht für *interaktive* Transformationen einsetzen, bei denen ein Benutzer bspw. die anzuwendenden Grammatikregeln beeinflusst. Da ein SViTL-Paar – im Unterschied zu TGGs – keine Kontextinformation beinhaltet kann dem Benutzer keine geeignete Hilfestellung bei der Wahl zwischen möglichen Alternativen gegeben werden. Hierdurch wird die Anwendbarkeit SViTLs im gegebenen Szenario nicht eingeschränkt da ein Übersetzer eindeutige Ergebnisse erzeugen sollte die keine benutzerabhängigen Entscheidungen erfordern.

6.9.2 Query/View/Transformation

Unter den industriellen Standards liegt die OMG Spezifikation Query/View/Transformation (vgl. Abschnitt 2.3.2 auf Seite 39) am nächsten an der Arbeitsweise von SViTL. Die hier anzubringenden Differenzierungsargumente entsprechen im Wesentlichen denen zu TGGs, was sich insbesondere auf die relationale Teilsprache QVT Relational bezieht [GK07].

QVT Relational erlaubt im Vergleich zu TGGs eine stärkere Formulierung von Anwendbarkeitsbedingungen anhand der OCL und somit der Prädikatenlogik erster Stu-

fe. Hierüber lassen sich komplexere Strukturen beschreiben als einfache Graphmuster (einschließlich Negation) in SViTL.

6.9.3 BOTL

Die *Bidirectional Object Transformation Language* [BM03] beschreibt einen Transformationsansatz, bei dem Ausschnitte des Quell- und Zielmodells paarweise in Relation gesetzt werden. Wie auch in SViTL werden partielle Graphstrukturen miteinander *verklebt* um die Transformationsverarbeitung reihenfolgeunabhängig gestalten zu können. In BOTL wird dazu die Existenz eines eindeutigen Objektbezeichners vorausgesetzt, wohingegen SViTL Verankerungen in Bezug auf die gewählte quellseitige Auffindungsstelle einführt.

BOTL-Regeln sind im Gegensatz zu SViTL-Paaren unter bestimmten Einschränkungen bidirektional auswertbar. Hierzu zählt u.a. die Forderung, dass zielseitig verwendeten Verbindungstypen eindeutig eine Übersetzerregel zugeordnet werden kann. Da diese Einschränkung in praktischen Szenarien nicht realistisch durchsetzbar ist, stellt die angedachte Bidirektionalität dieses Ansatzes keinen praktischen Vorteil dar.

6.9.4 Deklarative Transformationen nach de Lara & Guerra

Wie auch SViTL stellt [LG08] einen deklarativen Spezifikationsansatz für Modellübersetzungen vor. De Lara und Guerra kritisieren an TGGs die Unterscheidung zwischen Kontext und Nicht-Kontextelementen, wodurch ein operationaler Charakter entsteht. Aus diesem Grund stellen sie einen stärker deklarativ ausgerichtetes Verfahren zur Beschreibung von Modelltransformationen vor. TGGs bieten zudem keine direkte Unterstützung für negative Anwendbarkeitsbedingungen, wodurch bestimmte Sachverhalte nur durch eine explizite Priorisierung von Regeln ausdrückbar sind. Hierzu ist anzumerken, dass [SK08] eine formal gestützte Erweiterung von TGGs für negative Kontexte vorstellt.

Der in [LG08] eingeführte und in [BGL09] weiter detaillierte Ansatz besteht darin, Ergebnisse einer Transformationsregel mit Kontexten anderer Regeln abzugleichen. Hierdurch ergeben sich implizite Reihenfolgeabhängigkeiten, die durch einen entsprechenden Kontrollmechanismus in ausführbare TGG-Regeln einbezogen werden können. Somit kann auf die explizite Spezifikation von Kontexten verzichtet werden, auch negative Bedingungen können somit leicht in die Spezifikationssprache aufgenommen werden. Wie SViTL auch ist dieser Ansatz jedoch zunächst rein unidirektional anwendbar.

Derartige deklarative Transformationen bieten jedoch nur insoweiteinen Vorteil gegenüber TGGs, als dass die syntaktische Unterscheidung zwischen Kontext- und

Nicht-Kontextelementen aufgehoben wird. Wie bei TGGs sind Regeln anhand von *eins–zu–n*-Beziehungen aufzuteilen, da der *zu–eins*-Anteil zunächst erzeugt werden muss, um im zweiten Regelteil auf diesen impliziten Kontext Bezug nehmen zu können. Ohne eine solche Aufteilung würde der *zu–eins*-Anteil für jede Auffindungsstelle des *zu–n*-Anteils erneut erzeugt werden.

6.10 Zusammenfassung

Dieses Kapitel hat einen Mechanismus vorgestellt um die Kernsprache für Graphtransformationen DRAGULA leicht zur Realisierung verhaltensorientierter Spezifikationssprachen nutzen zu können. Durch automatisierte Transformation der abstrakten Syntaxstruktur werden ausführbare DRAGULA Spezifikationen erzeugt, die das erwünschte Verhalten der Eingangssprache repräsentieren. Bezüglich der Anforderungen aus Kapitel 1 wird insbesondere der *Erhalt des Abstraktionsniveaus* (Anforderung 2) bedient, was durch eine hochsprachliche Beschreibung der Sprachabbildung erfolgt. Zudem bleibt die Lösung als Transformationsbeschreibung dauerhaft anpassbar, was der *Anpassbarkeit der Lösung* entspricht (Anforderung 3).

6.10.1 Weiterführende Funktionalität

Im Folgenden werden kurz einige mögliche Erweiterungspunkte der Übersetzersprache SViTL behandelt. Dabei handelt es sich um technisch motivierte Verbesserungen sowie alternative Beschreibungsformen.

Inkrementelle Übersetzung

Im bisherigen Stand arbeitet SViTL rein batch-artig und berücksichtigt insbesondere keine bereits übersetzten Teilfragmente einer Spezifikation. Grundsätzlich wäre eine *inkrementelle* Erweiterung des Ausführungsverhaltens allerdings durchaus realisierbar und insbesondere aus Effizienzgründen wünschenswert. Ein möglicher Ansatzpunkt stellt dabei die Nachverfolgung von *Änderungen* am Quelldokument dar, für die jeweils eine *Implikationsanalyse* erfolgt. So würden bei Ergänzung weiterer quellseitiger Konstrukte ggf. weitere Auffindungsstellen einzelner SViTL-Paare ermittelt werden, bei Verwendung negierter Quellelemente allerdings auch möglicherweise bereits verarbeitete Auffindungsstellen ungültig.

Im ersten Fall kann die Auswertung der neuen Auffindungsstellen nachgetragen werden, was eine anschließende Verklebeoperation und ggf. eine veränderte Behälterzuordnung von Elementen zur Folge hat. Für letzteren Fall sollten alle Elemente

der Nachbarschaft, die der Auswertung der ungültig gewordenen Auffindungsstelle zugeordnet sind, gelöscht werden. Da dies auch bereits verklebte Elemente betrifft, sind alle Auffindungsstellen, deren zugeordnete Nachbarschaften ein zu löschendes Element enthalten, neu zu verarbeiten. Auf diese Weise könnte die zielseitige DRAGULA-Struktur mit reduziertem Aufwand dahingehend verändert werden, dass diese einer vollständigen Übersetzung auf Basis des veränderten Ausgangsmodells entspricht.

Hilfestellung bei der Übersetzerbeschreibung

Derzeit enthält SViTL keinerlei Unterstützung für den Entwickler für den Entwurf konsistenter und vollständiger Übersetzerspezifikationen. Dadurch können Laufzeitprobleme auftreten, etwa falls identisch verankerte Elemente aufgrund ihrer Typinformationen nicht miteinander verklebt werden können. Ggf. könnten Fehler jedoch auch durch eine statische Analyse abgefangen und der Entwickler benachrichtigt werden.

Vollständige Spezifikationen setzen die Abbildung aller quellseitigen Konzepte auf äquivaltente Strukturen in DRAGULA voraus. Zwar können grundsätzlich einfache Abdeckungsanalysen vorgenommen werden, etwa ob jedes Syntaxelement und dessen Referenzen in wenigestens einem SViTL-Paar auftreten. Aufgrund der freien Formulierbarkeit der Eingabesprache kann eine vollständige Abdeckung durch die Übersetzerspezifikation jedoch nicht automatisiert festgestellt werden.

Grammatikinterne Übersetzerspezifikation

Ein alternativer Ansatz, um eine zu realisierende Spezifikationssprache auf DRAGULA abzubilden, liegt in einer integrativen Übersetzungsspezifikation zusammen mit der syntaktischen Sprachdefinition. Falls eine grammatikbasierte Sprachdefinition zum Einsatz kommt, könnte bspw. die Attributierung der Grammatik verwendet werden, um zu einer abstrakten Syntaxstruktur simultan korrespondierende DRAGULA-Fragmente zu erstellen. Vergleichbare Techniken sind im Compilerbau bereits seit langer Zeit verbreitet. Im Unterschied zum hier vorgestellten *separierten* Ansatz bestünde so die Möglichkeit zu einer kompakteren Spezifikation, die sowohl Syntax als auch eine zugehörige semantische Abbildung beinhaltet. Als Nachteilig könnte sich der hohe Komplexitätsgrad einer solchen Transformationsrealisierung erweisen, die durch SViTL als eigenständige Übersetzersprache verringert wird.

7 Zusammenfassung, Fazit und Ausblick

Dieses abschließende Kapitel fasst die wesentlichen Ergebnisse der vorliegenden Arbeit zusammen. Im Rahmen der Diskussion wird zunächst die Anwendbarkeit des Ansatzes überprüft und dessen Vor- sowie mögliche Nachteile aufgegriffen. Abschließend erfolgt ein Ausblick auf mögliche zukünftige Ergänzungen.

7.1 Zusammenfassung

Diese Arbeit hat den Bedarf aufgedeckt, die Realisierung operationaler Spezifikationssprachen effektiv zu unterstützen. Hierbei haben sich Anfragen, Graphtransformationen und Modellübersetzungen als wesentliche Anwendungsgebiete dieses Ansatzes herausgestellt. Die entsprechenden Anforderungen sind dabei in die weitere Entwicklung eingeflossen.

Durch Definition einer Kernsprache wird grundlegende Funktionalität für obige Anwendungsgebiete breitgestellt. Die Arbeit gibt eine vollständige syntaktische und semantische Definition der Kernsprache, die insbes. einem interpretationsfreien Verständnis der Sprachbeschreibung dient. Die Kernsprache ist zudem direkt im Hinblick auf Erweiterbarkeit ausgelegt worden, was die Integration von Spracherweiterungen erleichtert.

Eine Ausführungsmaschine abstrahiert vom technischen Speichersystem des Laufzeitgraphen und ist in der Lage, spezifische Vorteile eines Hintergrundspeichers für Graphtransformationen zu nutzen. Zudem bietet die Ausführungsmaschine eine implementierungsseitige Erweiterbarkeit zur Einbettung eigener Sprachkonstrukte.

Schlussendlich wird durch eine Übersetzersprache die Möglichkeit geschaffen, die Kernsprache und deren Ausführungsumgebung leicht nutzbar zu machen. Dabei werden bekannte Ergebnisse wie etwa TGGs zu einem neuartigen Ansatz kombiniert. Zusätzlich unterstützt die Übersetzersprache den gewählten Anwendungsfall durch Berücksichtigung der Kernsprache als Transformationsziel. Im Endeffekt kann die Übersetzung daher mit weniger Abbildungsregeln erfolgen als in herkömmlichen Ansätzen.

7.2 Fazit

Wesentliche Motivation dieser Forschungsarbeit ist die Schaffung einer allgemein nutzbaren Plattform für operationale Spezifikationssprachen. Daher wird im Folgenden zunächst dargestellt, für welche konkreten Sprachen die Ergebnisse bereits eingesetzt und welche Klassen von Spezifikationssprachen somit addressiert worden sind. Anschließend werden Vor- und Nachteile diskutiert.

7.2.1 Anwendbarkeit

Die Anwendbarkeit des vorgestellten Ansatzes ist in mehreren unterschiedlichen Szenarien nachgewiesen worden. In der vorliegenden Arbeit ist primär die Anfragesprache EMAA behandelt worden, was dem Anwendungsszenario aus Abschnitt 2.1.5 auf Seite 35 entspricht. EMAA wurde insbesondere aus dem Grund vertiefend behandelt, da die Sprache vom Umfang her gut darstellbar ist. Zudem erfordern Anfragen komplexere Attributausdrücke durch Graphtransformationsansätze als üblicherweise angeboten werden, somit stellt die Anfragesprache auch eine gewisse fachliche Herausforderung dar. Andererseits berücksichtigt EMAA Srachelemente wie Transformationen und Kontrollfluss nicht.

Als zweites Anwendungsszenario ist die Graphtransformationssprache PROGRES als Repräsentant programmierter Graphersetzung (vgl. Abschnitt 2.1.3) konzeptionell auf die DRAGULA Sprache und deren Erweiterungsmodule abgebildet worden. Dabei zeigt sich insbesondere für anfragende und transformierende Graphmuster eine leichte Abbildbarkeit. Durch seine Unterstützung zur nichtdeterministischen Auswertung von Transformationsregeln einschließlich Backtracking ist auch dieser Aspekt leicht auf DRAGULA übertragbar. Da DRAGULA jedoch als Kernsprache und Ausführungsplattform konzipiert worden ist, können einige strukturelle Aspekte jedoch nicht direkt umgesetzt werden. So fehlt insbesondere die Darstellung abgeleiteter Attribute, deren Werte durch Berechnungsfunktionen dargestellt werden. Technisch lässt sich dies jedoch leicht über den DRAGOS Graphspeicher nachliefern, etwa in dem bei Abfrage entsprechend markierter Attribute die DRAGULA Ausführungsmaschine angesprungen wird.

Architekturbezogene Aspekte, die eher geringen Einfluss auf die Laufzeitsemantik eines Graphtransformationssystems besitzen, werden ebenfalls nicht in DRAGULA dargestellt. Hierbei ist die Modularisierung von Winter [Win00] sowie die objektorientierte Erweiterung von Münch [Mü03] zu nennen. Aus Sicht der Sprachrealisierung können diese leicht simuliert oder durch kleinere Spracherweiterungen etwa zur polymorphen Überladung von Transformationen umgesetzt werden. In verteilten Applikationen agierende Graphtransformationssysteme [Ran08] können durch die aus-

tauschbare Sprachrealisierung ergänzt werden.

Das dritte Anwendungsszenario des vorgestellten Ansatzes stellt die Übersetzung von Graphstrukturen in Abschnitt 2.1.4 dar. Dazu wurde eine Übersetzung der Sprache SViTL unter Nutzung von SViTL-Konzepten vorgenommen und auf die Ausführungsplattform abgebildet. Zur Verklebung von Graphelementen kommt dabei eine spezifische Spracherweiterung zum Einsatz, die das in Kapitel 6 beschriebene Verhalten realisiert. Verankerungen werden als Nachverfolgbarkeitsinformationen über Kanten im Graphspeicher repräsentiert. Auf die gleiche Weise werden auch Nachbarschaften behandelt. Über einen Operator zur Verschiebung von Graphelementen wird schlussendlich die Behälterzuordnung realisiert. Durch die Möglichkeit zur Erweiterung der Kernsprache wird somit auch dieses Anwendungsszenario effektiv unterstützt.

7.2.2 Vorteile

Durch die vorliegende Arbeit wird eine Grundlage zur Realisierung operationaler Spezifikationssprachen auf hohem Abstraktionsniveau geschaffen. Als wesentlicher Vorteil des Ansatzes wird deren *Realisierungsaufwand* deutlich *reduziert*, da zahlreiche wiederkehrende Realisierungsaspekte wie etwa die Graphmustersuche nicht erneut implementiert werden müssen. Zudem stehen die bereitgestellten *Optimierungstechniken* allen Sprachrealisierungen zur Verfügung. Die Transformationsausführung ist zudem unabhängig vom verwendeten Hintergrundspeicher und ist daher *plattformunabhängig* bezüglich der Speicherlösung.

Die transformationsbasierte Übersetzung einer zu realisierenden Spezifikationssprache nach DRAGULA kann auf hohem Abstraktionsniveau beschrieben werden, wie Kapitel 6 gezeigt hat. Hierdurch kann eine vornehmliche konzeptorientierte Abbildung zwischen den Spezifikationssprachen und DRAGULA vorgenommen werden, bei der technische Fragestellungen zunächst unberücksichtigt bleiben können. Auf diese Weise kann die zu realisierende Sprache in großen Teilen unabhängig von DRAGULA entworfen werden. Somit schränkt die Strukturierung und angebotene Funktionalität der Kernsprache ihre Anwendbarkeit nicht auf äquivalente Syntaxformen etc. ein.

7.2.3 Nachteile

Als nachteilig an dem gewählten Ansatz kann die Beschränkung auf die Transformationsdefinition und -ausführung erachtet werden, da keine Visualisierungsfunktionalität oder andere Frontend-Unterstützung angeboten wird. Eine so realisierte Spezifikationssprache muss zur Verwendung also zusätzlich mit einer Benutzerschnittstelle

angeboten werden. Dies ist jedoch mit gängigen Rahmenwerken sowohl für textuelle als auch für visuelle Syntaxformen leicht möglich. Durch eine enge Integration mit etablierten Basistechnologien, wie etwa Eclipse EMF, ist eine solch gesonderte Lösung auch technisch einfach umsetzbar.

Durch den gewählten Ansatz wird ein eng umrissenes Feld von Spezifikations- oder domänenspezifischen Modellierungssprachen abgedeckt. Hierbei handelt es sich um Sprachen mit einer operationalen Laufzeitsemantik, die also Verarbeitungen von Laufzeitdaten beschreiben. Die in dieser Arbeit referenzierten Hauptkategorien der Anfragen, Graphtransformationen und Modellübersetzungen werden jedoch bereits durch eine Vielzahl verfügbarer Sprachbeschreibungen abgedeckt. Zusätzlich werden regelmäßig neue Vorschläge unterbreitet, bzw. entsteht in existierenden Implementierungen regelmäßig der Bedarf nach weiterer Funktionalität. Aus diesem Grund wird auch in Zukunft ein signifikanter Bedarf nach einer Basislösung bestehen bleiben.

7.3 Ausblick

Abschließend werden einige Ansatzpunkte vorgestellt, an denen die Ergebnisse der vorliegenden Arbeit weiter entwickelt werden könnten.

7.3.1 Kernsprache

Hinsichtlich der Kernsprache ließen sich verschiedene Konzepte existierender Graphtransformationssprachen in den Sprachkern integrieren bzw. als Erweiterungsmodule hinzufügen. Hierzu zählen die bereits erwähnten Konstrukte aus PROGRES zum polymorphen Überladen sowie abgeleitete Attribute. Konzeptionell könnte zudem eine integrierte Spezifikation von Spracherweiterungen sinnvoll sein, was entsprechend funktionaler Programmiersprachen wie LISP oder Ruby erfolgen könnte.

7.3.2 Ausführungsmaschine

Im Bereich der Ausführungsmaschine sind verschiedene Ansätze zur Weiterentwicklung denkbar. So ist insbesondere die Werkzeugunterstützung zum Debugging von Graphspezifikationen ein möglicher technischer Ansatzpunkt. Da hierzu bislang lediglich eine Programmierschnittstelle vorliegt, könnte durch einen Anschluss an die Visualisierung von DRAGULA-Spezifikationen ein graphisches Debugging ermöglicht werden.

Ein zweiter Aspekt liegt in der verbesserten Nutzung von SQL-basierten Hintergrundspeichern, was bislang nur für die Verarbeitung von Anfragen erfolgt. Überle-

genswert wäre, das vollständige spezifizierte System in SQL oder eine vergleichbare technische Darstellung zu überführen. Hierzu wäre allerdings ein größerer Umbau des DRAGOS-Systems notwendig, bspw. um analog zum Ansatz von Varró einen stärkeren Nutzen aus der Tabellenstrukturierung einer SQL-Datenbank zu ziehen. Eine solche Lösung würde zudem eigene Optimierungsansätze für Graphtransformationen erfordern.

7.3.3 Übersetzerbau

Obwohl SViTL als Übersetzersprache für den spezifischen Anwendungsfall dieser Arbeit entworfen worden ist, steht ihrer breiten Anwendbarkeit nichts im Wege. Hierfür sollte die implementierungstechnische Bindung an die DRAGULA-Sprache aufgehoben werden, was jedoch kein konzeptionelles Problem darstellt. Zudem würden die angewandten Heuristiken, etwa zur Behälterzuordnung oder zur Zuweisung von Typen bei mehrdeutigen Informationen, verallgemeinert bzw. konfigurierbar gestaltet werden.

Technisch ist wiederrum die Werkzeugunterstützung zu nennen, die bisher prototypisch in Form eines visuellen Editors für Spezifikationen und einer Übersetzungskomponente nach DRAGULA vorliegt. Hier wären wiederrum das Debugging bzw. das Aufzeigen potenzieller Spezifikationsfehler hilfreiche Ergänzungen.

Schlussbemerkung

Mit der vorliegenden Arbeit ist ein Realisierungsansatz für operationale Spezifikationssprachen vorgestellt worden. Abseits der konkreten technischen Ausprägung, hier auf Basis von Graphtransformationen und Graphdatenbanken, wird dabei auch ein konzeptioneller Beitrag im Bereich operationaler Spezifikationssprachen erbracht.

Die Erfahrung in der Softwaretechnik zeigt, dass sich langfristig höhere Abstraktionsniveaus als neuer Schwerpunkt der Entwicklungstätigkeit etablieren. Hierzu müssen auch Spezifikationssprachen bereitgestellt werden können, die mit operationalen Ausdrucksmitteln das Verhalten eines Systems beschreiben. Zugleich bewirkt ein gesteigertes Abstraktionsniveau eine stärkere Spezialisierung von Lösungen auf spezifische Domänen, was eine größere Diversität fördert. Ein Schlüssel, um diese handhaben zu können, ist die kostengünstige Realisierung von Spezifikationssprachen. Die vorliegende Arbeit kann im Hinblick auf deren operationalen Anteile einen Ansatz liefern.

Literaturverzeichnis

[ACS90] Angelaccio, Michele ; Catarci, Tiziana ; Santucci, Giuseppe: QBD*: A Graphical Query Language with Recursion. In: *IEEE Transactions on Software Engineering* 16 (1990), Nr. 10, S. 1150–1163

[AEH+99] Andries, Marc ; Engels, Gregor ; Habel, Annegret ; Hoffmann, Berthold ; Kreowski, Hans-Jörg ; Kuske, Sabine ; Plump, Detlef ; Schürr, Andy ; Taentzer, Gabriele: Graph Transformation for Specification and Programming. In: *Science of Computer Programming* 34 (1999), Nr. 1, S. 1–54

[AKRS06] Amelunxen, Carsten ; Königs, Alexander ; Rötschke, Tobias ; Schürr, Andy: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In: Rensink, Arend (Hrsg.) ; Warmer, Jos (Hrsg.): *ECMDA-FA* Bd. 4066, Springer, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–35909–5, S. 361–375

[Amb03] Ambler, Scott: *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. Wiley, 2003. – ISBN 0–471–20283–5

[Apa10] Apache Foundation (Hrsg.): *Apache Derby Project*. : Apache Foundation, Juli 2010. http://db.apache.org/derby/

[ASM80] Abrial, Jean-Raymond ; Schuman, Stephen A. ; Meyer, Bertrand: Specification Language. In: [MM80], S. 343–410

[BA99] Bertrand, Frédéric ; Augeraud, Michel: BDL: A Specialized Language for Per-Object Reactive Control. In: *IEEE Transactions on Software Engineering* 25 (1999), Nr. 3, S. 347–362

[Baa02] Baar, Thomas: *Über die Semantikbeschreibung OCL-artiger Sprachen*, Humboldt-University Berlin, Diss., 2002

[Baa05] Baar, Thomas: Non-deterministic Constructs in OCL - What Does any() Mean. In: Prinz, Andreas (Hrsg.) ; Reed, Rick (Hrsg.) ; Reed, Jeanne

(Hrsg.): *SDL Forum* Bd. 3530, Springer, 2005 (Lecture Notes in Computer Science). – ISBN 3–540–26612–7, S. 32–46

[Bau99] Baumann, Roland: *Ein Datenbankmanagementsystem für verteilte, integrierte Software-Entwicklungsumgebungen*, RWTH Aachen University, Diss., 1999

[BFH87] Boehm, Paul ; Fonio, Harald-Reto ; Habel, Annegret: Amalgamation of Graph Transformations: A Synchronization Mechanism. In: *Journal of Computer System Science* 34 (1987), Nr. 2/3, S. 377–408

[BGL09] Bottoni, Paolo ; Guerra, Esther ; Lara, Juan de: Formal Foundation for Pattern-Based Modelling. In: Chechik, Marsha (Hrsg.) ; Wirsing, Martin (Hrsg.): *FASE* Bd. 5503, Springer, 2009 (Lecture Notes in Computer Science). – ISBN 978–3–642–00592–3, S. 278–293

[BHLW07] Becker, Simon M. ; Herold, Sebastian ; Lohmann, Sebastian ; Westfechtel, Bernhard: A graph-based algorithm for consistency maintenance in incremental and interactive integration tools. In: *Software and System Modeling* 6 (2007), Nr. 3, S. 287–315

[BHRV08] Bergmann, Gábor ; Horváth, Ákos ; Ráth, István ; Varró, Dániel: A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation. In: [EHRT08], S. 396–410

[BJ06] Bézivin, Jean ; Jouault, Frédéric: Using ATL for Checking Models. In: *Electronic Notes in Theoretical Computer Science* 152 (2006), S. 69–81

[BJSW02] Böhlen, Boris ; Jäger, Dirk ; Schleicher, Ansgar ; Westfechtel, Bernhard: UPGRADE: A Framework for Building Graph-Based Interactive Tools. In: *Electronic Notes in Theoretical Computer Science* 72 (2002), Nr. 2

[BM03] Braun, Peter ; Marschall, Frank: Transforming Object Oriented Models with BOTL. In: *Electronic Notes in Theoretical Computer Science* 72 (2003), Nr. 3

[BN99] Baader, Franz ; Nipkow, Tobias: *Term Rewriting and All That*. Cambridge University Press, 1999. – ISBN 0–521–77920–0

[Bro03] Broekstra, Jeen: SeRQL: Sesame RDF Query Language. In: Ehrig, M. (Hrsg.) ; Haase, P. (Hrsg.) ; Tempich, C. (Hrsg.): *SWAP Deliverable 3.2 Method Design*. 2003, S. 55–68

[Bro04] Brown, Alan W.: Model driven architecture: Principles and practice. In: *Software and System Modeling* 3 (2004), Nr. 4, S. 314–327

[BS03] Börger, Egon ; Stärk, Robert F.: *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003

[BV06] Balogh, András ; Varró, Dániel: Advanced model transformation language constructs in the VIATRA2 framework. In: Haddad, Hisham (Hrsg.): *SAC*, ACM, 2006. – ISBN 1–59593–108–2, S. 1280–1287

[BW02] Brucker, Achim D. ; Wolff, Burkhart: A Proposal for a Formal OCL Semantics in Isabelle/HOL. In: Carreño, Victor (Hrsg.) ; Muñoz, César (Hrsg.) ; Tahar, Sofiène (Hrsg.): *TPHOLs* Bd. 2410, Springer, 2002 (Lecture Notes in Computer Science). – ISBN 3–540–44039–9, S. 99–114

[Bö06] Böhlen, Boris: *Ein parametrisierbares Graph-Datenbanksystem für Entwicklungswerkzeuge*, RWTH Aachen University, Diss., 2006

[CAS] microTOOL (Hrsg.): *case/4/0*. : microTOOL, http://www.case40.de/

[CEER96] Cuny, Janice E. (Hrsg.) ; Ehrig, Hartmut (Hrsg.) ; Engels, Gregor (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Graph Gramars and Their Application to Computer Science, 5th International Workshop, Williamsburg, VA, USA, November 13-18, 1994, Selected Papers*. Bd. *1073*. Springer, 1996 (Lecture Notes in Computer Science). – ISBN 3–540–61228–9

[CEKR02] Corradini, Andrea (Hrsg.) ; Ehrig, Hartmut (Hrsg.) ; Kreowski, Hans-Jörg (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*. Bd. *2505*. Springer, 2002 (Lecture Notes in Computer Science). – ISBN 3–540–44310–X

[CEL+94] Corradini, Andrea ; Ehrig, Hartmut ; Löwe, Michael ; Montanari, Ugo ; Padberg, Julia: The Category of Typed Graph Grammars and its Adjunctions with Categories. In: [CEER96], S. 56–74

[CEM+06] Corradini, Andrea (Hrsg.) ; Ehrig, Hartmut (Hrsg.) ; Montanari, Ugo (Hrsg.) ; Ribeiro, Leila (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*. Bd. *4178*. Springer, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–38870–2

[CFP+04] Cortes, Corinna ; Fisher, Kathleen ; Pregibon, Daryl ; Rogers, Anne ; Smith, Frederick: Hancock: A language for analyzing transactional data streams. In: *ACM Transactions on Programming Languages and Systems* 26 (2004), Nr. 2, S. 301–338

[CH06] Czarnecki, Krzysztof ; Helsen, Simon: Feature-based survey of model transformation approaches. In: *IBM Systems Journal* 45 (2006), Nr. 3, S. 621–646

[Che76] Chen, Peter P.: The Entity-Relationship Model – Toward a Unified View of Data. In: *ACM Transactions on Database Systems* 1 (1976), Nr. 1, S. 9–36

[CHM+02] Csertán, György ; Huszerl, Gábor ; Majzik, István ; Pap, Zsigmond ; Pataricza, András ; Varró, Dániel: VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. In: *ASE*, IEEE Computer Society, 2002. – ISBN 0–7695–1736–6, S. 267–270

[CMR+97] Corradini, Andrea ; Montanari, Ugo ; Rossi, Francesca ; Ehrig, Hartmut ; Heckel, Reiko ; Löwe, Michael: Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach. In: [Roz97], S. 163–246

[DDH72] Dahl, Ole J. ; Dijkstra, Edsger W. ; Hoare, Charles Antony R.: *Structured programming*. London, UK : Academic Press Limited, 1972. – ISBN 0–12–200550–3

[DHJ+06] Drewes, Frank ; Hoffmann, Berthold ; Janssens, Dirk ; Minas, Mark ; Eetvelde, Niels V.: Adaptive Star Grammars. In: [CEM+06], S. 77–91

[DHJ+07] Drewes, Frank ; Hoffmann, Berthold ; Janssens, Dirk ; Minas, Mark ; Eetvelde, Niels van: Shaped Generic Graph Transformation. In: [SNZ08], S. 201–216

[DHT96] Douglass, Bruce P. ; Harel, David ; Trakhtenbrot, Mark B.: Statecharts in Use: Structured Analysis and Object-Orientation. In: Rozenberg, Grzegorz (Hrsg.) ; Vaandrager, Frits W. (Hrsg.): *European Educational Forum: School on Embedded Systems* Bd. 1494, Springer, 1996 (Lecture Notes in Computer Science). – ISBN 3–540–65193–4, S. 368–394

[Dij59] Dijkstra, Edsger W.: A note on two problems in connexion with graphs. In: *Numerische Mathematik* 1 (1959), S. 269–271

[dV02] de Lara, Juan ; Vangheluwe, Hans: AToM^3: A Tool for Multi-formalism and Meta-modelling. In: Kutsche, Ralf-Detlef (Hrsg.) ; Weber, Herbert (Hrsg.): *FASE* Bd. 2306, Springer, 2002 (Lecture Notes in Computer Science). – ISBN 3–540–43353–8, S. 174–188

[Dö95] Dörr, Heiko: *Efficient Graph Rewriting and Its Implementation*, Freie Universität Berlin, Diss., 1995

[EBN96] Norm ISO/IEC 14977:1996 1996. *Extended BNF*

[Edm67] Edmonds, Jack: Optimum Branchings. In: *Journal of Research of the National Bureau of Standards* 71b (1967), S. 233–240

[EEHT05] Ehrig, Karsten ; Ermel, Claudia ; Hänsgen, Stefan ; Taentzer, Gabriele: Generation of visual editors as Eclipse plug-ins. In: Redmiles, David F. (Hrsg.) ; Ellman, Thomas (Hrsg.) ; Zisman, Andrea (Hrsg.): *ASE*, ACM, 2005, S. 134–143

[EEKR99] Ehrig, Hartmut (Hrsg.) ; Engels, Gregor (Hrsg.) ; Kreowski, Hans-Jörg (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages and Tools*. World Scientific, 1999 . – ISBN 9810240201

[EEKR00] Ehrig, Hartmut (Hrsg.) ; Engels, Gregor (Hrsg.) ; Kreowski, Hans-Jörg (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Theory and Application of Graph Transformations, 6th International Workshop, TAGT'98, Paderborn, Germany, November 16-20, 1998, Selected Papers*. Bd. *1764*. Springer, 2000 (Lecture Notes in Computer Science). – ISBN 3–540–67203–6

[EEPT06] Ehrig, Hartmut ; Ehrig, Karsten ; Prange, Ulrike ; Taentzer, Gabriele: *Fundamentals of Algebraic Grpah Transformation*. Springer, 2006. – ISBN 3–5403–1187–4

[EHK+97] Ehrig, Hartmut ; Heckel, Reiko ; Korff, Martin ; Löwe, Michael ; Ribeiro, Leila ; Wagner, Annika ; Corradini, Andrea: Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In: [Roz97], S. 247–312

[EHKPP90] Ehrig, Hartmut ; Habel, Annegret ; Kreowski, Hans-Jörg ; Parisi-Presicce, Francesco: From Graph Grammars to High Level Replacement Systems. In: [EKR91], S. 269–291

[EHL+98] Ehrig, Hartmut ; Heckel, Reiko ; Llabrés, Mercè ; Orejas, Fernando ; Padberg, Julia ; Rozenberg, Grzegorz: Double-Pullback Graph Transitions: A Rule-Based Framework with Incomplete Information. In: [EEKR00], S. 85–102

[EHPP04] Ehrig, Hartmut ; Habel, Annegret ; Padberg, Julia ; Prange, Ulrike: Adhesive High-Level Replacement Categories and Systems. In: Ehrig, Hartmut (Hrsg.) ; Engels, Gregor (Hrsg.) ; Parisi-Presicce, Francesco (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *ICGT* Bd. 3256, Springer, 2004 (Lecture Notes in Computer Science). – ISBN 3–540–23207–9, S. 144–160

[EHRT08] Ehrig, Hartmut (Hrsg.) ; Heckel, Reiko (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.) ; Taentzer, Gabriele (Hrsg.): *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings*. Bd. *5214*. Springer, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–540–87404–1

[EJS88] Engels, Gregor ; Janning, Thorsten ; Schäfer, Wilhelm: A Highly Integrated Tool Set for Program Development Support. In: *SIGSMALL/PC*, 1988, S. 1–10

[EKR91] Ehrig, Hartmut (Hrsg.) ; Kreowski, Hans-Jörg (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Graph-Grammars and Their Application to Computer Science, 4th International Workshop, Bremen, Germany, March 5-9, 1990, Proceedings*. Bd. *532*. Springer, 1991 (Lecture Notes in Computer Science). – ISBN 3–540–54478–X

[EKRW02] Ebert, Jürgen ; Kullbach, Bernt ; Riediger, Volker ; Winter, Andreas: GUPRO - Generic Understanding of Programs. In: *Electronic Notes in Theoretical Computer Science* 72 (2002), Nr. 2

[ELS86] Engels, Gregor ; Lewerentz, Claus ; Schäfer, Wilhelm: Graph Grammar Engineering: A Software Specification Method. In: [ENRR87], S. 186–201

[EN94] Elmasri, Ramez ; Navathe, Shamkant B.: *Fundamentals of Database Systems, 2nd Edition*. Benjamin/Cummings, 1994. – ISBN 0–8053–1748–1

[ENRR87] Ehrig, Hartmut (Hrsg.) ; Nagl, Manfred (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.) ; Rosenfeld, Azriel (Hrsg.): *Graph-Grammars and Their Application to Computer Science, 3rd International Workshop, Warrenton, Virginia, USA, December 2-6, 1986*. Bd. *291*. Springer, 1987 (Lecture Notes in Computer Science). – ISBN 3–540–18771–5

[EOSW07] Engels, Gregor (Hrsg.) ; Opdyke, Bill (Hrsg.) ; Schmidt, Douglas C. (Hrsg.) ; Weil, Frank (Hrsg.): *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*. Bd. *4735*. Springer, 2007 (Lecture Notes in Computer Science). – ISBN 978–3–540–75208–0

[Epp99] Eppstein, David: Subgraph Isomorphism in Planar Graphs and Related Problems. In: *Graph Algorithms & Applications* 3 (1999), Nr. 3, S. 1–27

[EPS73] Ehrig, Hartmut ; Pfender, Michael ; Schneider, Hans J.: Graph-Grammars: An Algebraic Approach. In: *FOCS*, IEEE, 1973, S. 167–180

[ESU97] Ebert, Jürgen ; Süttenbach, Roger ; Uhe, Ingar: Meta-CASE in Practice: a Case for KOGGE. In: Olivé, Antoni (Hrsg.) ; Pastor, Joan A. (Hrsg.): *CAiSE* Bd. 1250, Springer, 1997 (Lecture Notes in Computer Science). – ISBN 3–540–63107–0, S. 203–216

[FHH03] Fikes, Richard ; Hayes, Pat ; Horrocks, Ian: OWL-QL: A Language for Deductive Query Answering on the Semantic Web / Stanford University. 2003 (KSL-03-14). – Forschungsbericht

[FNTZ98] Fischer, Thorsten ; Niere, Jörg ; Torunski, Lars ; Zündorf, Albert: Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In: [EEKR00], S. 296–309

[For79] Forgy, Charles L.: *On the efficient implementation of production systems*, Carnegie Mellon University Pittsburgh, PA, USA, Diss., 1979

[Fow03] Fowler, Martin: *UML Distilled*. Addison-Wesley Professional, 2003. – ISBN 0–321–19368–7

[GA02] Garland, Jeff ; Anthony, Richard: *Large-Scale Software Architecture: A Practical Guide using UML*. New York, NY, USA : John Wiley & Sons, Inc., 2002. – ISBN 0–470–84849–9

[GBG+06] Geiss, Rubino ; Batz, Gernot V. ; Grund, Daniel ; Hack, Sebastian ; Szalkowski, Adam: GrGen: A Fast SPO-Based Graph Rewriting Tool. In: [CEM+06], S. 383–397

[Gd07] Guerra, Esther ; de Lara, Juan: Adding Recursion to Graph Transformation. In: *Electronic Communications of the EASST* 6 (2007)

[GHJV94] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, 1994

[GHS09] Giese, Holger ; Hildebrandt, Stephan ; Seibel, Andreas: Improved Flexibility and Scalability by Interpreting Story Diagrams. In: *Electronic Communications of the EASST* 18 (2009)

[GK07] Greenyer, Joel ; Kindler, Ekkart: Reconciling TGGs with QVT. In: [EOSW07], S. 16–30

[GNS97] Gruner, Stefan ; Nagl, Manfred ; Schürr, Andy: Integration Tools Supporting Development Processes. In: Broy, Manfred (Hrsg.) ; Rumpe, Bernhard (Hrsg.): *Requirements Targeting Software and Systems Engineering* Bd. 1526, Springer, 1997 (Lecture Notes in Computer Science). – ISBN 3–540–65309–0, S. 235–256

[GSR05] Geiger, Leif ; Schneider, Christian ; Reckord, Carsten: Template- and modelbased code generation for MDA-Tools. In: Giese, Holger (Hrsg.) ; Zündorf, A. (Hrsg.): *Proceedings of the Fujaba Days 2005* Bd. tr-ri-05-259, University of Paderborn, Germany, 2005 (Technical Report)

[HHT96] Habel, Annegret ; Heckel, Reiko ; Taentzer, Gabriele: Graph Grammars with Negative Application Conditions. In: *Fundamenta Informaticae* 26 (1996), Nr. 3/4, S. 287–313

[HJG08] Hoffmann, Berthold ; Jakumeit, Edgar ; Geiss, Rubino: Graph Rewrite Rules with Structural Recursion. In: Mosbah, Mohamed (Hrsg.) ; Habel, Annegret (Hrsg.): *Workshop on Graph Computational Models*, University of Leicester, 2008, S. 5–16

[HJK+09] Heidenreich, Florian ; Johannes, Jendrik ; Karol, Sven ; Seifert, Mirko ; Wende, Christian: Derivation and Refinement of Textual Syntax for Models. In: Paige, Richard F. (Hrsg.) ; Hartman, Alan (Hrsg.) ; Rensink, Arend (Hrsg.): *ECMDA-FA* Bd. 5562, Springer, 2009 (Lecture Notes in Computer Science). – ISBN 978–3–642–02673–7, S. 114–129

[HJS+04] Heller, Markus ; Jäger, Dirk ; Schlüter, Marcus ; Schneider, Ralph ; Westfechtel, Bernhard: A management system for dynamic and inter-organizational design processes in chemical engineering. In: *Computers & Chemical Engineering* 29 (2004), Nr. 1, S. 93–111

[HK86] Habel, Annegret ; Kreowski, Hans-Jörg: May we introduce to you: hyperedge replacement. In: [ENRR87], S. 15–26

[HKSS08] Habela, Piotr ; Kaczmarski, Krzysztof ; Stencel, Krzysztof ; Subieta, Kazimierz: OCL as the Query Language for UML Model Execution. In: Bubak, Marian (Hrsg.) ; Albada, G. D. (Hrsg.) ; Dongarra, Jack (Hrsg.) ; Sloot, Peter M. A. (Hrsg.): *ICCS (3)* Bd. 5103, Springer, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–540–69388–8, S. 311–320

[HKT02] Heckel, Reiko ; Küster, Jochen M. ; Taentzer, Gabriele: Confluence of Typed Attributed Graph Transformation Systems. In: [CEKR02], S. 161–176

[HL89] Horebeek, Ivo van ; Lewi, Johan: *Algebraic Specifications in Software Engineering*. Springer, 1989

[HM00] Hoffmann, Berthold ; Minas, Mark: Towards rule-based visual programming of generic visual systems. In: *CoRR* cs.PL/0010016 (2000)

[Hoa80] Hoare, Charles Antony R.: A Model for Communicating Sequential Processes. In: [MM80], S. 229–254

[HVV07] Horváth, Ákos ; Varró, Gergely ; Varró, Dániel: Generic Search Plans for Matching Advanced Graph Patterns. In: *Electronic Communications of the EASST* 6 (2007)

[HWS00] Holt, Richard ; Winter, Andreas ; Schürr, Andy: GXL: Towards a Standard Exchange Format. In: *Reverse Engineering*, IEEE Computer Society, 2000. – ISBN 0–7695–0881–2, S. 162–171

[Jah99] Jahnke, Jens-Holger: *Management of Uncertainty and Inconsistency in Database Reengineering Processes*, University of Paderborn, Diss., 1999

[KASS03] Karsai, Gabor ; Agrawal, Aditya ; Shi, Feng ; Sprinkle, Jonathan: On the Use of Graph Transformation in the Formal Specification of Model Interpreters. In: *Journal of Universal Computer Science* 9 (2003), Nr. 11, S. 1296–1321

[KGKK02] Kuske, Sabine ; Gogolla, Martin ; Kollmann, Ralf ; Kreowski, Hans-Jörg: An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In: Butler, Michael J. (Hrsg.) ; Luigia Petre and, Kaisa S. (Hrsg.): *IFM* Bd. 2335, Springer, 2002 (Lecture Notes in Computer Science). – ISBN 3–540–43703–7, S. 11–28

[KKH05] Kline, Kevin ; Kline, Daniel ; Hunt, Brand: *SQL In A Nutshell, 2nd Edition*. O'Reilly, 2005

[Kle67] Kleene, Stephen C.: *Mathematical logic*. New York, US : Wiley, 1967

[KLN96] Kohring, Christine ; Lefering, Martin ; Nagl, Manfred: A Requirements Engineering Environment within a Tightly-Integrated SDE. In: *Requirements Engineering* 1 (1996), Nr. 3

[KLR96] Kelly, Steven ; Lyytinen, Kalle ; Rossi, Matti: MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In: Constantopoulos, Panos (Hrsg.) ; Mylopoulos, John (Hrsg.) ; Vassiliou, Yannis (Hrsg.): *CAiSE* Bd. 1080, Springer, 1996 (Lecture Notes in Computer Science). – ISBN 3–540–61292–0, S. 1–21

[Knu64] Knuth, Donald E.: Backus Normal Form vs. Backus Naur form. In: *Communications of the ACM* 7 (1964), Nr. 12, S. 735–736

[KRV07] Krahn, Holger ; Rumpe, Bernhard ; Völkel, Steven: Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In: [EOSW07], S. 286–300

[KS06a] Königs, Alexander ; Schürr, Andy: MDI: A Rule-based Multi-document and Tool Integration Approach. In: *Software and Systems Modeling* 5 (2006), Nr. 4, S. 349–368

[KS06b] Königs, Alexander ; Schürr, Andy: Tool Integration with Triple Graph Grammars - A Survey. In: *Electronic Notes in Theoretical Computer Science* 148 (2006), Nr. 1, S. 113–150

[KSW95] Kiesel, Norbert ; Schürr, Andy ; Westfechtel, Bernhard: GRAS, a Graph-Oriented (Software) Engineering Database System. In: *Information Systems* 20 (1995), Nr. 1, S. 21–51

[KW99] Kullbach, Bernt ; Winter, Andreas: Querying as an Enabling Technology in Software Reengineering. In: *CSMR*, IEEE Computer Society, 1999. – ISBN 0–7695–0090–0, S. 42–50

[KWB03] Kleppe, Anneke ; Warmer, Jos ; Bast, Wim: *MDA explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003. – ISBN 0–321–19442–X

[LBE+07] Lara, Juan de ; Bardohl, Roswitha ; Ehrig, Hartmut ; Ehrig, Karsten ; Prange, Ulrike ; Taentzer, Gabriele: Attributed graph transformation with node type inheritance. In: *Theorertical Computer Science* 376 (2007), Nr. 3, S. 139–163

[LG08] Lara, Juan de ; Guerra, Esther: Pattern-Based Model-to-Model Transformation. In: [EHRT08], S. 426–441

[LLMC04] Levendovszky, Tihamér ; Lengyel, László ; Mezei, Gergely ; Charaf, Hassan: A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS. In: *Electronic Notes in Theoretical Computer Science* 127 (2004), Nr. 1, S. 65–75

[LLMC06] Lengyel, László ; Levendovszky, Tihamer ; Mezei, Gergely ; Charaf, Hassan: A Visual Control Flow Language for Model Transformation Systems. In: Kokol, Peter (Hrsg.): *IASTED Conf. on Software Engineering*, IASTED/ACTA Press, 2006. – ISBN 0–88986–574–4, S. 194–199

[Llo84] Lloyd, John W.: *Foundations of Logic Programming*. Springer, 1984

[LLP07] Lindqvist, Johan ; Lundkvist, Torbjörn ; Porres, Ivan: A Query Language With the Star Operator. In: Ehrig, Karsten (Hrsg.) ; Giese, Holger (Hrsg.): *Graph Transformation and Visual Modeling Techniques* Bd. 6, 2007 (Electronic Communications of EASST). – ISSN 1863–2122, S. 69–80

[LV02] Larrosa, Javier ; Valiente, Gabriel: Constraint Satisfaction Algorithms for Graph Pattern Matching. In: *Mathematical Structures in Computer Science* 12 (2002), Nr. 4, S. 403–422

[Mar78] Marco, Tom d.: *Structural Analysis and System Specification*. Yourdon, 1978. – ISBN 0–917–07207–3

[MBL+03] Magyari, Endre ; Bakay, Arpad ; Lang, Andras ; Paka, Tamas ; Vizhanyo, Attila ; Agarwal, Aditya ; Karsai, Gabor: UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages. In: 3^{rd} *OOPSLA Workshop on Domain-Specific Modeling*, 2003

[MBP+04] Moll, Karl-Rudolf ; Broy, Manfred ; Pizka, Markus ; Seifert, Tilman ; Bergner, Klaus ; Rausch, Andreas: Erfolgreiches Management von Software-Projekten. In: *Informatik Spektrum* 27 (2004), Nr. 5, S. 419–432

[Mey00] Meyer, Bertrand: Design By Contract and the Component Revolution. In: Li, Qizoyan (Hrsg.) ; Firesmith, Donald (Hrsg.) ; Riehle, Richard (Hrsg.) ; Meyer, Bertrand (Hrsg.): *TOOLS (34)*, IEEE Computer Society, 2000. – ISBN 0–7695–0774–3, S. 515–518

[Mis01] Misra, Jayadev: A walk over the shortest path: Dijkstra's Algorithm viewed as fixed-point computation. In: *Information Processing Letters* 77 (2001), Nr. 2-4, S. 197–200

[ML05] McAffer, Jeff ; Lemieux, Jean-Michel: *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications*. Addison-Wesley Professional, 2005. – ISBN 0–321–33461–2

[MM80] McKeag, R. M. (Hrsg.) ; Macnaughten, A. M. (Hrsg.): *On the Construction of Programs*. Cambridge University Press, 1980

[MML08] Mészáros, Tamás ; Mezei, Gergely ; Levendovszky, Tihamer: A flexible, declarative presentation framework for domain-specific modeling. In: Levialdi, Stefano (Hrsg.): *AVI*, ACM Press, 2008. – ISBN 978–1–60558–141–5, S. 309–312

[Mon05] Monnox, Alan: *Rapid J2EE Development: An Adaptive Foundation for Enterprise Applications*. Prentice Hall, 2005

[MW03] Marburger, André ; Westfechtel, Bernhard: Tools for Understanding the Behavior of Telecommunication Systems. In: *ICSE*, IEEE Computer Society, 2003, S. 430–443

[Mü03] Münch, Manfred: *Generic Modeling with Graph Rewriting Systems*, RWTH Aachen University, Diss., 2003

[Nag78] Nagl, Manfred: *Graph-Grammatiken: Theorie, Anwendungen, Implementierung*, University of Erlangen-Nürnberg, Habilitationsschrift, 1978

[Nag96] Nagl, Manfred (Hrsg.): *Building Tightly Integrated Software Development Environments: The IPSEN Approach*. Bd. *1170*. Springer, 1996 (Lecture Notes in Computer Science). – ISBN 3–540–61985–2

[NNZ00] Nickel, Ulrich ; Niere, Jörg ; Zündorf, Albert: The FUJABA environment. In: *ICSE*, ACM Press, 2000, S. 742–745

[NPW02] Nipkow, Tobias ; Paulson, Lawrence C. ; Wenzel, Markus: *Lecture Notes in Computer Science*. Bd. 2283: *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer, 2002. – ISBN 3–540–43376–7

[NS94] Nagl, Manfred ; Schürr, Andy: Software Integration Problems and Coupling of Graph Grammar Specifications. In: [CEER96], S. 155–169

[NWS03] Nagl, Manfred ; Westfechtel, Bernhard ; Schneider, Ralph: Tool support for the management of design processes in chemical engineering. In: *Computers & Chemical Engineering* 27 (2003), Nr. 2, S. 175–197

[oAW10] Eclipse Foundation (Hrsg.): *openArchitectureWare Xpand*. : Eclipse Foundation, Juli 2010. http://wiki.eclipse.org/Xpand

[OMG05] Norm ISO/IEC 19503:2005 2005. *XML Metadata Interchange*

[OMG06a] Object Management Group (Hrsg.): *Meta Object Facility 2.0 Specification*. : Object Management Group, Januar 2006. http://www.omg.org/spec/MOF/2.0/

[OMG06b] Object Management Group (Hrsg.): *Object Constraint Language 2.0 Specification*. : Object Management Group, Mai 2006. http://www.omg.org/spec/OCL/2.0/

[OMG08] Object Management Group (Hrsg.): *Query/View/Transformation 1.0 Specification*. : Object Management Group, April 2008. http://www.omg.org/spec/QVT/1.0/

[OMG10] Object Management Group (Hrsg.): *Unified Modeling Language 2.3 Superstructure Specification*. : Object Management Group, Mai 2010. http://www.omg.org/spec/UML/2.3/Superstructure/

[Plu09] Plump, Detlef: The Graph Programming Language GP. In: Bozapalidis, Symeon (Hrsg.) ; Rahonis, George (Hrsg.): *CAI* Bd. 5725, Springer, 2009 (Lecture Notes in Computer Science). – ISBN 978–3–642–03563–0, S. 99–122

[PNB04] Pfaltz, John L. (Hrsg.) ; Nagl, Manfred (Hrsg.) ; Böhlen, Boris (Hrsg.): *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers*. Bd. *3062*. Springer, 2004 (Lecture Notes in Computer Science). – ISBN 3–540–22120–4

[Pos10] PostgreSQL Community (Hrsg.): *PostgreSQL*. : PostgreSQL Community, Juli 2010. http://www.postgresql.org

[PR69] Pfaltz, John L. ; Rosenfeld, Azriel: Web Grammars. In: *IJCAI*, 1969, S. 609–620

[Ran08] Ranger, Ulrike I.: *Deklarative Modellierung von verteilten Systemen mit Graphersetzungssprachen*, RWTH Aachen University, Diss., 2008

[RBP+90] Rumbaugh, James ; Blaha, Michael ; Premerlani, William ; Eddy, Frederick ; Lorensen, William: *Object-Oriented Modeling and Design*. Prentice Hall, 1990. – ISBN 0–136–29841–9

[Ren03] Rensink, Arend: The GROOVE Simulator: A Tool for State Space Generation. In: [PNB04], S. 479–485

[Ren06] Rensink, Arend: Isomorphism Checking in GROOVE. In: *Electronic Communications of the EASST* 1 (2006)

[Ren09] Rensink, Arend: Repotting the Geraniums: On Nested Graph Transformation Rules. In: *Electronic Communications of the EASST* 18 (2009)

[RK08] Rensink, Arend ; Kleppe, Anneke: On a Graph-Based Semantics for UML Class and Object Diagrams. In: *Electronic Communications of the EASST* 10 (2008)

[Ros75] Rosen, Barry K.: A Church-Rosser theorem for graph grammars. In: *SIGACT News* 7 (1975), Nr. 3, S. 26–31. http://dx.doi.org/http://doi.acm.org/10.1145/1008343.1008344. – DOI http://doi.acm.org/10.1145/1008343.1008344. – ISSN 0163–5700

[Roz97] Rozenberg, Grzegorz (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997 . – ISBN 9810228848

[Rud98] Rudolf, Michael: Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In: [EEKR00], S. 238–251

[Rum04] Rumpe, Bernhard: *Modellierung mit UML: Sprache, Konzepte und Methodik*. Xpert.Press, 2004. – ISBN 3–540–20904–2

[RW07] Ranger, Ulrike ; Weinell, Erhard: The Graph Rewriting Language and Environment PROGRES. In: [SNZ08], S. 575–576

[SBPM08] Steinberg, David ; Budinsky, Frank ; Paternostro, Marcelo ; Merks, Ed: *EMF: Eclipse Modeling Framework*. Addison-Wesley Longman, 2008

[Sch70] Schneider, Hans J.: Chomsky-Systeme für partielle Ordnungen / Universität Erlangen. Erlangen, 1970 (3). – Forschungsbericht

[Sch90] Schürr, Andy: PROGRESS: A VHL-Language Based on Graph Grammars. In: [EKR91], S. 641–659

[Sch91] Schürr, Andy: *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*, RWTH Aachen University, Diss., 1991

[Sch94] Schürr, Andy: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, Ernst W. (Hrsg.) ; Schmidt, Gunther (Hrsg.) ; Tinhofer, Gottfried (Hrsg.): *WG* Bd. 903, Springer, 1994 (Lecture Notes in Computer Science). – ISBN 3–540–59071–4, S. 151–163

[Sch01] Schürr, Andy: Adding Graph Transformation Concepts to UML's Constraint Language OCL. In: *Electronic Notes in Theoretical Computer Science* 44 (2001), Nr. 4

[SE76] Schneider, Hans J. ; Ehrig, Hartmut: Grammars on partial graphs. In: *Acta Informatica* 6 (1976), Nr. 2, S. 297–316

[Sel03] Selic, Bran: The Pragmatics of Model-Driven Development. In: *IEEE Software* 20 (2003), Nr. 5, S. 19–25

[Sel06] Selic, Bran: Model-Driven Development: Its Essence and Opportunities. In: *ISORC*, IEEE Computer Society, 2006. – ISBN 0–7695–2561–X, S. 313–319

[SK03] Sendall, Shane ; Kozaczynski, Wojtek: Model Transformation: The Heart and Soul of Model-Driven Software Development. In: *IEEE Software* 20 (2003), Nr. 5, S. 42–45

[SK08] Schürr, Andy ; Klar, Felix: 15 Years of Triple Graph Grammars. In: [EHRT08], S. 411–425

[SNZ08] Schürr, Andy (Hrsg.) ; Nagl, Manfred (Hrsg.) ; Zündorf, Albert (Hrsg.): *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers*. Bd. *5088*. Springer, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–540–89019–5

[SPA08] Eric Prud'hommeaux AND Andy Seaborne (Hrsg.): *SPARQL Query Language for RDF*. : Eric Prud'hommeaux AND Andy Seaborne, Januar 2008. http://www.w3.org/TR/rdf-sparql-query

[SRB06] Schultchen, Erhard ; Ranger, Ulrike ; Böhlen, Boris: Graph-oriented Storage for Fujaba Applications. In: Giese, Holger (Hrsg.) ; Westfechtel, Bernhard (Hrsg.): *Proceedings of the Fujaba Days 2006* Bd. tr-ri-06-275, University of Paderborn, Germany, 2006 (Technical Report), S. 68–72

[SVEH07] Stahl, Thomas ; Völter, Markus ; Efftinge, Sven ; Haase, Arno: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Dpunkt Verlag, 2007. – ISBN 3–898–64448–0

[SWZ95] Schürr, Andy ; Winter, Andreas J. ; Zündorf, Albert: Graph Grammar Engineering with PROGRES. In: Schäfer, Wilhelm (Hrsg.) ; Botella, Pere (Hrsg.): *ESEC* Bd. 989, Springer, 1995 (Lecture Notes in Computer Science). – ISBN 3–540–60406–5, S. 219–234

[Tae94] Taentzer, Gabriele: Hierarchically Distributed Graph Transformation. In: [CEER96], S. 304–320

[Tae99] Taentzer, Gabriele: AGG: A Tool Environment for Algebraic Graph Transformation. In: Nagl, Manfred (Hrsg.) ; Schürr, Andy (Hrsg.) ; Münch, Manfred (Hrsg.): *AGTIVE* Bd. 1779, Springer, 1999 (Lecture Notes in Computer Science). – ISBN 3–540–67658–9, S. 481–488

[Tae03] Taentzer, Gabriele: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: [PNB04], S. 446–453

[TB93] Taentzer, Gabriele ; Beyer, Martin: Amalgamated Graph Transformations and Their Use for Specifying AGG - an Algebraic Graph Grammar System. In: Schneider, Hans J. (Hrsg.) ; Ehrig, Hartmut (Hrsg.): *Dagstuhl Seminar on Graph Transformations in Computer Science* Bd. 776, Springer, 1993 (Lecture Notes in Computer Science). – ISBN 3–540–57787–4, S. 380–394

[TEG+05] Taentzer, Gabriele ; Ehrig, Karsten ; Guerra, Esther ; Lara, Juan de ; Lengyel, Laszlo ; Levendovszky, Tihamer ; Prange, Ulrike ; Varró, Dániel ; Varró-Gyapay, Szilvia: Model Transformations by Graph Transformations: A Comparative Study. In: *Model Transformations in Practice Workshop at MoDELS 2005, Montego*, 2005

[VAB+07] Varró, Dániel ; Asztalos, Márk ; Bisztray, Dénes ; Boronat, Artur ; Dang, Duc-Hanh ; Geiss, Rubino ; Greenyer, Joel ; van Gorp, Pieter ; Kniemeyer, Ole ; Narayanan, Anantha ; Rencis, Edgars ; Weinell, Erhard: Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools. In: [SNZ08], S. 540–565

[van08] van Gorp, Pieter: *Model-driven Development of Model Transformations*, University of Antwerp, Diss., 2008

[Var02] Varró, Dániel: A Formal Semantics of UML Statecharts by Model Transition Systems. In: [CEKR02], S. 378–392

[Var08] Varró, Gergely: *Advanced Techniques for the Implementation of Model Transformation Systems*, Budapest University of Technology and Economics, Diss., 2008

[VB07] Varró, Dániel ; Balogh, András: The model transformation language of the VIATRA2 framework. In: *Science of Computer Programming* 68 (2007), Nr. 3, S. 214–234

[VFV06a] Varró, Gergely ; Friedl, Katalin ; Varró, Dániel: Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans. In: *Electr. Notes Theor. Comput. Sci.* 152 (2006), S. 191–205

[VFV06b] Varró, Gergely ; Friedl, Katalin ; Varró, Dániel: Implementing a Graph Transformation Engine in Relational Databases. In: *Software and System Modeling* 5 (2006), Nr. 3, S. 313–341

[VHV07] Varró, Gergely ; Horváth, Ákos ; Varró, Dániel: Recursive Graph Pattern Matching. In: [SNZ08], S. 456–470

[VP03] Varró, Dániel ; Pataricza, András: VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML (The Mathematics of Metamodeling is Metamodeling Mathematics). In: *Software and System Modeling* 2 (2003), Nr. 3, S. 187–210

[VP04] Varró, Dániel ; Pataricza, András: Generic and Meta-transformations for Model Transformation Engineering. In: Baar, Thomas (Hrsg.) ; Strohmeier, Alfred (Hrsg.) ; Moreira, Ana M. D. (Hrsg.) ; Mellor, Stephen J. (Hrsg.): *UML* Bd. 3273, Springer, 2004 (Lecture Notes in Computer Science). – ISBN 3–540–23307–5, S. 290–304

[VSV05] Varró, Gergely ; Schürr, Andy ; Varró, Dániel: Benchmarking for Graph Transformation. In: *VL/HCC*, IEEE Computer Society, 2005. – ISBN 0–7695–2443–5, S. 79–88

[VV04] Varró, Gergely ; Varró, Dániel: Graph Transformation with Incremental Updates. In: *Electronic Notes in Theoretical Computer Science* 109 (2004), S. 71–83

[VVS06] Varró, Gergely ; Varró, Dániel ; Schürr, Andy: Incremental Graph Pattern Matching: Data Structures and Initial Experiments. In: *Electronic Communications of the EASST* 4 (2006)

[Wei07] Weinell, Erhard: Adaptable Support for Queries and Transformations for the DRAGOS Graph-Database. In: [SNZ08], S. 394–409

[Wei08a] Weinell, Erhard: Extending Graph Query Languages by Reduction. In: *Electronic Communications of the EASST* 10 (2008)

[Wei08b] Weinell, Erhard: Transformation-Based Operationalization of Graph Languages. In: [EHRT08], S. 520–522

[Wei09] Weinell, Erhard: Visual compilation of behavioral modeling languages. In: Heckel, Reiko (Hrsg.) ; Boronat, Arthur (Hrsg.): *Pre-Proceedings of GT-VMT'09*, University of York, 2009. – Technical Report

[Wij74] Wijngaarden, Adriaan van: The Generative Power of Two-Level Grammars. In: Loeckx, Jacques (Hrsg.): *ICALP* Bd. 14, Springer, 1974 (Lecture Notes in Computer Science). – ISBN 3–540–06841–4, S. 9–16

[Win00] Winter, Andreas J.: *Visuelles Programmieren mit Graphtransformationen*, RWTH Aachen University, Diss., 2000

[Xte10] Eclipse Foundation (Hrsg.): *Xtext*. : Eclipse Foundation, Juli 2010. http://www.eclipse.org/Xtext

[Zün93] Zündorf, Albert: A Heuristic Solution for the Subgraph Isomorphism Problem in Executing PROGRES / RWTH Aachen University. 1993 (AIB 93-05). – Forschungsbericht

[Zün94] Zündorf, Albert: Graph Pattern Matching in PROGRES. In: [CEER96], S. 454–468

[Zü96] ZÜNDORF, Albert: *PROgrammierte GRaphErsetzungsSysteme*, RWTH Aachen University, Diss., 1996

[Zü01] ZÜNDORF, Albert: *Rigorous Object Oriented Software Development*, University of Paderborn, Habilitationsschrift, 2001