DRAGULA - Eine Anfrage- und Transformationssprache für graphbasierte Daten

Diplomarbeit
Aachen, 3. September 2008

vorgelegt am Lehrstuhl für Informatik 3, RWTH Aachen

vorgelegt von

Andrej Scheuermann

aus Georgiewka, UdSSR

Gutachter: Universitätsprofessor Dr.-Ing. M. Nagl
Universitätsprofessor Dr. rer.nat. O. Spaniol
Betreuer: Dipl.-Inform. E. Weinell

RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

Lehrstuhl für
Informatik 3

Department of
Computer Science 3

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 3. September 2008

Andrej Scheuermann

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

In dieser Arbeit wird eine Anfrage- und Transformationssprache für Graphen und graph-basierten Daten konzipiert. Durch die Sprache soll eine Grundlage zu Entwicklung Graphwerkzeuge geschafft werden. Ein besonderer Wert wird auf die Allgemeinheit und die Anpassungsfähigkeit der Sprache gelegt.

1.1 Motivation

Graphen als Datenstruktur haben bereits eine feste Position in unterschiedlichen technischen Anwendungen eingenommen. Entsprechend ist auch der Bedarf nach einer Werkzeugunterstützung von graph-basierten Systemen gestiegen.

Zum einen lassen sich Graphen einfach und übersichtlich grafisch veranschaulichen. Zum anderen lassen sich durch Graphen verschiedene Tatsachen der realen Welt modellieren: Knoten repräsentieren Objekte oder Fakten, wohingegen Kanten Zusammenhänge zwischen diesen ausdrücken. Durch Attribute können Knoten und Kanten mit zusätzlichen Daten versehen werden. Auf diese Weise lassen sich strukturierte Informationen durch Graphen darstellen. Die oben genannte Eigenschaften machen Graphen unabdingbar bei der Modellierung von unterschiedlichen Systemen.

Ein Beispiel dazu ist eine Datenverwaltung in einem Prozessmanagementsystem. In einem Graph werden Ressourcen, Dokumente und Abläufe zusammengefasst und verwaltet. Durch die Speicherung der Daten in Form von Graphen können Informationen über ein Projekt oder einen Prozess durch eine Anfrage an einen Graph gewonnen werden. Graphtransformationen realisieren Änderungen in einem Prozesszustand.

Speziell auf dem Gebiet der Softwareentwicklung werden Graphen intensiv zur Modellierungszwecken eingesetzt. Ein Beispiel dafür sind die Zustandsdiagramme, die oft in der eingebetteten Software angewendet werden und allgemein zur Beschreibung von Interaktionen in einem Protokoll, der Formalisierung von Systemverhalten etc. dienen. Letztendlich sind die allseits bekannten UML- [UML] und ER-Diagramme [Vos94] auch ein Spezialfall von Graphen.

Zwei spezifische Aufgabengebiete des Softwareengineering, die in der letzten Zeit sehr an Bedeutung gewonnen haben, sind Model Driven Architecture (MDA) [KWB03]

und Reverse Engineering [CMW02]. In beiden Ansätzen spielen Graphen und Graphersetzungssysteme eine wichtige Rolle.

Bei MDA wird aus einem Modell, das meistens in Form von Zustands- und Klassendiagrammen gegeben ist, automatisiert Quelltext erzeugt. Da das Modell dabei eine zusätzliche Abstraktionsebene zum Quellcode darstellt, können die Anwendungen besser und schneller entwickelt werden. In manchen Anwendungsfallen wird durch den MDA-Entwicklungszyklus ein Rahmenwerk aufgestellt, in das von Hand implementierte Module eingebunden werden. Oft wird das Modell nach bestimmten Regeln verändert, zum Beispiel wird die Architektur spezifischen plattformabhängigen Kriterien angepasst. Das geschieht meistens automatisiert mit Hilfe von Graphtransformationen und kann mit einem Precompiler für Modelle verglichen werden. [PLGCGB06]

Beim Reverse Engineering wird das umgekehrte Ziel verfolgt, indem aus Quellcode die zugrunde liegende Architektur extrahiert wird. Das wird oft benutzt um ältere Programme, für die keine Dokumentation vorliegt oder der Entwurf nicht methodisch stattgefunden hat, weiterzuentwickeln. Meistens wird das Programm analysiert und in einer Neuentwicklung implementiert oder durch Refactoring an aktuelle Anforderungen angepasst. [MW02] Die Veränderung des durch die Analyse gewonnenen Modells wird mit Hilfe von Graphtransformationen realisiert. Es wird in der Architektur nach bestimmten Mustern, die geforderten Eigenschaften verletzen, gesucht und entsprechend transformiert.

Alle oben vorgestellten Beispiele arbeiten mit Graphen. Die verwendeten Graphen weisen aber sowohl zum einen unterschiedlichen Aufbau auf, zum anderen müssen sie unterschiedlichen Anforderungen genügen. Graphen in der MDA-Entwicklung bestehen aus Klassendiagramen und Architekturbeschreibungen. Sie sind grobgranular und drücken allgemeinen Zusammenhänge aus. Diese Daten werden durch verschiedene Werkzeuge verarbeitet. In Gegensatz dazu sind die graph-basierten Daten in Reverse Engineering relativ feingranular und werden eventuell bis auf den Quelltext zurückgeführt. Für die Arbeit mit diesen graphbasierten Daten werden Graphtransformationssysteme programmiert. Die Spezifikation von solchen Systemen wird mit Hilfe von Graphwerkzugen bewerkstelligt, die oft bestimmten Einschränkungen bezüglich verwendeten Daten oder Funktionalität unterliegen.

Daraus ergibt sich die Problemstellung, zu der in dieser Arbeit ein Lösungsansatz ausgearbeitet wird. Es soll eine Infrastruktur für die Entwicklung graphbasierter Werkzeuge erarbeitet werden. Die Graphen sollen angemessen gespeichert, verarbeitet und abgefragt werden können. Es existieren bereits einige ausgereifte Lösungen um graph-basierte Daten zu speichern. Die Unterstützung für die Entwicklung von Graphtransformationswerkzeugen und als Folge flexibler Einsatz von Graphersetzungssystemen bleibt aber unzureichend.

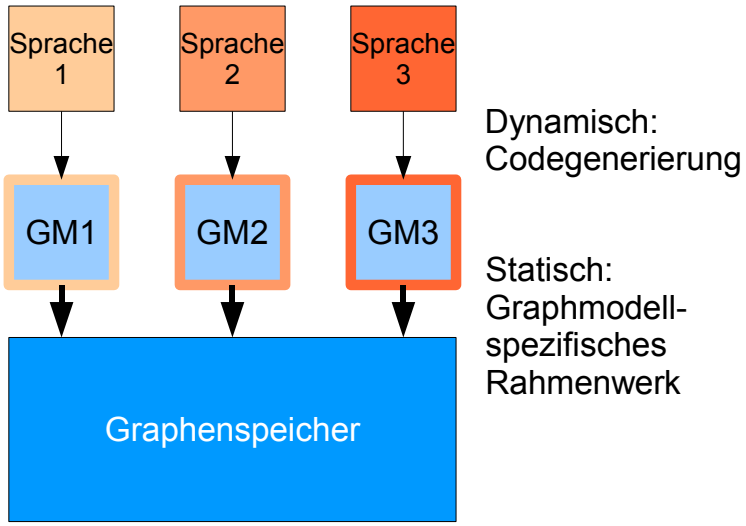

Abbildung 1.1: Herkömmliche Entwicklungsschritte

In diesem Ansatz (siehe Abb. 1.1) werden Graphersetzungssysteme in einem Graphtransformationswerkzeug mit Hilfe einer Sprache definiert. Die Sprache ist an ein Werkzeug gebunden und kann weder gegen eine anwendungspezifische ausgetauscht noch aus dem Werkzeug losgelöst werden. Aus der Spezifikation wird Quelltext erzeugt, um die modellierten Graphtransformationen anzuwenden. Dieser wird in einem an das Werkzeug angepassten Rahmenwerk ausgeführt. Das Rahmenwerk ist fest an den Graphspeicher angebunden und führt durch den generierten Code veranlassten Änderungen am Datenbestand aus. Der Graphenspeicher hat ein allgemeines Graphmodell, das auch mit unterschiedlichen Werkzeugen und Transformationssystemen vereinbart werden kann. Dieser Ansatz hat eine Reihe von Nachteilen, die im weiteren Verlauf dieser Arbeit genauer besprochen werden. Auf diese Weise werden laufzeiteffiziente Werkzeuge erzeugt, die Entwicklung ist aber aufwändig und nicht flexibel genug, um unterschiedlichen Anforderungen gerecht zu werden.

Als Beispiel wird hier ein Entwurf eines Graphtransformationssystems in PROGRES betrachtet: man spezifiziert Graphtransformationsregel in dem von der PROGRES Umgebung bereitgestellten Editor. Aus den Transformationen wird automatisch Quelltext erzeugt. Er benötigt eine Reihe von Bibliotheken und kann in einem Rahmenwerk namens UPGRADE auf der Graphdatenbank ausgeführt werden.

Trotz aller Ähnlichkeit zwischen den von verscihiedenen Graphtransformations-

werkzeugen Graphdaten müssen die Unterschiede bei der Implementierung bis auf das niedrigste Niveau heruntergebrochen werden. Ein Beispiel dazu sind Graphersetzungswerkzeuge PROGRES und Fujaba [KNNZ99], die beide auf beschrifteten gerichteten Graphen arbeiten. Die Werkzeuge haben aber sowohl eine unterschiedliche funktionale Schnittstelle, als auch einen nicht austauschbaren Format zur Speicherung von Daten. Eine Lösung besteht darin, ein allgemeines Graphmodell beim Speichern und Verarbeiten von Daten zu verwenden. So wird das spezielle Graphmodell der generierten Graphersetzungssysteme auf das allgemeine abgebildet, wodurch in einem gemeinsamen Speicher Graphdaten gespeichert und verwaltet werden können.

Eine Verbesserung in diesem Punkt ist eine universelle Infrastruktur und Werkzeuge zum Entwurf von Graphtransformationssystemen. Es soll die Kernfunktionalität eines Werkzeuges angeboten werden, und zwar so, dass ein Werkzeug an die jeweilige Anwendungsdomäne angepasst werden kann. Dazu werden die benötigten Datenstrukturen, sowie die Funktionen, wie Speicherung, Suchen auf Graphen und Transformieren, bereitgestellt. Alle Teile sind insoweit generisch, so dass sie ohne großen Aufwand verändert oder um weitere Funktionen erweitert werden können. Die einzelnen Schichten kommunizieren miteinander durch vordefinierte Schnittstellen und sind somit frei austauschbar. Das gibt dem Entwickler die Freiheit bei der Wahl von Graphmodellen und Methoden zur Beschreibung und Ausführung von Transformationen.

Die Codegenerierung, die dem herkömmlichen Ansatz zugrunde liegt, bringt sowohl Vorteile als auch Nachteile mit sich. Ein Vorteil ist die Laufeffizienz, die mit anderen Methoden kaum zu erreichen ist. Als ein Nachteil gilt die aufwendige Realisierung eines Codegenerators. Die Codegenerierung ist keine triviale Aufgabe und muss sorgfältig implementiert werden. Auch bei ähnlichen von der Sprache verwendeten Graphmodellen ist eine Wiederverwendung nur begrenzt möglich. Also stellt die Implementierung oder Veränderung von einem Codegenerator einen erheblichen Aufwand dar.

Ein weiterer Aspekt ist die Verifizierung des Generators [BGL05]. Es muss eine Übereinstimmung zwischen der Semantik der formulierten Regeln und des daraus generierten Quelltextes garantiert werden. Diese Frage kann nur für eine begrenzte Menge der Sprachen beantwortet werden, die einer Reihe erheblicher Einschränkungen unterliegen.

Der resultierende Code enthält neben den Regeldefinitionen auch die Steuerung zur Ausführung, Zusatzinformation für Backtracking, etc. Das macht den Quelltext nur schwer lesbar und kaum für eine weitere manuelle Entwicklung verwendbar.

Die aufwändige Implementierung des Codegenerators bringt ferner eine Reihe folgende Nachteile mit sich: Die Sprache lässt sich nur schwer um weitere Konstrukte erweitern. Auch eine Erweiterung des Graphmodells um spezifische Graphobjekte,

zum Beispiel Hyperkanten, ist kaum möglich und würde praktisch eine Neuimplementierung des Generators bedeuten.

Ein weiterer Nachteil des herkömmlichen Ansatzes ist der Verlust des Zusammenhanges innerhalb der Spezifikation. Durch mehrstufige Transformation der Spezifikation zum Code gehen die Idee und die Entscheidungen, die durch Regeln ausgedrückt sind, verloren. Die Information wird in einer Menge elementarer Operationen des generierten Codes verborgen. Das beeinträchtigt die Lesbarkeit und somit auch die Wartbarkeit einer Anwendung.

Der Ansatzpunkt, um die oben beschriebenen Mängel zu beheben, der auch in dieser Arbeit weiter verfolgt wird, ist neben dem allgemeinen Graphmodell auch eine allgemeine Graphtransformationssprache anzubieten. Die Entwicklung eines Werkzeugs ändert sich daher entsprechend (siehe Abbildung 1.2.) Ein Werkzeug bezieht sich unter der Benutzung eigener Graphtransformationssprache auf ein allgemeines Modul. Das Modul ist, aus der Sicht einer Anwendung, ein Teil der Datenbank. Die Sprache bildet eine klare, erweiterbare Schnittstelle zwischen einer Anwendung und dem Graphenspeicher.

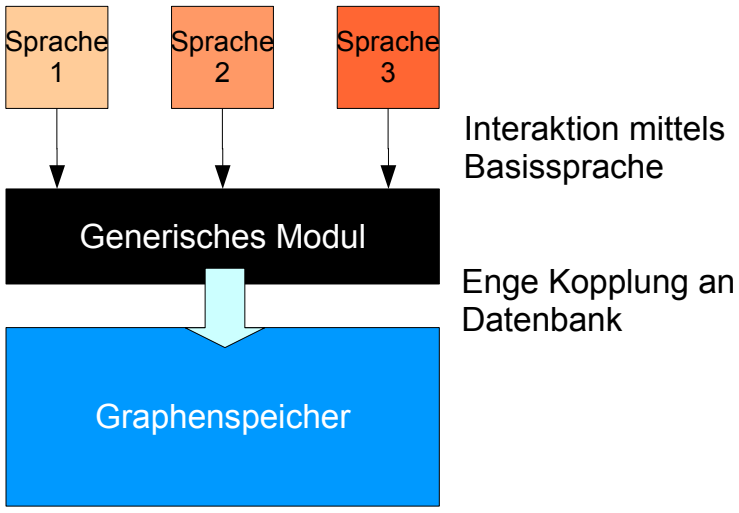

Abbildung 1.2: Interpretativer Ansatz

Die Modelltreue eines Werkzeugs wird durch ein spezifisches, vom Benutzer vorgegebenes Graphschema sichergestellt, dadurch wird die Konsistenz der Daten überwacht. Einzelne Operationen werden auf Graphen mit Hilfe der Basissprache be-

schrieben. Die Information, die bei der Modellierung der Anwendung entsteht, bleibt beibehalten. Das soll die Wartbarkeit und Anpassungsfähigkeit so entwickelter Werkzeuge steigern.

Der aus der Sicht einer Anwendung interpretative Ansatz erlaubt auf die häufig eingesetzte Codegenerierung zu verzichten und koppelt die Sprache von dem Werkzeug ab. Sprachen und Werkzeuge beziehen sich somit auf eine einheitliche Schnittstelle des Datenbankmoduls. Auf diese Weise lässt sich die Sprache flexibler gestalten und kann auf bestimmte Anwendungsfälle sowohl in Sprachkonstrukten als auch in dem Graphmodell angepasst werden.

Durch den Verzicht auf automatische Codegenerierung entfällt auch die Notwendigkeit eines Rahmenwerkes. Das steigert die Wiederverwendbarkeit: eine Anwendung soll nicht unbedingt an einen Rahmenwerk für Graphtransformationen angebunden sein. So kann ein Regelsatz von der Anwendung und der Anwendungsumgebung losgelöst und in einem anderem Programm wiederverwendet werden.

Durch eine viel engere Kopplung an die Datenbank kann der Durchsatz gesteigert werden. Das trifft insbesondere auf die Anwendungsfälle zu, bei denen Konstrukte gezielt von Information über den Datenbankinhalt profitieren.

Wiederverwendung wird auch durch die verbesserte Lesbarkeit und die Wartbarkeit unterstützt. Die Regeln werden auch als solche im Programm verwendet und nicht auf atomare Anweisungen heruntergebrochen. Eine spätere Überarbeitung oder Anpassung der Spezifikation benötigt nicht so viel Aufwand wie in herkömmlichen Systemen.

Somit ändert und vereinfacht sich der gesamte Prozess der Entwicklung eines Graphwerkzeuges. Das Graphmodell sowie die Transformationsvorschriften werden in der Basissprache beschrieben und von der Datenbank verarbeitet, ohne dass das Programm mit Nebeninformationen zur Auswertung überladen wird. Das erlaubt eine schnelle und flexible Entwicklung und einen breiten Einsatz von Graphwerkzeugen.

1.2 Ziel der Arbeit

In dieser Arbeit wird eine universelle Sprache entworfen. Sie bildet die Grundlage für die Kommunikation zwischen einer Anwendung und der Datenbank bzw. dem Datenbankmodul. Basierend auf der Sprachdefinition können in weiteren Projekten die ergänzenden Instrumente erarbeitet werden.

Die Sprache wird im Hinblick auf die Allgemeinheit und die Anpassungsfähigkeit entworfen. Sie soll existierende Graphtransformationssprachen abdecken und eine Basis für Neuentwicklungen auf dem Bereich der Graphtransformationssysteme bilden. Die Sprache soll so konzipiert werden, dass sie auch für mögliche Anforderungen

zukünftiger Anwendungen angemessene Unterstützung bringt.

Es werden die grundlegenden Formalismen für die Sprachbeschreibung eingeführt und mit deren Hilfe die Graphtransformationssprache definiert. Die syntaktische Zusammensetzung der Sprache, sowie die Semantik der Konstrukte werden festgelegt.

1.3 Überblick

Der Rest der Ausarbeitung ist wie folgt aufgebaut: zunächst wird im Kapitel 2 eine Übersicht über bereits existierende Graphtransformationssysteme gegeben. Diesbezüglich werden sowohl eingesetzte Sprachen, als auch Graphspeicher angesprochen. Exemplarisch werden Vor- und Nachteile einzelner Systeme aufgezeigt, diese bilden eine Grundlage für spätere Diskussion über die Eigenschaften der Sprache.

Im darauf folgenden Kapitel 3 wird das Gesamtprojekt vorgestellt und allgemeine Ziele des Gesamtprojektes angesprochen.

Anschließend, im Kapitel 4, wird die Graphdatenbank DRAGOS näher beschrieben und das zugrunde liegende Graphmodell und das Graphschema werden formalisiert.

Im Kapitel 5 wird die Basissprache eingeführt. Die Zusammensetzung der Sprache und die Sprachkonstrukte werden erläutert. Durch Beispiele und formale Definitionen wird die Semantik der Sprache erklärt. Die erreichten Eigenschaften der Sprache werden diskutiert.

Weiterführend, im Kapitel 6 werden einige technische Einzelheiten über die Anbindung der Sprache an die Datenbank besprochen. Unterschiedliche Ansätze werden mittels Beispielen erklärt.

Zum Schluss, im Kapitel 7 gibt es eine abschließende Zusammenfassung sowie einen kurzen Ausblick.

2 Grundlagen

In diesem Kapitel wird ein Überblick über bereits existierende und eingesetzte Graphtransformationsysteme gegeben. Es werden sowohl Graphtransformationssysteme, sowie Graphenspeicher charakterisiert. Ferner wird eine kurze Schilderung von Nach- und Vorteilen einzelner Konzepte gegeben.

2.1 Graphtransformationssysteme

Wie im vorherigen Kapitel beschrieben, werden spezielle Graphwerkzeuge benötigt um unterschiedliche anwendungsspezifische Anforderungen zu erfüllen. Im weiteren wird ein kurzer Überblick über einige bereits existierende Graphtransformationssysteme und Graphdatenbanken gegeben. [FMRS07] Die Vor- und Nachteile einzelnen Systemen bilden den Ausgangspunkt für Anforderungen an die Basissprache.

2.1.1 PROGRES

PROGRES (**PRO**grammierte **G**raph **ER**setzung**S**ysteme) [Sch90] [SWZ99] [SWZ96] entstand als ein Hilfswerkzeug im Rahmen des Projektes IPSEN [Nag96] am Lehrstuhl 3 für Informatik an der RWTH Aachen und verfolgt einen logisch orientierten Ansatz. PROGRES ist eines der ältesten eingesetzten Graphtransformationswerkzeuge.

Die Graphtransformationsregeln werden in PROGRES in einer grafisch-textuellen Notation beschrieben. Anschließend wird aus den Regeln der Code erzeugt. Der generierte Code enthält die komplette Information über den Transformationsregeln und Strukturen zur Steuerung der Ausführung, benötigt aber einige Bibliotheken und eine Laufzeitumgebung. Der Quelltext kann direkt in IPSEN-Werkzeuge eingebettet oder mit Hilfe eines Rahmenwerks ausgeführt werden. Es existiert auch die Möglichkeit die Regeln mit Hilfe eines internen Interpreter in dem PROGRES Editor abzuarbeiten.

Von PROGRES werden Anfragen und Transformationen auf attributierten und beschrifteten Graphen ausgeführt. Graphen bestehen aus attribuierten und beschrifteten Knoten und Kanten. Knoten und Kanten sind stets typisiert, wobei Klassen durch

Mehrfachvererbung definiert werden können. Kanten sind nur zwischen zwei Knoten erlaubt.

Die Struktur des verwendeten Graphen wird durch ein Graphschema vorgegeben. PROGRES unterstützt statische Regeln, die auf eine Inkonsistenz einen Graph reagieren können.

Die Knoten und Kanten werden mit Attributen ausgestattet. Es gibt folgende Arten der Attributen.

- intrinsische: Es sind eigenständige Attribute eines Objektes, die durch Zuweisung geändert werden.
- abgeleitete: Im Gegenteil zu intrinsische Attributen werden sie nicht zugewiesen, sondern aus anderen Attributen berechnet. Dadurch lassen sich komplexere Informationen, auch in Abhängigkeiten zu anderen Graphobjekten, sammeln.
- Metaattribute: In diesen Attributen werden Arbeitsinformationen beibehalten, die für eine korrekte, graphschematreue Ausführung der Transformationen benötigt werden. Die Metaattribute gehören zu Klassen, anstatt zu Knoten und werden im Laufe der Regelausführung nicht geändert.

PROGRES arbeitet sowohl mit einfachen, als auch mit mengenwertigen Attributen. Eine Ansammlung von Datensätzen kann durch ein mengenwertiges Attribut repräsentiert werden.

Regeln in PROGRES werden durch die Angabe einer linken und einer rechten Seiten beschrieben. Die Übereinstimmung von Knoten und Kanten in der linken und rechten Seite einer Regel wird durch die textuelle Notation angedeutet, Knoten in der rechten Seite tragen die gleiche Markierung wie in linken (siehe dazu das Beispiel 2.1.) Neben den grundlegenden Ausdrücken auf Knoten und Kanten existieren auch abgeleitete Sprachkonstrukte für Ausdrücke mit Pfaden.

Bei der Beschreibung einer linken Seite ist auch eine Verneinung von Objekten möglich. Es kann das Fehlen einzelner Graphentitäten, etwa eines Knotens oder einer Kante formuliert werden. Es ist nicht möglich komplexere, zusammenhängende Muster zu verneinen.

Beispiel 2.1. (*Beispieltransformation in PROGRES*)
In der Datenbank sind verketteten Listen gespeichert. Sie haben folgende Struktur: ein Knoten vom Typ `list` ist mit Kanten vom Typ `first` und `last` jeweils mit dem ersten bzw. dem letzten Element der Liste verbunden. Die Elementen in der Liste sind vom Typ `item` und werden untereinander mit Kanten vom Typ `next` organisiert (Abb. 2.1).

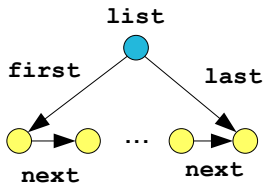

Abbildung 2.1: Verkettete Liste

In diesem Beispiel wird eine Produktion aus PROGRES dargestellt. Die Produktion nimmt als einen Eingabeparameter einen Knoten der Klasse `list`. Es wird nach einem für die linke Seite passenden Teilgraphen in der Datenbank gesucht. Falls die Suche erfolgreich ist und ein Teilgraph entsprechend der linken Seite gefunden wird, wird er durch die rechte Seite ersetzt. Hier wird ein neuer Knoten an eine verkettete Liste angehängt, indem ein Knoten erzeugt wird, markiert mit `3'` und die Kanten entsprechend der Regel umgeleitet werden.

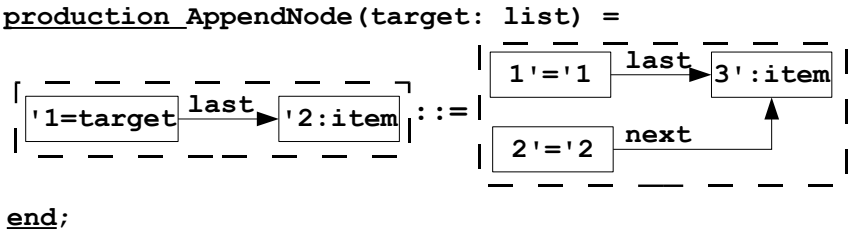

Abbildung 2.2: Transformation in PROGRES

■

Eine Produktion beinhaltet mehrere Regeln und kann mit einer Vor- und Nachbedingung versehen werden. Die Produktionen haben Signaturen, die Ein- und Ausgabeparameter vorgeben. Das ist nützlich für die Konsistenzprüfungen zur Kompilierzeit und erlaubet zudem Produktionen ähnlich wie Prozeduren aufzurufen.

Mehrere Produktionen können zu einem Programm zusammengefügt werden. Dazu bietet PROGRES recht umfangreiche Methoden, um Ablauf der Graphtransformationen zu steuern. Darunter sind Aufrufe von Transformationen, sequenzielle Ausführung, bedingte Verzweigung, Schleifen und nichtdeterministische Wahl.

Beispiel 2.2. (*Steuerung in PROGRES*)
In diesem Beispiel wird eines von vielen Konstrukten zur Ablaufsteuerung vorgestellt.

Mittels `choose` wird eine Alternative ausgewählt. Hier wird ein vereinfachter Fall dargestellt. Bei einem positiven Test `ListExist` für die Eingabe `InputList` wird die Produktion `AppendNode` angestoßen. Falls entweder der Test oder der Aufruf negativ ausfällt wird `choose` nicht abgebrochen, sondern mittels `skip` erfolgreich verlassen.

```
choose
    when ListExist(InputList)
    then AppendNode(InputList)
    else
        skip
end;
```

Abbildung 2.3: Kontrollstrukturen in PROGRESS

■

PROGRES besitzt eine Reihe weiterer Ausdrücke, wie etwa eine logische Aussage über alle möglichen Belegungen einer Formel. Das macht PROGRES zu einem allgemeinen und ausdrucksstarken Werkzeug. PROGRES konnte sich bereits in industriegroßen Anwendungen beweisen, zum Beispiel in dem Prozessmanagementsystem AHEAD. [JSW99]

Die PROGRES Spezifikation wird in einem syntaxgesteuertem Editor entworfen. Die Syntaxsteuerung des Editors bringt mit sich sowohl Vorteile, als auch Nachteile. Als eine positive Eigenschaft gilt, dass die Spezifikation immer bis zu einem gewissen Grad wohlgeformt ist. Der Editor überwacht die syntaktische Korrektheit der Ausdrücke und weist auf Fehler bezüglich statischer Semantik hin. Der Umgang mit dem Editor gestaltet sich für ungeübte Anwender aber nur schwer.

Ein weiterer Nachteil von PROGRES ist die veraltete Programmiersprache in der es implementiert ist. Es wurde in der Programmiersprache Modula2 entwickelt und ist nur in der Umgebung von älteren linuxbasierten Betriebssystemen lauffähig. Eine Anpassung der bestehenden Codebasis an die modernen Betriebssysteme ist nur mit viel Aufwand möglich. Das macht PROGRES nur begrenzt in modernen Neuentwicklungen einsetzbar.

2.1.2 DiaPlan

DiaPlan (**DIA**gramm **P**rogramming **LAN**guage) wird an der Universität Bremen entwickelt. [HM00] [Kle04] [Hof00] DiaPlan wurde als konzeptionelles Werkzeug entwickelt, es gibt nur eine nicht veröffentliche Implementierung.

DiaPlan verfolgt einen recht ungewöhnlichen für Graphtransformationswerkzeuge Ansatz, da als Grundstruktur Hypergraphen gewählt wurden, somit kann man hier auch von einem Kantenersetzungsystem sprechen. Wie die theoretische Grundlage jedoch zeigt, sind Kanten- und Knotenersetzungssysteme äquivalent zueinander. [Roz97]

Die Hypergraphen in DiaPlan bestehen aus Knoten und Hyperkanten, die mehrere Knoten verbinden. Knoten und Kanten können beschriftet werden. Ein nützliches Konzept, das von DiaPlan unterstützt wird sind hierarchische Graphen. Die Kanten in DiaPlan können so genannte Behälter enthalten, die wiederum Graphobjekte enthalten. Dies erlaubt dem Benutzer geschachtelten Graphen zu definieren, wodurch die Lesbarkeit und Wartbarkeit der Spezifikation steigert. Der hierarchische Aufbau von Graphen wurde bereits in vielen Arbeiten behandelt, aber bisher in keiner Sprache konsequent realisiert.

Alle Graphobjekte sind in DiaPlan streng typisiert. Anders als bei PROGRES wird die Konsistenz des Graphs durch einen Typgraph und nicht durch ein Graphschema gegeben. Der Typgraph enthält eine Beschreibung von Knoten- und Kantenklassen, sowie Beziehungen zwischen denen. Die einzelne Klassen können von anderen abgeleitet werden. Der Typgraph stellt ein Muster für den Aufbau einen Graphen, im Gegensatz zu PROGRES, in dem eine Klassenhierarchie auf Graphobjekten und die strukturelle Information nur implizit vorgegeben wird.

Eine Regel besteht in DiaPlan aus einer oder mehreren Anwendbarkeitsbedingungen, einer linken und einer rechten Seite. Die linke Seite wird mit Hilfe von Variablen definiert. Unter einer Variable wird in DiaPlan eine Hyperkante, die mehrere Graphobjekte umfasst verstanden. Beim Transformieren wird die Variable mit passenden Graphobjekten belegt und anschließend wird sie durch die rechte Seite, eine andere Hyperkante, ersetzt. Um die Regeln für hierarchische Graphen definieren zu können existiert der Mechanismus der geschachtelten Kantenersetzung. In einer Regel wird zuerst mit einer tiefst gelegenen Variable angefangen und dann von innen nach außen weitergearbeitet.

Beispiel 2.3. (*Beispieltransformation in DiaPlan*)
In diesem Beispiel wird ein Knoten `X` an die Liste `Q` angehängt. Zu Beachten hier, dass die komplette Liste durch eine Variable von einem entsprechenden Typ repräsentiert wird. Die Variable, in der Abbildung 2.4 durch graue Rechtecke dargestellt) wird durch die Konstruktion auf der rechten Seite ersetzt, so dass die Liste aus der ursprünglichen Liste und dem Knoten `X` kombiniert wird.

■

Der Kontrollfluss wird in DiaPlan nicht imperativ, wie in PROGRES, vorgeschrieben, sondern ähnlich dem funktionalen Programmieren allein durch eine Ansamm-

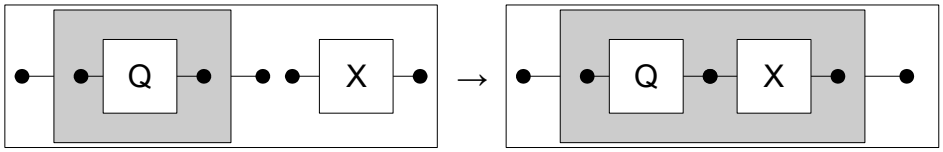

Abbildung 2.4: Transformation in DiaPlan

lung von Regeln gegeben. Sehr nützlich erweisen sich an dieser Stelle die Anwendbarkeitsbedingungen, sodass eine Regel, für die die linke Seite zwar instanziert werden kann, aber im Kontext nicht einsetzbar ist, von der Ausführung ausgeschlossen wird.

Die Interaktion zwischen einzelnen Regeln erfolgt mittels Abschlussklauseln, so wird Erfolg oder Fehlschlag festgestellt und an nachfolgende Regel weitergegeben. Dem Programmierer stehen Ausnahmen zu Verfügung, so dass eine erweiterte Interaktion zwischen Regeln möglich ist. Diese Herangehensweise lässt sich gut formalisieren und mit Hilfe der mathematischen Logik beschreiben, benötigt aber eine gewisse Erfahrung auf dem Bereich der funktionalen Programmierung.

Als ein weiterer Mechanismus der Ablaufkontrolle stellt die Sprache die Kapselung von Transformationen bereit. Knoten repräsentieren Daten und Regeln eines Programms. Die Kanten zwischen solchen Knoten ordnen die Ein- und Ausgabeparameter zu. Es erlaubt eine Wiederverwendung von Konstrukten und bringt die Möglichkeit Transformationsspezifikationen zu strukturieren und methodisch zu entwerfen.

Beispiel 2.4. (*Kappselung in DiaPlan*)
Hier wird eine gekapselte Transformation `T` dargestellt. Die Interna der Transformation bleibt verborgen, bis auf wenige, explizit angegebene Ein- und Ausgabeparameter, hier `init`, `open` und `close`. ■

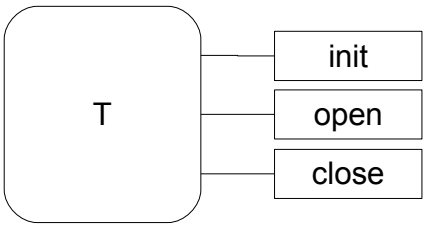

Abbildung 2.5: Kapselung in DiaPlan

Die Sprachkonstrukte in DiaPlan lassen sich gut formal beschreiben und bilden somit eine theoretische Grundlage der Sprache.

Ein recht ungewöhnlicher Ansatz der Kantenersetzung kann die Schwierigkeiten bei einem praktischem Einsatz bereiten, ferner sind nicht alle Anwendungsfälle einfach auf Hypergraphen zurückzuführen. Eine Graphersetzungsspezifikation ist einem Benutzer, der nur mit prozeduralem Programmieren vertraut ist, nur schwer verständlich.

Die Sprache wurde unter Anderem auch für weitere experimentelle Anwendungen auf dem Bereich Sterngrammatiken [DHJ+06] und Shaped Nested Graphs [HM01] konzipiert.

2.1.3 AGG

AGG (**A**lgebraische **G**raph**G**rammatik) wurde an der TU Berlin entwickelt [LB93] und verfolgt strikt den algebraischen Ansatz [Roz97]. Transformationen auf Graphen werden als algebraische Operationen auf der Menge aller möglichen Graphen zusammengefasst.

AGG arbeitet auf Graphen mit attribuierten Knoten und Kanten. AGG enthält wie DiaPlan einen Typgraph, der die Definitionen für Knoten- und Kantenklassen enthält. Knotenklassen können von den anderen Knotenklassen durch die Mehrfachvererbung abgeleitet werden. Zusätzlich werden im Typgraphen die Attributdefinitionen für Knoten und Kanten beschrieben. Die Graphobjekte müssen aber nicht unbedingt getypt sein, AGG unterstützt auch typlose Graphen. Abgeleitete Attribute oder Metaattribute werden nicht unterstützt.

Die Regeln in AGG werden durch die Angabe der linken und der rechten Seite beschrieben. Die linke Seite darf nur aus Knoten und Kanten bestehen. Es werden keine abgeleitete Konstrukte, wie etwa Pfadanfragen unterstützt.

Beispiel 2.5. (*Beispieltransformation in AGG*)
Mit dieser Transformation (Abb. 2.6) wird ein Knoten an eine verkettete Liste angefügt. Die linke Seite beschreibt die Suche nach dem letzten Knoten in der Liste. Die linke Seite wird anschließend durch die Rechte ersetzt. Die Knoten und die Kanten mit gleichen Markierungen auf beiden Seiten werden beibehalten, mit neuen Markierungen werden erzeugt.

■

Es besteht die Möglichkeit einen Graphen, der nicht gefunden werden soll, "außerhalb" der linken Seite zu definieren. Auf diese Weise lassen sich so genannte NACs (negative application conditions) darstellen. Im Gegensatz zu PROGRES können die

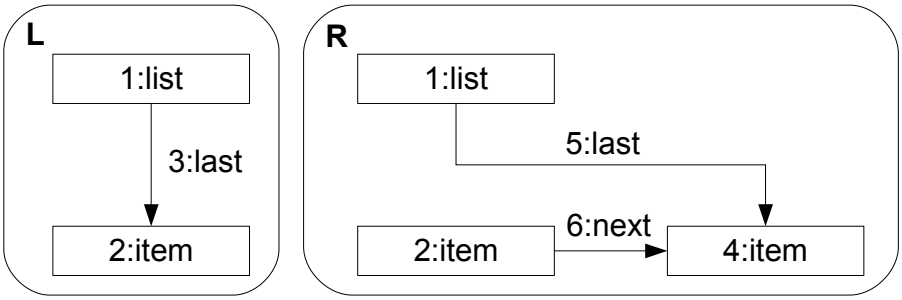

Abbildung 2.6: Transformation in AGG

zusammenhängende Strukturen verneint werden, sodass AGG in dieser Hinsicht ausdrucksstärker ist.

Eine passende Belegung der linken Seite wird bezüglich NACs und Bedingungen an Attribute gesucht. Falls die linke Seite gefunden werden kann, wird sie entsprechend dem SPO-Ansatz (siehe [Roz97]) zur rechten Seite umgeschrieben.

In AGG fehlen jegliche Mechanismen für Konsistenzprüfungen. Bis auf das Graphmuster aus dem Typgraph muss die Prüfung der Übereinstimmung von Attributtypen, Anfang- und Endtypen einer Kante „von Hand“, in Regeln, geschehen.

AGG bietet in Unterschied zu PROGRES und DiaPlan eine Möglichkeit zwei Knoten zu einem zu verschmelzen, dabei werden die Attribute von den beiden Knoten beibehalten. Eventuelle Konflikte löst der Benutzer interaktiv.

Die Sprache beinhaltet kaum Ausführungskonstrukte. Graphtransformationsregel können entweder einzeln manuell angestoßen werden oder in einer sequentieller Reihenfolge ausgeführt. Die Abfolge wird durch so genannten Ebenen vorgegeben: die Regel auf einer Ebene werden solange ausgeführt bis keine greift, dann wird mit den Regeln auf der nächsten Ebene fortgefahren.

Die Bibliotheken, auf denen AGG implementiert ist, erlauben hierarchische Graphen und komplexe Kanten, was von der Sprache aber nicht umgesetzt wird. Trotzt der fundierten Grundlage findet AGG keine breite Verwendung. Die Sprache ist relativ ausdruckschwach, das Fehlen von den Konsistenzprüfungen und den Ablaufkontrollstrukturen macht AGG eher ungeeignet für größere Anwendungen. Dieses Graphwerkzeug wurde als eine Laboranwendung entwickelt und ist nicht für große Datenmengen ausgelegt. AGG findet ihre sinnvolle Verwendung für Test- und Prototypingzwecken.

Eine nützliche Funktion von AGG ist die im Editor eingebaute Regelanalyse (siehe dazu auch [Tae04]). Die gegebene Spezifikation kann auf kritische Regelpaare untersucht werden. Es werden die Paare von Regeln ausgesucht, die sich bei der Ausfüh-

rung gegenseitig stören. Zum Beispiel wird ein Knoten durch eine Regel gelöscht, der für eine andere Regel benötigt ist, oder durch eine Regel wird ein NAC-Konstrukt für eine andere Regel verletzt. Die Regel können auch bezüglich der definierten Ausführungsebenen untersucht werden. Dies erlaubt die Spezifikationen auf mögliche Konflikte und Unbestimmtheiten zu untersuchen.

2.1.4 GrasQL

Eine speziell für Anfragen entwickelte Sprache ist GrasQL (Gras/GXL Query Language). [Thy05] Sie entstand parallel zur Entwicklung von der Datenbank DRAGOS (siehe Kapitel 2.2.2) und sollte die Sprache GreQL erweitern. Der Name bezieht sich auf den früheren Arbeitsname der Datenbank Gras/GXL.

Die Sprache basiert auf dem Graphmodell vom DRAGOS und kann somit verschiedene Graphmodelle von anderen Sprachen darauf abbilden. GrasQL unterstützt getypte, beschriftete Graphen, die aus Knoten, Kanten und n-ären Relationen bestehen. Alle Elemente können mit Attributen versehen werden. Als Attributwerte dienen alle serialisierbare Java-Datentypen. Dies ermöglicht sehr komplexe Beschreibungen in den Attributen zu verwalten, bringt allerdings den Nachteil, dass die Attribute nur auf die Gleichheit überprüft werden können. Für weitere komplexere Operationen auf den Attributen wird eine Schnittstelle für externe Java-Funktionen angeboten.

Die Sprache bietet unter Anderem auch abgeleitete Anfragen auf Kanten, es können Ausdrücke mit Pfaden definiert werden.

Beispiel 2.6. (*Beispielanfrage in GrasQL*)
In diesem Beispiel wird nach zwei Knoten `a` und `b` aus den Graphen `G1` und `G2` gesucht. Es werden folgende Bedingungen an die Knoten gestellt: die Knoten müssen gleiche Werte des Attributs `Name` haben und das Wert des Attributes `Wert` soll positiv sein. Als die Ausgabe liefert die Anfrage die beide Knoten.

■

Als Eingabe und Ausgabe einer Anfrage wird ein Mengen von Graphen, ein so gennante Graphpool benutzt. Dadurch können mehrere Anfragen hintereinander geschaltet werden, indem die Ausgabe einer Anfrage als die Eingabe der nächsten verwendet wird. Durch die Anfragen wird eine Sicht auf die Daten definiert und schrittweise verfeinert.

Um eine bessere Lesbarkeit zu erreichen, können komplexe Anfragen in eine Reihe kleinerern, logisch abgeschlossenen Unteranfragen zerteilt werden. Das erscheint besonders sinnvoll bei Anfragen, wo eine Gruppe von Ergebnissen konstruiert werden muss, bevor weitere Anfragen stattfinden. Auf diesem Weg besteht auch das Potenzial zur manuellen Optimierung der Anfragen.

```
DECLARE
    a: G1.NODE
    b: G2.NODE
WITH
    a.Name = b.Name
    AND a.Wert > 0
REPORT
    a,b
```

Abbildung 2.7: GrasQL Anfrage

Die Sprache bietet auch Mengenoperationen sowie die Aggregationsfunktionen. Bei Mengen werden gewöhnliche Operationen angeboten, wie Vereinigung, Durchschnitt, Differenz und Inklusion in einer Menge. Es werden zwei Klassen der Mengen unterschieden: die Mengen von Graphentitäten, die durch Teilgraphen repräsentiert werden, und die Mengen von Graphen, durch einen Graphpool. Eine Aggregation liefert eine Aussagen über eine Menge von Graphentitäten und bestimmt, zum Beispiel, Minimum, Maximum oder Kardinalität einer Menge. GrasQL stellt Konstrukte für die Anfragen auf hierarchischen Graphen zur Verfügung.

Vorteile der Sprache liegen in ihrer Ausdruckstärke, sie basiert auf einem sehr umfassenden und allgemeinen Graphmodell. Die Sprache hat eine rein textuelle Notation, die an der Syntax verbreiteten SQL angelehnt ist. Das gestattet die Verwendung der Anfragen in GrasQL innerhalb der Ausdrücken einer Wirtsprache oder als Parameter einer Funktion.

2.2 Graphdatenbanken

Im vorherigen Abschnitt wurden unterschiedliche Graphersetzungssysteme vorgestellt. Nun sollen die von Systemen eingesetzten Datenbanken diskutiert werden. Einige Graphtransformationswerkzeuge benutzen keine Datenbank für die Speicherung der Graphen. Beispielsweise verwaltet Fujaba alle Graphobjekte im Hauptspeicher verwaltet. Da Java keine direkte Möglichkeit vorsieht, die Hauptspeicherobjekte zu speichern, greift man auf ein Framework CoObRA zurück. Dies erlaubt eine persistente Speicherung, mit einigen Zusatzfunktionen wie undo/redo und Langzeitversionierung. Es wird einer Reihe Änderungen an Objekten gespeichert. Das kann zu Schwierigkeiten bei Konsistenzprüfungen führen, da dafür die Sicherstellung von exklusiven Schreib-/Leserechten benötigt wird.

Es spricht eine Reihe von Gründen anstatt der funktionalen Speicherung eine Datenbank zu benutzen [SRB06] :

- Die Speicherung ist nicht mehr an die interne Darstellung der Daten gebunden und erlaubt eine flexiblere Architekturgestalltung und cross-platform Applikationen.
- Zu undo/redo und Versionierung werden Daten zentralisiert und mit gewünschten Redundanzen gespeichert. Außerdem erleichtert die Verwendung von einer Datenbank die Skalierbarkeit einer Anwendung bezüglich dem Volumen und der Art der Speicherung von Daten.
- Transaktionen laufen atomar und isoliert voneinander ab. Nebenläufige Zugriffe werden intern optimiert und gesteuert.
- Die Datenbank überwacht automatisch Transaktionen. Es wird sichergestellt, dass die Daten nicht beschädigt werden und konsistent bleiben.
- Datenbank kann abgeleitete Informationen automatisch aktualisieren, ohne dass die Konsistenz von Daten programmiertechnisch überwacht werden muss. Das bedeutet eine direkte Unterstützung für abgeleitete Attribute in Graphersetzungssystemen.
- Eine Datenbank kann Informationen verteilt speichern, verwalten und unterschiedliche Benutzersichten auf Datenbestand bieten, was sehr behilflich bei großen Projekten mit mehreren Entwicklern ist.

Im Folgenden werden die im Projekt IPSEN entstandene Datenbank GRAS und sein Nachfolger DRAGOS präsentiert.

2.2.1 GRAS

Die Datenbank GRAS entstand am Lehrstuhl 3 für Informatik an RWTH Aachen im Rahmen des Projekts IPSEN, und ist daher gemäß dessen Anforderungen entwickelt worden [LS88]. GRAS bietet die Grundfunktionalität, die für Entwicklungswerkzeuge benötigt wird. Dies sind unter Anderem: persistente Speicherung, undo/redo Operationen, Versionierung von Graphen. Um die inkrementelle Analyse zu ermöglichen werden bei Änderungen der Daten entsprechende Ereignisse ausgelöst. Mehrere Entwicklungswerkzeuge können mit der Datenbank gleichzeitig kommunizieren.

In seinen ersten Versionen unterstützte GRAS nur ungetypte Graphen, erst später durch Anforderungen von IPSEN Werkzeugen kamen typisierte Graphen hinzu.

Dadurch, dass die Datenbank im Laufe ständig an neue Bedürfnisse des Projektes angepasst wurde, wurde es schließlich schlecht wartbar. Als ein Ausweg wurde GRAS einem Neuentwurf unterzogen in dem GRAS3 entstanden ist.

Die bereits benötigte Funktionalität wurde in GRAS3 neukonzeptioniert und von Grund auf implementiert. Eine Reihe von Diensten wurde überarbeitet, so zum Beispiel müssen hierarchische Graphen nicht mehr auf flachen Graphen abgebildet, sondern können auch als solche gespeichert. Es wurden auch Sichten für verschiedene Benutzergruppen eingeführt. Änderungen betreffen nicht nur die Funktionalität der Datenbanken, sondern auch die Ausführung. Wenn in GRAS die Operationen von dem Server durchgeführt werden, so sind sie jetzt zu Klienten ausgelagert.

Ein wichtiges Problem der GRAS Familie ist die strikte Entwicklung als Datenrepository für IPSEN-Werkzeuge. Das bedeutet, dass es unter Umständen mit unnötigen Funktionalität belastet ist und gleichzeitig nicht allgemein genug ist, um der Anforderungen aus anderen Bereichen gerecht zu sein. Durch ihre monolithische Aufbau kann die Datenbank nur mit einem großem Aufwand in der Funktionalität erweitert oder verändert werden.

So wird bei Graphtransaktionen die Information für undo/redo Funktionalität, sowie Versionierung mitverwaltet. In einigen Fällen wird sie aber nicht benötigt und bedeutet nur einen zusätzlichen Aufwand und eine erhöhte Redundanz in der Datenbank.

Da das Graphmodell von GRAS nur um benötigte Konstrukte erweitert wurde, werden nur gerichtete, attributierte, knoten- und kantenmarkierte Graphen unterstützt. Das stellt eine erhebliche Einschränkung an dargestellte Graphen. Auch die Erweiterung in GRAS3 reicht nicht um zum Beispiel das Graphmodell von oben besprochenen DiaPlan darzustellen, man ist gezwungen auf eine aufwändige Abbildung auszuweichen.

Der Einsatz von GRAS und GRAS3 in industriegroßen Anwendungen wies einen weiteren Mangel auf. Bei der Bearbeitung von sehr großen Graphen, müssen temporär sehr große Datenmengen in dem Hauptspeicher verwaltet werden, wobei GRAS und GRAS3 sehr schnell an ihre Grenzen stoßen.

Da die Programmiersprache Modula nicht mehr aktiv verwendet wird, veraltet auch die GRAS-Datenbank. Die Portierung von der Datenbank auf andere Betriebssysteme ist schwer und aufwendig. Bis jetzt wird nur UNIX unterstützt. Ein weiteres Problem bei GRAS3 ist die mangelnde Kompatibilität mit dem Vorgänger. Die Datenbanken bieten unterschiedliche Schnittstellen an, aus diesem Grund wurde GRAS3 nie von IPSEN-Werkzeugen eingesetzt.

2.2.2 DRAGOS

Als eine konsequente Fortsetzung, basierend auf Erkenntnissen aus GRAS wurde DRAGOS (**D**atabase **R**epository for **A**pplications using **G**raph **O**riented **S**torage) entwickelt. Bei dem Entwurf des Datenbanksystems wurden nicht nur die Anforderungen aus bestehenden Projekten berücksichtigt, sondern auch ein besonderes Augenmerk auf die Erweiterbarkeit für künftige Problemstellungen gelegt.

Zum einen wurde bei DRAGOS ein allgemeines Graphmodell entworfen. Das sollte nach Möglichkeit alle zur Zeit von Graphwerkzeugen verwendete Modelle überdecken. Die existierenden Graphmodelle können dann mit einem geringen Aufwand, ohne Zusatzkonstruktionen auf die Datenbankmodell abgebildet werden. Aus diesem Grund beherrscht DRAGOS hierarchische Graphen mit sehr allgemein definierten Enthaltenbeziehungen. Neben Kanten gibt es auch Relationen von beliebigen Kardinalität. Die Kanten, sowie die Relationen sind graphübergreifend. Im Gegensatz zu GRAS lässt DRAGOS keine ungetypten Graphobjekten zu. Alle verwendete Graphelementklassen werden in einem Graphschema gespeichert. Die Klassen können durch mehrfache Vererbung definiert werden. Es werden auch abstrakte Klassen unterstützt. Neben Graphelementendefinitionen werden auch Attributdefinitionen im Graphschema gespeichert, sodass auch abgeleitete Graphelemente entsprechende Eigenschaften von Vorfahren erben.

DRAGOS unterstützt sowohl interne als auch abgeleitete Attribute. Die abgeleiteten Attribute erlauben Eigenschaften eines Objektes ohne großen Laufzeitkosten aktuell zu halten. Das Programm muss die Werte von solchen Attributen weder selbst führen, noch auf Ereignisse aus der Datenbank warten, um Werte entsprechend anzupassen.

Die Datenbank stellt einen Ereignisdienst zur Verfügung. Bei jeder Änderung wird die Information über den geänderten Objekt und dem Graphspeicher in einem Ereignis übermittelt. Es können neben den vordefinierten auch die eigene Ereignisse beschrieben werden.

Der Hauptgrund, wieso GRAS nicht mehr weiter verwendet wird ist die schlechte Wartbarkeit. Um dieser Fehler nicht zu wiederholen, wurde die Architektur von DRAGOS methodisch mit der Rücksicht auf eventuelle zukünftige Anforderungen erarbeitet. Die Struktur der Datenbank ist von strikter Funktionalität- und Datenabstraktion geprägt. Der Kernel von DRAGOS bietet eine minimale Ansammlung der Funktionen, die von einer Datenbank verlangt werden: persistente Speicherung, Transaktionenverwaltung und Ereignisse. Die aufgezählte Funktionalität lässt sich nicht außerhalb des Kernels realisieren, ohne an Performanz oder klare Strukturierung zu verlieren.

Alle weiteren gewünschten Funktionen werden mittels Plug-ins dazugeschaltet. Dieser serviceorientierte Aufbau erlaubt eine flexible Gestaltung von der Datenbank, ohne sie mit unnötigen Modulen zu überlasten und führt somit auch zu einer besserer

Performanz. Somit wird DRAGOS auch für zukünftige Anwendungen konzipiert.

Die anpassungsfähige Konstruktion von DRAGOS betrifft auch die Speicherung von Graphen. Unterschiedliche Hintergrundspeicher können an DRAGOS angeschlossen werden. Dies bringt Vorteile zum Beispiel beim Testen. Eine schnelle Hauptspeicherimplementierung wird bei der Entwicklung benutzt und für den Einsatz wird eine relationale Datenbank angeschlossen. Der Integrationsschritt von der Entwicklungsumgebung zur Einsatzumgebung im Bezug auf Datenspeicherung verläuft dann durch DRAGOS abstrahiert und erfordert keine zusätzlichen Kosten.

Eine Neuerung in der Speicherung ist nicht nur die flexible Austauschbarkeit von Hintergrundspeicher. Es ist möglich mehrere, unter Umständen unterschiedliche, Datenbanken als Backend zu nutzen. Also werden es nicht nur graphübergreifende Transaktionen unterstützt, sie können sich sogar über mehrere Hintergrundspeicher erstrecken.

Neben dem Transaktionsmanager gibt es auch eine Regelmaschine. Sie ruft beim Auffinden eines bestimmtes Graphmusters eine dazugehörige Aktion auf. Das erweitert das Konzept der Ereignisse um Aktionen, die von der Datenbank ausgeführt werden. So kann der Datenbestand ständig überwacht werden und notfalls, durch Maßnahmen instand gesetzt werden. Der Graph wird nicht nur passiv, von der Seite Graphmodells, sondern auch aktiv durch die Regelmaschine, konsistent gehalten werden.

DRAGOS ist so aufgebaut, dass alle Module wie der Transaktionsmanager, die Regelmaschine, die Verbindung zum Hintergrundspeicher durch eine Ansammlung von wohl definierten Schnittstellen an den Datenbankkernel angeschlossen sind. Das bedeutet, dass sie sich durch andere, an bestimmte Aufgaben angepassten Implementierungen ersetzen lassen. Zum Beispiel kann der Transaktionsmanager durch einen anderen, mit der Unterstützung von verteilten Transaktionen auf der Basis von CORBA oder verwandten Systemen ersetzt werden. Der Ereignismanager wird auch entsprechend angepasst und stellt eine ähnliche Funktionalität wie die Ereignisdienst in CORBA. Er ermöglicht die Kommunikation zwischen den, an der Datenbank angebundenen, Anwendungen. Änderungen an Daten werden durch Ereignisse weiter vermittelt.

Als ein zusätzliches Werkzeug zum DRAGOS wird SUMAGRAM (**S**upporter for the **MA**pping of **GRA**ph **M**odels) angeboten. Es dient dazu, den Aufwand für die Realisierung von spezifischen Graphmodellen zu reduzieren. Das spezifische Graphmodell wird mit Hilfe von UML beschrieben. Nicht nur bestimmte Datentypen, sondern auch die Beziehungen zwischen denen werden festgehalten. Darüber hinaus wird das Modell mit einfachen und verbreiterten Mitteln definiert, es bedarf keine zusätzliche Werkzeuge für die Beschreibung von Graphobjekten. Aus UML-Diagramm wird mit Hilfe von SUMAGRAM eine XML-Spezifikation erzeugt, die neben der Infor-

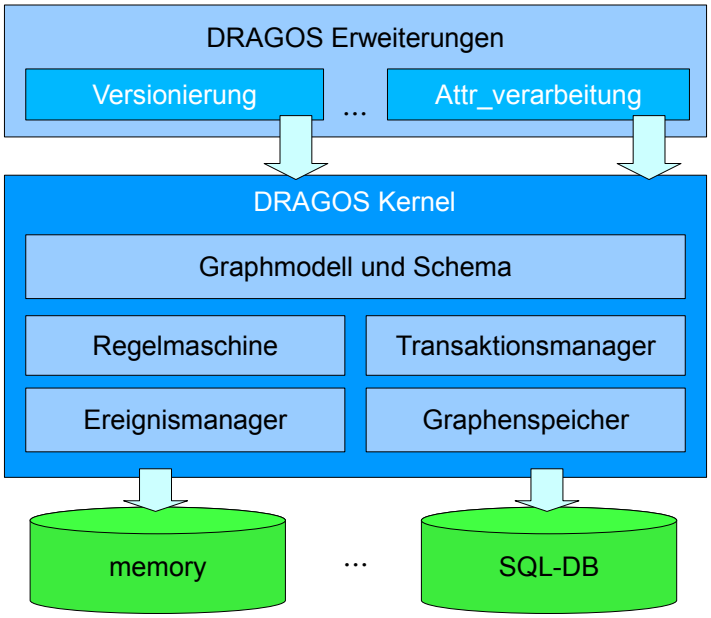

Abbildung 2.8: DRAGOS Architektur

mation aus UML noch Stereotypen für Spezifikation in DRAGOS enthält. Sie werden dazu benötigt, um einzelne Datentypen durch passende Graphobjekte zu repräsentieren. Aus diesen Dokumenten wird eine Reihe von Java-Klassen erzeugt, die die Abbildung vom spezifischen Graphmodell auf das DRAGOS-Graphmodell realisieren.

Nicht alle Graphmodelle können auf diese Weise abgebildet werden, vor allem ein Graphmodell, das rein prozedural beschrieben wird. Als eine Alternative kann solch eines Modell als Quelltext an vordefinierte Klasse angebunden werden. So fallen zusätzliche Kosten für die Implementierung von Modellspezifikation ein, durch das bereitgestellte Rahmenwerk werden sie jedoch in Rahmen gehalten.

3 Gesamtprojekt

In dieser Arbeit behandelten Probleme und Lösungsansätze sind ein Teil des Gesamtprojekts, das weiter in diesem Kapitel in Detail dargestellt wird.

Das Ziel des Projektes ist eine bessere Unterstützung für den Entwurf, die Erstellung und den Einsatz von Graphwerkzeugen. Das umfasst sowohl Werkzeuge zum Speichern und Bearbeiten von Graphen, als auch formale Spezifikationen von Graphen und Anfrage- und Transformationssprachen. Es wird auch eine entsprechende Unterstützung für Rapid Prototyping erarbeitet.

Die meisten der heutzutage existierenden Systeme (siehe Kap. 2.1) haben eine strikt vorgeschriebene Werkzeugkette. Einzelne Instrumente und Spezifikation sind somit gebunden und können weder einzeln verwendet, noch geändert werden. Das stellt einen Gegensatz zur Datenverarbeitung mit der Hilfe von relationalen Datenbanken. Eine relationale Datenbank und die mit ihr agierenden Anwendungen sind von einer strikter Daten- und Funktionsabstraktion geprägt.

Die Veränderungen in Graphregeln erfordern eine Neuegenerierung und eine Einbindung von dem resultierenden Code in das Programm, wodurch ein höhere Aufwand durch mehr Dokumente und deren Verwaltung während der Entwicklung entsteht. Das wirkt sich negativ bei der Entwicklung von Prototypen aus, da ein Prototyp laufend verändert und an zusätzliche Anforderungen angepasst wird. Das verursacht hohe zusätzliche Kosten.

Die Information, die in Graphtransformationspezifikation enthalten ist, wird durch die automatische Codegenerierung auf atomare ausführbare Operationen heruntergebrochen und ist somit für einen Menschen kaum lesbar. Das schränkt die Wiederverwendung von Teilen der Spezifikation ein.

Diese Probleme werden durch den Verzicht auf die obligatorische Codegenerierung und auf das dazugehörige Rahmenwerk gelöst. Dadurch werden die Anwendungen flexibler gestaltet, die Funktionalität einer Anwendung und einer Datenbank werden klar getrennt. Die Kommunikation zwischen der Datenbank und dem Graphwerkzeug findet mittels einer Anfrage- und Transformationssprache statt, wie es in kommerziellen Datenbanken üblich ist.

Das Gesamtprojekt basiert auf der Datenbank DRAGOS und besteht aus mehreren Unterprojekten. Es soll eine Basissprache, sowie eine Reihe von Erweiterungsmodulen zu DRAGOS entstehen. Durch diese Module soll die Sprache realisiert werden, sowie

notwendige Werkzeuge angeboten werden.

Das Gesamtprojekt setzt sich aus folgenden Teilen zusammen.

- Der Entwurf und das Formalisieren der Sprache DRAGULA (**DRAG**OS **U**nifed **LA**nguage).
  Es soll eine Basissprache definiert werden, die hinreichend ausdrucksstark ist, um andere Sprachen und Graphmodelle abzubilden. Die Sprache bildet die Grundlage für die Graphtransformationswerkzeuge und erlaubt die Kommunikation zwischen einer Anwendung und der Datenbank.

- Die Implementierung eines generischen Moduls zur Interpretation von DRAGULA Ausdrücken.
  Die Sprache soll nicht nur als eine abstrakte Beschreibung existieren, sondern auch praktisch anwendbar sein. Zu diesem Zweck wird ein generisches Erweiterungsmodul für DRAGOS entwickelt. Das Modul soll die Konstrukte in DRAGULA auswerten/ausführen und an DRAGOS weiterleiten. Dadurch wird die Schnittstelle zur Datenbank geschafft.

- Die Implementierung von Hintergrundspeicher-spezifischen Modulen.
  Da DRAGOS verschiedenen Hintergrundspeicher für die Speicherung der Daten einsetzen kann, werden für einige Type der Datenbanken Module für einen schnelleren und effizienten Zugriff gebaut. Man nutzt das Wissen über den jeweiligen Backend aus und transformiert und optimiert Anfragen entsprechend.

  Ein Beispiel für eine relationale Datenbank: eine Anfrage in DRAGULA wird in eine Anfrage in SQL überführt. Durch Zusammenfassen von mehreren Anfragen, die sonst in viele kleine zerstreut wären, entsteht das Optimierungspotenzial auf der Seite der Datenbank.

- Gängige Sprachen wie PROGRES oder GrasQL werden auf DRAGULA abgebildet.
  So profitieren auch die bereits bestehenden Entwicklungen, die in anderen Graphtransformationssprachen formuliert sind, von der besseren Unterstützung der Graphwerkzeugen. Eine rein textuelle Notation von GrasQL kann sehr vorteilhaft sein, bei der direkten Benutzung der Sprache DRAGULA in Anwendungen.

Wie oben beschrieben ist, wird eine Reihe von den Instrumenten entwickelt und theoretisch fundiert, um die Arbeit mit den graph-basierten Daten zu erleichtern und einen methodischen Entwurf von Graphwerkzeugen, auch basierend auf unterschiedlichen Graphmodellen, zu ermöglichen.

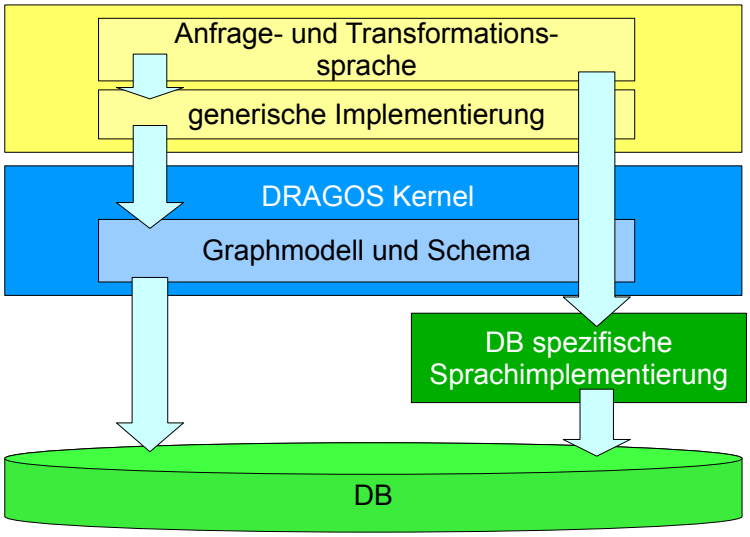

Abbildung 3.1: Datenbankarchitektur mit zusätzlichen Modulen

Die Datenbank wird wie es in der Abbildung 3.1 gezeigt mit Zusätzlichen modulen versehen.

In weiteren Kapiteln wird der Kern dieser Arbeit, eine universelle Basissprache, besprochen. Es werden bereits existierende Grundlagen und die Anforderungen an die Sprache ausformuliert und die Sprache nach diesen Richtlinien entworfen.

4 Graphspeicher

Nun wird die für diese Arbeit relevante Graphdatenbank DRAGOS detaillierter beschrieben (siehe auch Kapitel 2.2.2). Es werden die Grundzuge und die Eigenschaften DRAGOS dargestellt. Anschließend, als der Ausgangspunkt für spätere Definitionen, werden das Graphschema und das Graphmodell formal festgelegt.

Als eine Grundlage für eine Graphsprache dient das Graphmodell, auf dem die Sprache Anfragen und Transformationen formuliert. Es legt die atomare Graphobjekte und die zulässige Beziehungen zwischen diesen fest.

Wie im Kapitel 2.2 gezeigt wurde, werden für die Speicherung größeren Datenmengen, insbesondere bei der Benutzung durch unterschiedliche Werkzeuge, die Datenbänke bevorzugt. Es ist auch sinnvoll als die Basis einer Sprache ein Graphmodell einer Datenbank zu wählen, hier wurde für das Graphmodell der Datenbank DRAGOS entschieden.

Die Datenbank DRAGOS ist im Moment eine, der modernen Datenbanken für graph-basierte Daten. Sie ist in Java implementiert und läuft somit unter allen gängigen Betriebssystemen. Um die Fehler des Vorgängers GRAS zu vermeiden geschah die Entwicklung methodisch und gut dokumentiert, was auch die Einblicke in die Infrastruktur der Datenbank erlaubt. Die Architektur der Datenbank wurde mit einem besonderen Wert auf die Erweiterbarkeit und eventuellen Anforderungen, die in zukünftigen Anwendungen anfallen können, entworfen. Es bildet eine gute Grundlage für weitere Entwicklungen für DRAGOS.

Sehr günstig in diesem Fall ist auch die modulare Bauweise der Datenbank. Eine Reihe Diensten kann in Form von Modulen an den Kern der Datenbank angeschlossen werden. Die Module können sowohl die grundlegende Funktionalität der Datenbank implementieren, als auch sie erweitern.

Dadurch dass die Module eine gewisse Kenntnis von datenbankinternen Abläufen haben, besteht ein Potenzial zur Optimierung. Es können die Anfragen an der Anwendung-Datenbank Schnittstelle zusammengefasst, verzahnt und ausgeführt werden oder an der Schnittstelle zum Hintergrundspeicher backendspezifische Fähigkeiten ausgenutzt.

Außerdem bietet die Datenbank ein sehr universelles Graphmodell, was essenziell für die Entwicklung neuer Graphwerkzeuge ist.

In folgenden Unterkapiteln wird das Graphmodell von DRAGOS beschrieben. Es

besteht aus zwei Teilen: dem Abschnitt 4.1 mit dem Graphschema, das die Information zu den Datentypen von Graphentitäten enthält und dem Abschnitt 4.2 wo das Graphmodell definiert wird, das die gespeicherte Elemente beschreibt.

4.1 Graphschema

Ähnlich zu Data Definition Language in Datenbanken werden Typen sowie Beziehungen einzelner Objekttypen in einem Schemamodell definiert. Alle Entitäten in der Datenbank DRAGOS sind typisiert. Es gibt eine Möglichkeit auch untypisierte Elemente zu speichern, indem eine extra Klasse für untypisierten Elemente eingeführt wird, als Aggregation aller Typen.

Ein vordefiniertes Schema hilft grobe Fehler bei der Verarbeitung der Daten zu vermeiden und bringt wohldefinierte Verhältnisse zwischen verschiedenen Typen ein. Anderseits ist das Schema insofern allgemein definiert, dass sich auch spezifische Graphschemata einbetten lassen.

Mit Hilfe vom Schemamodell wird die Konsistenz der Daten automatisch von der Datenbank sichergestellt. Dadurch werden auch die externe Werkzeuge auf das Erhalten der Graphstruktur überprüft.

Das Graphschema wird in DRAGOS wie folgt in UML definiert (Abb. 4.1). [Böh04]

Jede Klasse wird durch ihren Namen repräsentiert. So müssen die gleich benannte Klassen, etwa in einer Pakethierarchie, durch ihren vollständig qualifizierten Namen bezeichnet werden.

Damit ein benutzerdefiniertes Graphschema möglichst den wenigen Restriktionen unterliegt, schreibt das DRAGOS Schema nur wenige Eigenschaften der Graphentitäten vor. Die Semantik der Vererbung wurde sehr allgemein definiert, so dass sich auch andere Graphschemata, mit unterschiedlichen Vererbungsmechanismen, darauf abbilden lassen. Die Eigenschaften von mehreren Graphobjektklassen können an eine Klasse weitergegeben werden.

Die Vererbung stellt eine nicht zyklische Beziehung dar. Ein Vorfahre einer Klasse muss die Klasse gleicher Art sein, zum Beispiel erbt eine Knotenklasse von einer Knotenklasse. Eine Ausnahme bilden hier die Graphentitätklassen, von denen jede beliebige Klasse abgeleitet werden darf. Die Klassen können auch als abstrakt markiert werden, solche Klassen können dann nicht instanziert werden und dienen nur den Vererbungszwecken.

Jedes Element im Graphschema, bis auf die Attribute selbst, kann mit den Attributen versehen werden. Als eine Wertemenge kann jeder beliebiger Datentyp dienen, die einzige Einschränkung, die er unterlegen soll, dass der Typ das Java-Interface `Serializable` implementieren muss. Um die Abbildung von verschiedenen spezifi-

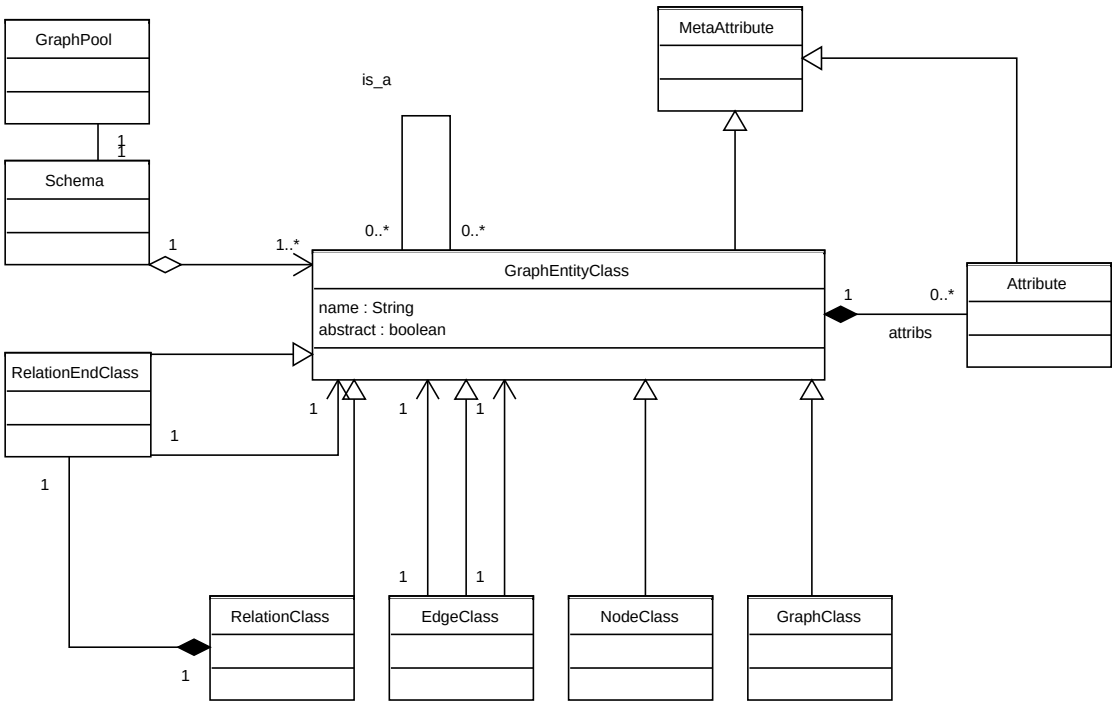

Abbildung 4.1: DRAGOS-Graphschema

schen Graphschemata zu erleichtern stellt DRAGOS die Metaattribute bereit. In diesen Attributen werden die internen Informationen zum Aufbau von Graphobjekten verwaltet und sind von dem Benutzer verborgen und beeinflussen die Ausführung von Transformationsregeln in dem Sinne nicht. Die Metaattribute können an alle Graphobjekte, auch an die Attribute, angehängt werden.

Die Attribute werden durch ihren Namen identifiziert, der Name muss eindeutig bezüglich der Klasse sein. Die Attributen aus der Oberklasse werden dabei nicht berücksichtigt.

Es wird an dieser Stelle vermieden, sich für eine bestimmte Semantik der Vererbung zu entscheiden. Es gibt verschiedene Strategien Konflikte bei gleich benannten Attributen aus der Oberklassen zu lösen. Entweder werden die Attribute umbenannt oder bei dem Zugriff explizit die Oberklasse angegeben. DRAGOS referenziert die Attribute nicht durch ihren Name, sondern durch einen eindeutigen systemweiten Bezeichner, dadurch können die Attribute mit einem gleichem Namen in einer Vererbungshierarchie unterschieden werden.

Zwei elementaren Klassen sind die Graph- und die Knotenklassen, sie enthalten neben den Klassennamen keine zusätzlicher Information in den vordefinierten Attributen.

Die Kantenklassen enthalten neben dem Namen der Klasse auch die Angaben zur zugelassenen Anfangs- und Endklassen, sowie die Kardinalitäten einer Kante. Ob eine Klasse der Kanten gerichtet ist wird durch einen Flag signalisiert.
Die Kardinalitäten werden beim Anlegen einer Kante vom DRAGOS Kernel überprüft. Würde es nicht zu diesem Punkt geschehen, müsste es später innerhalb der Regeln nachgeprüft werden, was die Transformationspezifikation überlädt und eventuell auch den Systemdurchsatz verringert.

Die Relationen können eine beliebige Anzahl der Graphentitäten verbinden, deswegen wird eine Klasse der Relationende eingeführt. Ähnlich zu den Kanten werden die zulässigen Klassen als Ziel der Tentakel und die Kardinalität festgelegt. Die Relationenden enthalten keine Rollenbeschriftungen und unterscheiden sich durch ihre Klassen. Durch das Schema wird vorgegeben, welche Relationenden eine Relation haben darf. Durch die Relationen lässt sich auch der Konzept der Hyperkanten realisieren.

Da das Graphschema nur in UML gegeben ist, wird für die späteren Formalismen eine Beschreibung benötigt, wo man auch die einzelne Klassen ansprechen kann und nicht nur eine Ansammlung von gleichartigen Klassen. Dazu existieren mehrere Möglichkeiten. Zum einen werden die Graphen und die Graphtransformationen mittels Algebra beschrieben. Alle zulässige Graphen bilden eine Grundmenge und Transformationen sind die Operationen auf Graphen. Diese Beschreibung ist recht kompakt und beruht auf einer solider mathematischer Grundlage.

Leider eignet sich die algebraische Beschreibung nicht für spätere Sprachdefinitionen. Deswegen benutzt man hier eine mengenorientierte Schreibweise. Alle relevanten Objekte werden in entsprechenden Mengen verwaltet. Das kann eventuell aufgrund von vielen zerstreuten Mengen umständlich sein, bietet aber den Vorteil, dass alle einzelne Objekte ansprechbar sind.

Man führt für jede Sorte von den Klassen eine Menge ein. Die Beziehungen zwischen Mengen und einzelnen Klassen werden durch die Funktionen gegeben. Nun wird das Graphschema wie folgt beschrieben:

Definition 4.1. Sei $\mathbb{SM}$ das Schemamodell

$$\mathbb{SM} = (\mathcal{V}, \mathcal{G}, \mathcal{N}, \mathcal{E}, \mathcal{R}, \mathcal{D}, \mathcal{AD}, Inh, TypAttr, TypSrc, TypTrg)$$

Als $\mathcal{U}$ wird die Menge der Klassen im Schemamodell bezeichnet $\mathcal{U} = \mathcal{V} \cup \mathcal{G} \cup \mathcal{N} \cup \mathcal{E} \cup \mathcal{R} \cup \mathcal{D}$

- $\mathcal{V}$ ist die Menge der abstrakten Klassen von Graphentitäten. Es werden Attributen an abgeleitete Klassen weitervererbt, eine Instantiierung ist nicht möglich.
- $\mathcal{G}$ ist die Menge von Graphklassen

- $\mathcal{N}$ ist die Menge von Knotenklassen
- $\mathcal{E}$ ist die Menge von Kantenklassen, dazu gehören zwei Funktionen $TypSrc$ und $TypTrg$ die jeder Kantenklasse $e \in \mathcal{E}$ eine Menge der Klassen aus $\mathcal{U}$ zuordnen, die Quelle bzw. Ziel einer Kante der Klasse e sein dürfen.
- $\mathcal{R}$ ist die Menge von Relationenklassen
- $\mathcal{D}$ ist die Menge von Relationenden zu einer Klasse $\mathcal{R}$
- $\mathcal{AD}$ ist die Menge von Attributdefinitionen, jeder Klasse aus $\mathcal{U}$ kann durch die Funktion
$$TypAttr : \mathcal{U} \longrightarrow \mathcal{AD}$$
eine Menge der Attributdefinitionen zugeordnet werden.
- $Inh : \mathcal{U} \longrightarrow 2^{\mathcal{U}}$ ist die Funktion, die Vererbungsbeziehung herstellt. Diese Funktion ordnet jeder Klasse eine Menge der Oberklassen zu.
Die Vererbungsbeziehung darf nicht zyklisch sein, also gilt
$$\forall c_1 \dots c_n \in \mathcal{U} | \Big(\forall i \in \{1 \dots n-1\} | \, (Inh(c_i) \in c_{i+1}) \wedge c_1 \neq c_n \Big)$$

■

Diese Beschreibung beinhaltet alle einzelne Klassen aus dem Graphschema. Nachteil dieser Definition ist der Verlust der gesamten Zusammenhangs, wie es in UML-Diagramm zu finden ist. Die Abhängigkeiten werden in der Funktionen gekapselt und liefern die Aussagen zur einzelnen Klassen, genau diese Information wird relevant bei dem Betrachten / der Erzeugen der Graphobjekten.

4.2 Graphmodell

Die eigentliche Objekte, aus denen Graphen zusammengesetzt werden, werden in einem Instanzmodell aufgefasst. Da die Enthaltensein-Beziehung und die Kanten im DRAGOS sehr allgemein definiert sind, ist man dazu gezwungen eine Oberstruktur, einen Graphpool, zu beschreiben. Auch das im Kapitel 4.1 beschriebene Schema ist innerhalb eines Graphpools gültig. Somit wird hier die Verwaltung von Graphobjekten auf der Ebene des Graphpools betrachtet.

Das DRAGOS Graphmodell wird durch die UML-Diagramm (Abbildung 4.2) wie folgt definiert.

Ein Graphpool enthält ausschließlich Graphen. Jeder Graph in einem Graphpool wird eindeutig durch seine Rollenangabe identifiziert.

Ein Graph kann jede beliebige Anzahl von Graphobjekten beinhalten, also dazu gehören Knoten, Kanten, Relationen und Graphen selbst. Die Relationenden, obwohl sie zu den Graphobjekten zählen, sind nicht in einem Graphen enthalten, sondern in einer bestimmten Relation.

Die Möglichkeit in einem Graphen weitere Graphen zu erzeugen liefert die Unterstützung für die Modellierung hierarchischen Graphen. Nur die Graphen können weitere Graphen enthalten, das Graphmodell verbietet es für andere Graphobjekte. Es gibt jedoch einen Umweg, indem man graphwertige Attribute definiert und dadurch beliebige hierarchische Strukturen ermöglicht.

Die Kanten werden nicht als der Kartesische Produkt $N \times N$ definiert, da das DRAGOS Modell mehrere parallele gleich beschriftete Kante zulässt und diese Darstellung zu schwach wäre. Stattdessen wird jede Kante als eine eigenständige Entität dargestellt. Das hat auch den Vorteil bei der Realisierung von spezifischen Graphmodellen, weil solch eine Darstellung von Kanten allgemeiner als binäre Relation ist.

Ähnlich wie bei Kanten, werden die Relationen allgemein nicht als eine Untermenge $\mathbf{U}^{\mathbb{N}}$ dargestellt, obwohl durch die Klassendefinition eine Typisierung der Relationenden vorgegeben wird. Relationenden werden als eigenständige Graphelementen behandelt, zum Beispiel können Kanten zu und von Relationenden aus gezogen werden. Durch die Typisierung der Relationenden sind die Rollenbeschriftungen nicht länger erforderlich.

Jedes Element, bis auf Attribute selbst, kann mit Attributen versehen werden, die zulässigen Attributen und die zugehörigen Attributwerten werden im Graphschema definiert. Da die Wertmenge recht allgemein vorgegeben wird und lediglich die Implementierung von Java-Interface `Serializable` voraussetzt, können auch Graphobjekte und insbesondere auch Ansammlungen von Graphobjekten als Attributwerte dienen. Eine weitere Besonderheit ist das Flag, das die Gültigkeit eines Attributes signalisiert, somit entfällt die Notwendigkeit ein extra Wert, wie etwa `null` für nicht gültige Attribute einzuführen.

Ähnlich wie bei Graphschema benötigen die späteren Sprachdefinitionen ein Graphmodell, das einzelne Instanzen der gespeicherten Daten repräsentiert. Durch die UML-Diagramm wurden nur die Zusammenhänge der Graphentitäten gegeben, nicht aber der Bezug auf einzelne Objekte.

Definition 4.2. Sei $\mathbb{IM}$ das Instanzmodell

$$\mathbb{IM} = (\mathbf{G}, \mathbf{N}, \mathbf{E}, \mathbf{R}, \mathbf{D}, \mathit{Typ}, \mathit{Cnt}, \mathit{Attr}, \mathit{Src}, \mathit{Trg}, \mathit{Rel}, \mathit{RelEnd})$$

wobei $\mathbf{U} = \mathbf{G} \cup \mathbf{N} \cup \mathbf{E} \cup \mathbf{R} \cup \mathbf{D}$ ist die Menge Graphelementen, die im Instanzmodell $\mathbb{IM}$ enthalten sind.

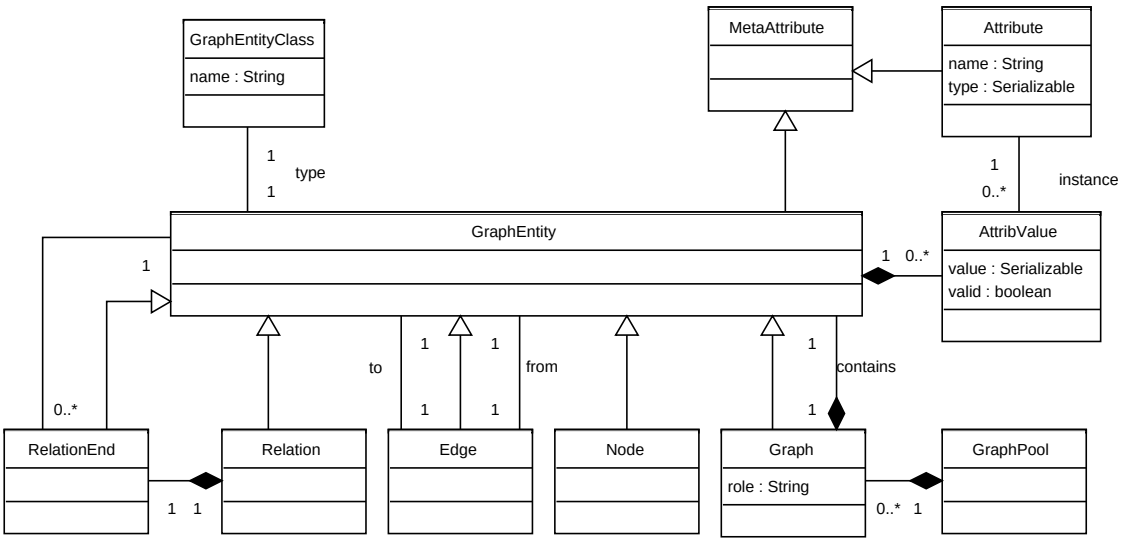

Abbildung 4.2: DRAGOS-Graphmodell

- **G** ist die Menge der im Graphpool $\mathbb{IM}$ enthaltenen Graphen.
- **N** ist die Menge von Knoten in den Graphen **G**
- **E** ist die Kantenmenge
- **R** ist die Menge von Relationen
- **D** ist die Menge von Relationenden
- Da jedes Element in dem Graphschema streng typisiert ist, ordnet die Funktion

$$Typ : (G \longrightarrow \mathcal{G}) \cup (N \longrightarrow \mathcal{N}) \cup (E \longrightarrow \mathcal{E}) \cup (R \longrightarrow \mathcal{R}) \cup (D \longrightarrow \mathcal{D})$$

  jedem Element eine entsprechende Klasse zu.

- Die Funktion

$$Cnt : (\mathbf{U} \longrightarrow \mathbf{G}) \cup (\mathbf{G} \longrightarrow \top)$$

  beschreibt Enthaltensein-Beziehung und ordnet jedem Element des Graphpools einen Graphen zu, in dem es enthalten ist. $\top$ bezeichnet hier das Graphpool und wird allen Topgraphen zugeordnet.
  Die beschriebene Beziehung darf nicht zyklisch sein, es gilt

$$\forall g_1 \dots g_n \in \mathbf{G} | \Big(\forall i \in \{1 \dots n-1\} | \, (Cnt(g_i) = g_{i+1}) \wedge g_1 \neq g_n \Big)$$

- $Attr : (\mathbf{U}, \mathcal{AD}) \longrightarrow Dom(\mathcal{AD})$ ist eine Funktion, die jeder Attributdefinition in einem Graphobjekt einen Attributwert zuweist. Eine Zuweisung hat nur dann ein Sinn wenn eine Attributdefinition $\mathcal{AD}$ für die entsprechende Klasse u in Graphschema beschrieben ist, also es gilt: $\mathcal{AD} \in AttrType(u)$
- Die Funktionen $Src : \mathbf{E} \longrightarrow \mathbf{U}$ und $Trg : \mathbf{E} \longrightarrow \mathbf{U}$ geben Quelle und Ziel einer Kante. Die Typen der Quelle und des Ziels müssen mit der Vorgabe im Schemamodell übereinstimmen.
- Die Funktion $RelEnd : \mathbf{D} \longrightarrow \mathbf{U}$ gibt das Graphobjekt, das durch das Relationende $d \in \mathbf{D}$ referenziert wird.
- Die Funktion $Rel : \mathbf{D} \longrightarrow \mathbf{R}$ gibt die Zugehörigkeit eines Relationendes $d \in \mathbf{D}$ zu einer Relation $r \in \mathbf{R}$.

■

Durch das Instanzmodell werden alle Objekte in einem Graphpoll referenziert, sowie die Bezüge zwischen ihnen. Dadurch lassen sich die Ausdrucke bezüglich Graphstrukturen formulieren.

4.3 Transformationen

Die Datenbank DRAGOS unterstützt keine Transformationen im Sinne der Graphtransformationswerkzeuge. Also es gibt keine Möglichkeit eine linke Seite der Regel zu formulieren, was von der Datenbank dann zu der rechten Seite transformiert wird.

Auch wie bei Anfragen an die Daten gibt es eine rein prozedurale Schnittstelle, die erlaubt die Garphentitäten zu erzeugen, zu verändern und zu löschen. Die Veränderungen an den Daten laufen unabhängig voneinander in Transaktionen. Je nach verwendetem Hintergrundspeicher werden die Eigenschaften der Transaktionen garantiert. So kann man zum Beispiel von dem Hauptspeicher die Persistenz der gespeicherten Daten nicht erwarten, hingegen erlaubt eine relationale Datenbank Transaktionen mit so genannten ACID-Eigenschaften. Unter anderem erlauben verschiedene Implementierungen von Backends auch die geschachtelte Transaktionen.

Es können zwar Aktionen an den Ereignismanager gekoppelt werden und bei der entsprechenden Veränderungen in Graphen angestoßen werden. Dieses Mechanismus ist aber zu schwach um komplexere Graphtransformationen zu durchführen und eignet sich lediglich zu den Konsistenzprüfungen des Datenbestands.

Wie man sieht, fehlt der Datenbank DRAGOS eine geeignete Schnittstelle oder eine Sprache um komplexere Transformationsregeln auszudrucken. Dieser Mangel soll durch die Entwicklung der Erweiterungsmodulen behoben werden (siehe Kapitel 3).

5 Graphsprache DRAGULA

Nachdem in dem Kapitel 4 die grundlegenden Formalismen eingeführt sind, wird die Basissprache für die Interaktion mit einer graph-basierten Datenbank behandelt.

Als Erstes um die Grundlage für die spätere Diskussion zu schaffen werden im Kapitel 5.1 die formalen Anforderungen an die Sprache festgelegt.

Danach, nacheinander aufbauend werden im Kapitel 5.2 die Zusammensetzung der Sprache und Sprachkonstrukte eingeleitet und nachfolgenden Kapiteln genau erläutert.

In jedem Abschnitt wird die Grundidee mit Hilfe von Beispielen und einer informeller Beschreibung umrissen und hierauf formell und möglich präzise zusammengefasst.

Anschließend, im Kapitel 5.6, werden die Eigenschaften der Sprache in Hinsicht auf die Anforderungen diskutiert.

5.1 Anforderungen

In diesem Kapitel werden die wichtigen Punkte für den Entwurf einer Anfrage- und Transformationssprache auf graphbasierten Daten zusammengefasst und erläutert. Sie stellen die Motivation für spätere Entscheidungen bei der Definition der Sprache dar. An dieser Stelle wird versucht unterschiedliche positive Aspekte aus anderen Sprachen und Graphersetzungssystemen zu übernehmen, siehe dazu das Kapitel 2.1.

Der besondere Augenmerk bei dem Entwurf der Sprache liegt auf an der Erweiterbarkeit und Anpassungsfähigkeit an anwendugspezifische Aufgaben. Dadurch soll der Einsatzgebiet der Graphwerkzeuge erweitert werden und das Bauen von Graphersetzungsystemen vereinfacht werden.

Die Sprache soll folgenden funktionalen Anforderungen genügen.

Anfragen und Transformationen auf Graphen. Die Sprache soll auf dem Aufgabengebiet der Graphersetzungssystemen und Graphwerkzeuge eingesetzt werden. Damit sollen die Anfragen an die gespeicherten graph-basierten Daten, sowie das Verändern von Daten möglich sein.

Unterstützte Graphobjekte. Die Sprache soll unterschiedliche Graphelemente unterstützen. Das gilt besonders für im DiaPlan erwähnten hierarchischen Graphen und Hyperkanten. Selbstverständlich müssen auch die herkömmlichen Graphen, Knoten und Kanten verarbeitet werden. Es müssen auch die Eigenschaften der Objekte, sowie Beziehungen untereinander in der Sprache ausdrückbar sein.

Typisierung von Graphobjekten. Eine große, strukturierte Entwicklung ist kaum ohne Typisierung von Objekten denkbar. Also wird es nach Konstrukten in der Sprache verlangt, die Zugehörigkeit zu einer Klasse überprüfen.

Eigenschaften eines Objektes. Eng verbunden mit einer Typbeschreibung eines Objektes sind dessen Eigenschaften. In jedem Graphbjekt kann die Information in Form von Attributen gespeichert werden. Aus diesem Grund soll die Sprache die Attribute abfragen und konsistent, entsprechend der Typbeschreibung, setzen können.

Abgeleitete Konstrukte. Die Sprache soll auch abgeleitete Sprachkonstrukte, etwa wie Pfadausdrücke, zur Verfügung stellen.

Strukturierung von Ausdrücken. Da die Sprache sowohl für das Prototyping, als auch für große Graphersetzungssysteme konzipiert wird, muss eine Möglichkeit vorgesehen werden komplexe Transformationsvorschriften strukturiert, zum Beispiel geeignet abgekürzt oder vereinfacht darzustellen und zu bearbeiten.

Kontrollstrukturen. Es sollen Sprachkonstrukte zur Ablaufkontrolle der Transformationen zur Verfügung stehen. Einzelne Transformationsvorschriften können so zu einem komplexem Programm zusammengefügt werden.

Alle oben vorgestellten Anforderungen betreffen die Funktionen und Konstrukte der Sprache. Es werden aber die Anforderungen auch an die Sprachrealisierung und Umgebung, in der die Sprache angeboten wird, gestellt, die nicht unmittelbar mit der Formulierung von Graphtransformationen verbunden sind.

Erweiterbarkeit. Neue Sprachkonstrukte sollen einfach beschrieben und in die Sprache eingebettet werden. Diese Konstrukte können aus elementaren vordefinierten Konstrukten zusammengesetzt werden, oder auch eine vollkommen unterschiedliche Semantik haben und direkt von der Auswertungsmaschine interpretiert werden. Das erlaubt dem Entwickler eigene, an den Aufgabenbereich angepassten Konstrukte zu entwerfen.

Anpassungsfähigkeit. Die Sprache soll einfach an eine neue Umgebung angepasst werden. Bereits definierte Konstrukte und Transformationsvorschriften sollen auch in neuen Kontext verwendbar sein. Es kann eine spezifische Palette der Konstrukte und der Werkzeuge entwickelt und verwendet werden. Dadurch steigt die Wiederverwendbarkeit und der Aufwand für die Entwicklung ähnlicher Produkte reduziert sich.

Verschiedene Graphmodelle. Damit die Entwicklung von Graphwerkzeugen möglich wäre, benötigt man ein möglichst allgemeines zugrunde liegendes Graphmodell. Weitere Graphmodelle werden dann darauf abgebildet.

Anwendungsunabhängigkeit. Die Auswertungsmechanismen sollen nach Möglichkeit von der Anwendung unabhängig sein. Das wird zum Einem durch eine klar definierte Schnittstelle zwischen der Anwendung und der Datenbank gewährleistet. Zum anderem wird es auf Rahmenwerke verzichtet, die eventuelle Einschränkungen an die Anwendung stellen könnten.

Ausdruckstärke. Die Sprache soll ausreichend ausdrucksstark sein. Es ist nicht sinnvoll eine universelle Sprache zu konzipieren, die zwar ein allgemeines Graphmodell verwendet, aber von vornherein sehr an Ausdrucksstarke eingeschränkt ist.
Insbesondere wird von der Sprache die Unterstützung der gängigen Anfragen/Transformationen auf Knoten, Kanten und Relationen erwartet, sowie Pfad- und Attributfunktionen. Da das verwendete Graphmodell es hierarchischen Graphen erlaubt, soll auch die Möglichkeit bestehen sie zu benutzen.

Deklarativität. Die Transformationsvorschriften sollen leicht und verständlich beschrieben werden. Da die Sprache als ein Werkzeug zum Entwurf von Graphtransformationspezifikationen dienen soll, sollen damit verfasste Dokumente von Menschen geschrieben und gelesen werden können.

Die Erwartungen an die Erweiterbarkeit und die Anpassungsfähigkeit stellen neben einem ausreichend allgemeinem Graphmodell und der Ausdrucksstärke der Sprache einen sehr wichtigen Punkt für die Unterstützung von Graphwerkzeuge dar. Der Benutzer soll in der Lage sein seine eigenen Konstrukte zu definieren und konsequent zu benutzen. Es besteht auch die Möglichkeit die selbst definierten Ausdrucke, etwa aus Effizienzgründen, in die Auswertungsmaschine „festzuverdrahten“.

Die Anforderung an das Graphmodell ist zum Teil durch die Benutzung des Graphmodells von DRAGOS gelöst. Seitens der Datenbank wird ein allgemeines Graphmodell sowie die Werkzeuge für die Abbildung von spezifischen Graphmodellen angeboten.

5.2 Sprachdefinition

In diesem Abschnitt wird die Zusammensetzung der Sprache DRAGULA beschrieben. [Wei08] [Wei07b] [Wei07a] Der Aufbau der Sprache kann in Schichten zusammengefasst vorgestellt werden. Die unten liegenden Schichten benutzen die oberen.

In der Abbildung 5.1 wird die Sprachaufbau grafisch veranschaulicht. Die einzelnen Schichten haben folgende Funktionen, von elementaren zu komplexen Schichten.

- **Graphmuster.** Damit wird die Struktur eines zu suchenden Teilgraphen definiert und nach einem entsprechendem Ansatzpunkt für eine weitere Verarbeitung in Graphdaten gesucht. Die Anfragen selbst setzen sich aus folgenden Teilen zusammen.
  - Atomare Definitionen auf Graphentitäten. Dadurch können elementare Beziehungen zwischen den Graphobjekten definiert werden.
  - Strukturierung und Kombinierung von atomaren Ausdrücken. Es werden Unteranfragen mit einer entsprechenden Sichtbarkeitsregelung und einer Auswertungsreihenfolge eingeführt. Die Anfragen können sich aus mehreren Unteranfragen zusammensetzen.
  - Die voneinander unabhängigen Unteranfragen können mittels logischer Verknüpfungen verbunden werden.

- **Transformationsvorschriften.** Durch Transformationen wird der Datenbestand aktiv manipuliert. Die eigentlichen Veränderungen an Graphen werden mittels Transformationsoperationen formuliert.

- **Ablaufsteuerung.** Einzelne Anfragen und Transformationen werden zu komplexen zusammengesetzten Transformationsvorschriften kombiniert. Die Reihenfolge der Ausführung wird durch Kontrollfluss beschrieben. Parallel dazu findet entsprechende Parameterübergabe über Datenfluss statt.

Folgend, im Abschnitt 5.3, werden Graphmuster eingeführt. Es werden nacheinander einfachen und dann komplexeren Graphmuster eingeführt. Darauf beruhend werden im Kapitel 5.4 die Transformationen auf Graphen vorgestellt. Anschließend, im Kapitel 5.5 werden die Möglichkeiten vorgestellt um Transformationsvorschriften gesteuert auszuführen. Es werden der Kontrollfluss und der Datenfluss zwischen einzelnen Vorschriften definiert.

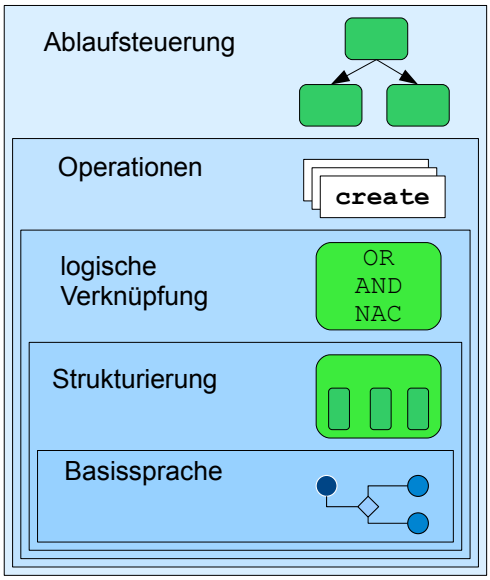

Abbildung 5.1: Zusammensetzung der Sprache DRAGULA

5.3 Suchmuster

In diesem Abschnitt werden Anfragen an Graphen und graph-basierten Daten vorgestellt. Durch die Angabe eines Suchmusters wird eine Anfrage an die zugrunde liegende Datenbank gestellt und nach einem Teilgraph in Daten gesucht. Auf diese Weise wird auch die linke Seite einer Transformationsregel formuliert.

Im Allgemeinem wird ein Suchmuster beschrieben, indem man anzufragende Graphobjekte durch Platzhalter, Variablen, repräsentiert und die erwünschten Beziehungen zwischen den Variablen mit den Bedingungen festlegt. Weiterführend werden die Anfragen strukturiert und mit einer weiteren Semantik durch logischer Verknüpfungen versehen.

Die Struktur eines Musters soll durch das UML-Diagramm in der Abbildung 5.2 veranschaulicht werden. Ein Graphmuster besteht aus mehreren Anfrageelementen, dazu zählen Variablen, Bedingungen und Mustern, die im Folgenden weiter erläutert werden. Analog für die Formalisierung des Schema- und Instanzmodell wird eine mengenorientierte Schreibweise für Anfragen eingeführt.

Ein Muster kann neben Variablen und Bedingungen weitere Muster enthalten. Die in einem Muster, aber nicht in einem Untermuster liegenden Variablen und Bedingungen werden als *direkt enthalten* bezeichnet. Die Enthaltensein-Beziehung darf nicht zyklisch sein. Die Variablen und die Bedingungen können in einer Beziehung zu wei-

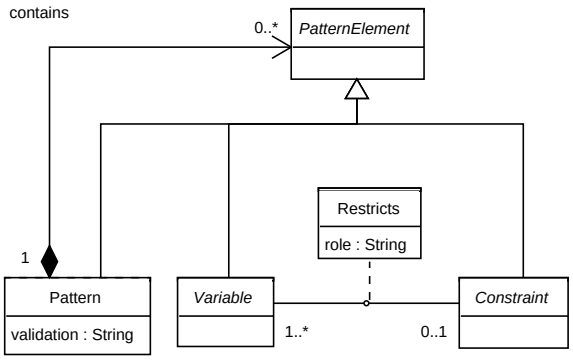

Abbildung 5.2: Metamodell: Anfragen

teren, tiefer liegenden Variablen und Bedingungen stehen.

Definition 5.1. Ein Suchmuster P ist ein 4-Tupel

$$P = (Var, Con, Pat, valid)$$

wird durch eine Ansammlung von den Mengen der Variablen Var, der Bedingungen Con und der Menge der Pattern Pat beschrieben.

- Var ist die Menge von Variablen
- Con ist eine Menge von Bedingungen
- Pat ist eine Menge von in dem Muster P enthaltenen Untermuster
- $valid$ gibt die Gültigkeitsbedingung des Musters an.

■

Im weiteren wird die in Kapitel 4 eingeführte Schreibweise für das Schemamodel $\mathbb{SM}$ und das Instanzmodell $\mathbb{IM}$ benutzt.

5.3.1 Variablen

Um einen gesuchten Graph zu beschreiben wird jedes für die Anfrage relevantes Objekt durch eine Variable repräsentiert. Jedes beliebige Objekt kann durch eine Variable angesprochen werden. Die Variablen sind insofern an das Graphmodell von DRAGOS angelehnt, da es für jede Objektklasse eine eigene Variablensorte gibt. Aus diesem Grund wird hier auch auf die im Kapitel 4 eingeführte Notation zurückgegriffen. So

werden die Knoten aus **N** durch die Variablen aus **NV** repräsentiert, entsprechend auch die Kanten aus **E** durch **EV** und so weiter. Die Menge aller möglichen Variablen wird, in Analogie zum Instanzmodell, durch **UV** bezeichnet. In dieser Ausarbeitung werden die Variablen durch Kreise mit einer passender Inschrift grafisch veranschaulicht.

Die Variablen an sich begrenzen die Suchergebnisse nur in dem Sinne, dass sie entsprechend der Sorten belegt werden. Weitere Beziehungen zwischen den Variablen oder die Einschränkung bestimmter Eigenschaften werden durch die Bedingungen gegeben.

Die Variablen stellen gewissermaßen den Kern der Sprache dar, da sowohl die Suchanfragen, als auch die, weiter unten beschriebene, Transformationen sich auf die Variablen stützen.

Um eine Verbindung zwischen variablen und tatsächlichen Graphobjekten zu herstellen wird der Begriff des Matches eingeführt.

Definition 5.2. Für die Anfrage $P = (Var, Con, Pat, valid)$, die die Variablen $Var = \mathbf{GV} \cup \mathbf{NV} \cup \mathbf{EV} \cup \mathbf{RV} \cup \mathbf{DV}$ beinhaltet, wird das Match M definiert, wobei durch M einer Variable höchstens ein Graphobjekt zugeordnet wird. Es müssen nicht alle Variablen belegt werden. Es muss beachtet werden, dass die Variablen entsprechend ihren Sorten belegt werden, die Funktion ist somit partiell definiert.

$$M : \mathbf{NV} \longrightarrow \mathbf{N} \cup \mathbf{EV} \longrightarrow \mathbf{E} \cup \mathbf{GV} \longrightarrow \mathbf{G} \cup \mathbf{RV} \longrightarrow \mathbf{R} \cup \mathbf{DV} \longrightarrow \mathbf{D}$$

■

Diese Definition beschreibt ein Match zunächst allgemein, ohne Bezugnahme auf die Bedingungen, die in einem Suchmuster enthalten sind.

5.3.2 Bedingungen

Die Variablen grenzen die Graphobjekte nur bezüglich deren Sorten ein. Um die Eigenschaften von Objekten und Beziehungen zwischen diesen zum Ausdruck zu bringen werden die Bedingungen (auch Constraints genannt) eingeführt. Sie grenzen mögliche Belegungen der Variablen weiter ein. Bei der grafischen Wiedergabe werden die Bedingungen als Rhomben mit passenden Beschriftungen dargestellt. Der Bezug einer Bedingung zu den Variablen wird durch Kanten dargestellt. Diese Kanten können mit zusätzlichen Beschriftungen, die auf eine vorgeschriebene Bedeutung hinweisen, versehen werden.

Beispiel 5.1. (*Eine einfache Anfrage*)
In diesem Beispiel wird nach zwei beliebigen, nicht notwendigerweise verschiedenen

Knoten gesucht, die mit einer Kante verbunden sind. Für jeden Graphobjekt in dieser Anfrage steht jeweils je eine Variable. Die Inzidenz zwischen den Knoten und der Kante wird mittels der `inc`-Bedingung sichergestellt (Abb. 5.3). Hierbei handelt es sich um eine ungerichtete Kante, andernfalls können die Kanten von den Bedingungen zu den Variablen mit den Rollenbeschriftungen wie `source` bzw. `target` ausgestattet werden.

Abbildung 5.3: Verbindung zwischen zwei Knoten

■

Nun werden die Bedingungen formal beschrieben. Ein Constraint c kann als eine Funktion angesehen werden, die einer Ansammlung von Werten $Var = x_1, x_2, \ldots, x_n$, hier den Graphobjekten, einen booleschen Wert zuordnet.

$$c = c(x_1, x_2, \ldots, x_n)$$

Zuordnungen zu Variablen können eine bestimmte Bedeutung haben, und aus diesem Grund mit Rollenanganben markiert werden. Das ist aus der textuellen Notation nicht sofort erkennbar und wird hier implizit durch die Position des Parameters gegeben. Zum Beispiel kommt eine Variable, die einen Container-Graphen bezeichnet, als erste in einer Bedingung $c(g, o_1, o_2, \ldots)$ vor und die in dem Graphen enthaltenen Objekte folgen nach.

In einer Anfrage $P = (Var, Con, Pat, valid)$ kann eine Bedingung $c \in Con$ einen Bezug auf jede in dem Muster enthaltenen Variablen nehmen. Diese Darstellung wird in dem Abschnitt 5.3.3 durch Einführen von Untermuster erweitert, sodass eine Bedingung zu weiteren Variablen in der Beziehung stehen darf.

Es sind das Schemamodell $\mathbb{SM}$ und das Instanzmodell $\mathbb{IM}$ wie bereits in Kapitel 4 beschrieben, definiert, und Suchmuster $P = (Var, Con, Pat, valid)$ gegeben. Eine Bedingung $c \in Con$ wird im Bezug auf ein Match M betrachten.

Bezeichnung	**Beschreibung**
`any` Beliebig	Diese Bedingung kann an beliebige Variablen $v_1, \ldots, v_n$ angebunden sein und liefert immer den Wert `true` zurück. $c(v_1, \ldots, v_n) =$ `true`
	Fortsetzung auf nächster Seite

Tabelle 5.1 – Fortsetzung

Bezeichnung	Beschreibung
`const` Konstante	Eine oder mehrere Variablen $v_1, \ldots, v_n$ werden auf einen festgelegten Wert $o \in \mathbf{U}$ eingeschränkt. Die Bedingung ist wahr genau dann, wenn gilt: $\forall i \in \{1..n\} \vert M(v_i) = o$
`inc` Inzidenz	Drückt die Inzidenz zwischen einer Kante, durch die Variable $e \in \mathbf{EV}$ bezeichnet, und einem Graphobjekt $o \in \mathbf{UV}$ aus und kann zusätzlich mit `target` und `source` Rollen beschriftet werden. Die Bedingung $c(e, o)$ ist genau dan erfüllt wenn gilt `target`: $Trg(M(e)) = M(o)$ `source`: $Src(M(e)) = M(o)$ Ohne Rollenangabe ist die Bedingung dann erfüllt, wenn entweder `target` oder `source` gilt.
`iso` Isomorphie	Formuliert paarweise Verschiedenheit der damit begrenzten Objekten $c(o_1 \ldots o_P)$ aus. $o_1, \ldots, o_P \in \mathbf{UV}$ $\forall i, j \in \{1...P\} \vert (i \neq j \Longrightarrow M(o_i) \neq M(o_j))$
`id` Identität	Die Identität von Objekten, bezeichnet durch die Variablen $\{o_1 \ldots o_P\} \in \mathbf{UV}$ wird sichergestellt. Bedingung $c(o_1 \ldots o_P)$ ist wahr falls gilt: $\forall i, j \in \{1...P\} \vert (M(o_i) = M(o_j))$
Fortsetzung auf nächster Seite	

Tabelle 5.1 – Fortsetzung

Bezeichnung	Beschreibung
`cont` Enthaltensein	Die Enthaltensein-Beziehung zwischen einem Graphen $g \in \mathbf{GV}$ und weiteren Objekten $\{o_1 \dots o_P\} \in \mathbf{UV}$ wird formuliert. Der Container wird durch Rollenangabe `container` gekennzeichnet. Die Bedingung $c(g, o_1 \dots o_P)$ ist erfüllt, wenn gilt $$\forall i \in \{1..P\} \mid (Cnt(M(o_i)) = M(g))$$
`type` Typangabe	Der Datentyp $t \in \mathcal{U}$ der Objekte $\{o_1 \dots o_P\} \in \mathbf{UV}$ wird vorgeschrieben. Die Angabe des Typs erfolgt als eine Eigenschaft der Bedingung und kommt somit nicht explizit in der textuellen Notation vor. Die Bedingung $c(o_1 \dots o_P)$ ist wahr, falls gilt: $$\forall i \in \{1..P\} \mid (Typ(M(o_i)) = t)$$
`relatesto` Zugehörigkeit Relationenden	Die Bedingung stellt die Inzidenz eines Relationendes $re \in \mathbf{DV}$ zu einer Relation $r \in \mathbf{RV}$ fest. Die Bedingung $c(r, re)$ ist wahr, falls gilt $$Rel(M(r)) = M(re)$$
`relate` Relationbeziehung	Durch diese Bedingung wird die Beziehung zwischen einem Graphobjekt $o \in \mathbf{UV}$ und einem Relationende $rd \in \mathbf{DV}$ beschrieben. Es soll für die Bedingung $c(rd, o)$ gelten: $$RelEnd(M(rd)) = M(o)$$
`AttrVal` Attributbedingung	Ein Attribut eines Objektes $o \in UV$ wird bezüglich seines Wertes eingeschränkt. Die Semantik der Bedingung c(o) wird mittels einer externen Sprache definiert und wird zunächst nicht näher beschrieben. Es ist möglich Werte mehrerer Attribute mit Hilfe der Bedingung abzufragen und auszuwerten.

Tabelle 5.1: Bedingungen

Um die hier eingeführte Schreibweise zu begründen, wird das PROGRES `query` Konstrukt gegenüber DRAGULA Anfragen gestellt.

Beispiel 5.2. (*Schreibweisevergleich*)
Hier wird eine Anfrage nach Existenz einer Liste mit genau zwei Knoten betrachtet. Die Listen werden wie im Beispiel 2.1 definiert. Eine solche Anfrage wird in PROGRES mit dem Konstrukt `query` realisiert. Alle Eigenschaften der Objekte sind durch Beschriftungen vorgegeben.

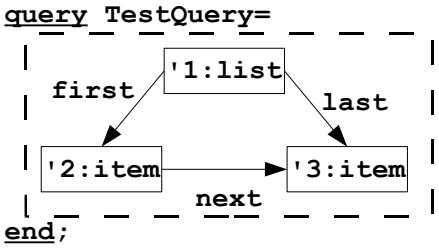

Abbildung 5.4: PROGRES Anfrage

Die dazu äquivalente Anfrage in DRAGULA sieht wie folgt aus (Abb. 5.5.) Die Eigenschaften, sowie die Inzidenz der Objekte werden durch Bedingungen festgehalten.

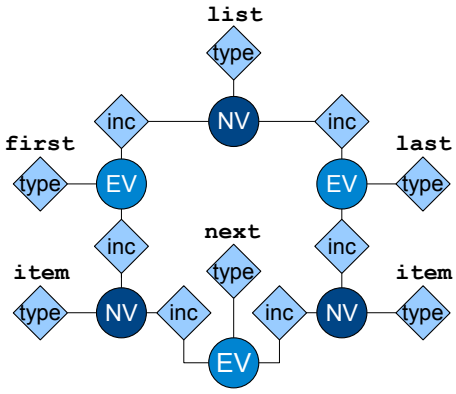

Abbildung 5.5: DRAGULA Anfrage

■

DRAGOS behandelt jedes Graphobjekt als ein sogenannten First-Class-Citizen und unterscheidet sie durch eine systemweite Bezeichnung. Daran knüpft DRAGULA an, und referenziert intern die Objekte durch DRAGOS Bezeichner. Dies erlaubt mehrere Objekte mit gleichwertigen Attributen, die von anderen Systemen nicht unterschieden

werden können. Zum anderen gibt es auf dem Graphschemaniveau viele Objekte mit gleichen Attributdefinitionen und Eigenschaften. So wäre eine PROGRES-ähnliche Schreibweise sehr überladen. Auf diese Weise werden für eine Anfrage nicht relevante Aspekte gar nicht aufgeführt. Das kann aber auch Nachteile bringen, da eine Spezifikation bei sehr vielen Attributen und festgelegten Eigenschaften überdetaliert wird.

Eine strikte Unterscheidung zwischen den Variablen, die für Objekte stehen, und den Bedingungen darauf erlauben es auf eine einfache Art und Weise neue Bedingungen einzuführen. Komplexeren Bedingungen werden aus den Variablen und den Bedingungen zusammengesetzt und können dann weiter als ein eigenständiger und abgeschlossener Konstrukt benutzt werden. Die Kapselung von den Bedingungen bringt mit sich auch die Vorteile für die Auswertung von Anfragen: die spezifischen Konstrukte müssen nicht unbedingt auf atomare Operationen zurückgeführt werden und können von einer anwendugngspezifischen Implementierung direkt ausgewertet werden. Das hilft auch die Beschreibung der Transformationen einfach zu halten.

Beispiel 5.3. (*Konstruktdefinition*)
Es wird ein Adjazenzbedingung basierend auf der Inzidenzbedingung definiert. Eine Kantenvariable und zwei Inzidenzbedingungen werden in einem geschlossenem Konstrukt gekapselt. Eine Anfrage nach zwei zusammenhängenden Knoten wie im Beispiel 5.1 würde wie folgt aussehen.

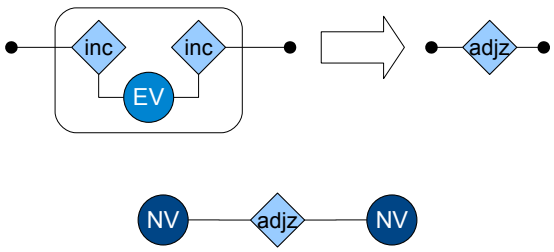

Abbildung 5.6: Konstruktione eigener Bedingung

Bei der Auswertung wird der Adjazenzbedingung auf die Inzidenzbedingungen abgeleitet werden. Es ist aber möglich den Konstrukt mit der Hilfe eines spezifisches Moduls auszuwerten, ohne ihn auf Atomaroperationen zurückzuführen. Etwa dadurch, dass in einer extra geführten Adjazenztabelle nachgeschlagen wird. ■

Die Suche nach einem Teilgraphen in der Daten wird auf ein bekanntes Bedingungserfüllungsproblem aus dem Bereich der künstlichen Intelligenz abgebildet. Das

Problem ist gut erforscht und hat eine solide theoretische Grundlage [Tsa93], was bei dem Entwurf von Auswertungsmaschinen von Vorteil sein kann. Auch wenn das Problem im Allgemeinen als NP schwer gilt, gibt es für einen Graphmuster endlicher Größe immer eine Lösung in Polynomialzeit.

Bei der Suchanfrage, die durch ein Muster formuliert wird, wird nicht nach einem beliebigen Match gesucht, sondern nach einem Teilgraphen, der die geforderten Eigenschaften erwartet. Es wird hier eine formale Definition von Eigenschaften eines Matches gegeben. In diesem Zusammenhang wird die Definition des Matches verfeinert.

Definition 5.3. Zu einem Graphmuster $P = (Var, Con, Pat, valid)$, hier ohne Untermuster, also $Pat = \emptyset$ und einer beliebigen Gültigkeitsbedingung $valid$, wird ein Match $M : Var^*(P) \rightarrow \mathbf{U}$ als bedingungskonform genannt genau dann, wenn folgendes zutrifft:

- M **ist vollständig**, wenn alle Variablen $v \in Var^*(P)$, die durch Bedingungen eingeschränkt sind, belegt werden .
- M **ist korrekt**, wenn alle Bedingungen $c(v_1 \ldots v_n) \in Con$ erfüllt sind.

■

Wie aus dieser Definition hervorgeht müssen nicht alle Variablen in einem Muster und eventuellen Obermustern belegt werden. Nur die Variablen, die mit einer Bedingung versehen sind, haben eine semantische Bedeutung, der Rest kann entweder mit beliebigen Graphobjekten oder überhaupt nicht belegt werden. Um eine Belegung zu erzwingen wird die `any`-Bedingung benutzt.

5.3.3 Muster

Um die Anfragen zu strukturieren und zu ordnen, und als Folge auch mehr Übersichtlichkeit zu schaffen, können die einzelnen Unteranfragen in Mustern (auch Patterns genannt) gekapselt werden. Im Folgenden werden weitere Funktionen, wie etwa logische Bindungen der Anfragen und ein rekursiver Aufbau der Patterns beschrieben. Die Muster werden grafisch als abgerundeten Rechtecken dargestellt. Sie können weitere Anfrageelemente beinhalten,

geschachtelte Muster

Es ist sinnvoll große Anfragen zu strukturieren. Die Anfragen können in einzelne abgeschlossene Bestandteile zerteilt werden. Es schadet aber der Lesbarkeit und der

Verständlichkeit, wenn es viele zerstreute Teile einer Anfrage gibt. Eine Lösung dieses Problems sind strukturierten Unteranfragen. Die logisch zusammengehörenden Teile einer Anfrage werden auch strukturell und dem entsprechend auch grafisch zusammengefasst. Dadurch werden Sichtbarkeitsregel auf den Variablen erstellt und die Auswertungsreihenfolge von den Unteranfragen beeinflusst.

Beispiel 5.4. (*Geschachtelte Anfragen*)
Ein Muster kann eine oder mehrere Anfragen zu einer neuen geschachtelten Anfrage zusammenfassen. In diesem Beispiel wird nach Knoten `item`, die mit einer Kante mit einem Knoten von Typ `list` verbunden sind, gesucht. Die Anfrage enthält eine Unteranfrage um die Suche nach `item`-Knoten zu kapseln (Abb. 5.7.)

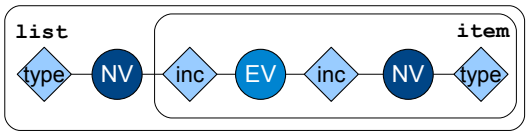

Abbildung 5.7: Eine geschachtelte Anfrage

■

In einer Anfrage $P = (Var, Con, Pat, valid)$ kann eine Bedingung $c \in Con$ einen Bezug auf jede in dem Muster und in Obermustern enthaltenen Variablen nehmen.

Die Beziehung zu Variablen in Untermustern kann zu gewissen Problemen bei der Auswertung führen und wird im Folgenden beschrieben (siehe Beispiel 5.5).

Die Variablen aus den Obermustern werden durch Var^* bezeichnet und wie folgt rekursiv definiert.

$$Var^*(P) = Var \cup Var^*(O) \big| P \in O$$

Die Auswertung der geschachtelten Muster wird strikt von außen nach innen durchgeführt. Entsprechend der Auswertungsreihenfolge werden die Sichtbarkeitsregeln festgesetzt: man stützt sich auf die bereits ausgewerteten Ausdrucke. Also sind die Variablen bereits mit Graphobjekten belegt, sodass alle Informationen für die Auswertung einer Bedingung vorliegen. Die Variablen werden stets so belegt, dass keine Bedingung dadurch explizit verletzt wird.

Sei p_i ein Untermuster, das in einer Reihe geschachtelter Muster

$$P = p_1, p_2 \ldots, p_i, \ldots, p_n$$

enthalten ist, wobei p_1 das äußerste und p_n das innereste Muster ist. Dann sind die Variablen, die in den äußeren Mustern $p_1, p_2, \ldots, p_{i-1}$ definiert sind, für das Untermuster p_i sichtbar.

Mit diesen Regeln sind die Querverweise zwischen zwei Untermustern auf dem gleichen Schachtelungsniveau verboten. Die Sichtbarkeitsregeln für die tiefer enthaltenen Untermuster haben keinen Sinn, da die Untermuster noch ausgewertet werden müssen und es kein Bezug auf die darin enthaltenen Variablen genommen werden kann.

Es soll beachtet werden, dass durch das Zulassen von Bezügen zu Variablen in Untermuster es zu Konflikten beim Auswerten von Graphmustern kommen kann. Zum Einen, da nicht alle Variablen zu dem Zeitpunkt der Auswertung einer Bedingung belegt werden, muss die Bedingung mehrmals auf Gültigkeit überprüft werden. Zum Anderen wird eine Bedingung in einigen Situationen niemals erfüllt, aber auch nicht verletzt werden. Das ist durch die definierte Sichtbarkeitsregelung bedingt. Beim Auswerten eines Untermusters bleiben Teile anderer Untermuster verborgen. Das macht solche Bedingungen nutzlos oder sie haben nicht den erwarteten Einfluss auf die Ergebnisse der Auswertung. Dies wird durch das folgende Beispiel verdeutlicht werden.

Beispiel 5.5. (*Bedingung über mehrere Muster*)
In diesem Beispiel wird nach zwei identischen Knoten gesucht. Das Problem ist die Schachtelung der beiden Knotenvariablen, sie befinden sich auf dem gleichem Niveau und sind somit füreinander unsichtbar.

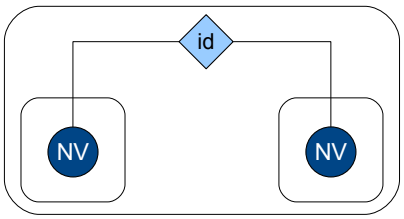

Abbildung 5.8: Geschachtelte Anfrage mit der überflüssigen Bedingung

Als Folge verläuft die Suche nach Knoten unabhängig voneinander, da jeweils die andere Variable nicht sichtbar ist. Es ist leicht nachvollziehbar, dass die gefundene Belegung nicht die erwarteten Eigenschaften aufweist und die Paare von nicht identischen Knoten enthält.

Das Muster kann formal beschrieben werden. Die Untermuster enthalten keine Bedingungen und haben folgende Struktur:

$$p_1 = (\{n1\}, \emptyset, \emptyset)$$
$$p_2 = (\{n2\}, \emptyset, \emptyset)$$

Das Gesamtmuster setzt sich dann aus p_1 und p_2 wie folgt zusammen.

$$P = (\emptyset, \{id(n1, n2)\}, \{p_1, p_2\})$$

Bei der Auswertung des Musters müssen für P keine Variablen belegt werden, es wird mit einem Untermuster fortgefahren. In dem Untermuster gibt es keine Bedingungen, die eine mögliche Belegung der Variable einschränken. Aus dem Kontext eines Untermusters ist die Variable in dem anderem Untermuster nicht sichtbar. Aus diesem Grund wird die Identitätsbedingung niemals verletzt. Somit wird die Knotenvariable mit einem beliebigen Knoten besetzt. Analog verläuft die Suche für das zweite Untermuster.

In diesem Beispiel wurde gezeigt, dass bei der Beziehung zwischen Bedingungen und Variablen in Untermustern zu Konflikten kommen kann. Die Bedingung in dem Beispiel wurde mehrmals ausgewertet und die Ergebnisse weisen nicht die gewünschten Eigenschaften aus. ■

Aus oben genannten Gründen, um eine eindeutige und effiziente Auswertung der Graphmustern zu ermöglichen, werden die Beziehungen von Bedingungen zu in Untemustern liegenden Variablen verboten.

logische Verknüpfungen der Muster

Eine naheliegende Idee, einzelne Unteranfragen nicht nur strukturell zusammenfassen, sondern auch logische Abhängigkeiten zwischen denen zum Ausdruck zu bringen. Es werden drei Grundtypen der Musterbindung angeboten: `OR`, `AND` und `NAC` Bindung.

Zum Beispiel wird eine Schachtelung, mit mehreren Unteranfragen, wobei mindestens eine davon gültig sein sollte, durch ein Muster mit `OR`-Verknüpfung vereinigt. Eine Anfrage, in der alle Untermuster gelten müssen, wird durch ein mit `AND` beschrifteten Muster gekapselt. Die Information über die Gültigkeitsbedingung wird in einem Muster $P = (Var, Con, Pat, valid)$ in dem Attribut $valid$ gespeichert.

Auf diese Weise werden die unterschiedlichen unabhängigen Anfragen, ohne gemeinsamen Variablen, zu einer Einheit zusammengefasst und verarbeitet. Wie man weiter unten im Kapitel 5.5.2 sieht, können die sich so zusammengefundenen Graphobjekte mit Hilfe des Datenflüsses weiterhin zusammen transformiert werden.

Ein interessanter Punkt ist dabei die Verneinung eines Musters mit der Hilfe von `NAC`-Pattern (**N**egative **A**pplication **C**ondition). Nur einige Sprachen, wie AGG und VIATRA2 [BV] können nicht nur einzelne Knoten oder Kanten verneinen, sondern wohldefinierte Muster. VIATRA2 kann außerdem auch die geschachtelten negierten Ausdrücke auswerten.

Beispiel 5.6. (*Logische Verknüpfung der Unteranfragen*)
Es wird eine Anfrage (Abb. 5.9) konstruiert, die die Konsistenz einer verketteter Liste

überprüfen soll. Es soll sichergestellt werden, dass es keinen Knoten mit zwei unterschiedlichen einlaufenden oder auslaufenden `next`-Kanten gibt.

Diese Bedingung wird in zwei Unteranfragen zerteilt. In der linken Unteranfrage wird die Existenz der einlaufenden Kanten überprüft, in der rechten der auslaufenden. Genau diese Muster sollen in einer konsistenten Liste nicht gefunden werden und werden aus diesem Grund verneint, die Unteranfragen tragen die `NAC` Beschriftungen. Die Tatsache, dass beide Unteranfragen zur gleichen Zeit gelten müssen wird durch `AND`-Beschriftung am enthaltenden Muster zum Ausdruck gebracht.

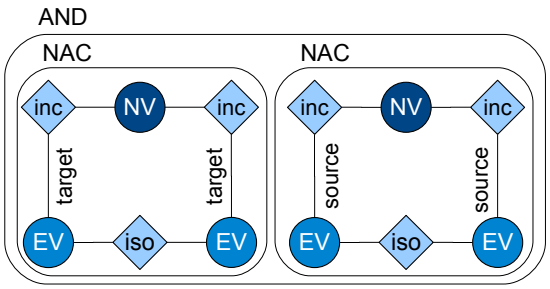

Abbildung 5.9: Anfrage mit der Verknüpfung

■

Durch die Schachtelung der Muster und die Negation können sehr komplexe Zusammenhänge zum Ausdruck gebracht werden, etwa wie Allquantoren. Dies wird durch folgendes Beispiel anschaulich gemacht wird.

Beispiel 5.7. (*Modellieren eines Allquantors*)
In dieser Anfrage (Abb. 5.10) wird die Tatsache dass alle Nachfolger von dem, durch die linke Knotenvariable repräsentierte, Knoten von Typ `item` sind, überprüft.

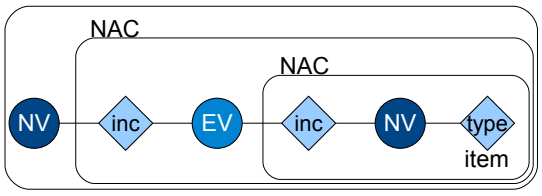

Abbildung 5.10: Allquantor

Die Anfrage wird mittels eines verneinten geschachtelten Muster ausgedrückt. Es wird nach so einem Knoten gesucht, von dem es keine Kante gibt, die zu einem Knoten führt, für den die Typbedingung nicht gilt. Die linke Knotenvariable repräsentiert

in dieser Anfrage der Kontext, in dem die quantisierte Ausdruck angewendet wird. Die Kantenvariable ist eine quantisierte Variable und die rechte Knotenvariable eine logische Bedingung an sie. Es wird also, nach so einem Kontext gesucht, für den für alle Belegungen der quantisierten Variable die Bedingung erfüllt ist.

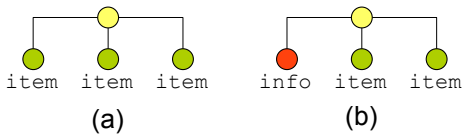

Abbildung 5.11: Beispielgraphen für Allquantor

Es ist offensichtlich, dass der Graph auf der Abb. 5.11a) die Anfrage erfüllt, der Graph 5.11b) hingegen nicht. Beim zweiten Graphen wird die rechte Knotenvariable mit dem Knoten von Typ `info` belegt. Das macht das inneres `NAC`-Muster gültig und verletzt somit die mittlere Unteranfrage, da sowohl die Unteranfrage, als auch die Kantenvariable gültig belegt werden können. Dadurch wird die Gesamtmuster verletzt. ■

Es muss an dieser Stelle dazu geraten werden, die logische Verknüpfungen durchdacht und vorsichtig einzusetzen, insbesondere gilt das für die Verneinung. Ansonsten werden die Anfragen schnell undurchschaubar und kaum zu verstehen sein.

rekursive Muster

Alle bisherige Konstrukte erlauben keine komplexeren und abgeleiteten Anfragen, etwa wie Pfad- oder Baumausdrucke. Ein Pfad kann mit der Hilfe eines unendlichen Musters beschrieben werden, und zwar wie die Abbildung 5.12 es schematisch darstellt. Das Muster beinhaltet implizit alle Pfade beliebiger Länge.

Natürlich lassen sich die unendliche Muster nur schwer handhaben. Um sie doch darstellen und benutzen zu können, macht man sich den systematischen Aufbau solcher Muster zunutze.

Ein Pfad der Länge 0 ist gleich einer Identitätsbedingung auf zwei Knotenvariablen. Aus einem Pfadausdruck der Länge n, bekommt man einen Pfad der Länge $n + 1$ indem man daran noch einen adjazenten Knoten anhängt und somit den Pfad verlängert. Zusätzlich muss in jedem Schritt überprüft werden, ob man von dem Anfangspunkt aus den erwünschten Punkten erreicht hat.

Es wird außer der Grundstruktur auch der Aufbauschritt beschrieben. Das Ausbauen von dem Muster kann mit einem sogenannten Verweis auf ein Pattern bewerkstelligt werden. Falls das Suchen erfolgreich ist, wird das Muster um ein weiteres

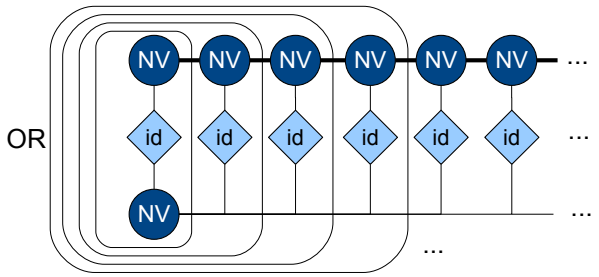

Abbildung 5.12: Unendliches Muster für Pfade

Untermuster expandiert und nochmals gesucht. Also bleibt das Muster endlich beschreibbar und kann bei der Suche zur Laufzeit expandiert werden.

Durch die Klebekanten zu einer Referenz wird die "Eingabe" eines kopierten Musters angegeben (siehe Beispiel 5.8). Als der Standard wird beim Kopieren die gleiche Eingabe benutzt wie ursprünglich.

Beispiel 5.8. (*Rekursives Muster*)
In diesem Beispiel wird das Muster für Pfadausdrücke aus der Abbildung 5.12 zusammengefaltet.

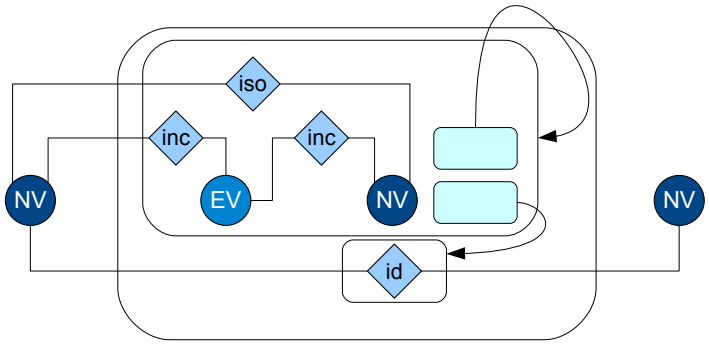

Abbildung 5.13: Rekursives Muster für Pfade

Es wird mit diesem Muster wie folgt gesucht. Es wird überprüft, ob der Anfangs- und Endknoten nicht identisch sind. Falls ja, dann existiert ein Pfad der Länge 0.

Falls es gelingt den Pfad zu verlängern, wird das Muster expandiert und zwar nach den Verweisen, die Untermuster noch ein Mal in das Muster reinkopieren, siehe Abbildung 5.14. Die Klebekanten, hier nicht abgebildet, mappen die Eingabe von

neueerzeugtem Muster. Der reinkopierte Untermuster bezieht sich mittels Inzidenzbedingung nicht mehr auf den Anfangsknoten, sondern wurde auf den Knoten auf dem Pfadverlängerung gemappt. Mittels neuekopierten Identitätsbedingung wird sichergestellt, ob ein Pfad vom Anfang- zum Endknoten gefunden ist.

Falls der Versuch den Pfad zu verlängern scheitert, bedeutet es, dass alle Pfade von dem Anfangsknoten aus abgearbeitet sind und das Verfahren bricht ab.

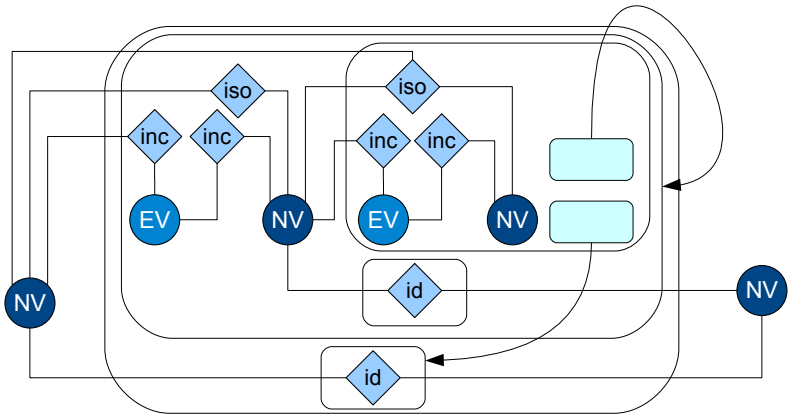

Abbildung 5.14: Expandieren des rekursiven Musters

Nach einem erfolgreichen Expansionsschritt wird das Kopieren von Mustern wiederholt. Diese Methode entspricht der Breitensuche und garantiert, falls ein Weg vom Anfangsknoten aus zum Endknoten existiert, in endlich viel Expansionsschritten den Pfad zu finden.

Das Verfahren terminiert in endlich viele Schritten, nämlich genau dann, wenn ein Pfad nicht mehre verlängert werden kann. Da Isomorphiebedingung die Kreisfreicheit auf Pfaden sicherstellt und in einem endlichen Graphen nur endlicher Länge kreisfreie Pfade gibt, kann ein Suchpfad nur endlich viele Male verlängert werden.

■

Auf gleiche Weise können auch andere Strukturen als Pfade beschrieben werden, zum Beispiel Ausdrucke auf Bäumen, wo auf jedem Pfad eine Bedingung gelten soll. Durch rekursive Muster wird dem Benutzer ein Ansatz der Graphgrammatik gegeben, um verwendete, domänespezfische Konstrukte zu beschreiben.

Es wurde bewusst auf ein vordefiniertes Pfadkonstrukt verzichtet. Obwohl fast jede Graphtransformationssprache mit Pfaden arbeitet und dazu auch bestimmte Ausdrücke bietet, unterscheiden sich die Ausdrücke in ihrer Semantik sehr stark. Stattdes-

sen bietet DRAGULA eine Möglichkeit eigene Konstrukte mit gewünschter Semantik zu beschreiben.

Von einem Match werden die folgende Eigenschaften gefordert: die Vollständigkeit und die Korrektheit. Ein vollständiges und korrektes Match wird dann gültig genannt. Nun wird die oben gegebene Definition der Eigenschaften eines Matches im Bezug auf eine Anfrage vollständig formalisiert.

Die Eigenschaften eines Matches M werden rekursiv auf der Anfrage P definiert.

Definition 5.4. Ein Match $M : Var^*(P) \rightarrow \mathbf{U}$ zu einem Muster $P = (Var, Con, Pat, valid)$ ist gültig genau dann, wenn gilt:

- P **ist nicht negiert,** $valid \neq \texttt{NAC}$
  Es wird gefordert, dass M bedingungskonform ist. Dazu kommt die Gültigkeitsbedingung für Untermuster.
  - `AND`: *Alle* Untermuster $p \in Pat$ müssen gültig sein.
  - `OR`: *Mindestens eines* der Untermuster muss gültig sein.

- P **ist negiert**
  Falls das Muster als die Gültigkeitsbedingung eine Beschriftung `NAC` hat, ist es nur dann gültig, falls es für $P' = (Var, Con, Pat, \texttt{AND})$ keine gültige Belegung gibt.

■

Für eine Anfrage kann es ein oder sogar mehrere gültige Matches geben. Falls für eine Anfrage keine gültige Belegung gefunden wurde, wird die Suche als erfolglos bezeichnet.

5.3.4 geschachtelte Matches

Die Schachtelung der Muster birgt in sich ein weiteres Problem, wenn es für Unteranfragen mehrere Matches gibt. In dem Match wird dann auch strukturelle Information über Aufbau des Graphmusters enthalten und muss auch konsequent benutzt werden, um einen gültigen Match ohne Mehrdeutigkeit zu bekommen. (Abb. 5.15)

Falls für ein Muster mehrere gültige Belegungen existieren, wird hier keine Mengenwertigkeit eingeführt. Die Transformationen werden auf einfachen "flachen" Graphen ausgeführt. Dabei wird eine Alternative aus eine Reihe möglichen Matches ausgewählt, dadurch wird die Komplexität der Ausführung und des Backtrakings in Rahmen gehalten.

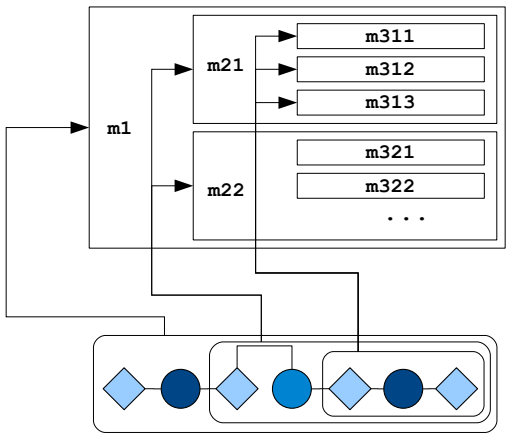

Abbildung 5.15: Match mit mehreren Untermatches

Die Ergebnisse der geschachtelten Unteranfragen müssen “entfaltet“ werden, indem die Matches aus der Unteranfragen zusammenkombiniert werden. Aus komplexen Matches für Untermuster wird ein Kreuzprodukt mit dem Ergebins für Obermuster gebildet. Die Problemstellung wird mit folgendem Beispiel verdeutlicht.

Beispiel 5.9. (*Match zu einem geschachtelten Muster*)
Es sind in der Datenbank zwei verketteten Listen mit mehr als einem Element gespeichert (Abb. 5.16.) Darauf wird die Anfrage aus dem Beispiel 5.4 ausgeführt.

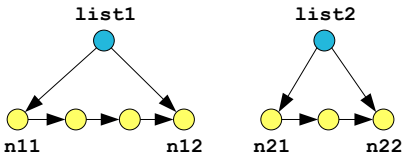

Abbildung 5.16: Schachtelung: Beispielinhalt der Datenbank

Zu dem Untermuster gibt es eine korrespondierende Belegung: für jede Liste werden ein Knoten von Typ `list` und zwei Knoten von Typ `item` gefunden. Da in der Datenbank zwei Listen sind, werden zwei Matches für jeweils `list1` und `list2` zurückgegeben, wobei sie zwei Matches für das Untermuster enthalten. In der Abbildung 5.17 sind Matches durch Rechtecken dargestellt und deren Inhalt durch Objejtbezeichner in dem Rechteck.

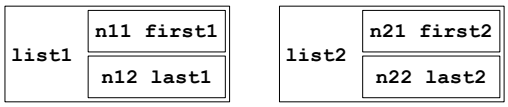

Abbildung 5.17: Schachtelung: Strukturierten Matches

Das Kreuzprodukt von äußerem und innerem Match bildet eine Reihe von gültigen Belegungn für das Suchmuster. Das strukturierte Match soll unter Befolgung der Schachtelung der Untermuster entfaltet werden. Aus jedem Match werden je zwei flachen gebildet, für die `list1` mit dem Knoten `n11` bzw. `n12` und analog für `list2` (Abb. 5.18.)

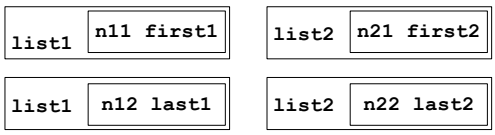

Abbildung 5.18: Schachtelung: Entfalteten Matches

■

Für einen Graphmuster

$$P = (Var, Con, Pat, valid)$$

wird ein Match M wie folgt entfaltet und dabei das Match $M^*(P)$ rekursiv konstruiert. Das Match für Variablen Var wird mit Matches für Untermuster $M^*(p), p \in Pat$ kombiniert.

$$M^*(P) = M(Var) \cup M^*(p) \big| p \in P$$

Falls zu einem Untermuster p mehrere Matches $M^*(p)$ gibt wird zufällig ein Match ausgewählt. Die Auswertungsmaschine stellt die Funktionalität zur Verfügung zur Laufzeit durch alle Kombinationen von Matches zu iterieren.

Für das Match in der Abbildung 5.15 würde das bedeuten, dass eine Reihe von Matches dafür gebildet wird.

```
m1 - m21 - m311
m1 - m21 - m312
m1 - m21 - m313

m1 - m22 - m321
m1 - m22 - m322
```

Auf die oben beschriebene Weise lassen sich verschiedene Graphmuster formulieren. Als Ergebnis einer Suche nach so einem Muster bekommt man ein Match, das eine Zuordnung von Graphobjekten zu Variablen darstellt. Die explizite Strukturinformation aus dem Muster wird an dieser Stelle verloren. Das ist aber auch nicht relevant für weitere Transformationen, da jedes Graphobjekt durch eine (eventuell auch mehrere) Variable referenziert wird und auch anhand dieser Referenz weiterverarbeitet wird.

5.4 Transformationsvorschriften

In diesem Abschnitt werden Operatoren beschrieben, die Daten in der Datenbank aktiv verändern. Ähnlich wie im vorherigen Abschnitt beziehen sich alle Operationen auf die Variablen.

Die Zusammenhang zwischen Variablen und Operatoren wird in der Abbildung 5.19 dargestellt. Die Grundidee ist, die Graphobjekte, hier durch die Variablen referenziert, mit der Hilfe von Operatoren zu verändern. An jede Variable kann ein oder mehrere Operatoren angehängt werden, die dann einen zugeordneten Graphobjekt entsprechend modifizieren. Obwohl die Operatoren an den Variablen angebunden sind, spricht man auch von Graphobjekten, da zu dem Zeitpunkt der Ausführung alle relevanten Variablen bereits belegt sind.

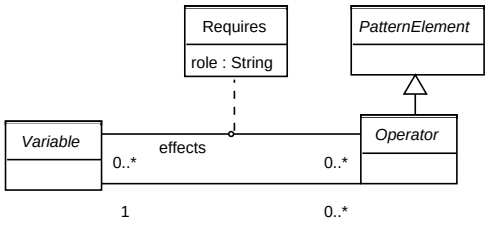

Abbildung 5.19: Metamodell: Transformationsvorschriften

Die Operatoren werden durch Rechtecke mit einer korrespondierenden Inschrift dargestellt.

Beispiel 5.10. (*Anhängen einen Knoten an eine Liste*)
In diesem Beispiel wird ein Knoten an das Ende einer verketteten Liste angehängt (Abb. 5.20.)

Als Erstes wird nach dem Ende der Liste gesucht, dafür wird die Kante `last` verfolgt. Im nächsten Schritt werden die Knoten und die Kanten, die mit den Operatoren

`create` versehen sind, erzeugt und miteinander, durch die Operatoren `connect` angegeben, verbunden. Damit die Liste konsistent bleibt, wird die Kante `last`, die jetzt auf den vorletzten Knoten zeigt, gelöscht.

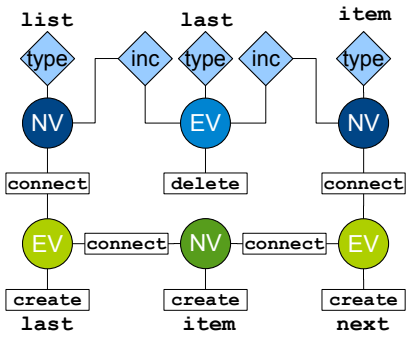

Abbildung 5.20: Einfügen eines Knoten

■

Nun werden die Operatoren formal beschrieben.
Sei $\mathbb{IM}$ das Instanzmodell wie in Kapitel 4.2 gegeben

$$\mathbb{IM} = (\mathbf{G}, \mathbf{N}, \mathbf{E}, \mathbf{R}, \mathbf{D}, Typ, Cnt, Attr, Src, Trg, Rel, RelEnd)$$

und $\mathbb{IM}'$ das Instanzmodell nach der Operatorausführung

$$\mathbb{IM}' = (\mathbf{G}', \mathbf{N}', \mathbf{E}', \mathbf{R}', \mathbf{D}', Typ', Cnt', Attr', Src', Trg', Rel', RelEnd')$$

Bezeichnung	Beschreibung
`create` Erstellen	Ein neues Graphobjekt wird erzeugt. Die Sorte des Objektes entspricht der Sorte der mit dem Operator verbundenen Variable. Zum Beispiel wird zu einer Kantenvariable $v \in \mathbf{EV}$ eine Kante e vom Typ $et \in \mathcal{E}$ in einem Graphen $g \in \mathbf{G}$ durch Operator `create` erzeugt, sodass die Gleichheit für Instanzmodelle gilt, bis auf $$\mathbf{E}' := \mathbf{E} \cup \{e\}$$ $$Typ' := Typ \cup \{e \to et\}$$ $$Cnt' = Cnt \cup \{e \to g\}$$ Analog für die anderen Graphentitäten.
`delete` Löschen	Das damit verbundene Graphobjekt wird gelöscht. So gilt für eine Knotenvariable $v \in \mathbf{NV}$ belegt mit dem Knoten $n \in \mathbf{N}$ nach der Ausführung die Gleichheit der Instanzmodelle bis auf $$\mathbf{N}' := \mathbf{N} \backslash \{n\}$$ Die internen Mechanismen in der Datenbank sorgen anschließend dafür, dass es keine hängenden Kanten bzw. Relationsenden gibt, indem sie gelöscht werden.
`connect` Verbinden	Der Operator `connect` verbindet eine Kante $e \in \mathbf{E}$ und ein beliebiges Objekt $o \in \mathbf{U}$. Für gerichtete Kanten kann der Operator mit einer Rollenangabe `target` oder `source` versehen werden. Nach der Ausführung gilt: `target`: $$Trg' := Trg \cup \{e \to u\}$$ `source`: $$Src' := Src \cup \{e \to u\}$$
`relateto` Relationenendenzugehörigkeit	Durch diesen Operator wird die Zugehörigkeit eines Relationenendes $re \in \mathbf{D}$ zu einer Relation $r \in \mathbf{R}$ festgelegt. Es gilt nach der Ausführung $$Rel' := Rel \cup \{re \to r\}$$
	Fortsetzung auf nächster Seite

Tabelle 5.2 – Fortsetzung

Bezeichnung	Beschreibung
`relate` Relation	Die Beziehungen zwischen einem Graphobjekt $o \in \mathbf{U}$ und einem Relationsende $re \in \mathbf{D}$ wird vereinbart. $$RelEnd' := RelEnd \cup \{re \rightarrow o\}$$
`SetAttr` Attributierung Setzen	Ein Attribut a eines Objektes $o \in \mathbf{U}$ wird auf entsprechenden Wert X gesetzt. $$Attr' := Attr \cup \{(o,a) \rightarrow X\}$$
`UnsetAttr` Attributierung Löschen	Ein Attribut a eines Objekts o wird als ungültig markiert. Die Attributwertzuordnung wird aus der Funktion $Attr$ gelöscht. Es gilt: $$Attr' := Attr \backslash \{(o,a) \rightarrow X\}$$

Tabelle 5.2: Operationen

Bei der Ausführung von Operationen kann es zu einem Konflikt kommen. Zum Beispiel wird versucht die Eigenschaften eines Knotens zu verändern, der noch nicht erzeugt wurde, oder etwa das Auslesen von Attributen von einem in diesem Transformation gelöschtem Objekt.

Um dieses Problem zu lösen, läuft die Ausführung von Operationen in mehreren Schritten ab. Die Reihenfolge der Ausführung wird durch die Prioritäten festgelegt, als ein Standard gibt es folgende 4 Prioritätsklassen.

- Die relevanten Eigenschaften Objekten werden gelesen.
- `create`-Operatoren werden ausgeführt, sodass alle, für weitere Schritte benötigte Objekte zur Verfügung stehen.
- Löschoperatoren werden ausgeführt. Die Graphobjekte, die gelöscht werden müssen, enthalten keine für die Weiterverarbeitung benötigte Information und können entfernt werden.
- Alle Schreiboperationen auf Eigenschaften und Attributen werden ausgeführt.

Auf diese Weise werden zyklische Abhängigkeiten von Attributoperationen gelöst. Durch die Reihenfolge der Operationen wird verhindert, dass nicht initialisierten Attribute aus neueerzeugten Objekten gelesen werden, und dass in zu löschenden Objekten geschrieben wird.

Durch eine Graphmusterbeschreibung und darauf angesetzten Operationen lassen sich beliebige Transformationen beschreiben. Die Schreibweise ist verschieden zu den meisten Graphtransformationsystemen und unterscheidet sich von der eingebürgerten algebraischen Schreibweise mit der Angabe von einer linken und einer rechten Seite einer Regel.

Man kann leicht nachvollziehen, dass die beiden Schreibweisen gleichstark sind und ohne viel Aufwand ineinander überführt werden können.

Beispiel 5.11. (*Überführen von Regelschreibweise*)
In diesem Beispiel wird gezeigt, wie man eine Regel von DRAGULA nach PROGRES transformiert und umgekehrt. Dazu wird die Regel aus dem Beispiel 5.10 benutzt.

Alle Knoten, die nach dem Matchen der Anfrage weder erzeugt noch gelöscht werden, kommen auf der linken und der rechten Seite vor. Die Knoten mit der Operation `create` kommen nur auf der rechten Seite einer Regel vor, analog die Knoten mit einem `delete` erscheinen nur auf der linken Seite. Entsprechend diesem Vorgehen werden die Knoten mit eindeutigen Markierungen versehen, um nachvollziehen, welche Knoten hinzukommen und welche gelöscht werden.

So wird die Regel von der Abbildung 5.20 wie folgt transformiert (Abbildung 5.21.)

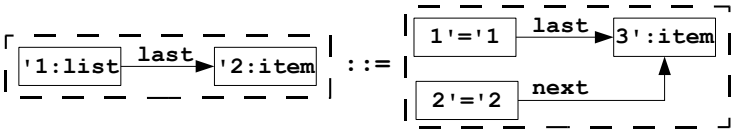

Abbildung 5.21: Transformierte Regel in PROGRES

Die zwei Knoten, die in der DRAGULA Regel mit keinem Operator versehen sind, kommen sowohl auf der linken, als auf der rechten Seite der PROGRES Regel, das wird auch durch Markierung `1' = '1` und `2' = '2` ausgedrückt. Der neue erzeugte Knoten erhält die noch nicht benutzte Markierung `3'`. Die Kanten werden automatisch umgeleitet.

Eine Transformation einer Regel von PROGRES nach DRAGULA erfolgt ähnlich. Es werden alle Knoten aus den rechten und linken Seiten der Regel gesammelt. Anschließend werden die Knoten entsprechend ihrer Markierung mit Operationen ausgestattet. Eine besondere Behandlung bekommen die Kanten, da sie in DRAGULA

nicht automatisch erzeugt / umgehängt werden. Sie müssen explizit verfolgt werden und gegebenenfalls erzeugt und verbunden oder gelöscht werden. ■

Ähnlich zu den Bedingungen können auch die Operationen aus mehreren Variablen und atomaren Operationen zusammengesetzt werden. Das kann die Verständlichkeit der Spezifikation erhöhen und bietet die Möglichkeit eigene spezifische Operationen zu definieren (Abb 5.22.) Die selbstdefinierten Bedingungen dürfen keine Bedingungen enthalten, genauso wie eigene Bedingungen keine Operatoren.

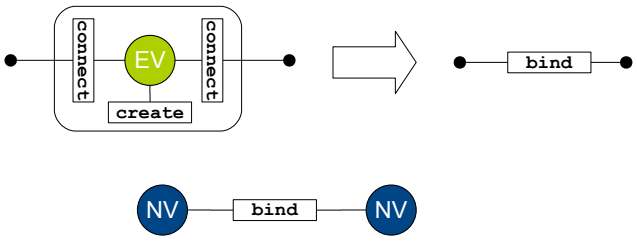

Abbildung 5.22: Definition eines eigenes Operator

5.5 Ablaufsteuerung

In diesem Kapitel werden die Sprachkonstrukte beschrieben, die mehrere Transformationen zu einem Graphtransformationssystem zusammenzuschalten erlauben. Das sind die Kontrollstrukturen, die den Ablauf der Ausführung steuern, und der Datenfluss, der eine konsequente Anwendung auf logisch zusammengehörenden Teilen eines Graphen sicherstellt [HP] [Zün95].

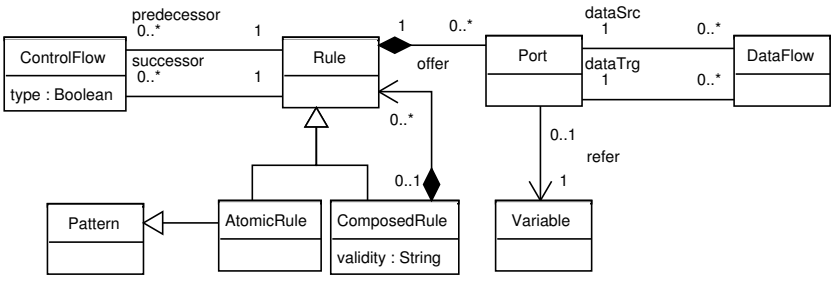

Abbildung 5.23: Metamodell: Ablaufsteuerung

Das UML-Diagramm in der Abbildung 5.23 gibt einen Überblick über die Ablaufsteuerung. Jede Regel bildet eine Ausführungseinheit. Die Regel selbst ist entweder atomar oder wird aus weiteren Regeln kombiniert. Die Angaben über die Reihenfolge der Ausführung wird durch Kontrollflusskanten vorgegeben. Die Kanten können mit Zusatzinformation für bedingte Verzweigungen ausgestattet werden. Die Informationsübertragung ist von der Ausführungskontrolle getrennt. Sie wird mittels Ports und Datenflusskanten angegeben.

In folgenden Abschnitten wird die Syntax und Semantik der Ablaufsteuerung und des Datenflusses beschrieben. Die Semantik wird nicht formal umschrieben, da ein dazu geeigneter Kalkül die Rahmen dieser Arbeit übersteigen würde.

5.5.1 Kontrollfluss

In diesem Abschnitt werden die grundlegende Konstrukte zur Ablaufsteuerung vorgestellt.

Als eine elementare Einheit für die weiteren Beschreibungen dient hier ein Transformationsvorschrift. So wird von dem Inhalt einer Vorschrift abstrahiert, man spricht auch von einer atomaren Regel. Auf dieser Sprachschicht wird die Ausführung von Transformationen als atomar angesehen.

Konkatenation

Eine einfachste Kontrollstruktur ist die Hintereinanderschaltung mehrerer Regeln. Die Regeln werden einfach nacheinander angestoßen. Grafisch wird es durch Kontrollflusskanten von einer Regel zu der nachfolgenden dargestellt (Abb. 5.24). Die Konkatenation von Regeln ist zu schwach für die meisten Anwendungsfälle, sie bildet aber die Basis für weiteren Kontrollstrukturen.

Abbildung 5.24: Konkatenation zwei Regeln

Falls eine Regel bei der Ausführung nicht erfolgreich war, werden die Entscheidungen in der vorherigen Regel revidiert. Der einzige Punkt an dieser Stelle für die Revision, ist das Match, auf dem die Regel angesetzt war. Falls es mehrere Matches

zu einer Transformation gibt werden die Änderungen auf dem Graph durch Backtracking zurückgenommen und mit einem neuen Match ausprobiert.

Bedingte Verzweigung

Weiter zieht man das Ergebnis einer Transformationvorschrift in die Betrachtung. Es gibt zwei Möglichkeiten: entweder war die Transformation erfolgreich oder es schlug fehl, weil etwa Ansatzpunkt für das Muster nicht gefunden werden konnte. Aus dieser Information können die Schlüsse über Graphbeschaffenheit gezogen werden und darauf basierend Entscheidungen über weiteren Verlauf der Bearbeitung getroffen werden.

Also werden die Kontrollflusskanten mit den Beschriftungen, die sie als entweder positiv oder negativ markieren, versehen. Eine der positiven Kanten wird im Falle einer erfolgreichen Ausführung weiterverfolgt und eine aus der negativen bei einem Misserfolg.

In der Abbildung 5.25 sieht man die Regel 1 mit mehreren sowohl positiven, als auch negativen Kontrollflusskanten. Wenn die Regel 1 erfolgreich ausgeführt wird, wird (eventuell, da mehrere Nachfolger gibt) mit der Regel 2 fortgefahren. Im Falle eines Misserfolgs wird unter Umständen die Regel 3 ausgeführt.

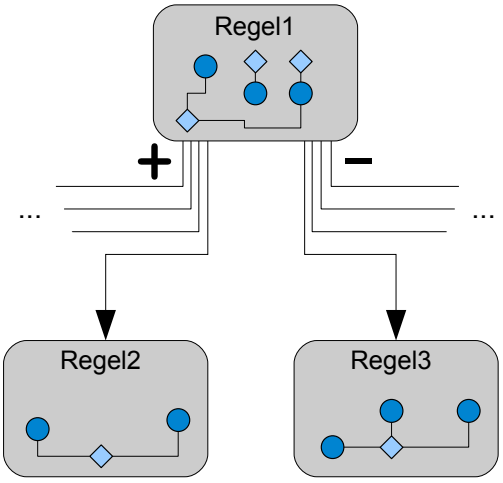

Abbildung 5.25: Bedingte Verzweigung

Falls eine Regel nicht erfolgreich bei der Ausführung war, wird es mit einer negativen Kante fortgefahren. Falls es keine negativ markierten Kontrollflusskanten gibt, wird versucht per Backtracking die zuletzt ausgeführte Regel zurückzunehmen und mit neuen Parametern erneut auszuführen, wie bei der Konkatenation.

Auf diese Weise wird solange ausprobiert, bis entweder die Regel erfolgreich ausgeführt wird oder bis alle Möglichkeiten für den aktuellen Schritt ausgeschöpft sind. In diesem Fall wird die zweitletzte Entscheidung revidiert und so weiter. Mehrere Kanten, die von einer Regel ausgehen erlauben beim Backtracking und Neueansetzen nicht nur Matches durchprobieren, sondern auch die Regel, die als Nachfolger ausgeführt werden.

Das Backtracking auf einer negativen Kontrollflusskante ist nur dann sinnvoll, falls es mehrere Nachfolger mit negativen Kanten gibt. Eine negative Kante wird nur im Falle erfolgloser Ausführung einer Regel verfolgt, deswegen gibt es keine alternativen Matches, die ausprobiert werden können.

Bereits diese Konstrukte erlauben dem Benutzer Hintereinanderschaltung mehreren Regeln, sowie bedingte Verzweigungen und Schleifen aus Transformationsvorschriften zu definieren.

Zusammengesetzten Regel

Um eine einfache und strukturierte Schreibweise für komplexeren Regeln anzubieten, werden die zusammengesetzten Regeln eingeführt. Im Gegensatz zu den atomaren Regeln dürfen die zusammengesetzten keine Variablen, Bedingungen oder Operatoren direkt enthalten, dafür aber andere Regeln (atomare sowie zusammengesetzte).

Die Ablaufsteuerung zwischen den zusammengesetzten Regeln wird, analog auch wie zwischen atomaren Regeln, durch die Kontrollflusskanten vorgegeben. Eine komplexe Regel wird von außen nach innen ausgeführt, wobei eine Regel eine initiale, unbeschriftete Kontrollkante zu einer inneren Regel haben darf, die als erste angestoßen werden soll. Falls es keine initiale Kante gibt, kommen alle direkt enthaltenen Regeln, die keine eingehenden Kontrollkanten haben, als möglichen Kandidaten, die als Einstigepunkten in die Regel dienen, in Betracht.

Um Konstruktionen aus Regeln besser zum Ausdruck zu bringen und eventuell einige Entscheidungen erst zur Laufzeit dem Backtracking zu überlassen, werden weitere Strukturen auf Regeln eingeführt. Es wird ein bestimmtes Verhalten für eine Gruppe der Regel definiert.

Manchmal wird die Ausführung nur einer Regel aus der Reihe mehreren benötigt. Diese optionale Verhalten bei der Ausführung wird für eine zusammengesetzte Regel durch eine `OR`-Beschriftung markiert.

Analog wird die Ausführung aller Regeln aus einer Sammlung in einer nicht festgelegten Reihenfolge verlangt, das wird durch `AND` markiert (Abb. 5.26).

Auch hier wird entsprechend dem Konzept des Backtrackings fortgefahren. Bei `OR` Regel versucht die Auswertungsmaschine eine erfolgreiche Ausführung für eine enthaltene Regel zu finden. Bei einem `AND` Konstrukt wird methodisch nach einer pas-

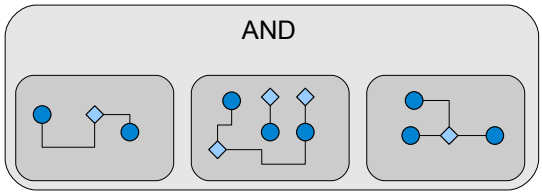

Abbildung 5.26: Bindung der drei Regeln mit `AND`

senden Reihenfolge der Ausführung von Regeln gesucht, bis alle enthaltenen Regeln erfolgreich angewendet sind.

Auch wenn die mit `OR` und `AND` markierten Regeln den oben beschriebenen Graphmustern (Kap. 5.3) ähneln, besitzen sie eine vollkommen unterschiedliche Semantik. Diese zwei Konstrukte unterscheiden sich bezüglich der Ausführung der Operationen. Muster dienen dazu, die Suche zu strukturieren. Also müssen die Operationen erst nach der Suche, nachdem die Muster abgearbeitet sind, ausgeführt werden. Bei den Regeln kennt man keinen Unterschied zwischen Suchen und Ausführen von Operationen, aus der Sicht der Kontrollflusses ist es eine einzige atomare Prozedur.

Um an mehreren Stellen verwendeten Regeln nicht mehrmals zu kopieren, können die Regeln referenziert werden, ähnlich wie bei rekursiven Mustern. Es können nur die Top-Level Regeln referenziert werden. Sie bilden eine abgeschlossene und unabhängige Ausführungseinheit, es führen keine Kontrollflusskanten zu oder von solchen Regeln. Das Referenzieren hat eine Aufrufsemantik und ist mit einem Sprung in eine Unterprogramm zu vergleichen.

Mit der Hilfe der oben beschriebenen Sprachkonstrukten zur Ablaufkontrolle lassen sich bereits komplexe Graphersetzungsprogramme beschreiben. Hier ein Beispiel, um die Verwendung von Ablaufsteuerung zu verdeutlichen.

Beispiel 5.12. (*Knoten kopieren*)
In diesem Beispiel wird die Arbeit mit der Ablaufsteuerung veranschaulicht.

Man will alle Knoten aus einem Graph in einen anderen kopieren. Das wird wie folgt gemacht. Man nehme einen unmarkierten Knoten in dem Quellgraph und kopiere ihn in den Zielgraph, markiere den Originalknoten. Es wird solange fortgefahren bis es keinen unmarkierten Knoten gibt, das bedeutet, dass alle Knoten in den Zielgraph kopiert sind. Jetzt wird der Quellgraph von den überflüssigen Markierungen bereinigt: alle Knoten werden abgearbeitet, bis alle Markierungen gelöscht sind.

Die Ausführung sieht dann in DRAGULA formuliert wie folgt aus(Abb. 5.27), der Inhalt der Regel wird hier nicht angegeben.

■

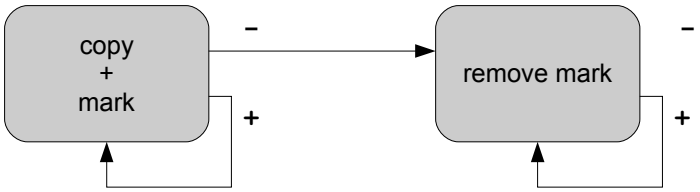

Abbildung 5.27: Kopieren von Knoten eines Graphs

5.5.2 Datenfluss

Für einige Fälle muss der Kontext, in dem eine Transformation angewendet wurde, weiter an die Verarbeitung gegeben werden. Zum Beispiel werden zwei Objekte, die zufällig ausgesucht wurden, als Ansatzpunkte für weitere Transformationen weitergegeben werden. Es wurde versucht die Ablaufsteuerung mit nicht relevanten Informationen nicht zu überladen. Der Kontrollfluss und der Datenfluss wurden strikt voneinander getrennt. Aus diesem Grund reichen die Methoden des Kontrollflusses nicht, um die Abhängigkeiten, die während der Auswertung entstehen, zu übermitteln. Man braucht einen Datenfluss zwischen den Transformationen.

Der Datenfluss wird durch spezielle Datenflusskanten und sogenannte Ports organisiert. Jede solche Kante ist eine gerichtete Kante von einem Port zu einem anderem. Unter Ports wird eine Referenz einer Variable verstanden. Erst durch einen Port wird eine Variable außerhalb einer Regel zugänglich und ist mit einer `public` Deklaration in OO-Sprachen zu vergleichen. Dadurch definieren Ports eine Schnittstelle einer Regel.

Durch Ports wird das Innenleben einer Transformation abstrahiert. So können die Regeln entsprechend ausgetauscht werden, oder in ihrer Funktion geändert werden, ohne dass das Gesamtsystem davon betroffen wird.

Durch die Kombination von Verweisen auf Regeln aus dem Kontrollfluss und Ports bekommt man eine zur Prozeduren ähnliche Semantik. Eine Regel hat eine Reihe von Ein- und Ausgabeparameter und kann an bestimmten Stellen “aufgerufen“ werden, indem es referenziert wird.

Wie in der Abbildung 5.28 werden Ports durch kleine Quader auf dem Rand einer Regel dargestellt. In dem Diagramm sind auch ein internes Leben der Regel dargestellt, um die Bindungen von Ports an die Variablen zu verdeutlichen.

Nachdem ein Graphmuster gefunden wird, wird ein Port, entsprechend der Belegung der mit ihm verbundenen Variablen mit einem Graphobjekt belegt. Bei der Ausführung weiterer Regeln wird die Information mittels Datenflusskante transportiert und an einen Port übergeben. Dadurch wird auch die durch den Port referenzierte

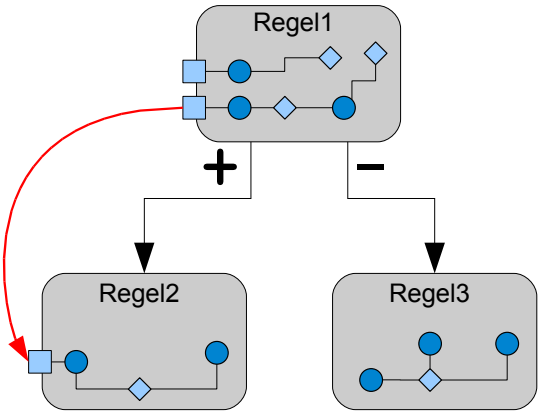

Abbildung 5.28: Datenfluss zwischen Regeln

Variable auf den festgelegten Wert gesetzt.

Die Zielregel einer Datenflusskante muss daher durch eine positive Verzweigung an der Quellregel erreichbar sein. Falls die Quellregel nicht erfolgreich war, gibt es auch kein gültiges Match zu den Variablen, so dass auch nichts übertragen werden kann.

Da Regeln in ihnen enthaltenen Transformationen abstrahieren, benötigt man eine Reihe von Ports und Datenflusskanten zwischen diesen, um eine Variable außerhalb einer mehrmals geschachtelten Regel zugänglich zu machen (Abb. 5.29.) Das Wert einer Variable wird auf diese Weise durch mehrere Schichten “durchgereicht“.

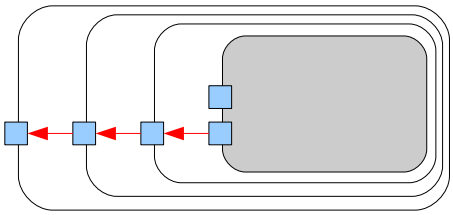

Abbildung 5.29: Datenfluss in einer geschachtelten Regel

Es muss auch beachtet werden, dass in manchen Fällen, zum Beispiel Graphmuster mit `NAC`-Untermuster, Teile von Muster keine gültige Belegung haben dürfen. Also sind unter Umständen nicht alle Variable, auch in einem Muster mit *gültigem* Match, belegt.

Mit der Hilfe den oben vorgestellten Mitteln lassen sich beliebige Konstrukte aus Transformationen definieren. Die einzelnen Regeln können in weiteren Regeln zu-

sammengefasst werden, die Reihenfolge der Ausführung wird durch die Kontrollflusskanten und Markierungen auf zusammengesetzten Regeln gegeben. Die Daten werden mittels Ports und Datenflusskanten für eine konsequente Verarbeitung weitergegeben.

Hier ein Beispiel um das Zusammenspiel von allen einzelnen Teilen von DRAGULA zu demonstrieren.

Beispiel 5.13. (*Abbildung von Turingmaschine in DRAGULA*)

Es wird in diesem Beispiel gezeigt, wie man eine Turingmaschine mit Hilfe von DRAGULA simulieren kann. Eine Turingmaschine ist ein Zustandsautomat, der auf einem unendlichen Band arbeitet.

Definition 5.5. Eine Turingmaschine wird als 7-Tupel definiert.

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$$

Wobei ist

- Q eine endliche Menge der Zustände
- Σ und Γ sind das Eingabe- und das Arbeitsalphabet der Maschine.
- $\delta : Q - \{q_{acc}, q_{rej}\} \times \Gamma \longrightarrow Q \times \Gamma \times \{L, N, R\}$ ist eine Übergangsfunktion. Sie ordnet jedem Zustand und einem Buschstaben aus dem Alphabet einen weiteren Zustand, in den die Maschine wechselt, einen Buchstaben, der auf den Band geschrieben wird und die Richtung, in die sich die Maschine auf dem Band weiterbewegt.
- q_0 ist der Anfangszustand
- q_{acc}, q_{rej} sind der akzeptierende und der verwerfende Zustände. Angekommen in so einem Zustand hält die Maschine und akzeptiert bzw. verwirft die Eingabe.

■

Es sind zwei Aufgaben: eine zum Band äquivalente Datenstruktur zu finden und die Maschine mit entsprechenden Zustandsänderungen darzustellen.

Band: Eine nahe liegende Möglichkeit ist, das Band als eine verkettete Liste darzustellen. Die Liste kann einfach hin und her traversiert werden und das Hinzufügen von neuen Elementen in die Liste erfordert nur einen konstanten Aufwand. Das Einfügen von den Elementen in der Mitte der Liste wird nicht betrachtet,

da es mit dem ursprünglichen Band nicht möglich ist und auch von der Turingmaschine nicht benutzt wird.

Die Liste wird wie folgt aufgebaut: sie besteht aus den Knoten von Typ `cell`, jeder Knoten hat ein Attribut `info`, wo der Inhalt der Bandzelle aufbewahrt wird. Die Knoten sind miteinander durch die `next`-Kanten zu einer Liste verbunden. Am Anfang und dem Ende werden die Extraknoten vom Typ `ListEnd` angebracht, die das Ende der Liste signalisieren, die Liste muss dann durch das Anhängen einen Knoten erweitert werden.

Maschine: Die Maschine wird auch mittels Knoten und Kanten abgebildet.

Jeder Zustand q_i aus der Menge Q wird durch einen Knoten vom Typ `state` repräsentiert.

Für jeden Zustandsübergang

$$\delta(q_i, a) = (q_j, b, m)$$

mit $q_i, q_j \in Q$, $a, b \in \Gamma$ und $m \in \{L, N, R\}$ wird eine Kante vom Typ `transition` definiert. Sie führt von dem Knoten q_i zum Knoten q_j und ist mit Attributen `in`, `out` und `mov` beschriftet. Die Attribute enthalten die Information zu dem Übergang, wie die Beschriftung der Bandzelle, der zu schreibende Buchstaben und die Richtung für den Schritt auf dem Band.

Die beschriebene Struktur kann in einer Polynomialzeit aus Q und δ gebaut werden. Alle Elemente in den beiden Mengen werden nur ein Mal angefasst und daraus ein korrespondierendes Zustandsdiagramm erzeugt.

So werden das Band und die Maschine als Graphen dargestellt. Jetzt braucht man die Transformationsvorschriften, die die Berechnungsschritte simulieren.

Um die aktuelle Kopfposition auf dem Band zu merken, wird ein neuer Knoten vom Typ `CurrPos` eingeführt. Eine Kante verbindet diesen Knoten mit der aktuellen Bandzelle. Analog wird auch der aktuelle Zustand mit der Hilfe eines Knotens von Typ `CurrState` gemerkt.(Abbildung 5.30)

Es wird der Berechnungsschritt betrachtet, bei dem die Lese-/Schreibkopf auf dem Band nicht bewegt wird. Die Maschine liest den Buchstaben a, schreibt in die Zelle b rein und wechselt dabei von dem Zustand q_i zum Zustand q_j.

In DRAGULA wird das wie folgt abgebildet (Abbildung 5.31). In dem oberem Teil wird die aktuelle Bandzelle, mittels der Attributbedingung, auf die Übereinstimmung mit dem `in` Attribut einer Transition geprüft. Falls in der aktuellen Bandzelle zu dem aktuellem Zustand richtige Beschriftung gefunden wird, wird der Zustandsübergang durchgeführt und der neue Zelleinhalt mittels `SetAttrVal` gesetzt.

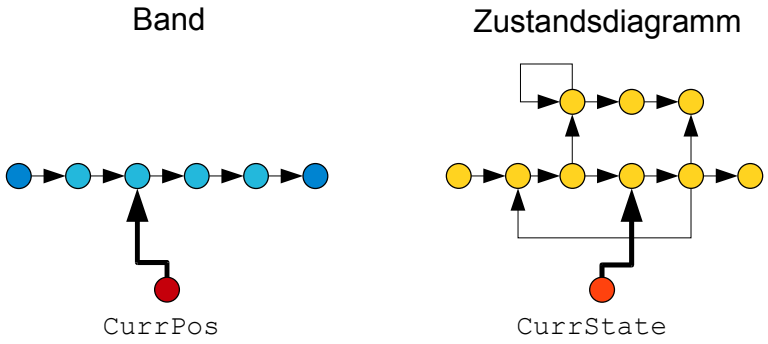

Abbildung 5.30: Das Band und die Turingmaschine als Graphen

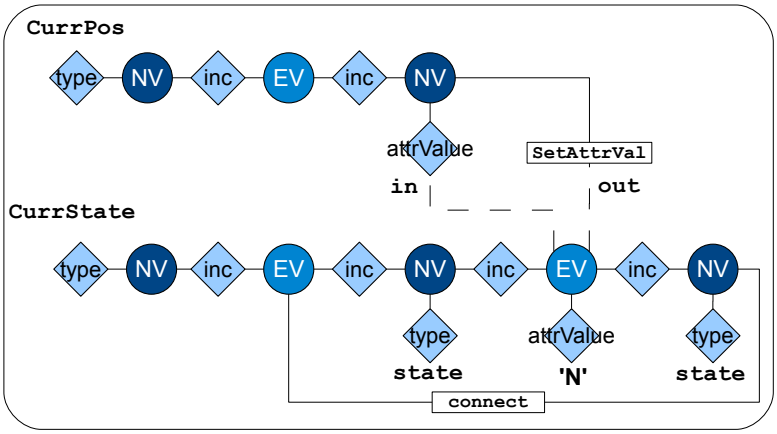

Abbildung 5.31: Transformation für einen N-Schritt

Ähnlich wird eine Transformation für einen Schritt mit dem Wechsel zur nächster Bandzelle konstruiert (Abb 5.32.) Hier muss man zwei unterschiedliche Fälle betrachten: die Nachfolgerzelle existiert bereits oder sie muss erst an die Liste angefügt werden. Diese Unterscheidung wird mit der Hilfe von `OR`-Muster gelöst. Entweder wird als nächste eine regulare Zelle gefunden und zu der gesprungen, oder man findet das Ende der Liste und muss eine Zelle erzeugen (siehe dazu das Beispiel 5.10.) Der Zustandwechsel wird analog zum N-Schritt durchgeführt. Analog gestaltet sich der Schritt mit der Bewegung auf dem Band nach links.

Einer der oben beschriebenen Transformationen soll immer anwendbar sein, ist es nicht der Fall, befindet sich die Maschine in einem Zustand ohne Nachfolger und kann somit die Eingabe nicht akzeptieren. Die Berechnung bricht in diesem Fall ab.

Nach jedem Schritt wird überprüft, ob der akzeptierende oder der verwerfende Zustand erreicht ist. (Abbildung 5.33) Das ist eine reine Abfrage, sie beinhaltet keine

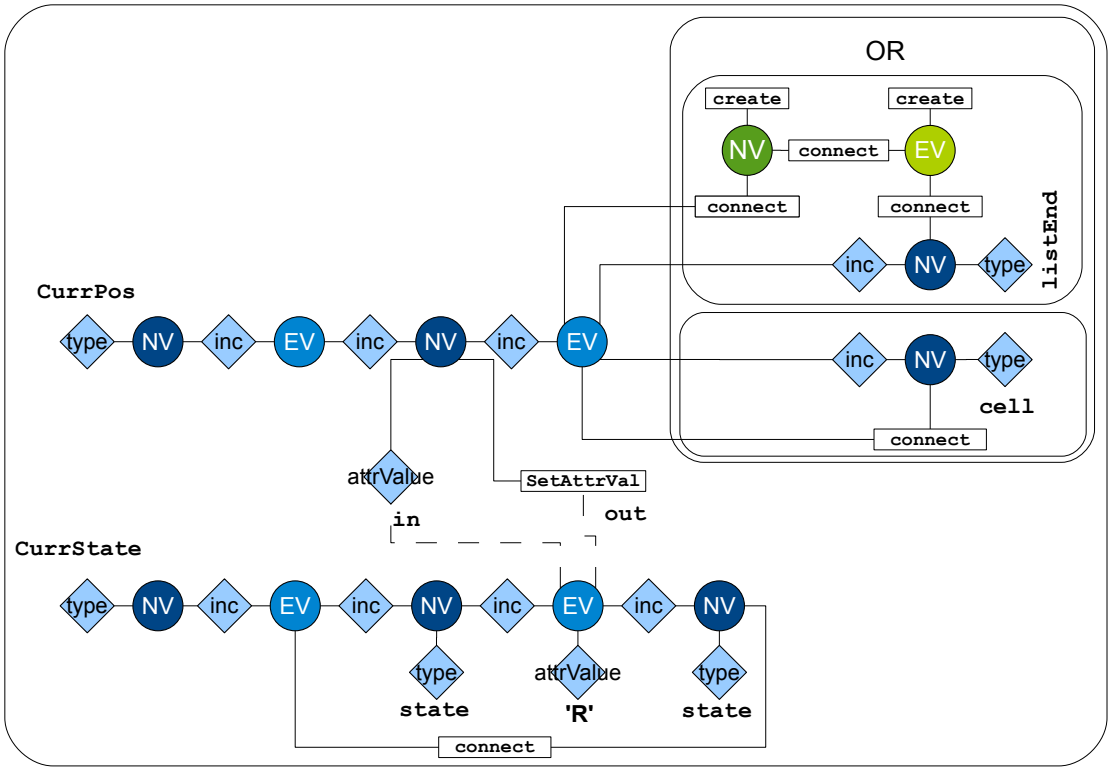

Abbildung 5.32: Transformation für einen R-Schritt

Operationen und ändert auch nicht den aktuellen Zustand der Maschine.

Die einzelnen Transformationen und die Anfragen werden zu einem Programm zusammengeschaltet (Abb 5.34.) Es werden solange die Transformationen ausgeführt, bis die Maschine sich in einem der Endzustände befindet.

Die Information über die aktuelle Bandzelle und den Zustand wurde explizit in dem Graphmodell zu der Maschine verwaltet. Alternativ dazu könnte man diese Information mittels Datenfluss von einer Regel zu einer anderen leiten.

Jede Transformation besteht aus endlich vielen Variablen und Bedingungen. Das bedeutet, dass die entsprechenden Homomorphismen in polynomielle Zeit gefunden werden können. Somit wird jeder Schritt einer Turingmaschine durch eine Abbildung in Polynomzeit simuliert. Damit ist gezeigt, dass die Sprache DRAGULA Turing-vollständig ist. ■

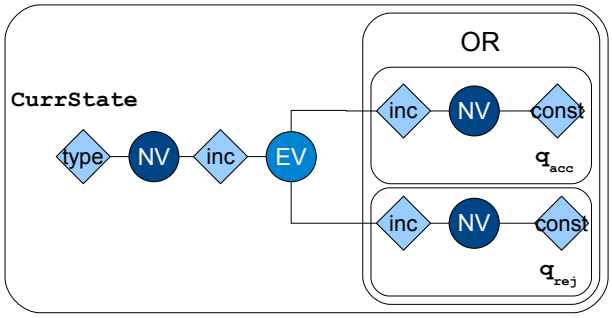

Abbildung 5.33: Anfrage: Erreichen von Endzuständen

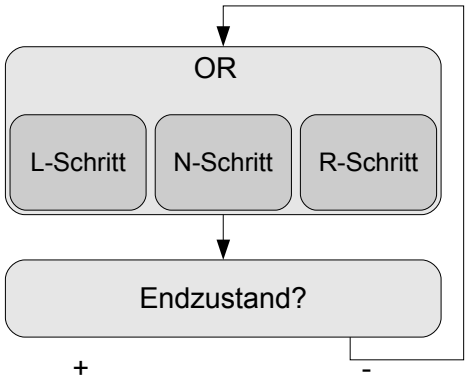

Abbildung 5.34: Turingmaschine: Zusammenschaltung der Transformationen

5.6 Diskussion

In diesem Kapitel werden die Eigenschaften von DRAGULA zusammengefasst und anhand der gestellten Anforderungen besprochen. Es werden die Vorteile der Sprache herausgearbeitet, sowie mögliche Verbesserungspunkte aufgezeigt.

Das Hauptziel der vorliegenden Arbeit ist der Entwurf einer hinreichend allgemeinen und erweiterbaren Basissprache, die sich auch für weitere Entwicklungen auf dem Gebiet Graphwerkzeuge eignet. Dafür unterstützt die Sprache die unterschiedlichen Graphentitäten, wie Knoten, Kanten, hierarchische Graphen und Relationen. Die Sprache bietet Konstrukte, um Anfragen und Transformationen auf graph-basierten Daten zu formulieren.

Mittels DRAGULA können Graphmuster beschrieben werden, zu denen dann ein Teilgraph in der Datenbank gesucht wird. Dabei wird sowohl Bezug auf die Eigenschaften einzelner Objekte, dazu gehören auch die Anfragen an den Typ einer Entität,

als auch auf die Kombination der Objekte und die Beziehungen untereinander genommen.

Die Sprache bietet außer den elementaren Konstrukten zur Beschreibung der Graphmuster auch die Möglichkeit eigene abgeleitete Konstrukte zu definieren. Dazu zählen die Pfad- und Baumausdrücke und im Allgemeinem alle Konstrukte, die durch die Rekursion beschrieben werden können.

Es bestehen mehrere Möglichkeiten spezifische Bedingungen und Muster zu organisieren und zu kapseln. Die vordefinierte Konstrukte können mittels Verweise und Ports in ihrer Funktionalität und internen Daten abstrahiert werden. Die Transformationen werden zu abgeschlossenen Programmen kombiniert. DRAGULA stellt eine Reihe der Konstrukte zur Organisation der Ausführung zur Verfügung. Für eine konsequente Übermittlung der Parameter von einer Transformation zu einer anderen dient der Datenfluss.

Die Sprache ist hinreichend allgemein und ausdrucksstark um andere Graphtransformationssprachen abzubilden. Zum Einen werden die Sprachkonstrukte, die von anderen Sprachen angeboten werden, von DRAGULA gedeckt. Zum Anderen ist die Sprache, wie in dem Beispiel 5.13 gezeigt wurde, Turing-vollständig und kann somit jede berechenbare Funktion ermitteln.

Ein wichtiger Aspekt einer Sprache ist nicht nur die Ausdrucksstärke, sondern auch die Infrastruktur, in der sie angeboten wird.

Die Flexibilität der Sprache ist durch eine Anbindung an der Datenbank gegeben. Ein generisches Modul zur Auswertung von Ausdrucken kann durch spezifische Module ersetzt oder erweitert werden (siehe dazu das Kap. 6.)

Die Sprache bietet vielfältige Möglichkeiten eigene Sprachelemente zu definieren. Beispiel dazu sind selbstdefinierten Bedingungen, oder abgeleiteten Konstrukte, die mittels rekursiven Verweisen entstehen. Einige der Ausdrücke dienen nicht nur der Strukturierung der Graphtransformationsregeln, sondern können von einem spezialisierten Auswertungsmodul effizient verarbeitet werden (siehe dazu das Kap. 6.2). Das kann die entscheidenden Vorteile bei speziellen Graphmodellen oder einer optimierten Abbildung auf Hintergrundspeicher bringen. Das macht die DRAGULA erweiterbar und an die spezifische Aufagabendomäne anpassbar.

Die Tatsache, dass eine in DRAGULA formulierte Transformationspezifikation selbst einen Graphen darstellt bringt eine zusätzliche Flexibilität mit sich mit. Eine Spezifikation kann automatisch, mit der Hilfe von DRAGULA selbst, an bestimmte Anforderungen oder Gegebenheiten angepasst werden. Eine Transformation kann so umstrukturiert, erweitert oder anwendungsspezifisch modifiziert werden.

Eine Anwendung muss nicht eine Vielzahl von Hilfsfunktionen importieren oder sich auf ein Rahmenwerk beziehen, um die Ausdrücke in DRAGULA auszuführen.

Das bringt die benötigte Anpassungsfähigkeit für die Sprache. Die Sprache ist auch in verschiedenen Laufzeitumgebungen anwendbar, da eine Auswertungsmaschine auf der Seite des Graphspeichers zur Verfügung steht. Die selbstdefinierte Sprachkonstrukte, sowie Transformationsvorschriften bleiben auch in anderen Programmen anwendbar.

Ein weiterer Vorteil von dieser Darstellung der Spezifikationen ist die Anwendung von Analysewerkzeugen. Eine Spezifikation kann auf bestimmte Eigenschaften automatisch überprüft werden, etwa Erreichbarkeit einer Regel oder gegenseitige Störung von mehreren Regeln.

Eine Basis für eine spezifische Graphtransformationssprache bildet ein spezifisches Graphmodell, auf dem gearbeitet wird. DRAGULA kann auf vielen verschiedenen Graphmodellen angewendet werden, da der Sprache ein sehr allgemeines Modell zugrunde liegt. Mit der Datenbank DRAGOS gelieferte Werkzeuge erlauben dem Anwender eine Abbildung des gewünschten Graphmodells mit einem vertretbarem Aufwand zu definieren. Somit ist die Sprache auch auf spezifischen Graphmodellen anwendbar.

Die Ausführung von Transformationen erfolgt in einem Modul und ist von der Anwendung abstrahiert. Somit kann die Ausführung von der Seite der Anwendung nicht beeinflusst werden.

Durch die Bedingungen und die vielfältige Varianten um ein Muster zu beschreiben kann DRAGULA komplexe Transformationsvorschriften formulieren. Die Ausdrucksstärke wird auch durch die im Beispiel 5.13 gezeigte Abbildung von Turingmaschine in DRAGULA gezeigt.

Die Sprache ist möglichst deklarativ gestaltet und für Menschen lesbar, sie wird nicht mit unnötigen Details überladen. Alle relevanten Aspekte werden durch die Bedingungen an die Objekte und den entsprechenden Strukturierung abgedeckt. Die Möglichkeit oft verwendete Ausdrücke durch kürzere Beschreibungen zu ersetzen steigert die Lesbarkeit von Transformationsregeln. Da die Sprache auf Graphen basiert, lässt sie sich gut in grafischer Form veranschaulichen.

Die Sprache DRAGULA beinhaltet mehrere Konzepte um Transformationvorschriften zu organisieren: von den selbstdefinierten Bedingungen bis zu geschachtelten Mustern und den Regeln für Ablaufsteuerung. Um von diesen Spracheigenschaft zu profitieren wird ein Editor benötigt, der die Graphen je nach Arbeitsschritt passend grafisch darstellt. Weitere Werkzeuge, die in der Verbindung mit DRAGULA benötigt werden, sind die Transformationswerkzeuge, die Spezifikationen in anderen Graphtransformationssprachen in DRAGULA überführen. Eine Reihe von weiteren generischen Modulen, basierend auf anderen als DRAGOS Datenbanken, würde auch die Flexibilität bei der Wahl von Graphspeicher erhöhen.

In diesem Kapitel wurde eine allgemeine, erweiterbare, ausdrucksstarke Sprache

definiert. Durch hohe Grad an Flexibilität kann sie in unterschiedlichen Anwendungen und Graphwerkzeugen eingesetzt werden. In weiteren Kapiteln werden Beispielweise technische Aspekte der Anbindung der Sprache an Datenbanken behandelt, damit soll die Infrastruktur um die Sprache gezeigt werden.

In der Tabelle 5.35 werden noch Mal die wichtige Eigenschaften der drei Sprachen: DiaPlan, PROGRES und DRAGULA aufgeführt.

Eigenschaft		AGG	DiaPlan	PROGRES	DRAGULA
Graph	Art	gerichtet, attributiert, beschriftet	gerichtet, attributiert, beschriftet, hierarchisch	gerichtet, attributiert, beschriftet	gerichtet, attributiert, beschriftet, hierarchisch
	Graphschema	Typgraph	Typgraph	Graphschema	Graphschema
Knoten		typisiert, attributiert, identifizierbar	typisiert, attributiert, identifizierbar	typisiert, attributiert, identifizierbar	typisiert, attributiert, identifizierbar
Kanten	Art	typisiert, attributiert, identifizierbar, zwischen Knoten	typisiert, attributiert, Hyperkanten	typisiert, zwischen Knoten	typisiert, attributiert, zwischen Graphobjekten, Hyperkanten
	Abgeleiteten	——	——	Pfade	Rekursive Strukturen
Attribute n	Art	intrinsische	intrinsische	intrinsische, abgeleitete, Metaattribute	intrinsische
	Wertmenge	Standardtypen, Javaobjekte	Standardtypen	Standardtypen, Knoten, Mengen	Standardtypen, Javaobjekte
Anfragen	Teilgraph	Knoten, Kanten	Knoten, Hyperkanten, Graphen, Prädikaten	Knoten, Mengen der Knoten, Kanten, Pfade, Bedingungen	Knoten, Kanten, Hyperkanten, Graphen, rekursive Konstruktionen
	NAC's	negierte Teilgraphen	——	negierte Knoten, Kanten, Pfade, negierte Bedingungen	negierte Knoten, Kanten, Teilgraphen, geschachtelte Negation
	Attrbutbedingung	Ja	Ja	Ja	Ja
Kontrollstrukturen		Iterationen über Ausführungsebenen	Kapselung, sequentielle und bedingte Ausführung, nichtdeterministische Wahl	equentielle und bedingte Ausführung, nichtdeterministische Wahl, Transformationsaufrufe	Kapselung, sequentielle und bedingte Ausführung, nichtdeterministische Wahl

Abbildung 5.35: Sprachvergleich

6 Anbindung an DRAGOS

Nachdem die theoretischen Grundlagen der Sprache ausgeführt sind, werden einige technische Einzelheiten der Anbindung der Sprache an die Datenbank DRAGOS besprochen. Es sind die Ansätze, die sich während der Erarbeitung des Gesamtprojekts noch ändern können.

Für einen Überblick über weiter unten angesprochene Module zu bekommen wird auf die Abbildung 6.1 verwiesen.

6.1 Generisches Modul

Das generische Modul hat als Aufgabe, die grundlegende Funktionalität, die für die Realisierung der Sprache DRAGULA benötigt wird, anzubieten. Durch das Modul wird die Datenbank, mit einer prozeduraler Schnittstelle abstrahiert. Eine Anwendung definiert Transformationen und lässt sie von dem Modul ausführen.

Das Modul wird als eine Erweiterung an die Datenbank angeschlossen. In diesem Modul sind nicht nur Prozeduren zur Auswertung der DRAGULA Ausdrücken enthalten, sondern auch die DRAGOS Graphschema mit der Beschreibung aller atomaren Elementen der Sprache. Siehe dazu das oberes Modul in der Abbildung 6.1.

Alle Konstrukte der Sprache werden auf Graphobjekte abgebildet. Zum Beispiel basieren Variablen, Bedingungen und Operationen auf Knoten, wie sie in DRAGOS zu finden sind. An diesem Punkt sieht man die Notwendigkeit eines klar definiertes Graphschema. Variablen und Bedingungen werden auf Knoten abgebildet, haben aber unterschiedliche Syntax und Semantik, was durch Knotenklassen zum Ausdruck gebracht wird.

Selbstverständlich kann das vordefinierte Graphschema erweitert werden. Dazu stehen die Grundklassen und die Vererbungsmechanismen zur Verfügung, sowie die Möglichkeiten eigene Klassen von Grund auf zu definieren.

Das generische Modul implementiert die elementaren Funktionen zu der Sprache, dazu gehören: das Auswerten und das Transformieren von Anfragen, das Verwalten und das Filtern von Matches, sowie die Ablaufsteuerung. Das Suchen nach einem gültigen Match passiert schrittweise, wie es durch die Sichtbarkeitsregelung vorgegeben ist. Gültige Belegungen werden durch Einschränkungen des Lösungsraumes

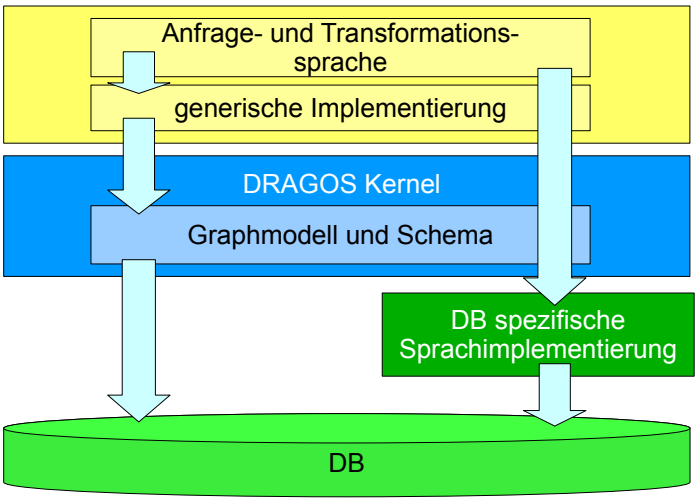

Abbildung 6.1: Verbindung zwischen zwei Knoten

gefunden, wobei für jede Variable eine Kandidatenliste mit Graphobjekten geführt wird und entsprechend die Kandidaten, die eine Bedingung verletzen, rausgefiltert werden. Bei der Ablaufsteuerung wird ein Entscheidungsbaum mit dazugehöriger Information verwaltet.

Einige der Funktionen benötigen auch weitere Erweiterungsmodule. Zum Beispiel für das Backtracking wird ein Modul, das die Versionierung realisiert, benötigt. Andernfalls müssen die Veränderungen an Datenbestand intern verwaltet werden. Das würde unter Umständen sehr viele Ressourcen kosten und für größere Transformationen nicht geeignet sein, da die Konsistenz der Daten nicht sichergestellt werden kann.

Das generische Modul kann auch in seiner Funktionalität ausgebaut werden. Mittels Java-Vererbung und eigenständig programmierten Klassen können neue Fähigkeiten eingebaut werden.

Also es wird durch das generische Modul nicht nur die Auswertung der Sprachausdrücke durchgeführt, sondern auch eine Grundlage für weitere Entwicklungen geschaffen.

6.2 Hintergrundspeicher-spezifisches Modul

In diesem Abschnitt werden Module besprochen, die zwar auch als eine Erweiterung für DRAGOS agieren, aber in der Architektur weiter unten, auf dem Niveau von Hintergrundspeicher anzusiedeln sind (siehe untere Module zwischen dem DRAGOS-

Kernel und dem Hintergrundspeicher.

In einigen Fällen ist es sinnvoll Ausdrücke in der Sprache nicht auf atomare Elemente zurückzuführen, sondern als eine Einheit auszuführen. Das trifft besonders in Spezialfällen zu, wo der Hintergrundspeicher für bestimmte Anfragen optimiert ist, oder durch Anforderungen an das System gewissen Einschränkungen unterliegt. Bei einer solchen Optimierung werden an die Speicher nicht viele kleine Anfragen gestellt, sondern eine große, komplexe, unter Ausnutzung von Speicherfähigkeiten aufgebaute [Ali08, Kapitel 5].

So wurde bereits ein Modul entwickelt, das Anfragen bezüglich der Inzidenz der Knoten und Kanten optimiert. Anstatt vielmals die Kandidatenlisten bezüglich der Knoten- / Kantenklassen durchzusuchen und inzidente Paare intern rauszusuchen, wird *eine* komplexere SQL-Anfrage an die relationale Datenbank konstruiert, die serverseitig gemäß einer internen Strategie optimiert wird.

Die Objekte werden sofort mit allen relevanten Eigenschaften, wie Klasse, Attributwerte, sowie mit Beziehungen zwischen Objekten, hier die Inzidenz beschrieben. Die Verarbeitung solch einer Anfrage kostet weniger Zeit als eine Vielzahl von den kürzeren Suchanfragen und wird besser von der Datenbank optimiert. Eine komplexere Anfrage liefert sofort die Ergebnisse und benötigt keine Nacharbeitung.

Beispiel 6.1. (*Bau der SQL-Anfrage aus DRAGULA*)
Aus einer Anfrage in DRAGULA kann man leicht eine SQL-Anfrage konstruieren, wenn die genauere Speicherung von Objekten in einer Datenbank bekannt ist.

So werden die Objekte, die durch Variablen repräsentiert sind, aus entsprechenden Tabellen mittels `SELECT` ausgewählt. Die Eigenschaften jedes Objektes, die durch Bedingungen eingeschränkt werden, kommen dann in die `WHERE`-Teil einer Anfrage.

Es wird hier ein vereinfachte Fall betrachtet. Die Knoten und die Kanten sind in zwei Tabellen gespeichert. Sie sind typisiert und werden eindeutig durch ein ID identifiziert.

Nodes

id	type
nId1	nClass1
nId2	nClass1
nId3	nClass2
nId4	nClass2
	⋮

Edges

id	type	src	trg
eId1	eClass1	nId1	nId2
eId2	eClass1	nId1	nId3
eId3	eClass2	nId3	nId4
	⋮		⋮

Eine Typeanfrage an einem Knoten in DRAGULA sieht wie folgt aus.
Und entsprechend wird es in SQL überführt.

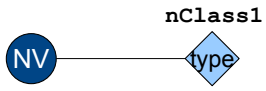

Abbildung 6.2: Typ-Bedingung in DRAGULA

```
SELECT n FROM Nodes WHERE n.type = nClass1
```

Eine Anfrage nach zwei adjazenten Knoten in DRAGULA, es wird nach zwei Knoten und einer, sie verbindende, Kante gesucht.

Abbildung 6.3: Verbindung zwischen zwei Knoten

Nun sieht es folgendermaßen in SQL.

```
SELECT Nodes.n1, Nodes.n2, Edges.e WHERE ((e.src=n1 AND e.trg=n2) OR
                    (e.src=n2 AND e.trg=n1))
```

Es lassen sich natürlich einzelne Bedingungen kombinieren und komplexere Anfragen nach SQL transformieren. Anschließend bekommt man eine Reihe von Matches, also fertige Ergebnisse zu einer Anfrage.

■

Solche Module müssen nicht Unbedingt für ganze Anfragen entwickelt werden, und können nur für spezifische, auf einen Anwendungsfall bezogene Sprachkonstrukte gebaut werden.

6.3 Diskussion

Der Unterschied zwischen den zwei oben dargestellten Modulen liegt in der Gestaltung der Module.

Das generische Modul soll die Grundfunktionalität der Sprache realisieren und stellt eine Schnittstelle für Anwendungen zur Verfügung und soll hinreichend allgemein sein. Das Modul stellt alle die benötigten Funktionen zur Auswertung und

Ausführung von DRAGOS Ausdrücken. Da die Möglichkeit zur Erweiterung als Teil der Sprachdefinition abgefasst wurde, bietet das Modul dazu auch entsprechende Mechanismen. Zum Einen ist es funktionale Schnittstelle, sowie Objekthierarchie in Java. Und zum anderen die Graphobjekte mit passenden Attributen in DRAGOS Vererbungshierarchie.

Das zweite Modul beruht auf dem Wissen über den Hintergrundspeicher und über eine Abbildung von Konstrukten auf ihn. Das Modul ist im Gegensatz zu dem generischen sehr anwendungsspezifisch und kann unter Umständen nicht in anderen Werkzeugen oder für andere Hintergrundspeicher verwendet werden.

Die oben vorgestellte Wege die Ausdrücke zu interpretieren und auszuführen erscheinen als sinnvoll in der Verbindung mit DRAGOS. Ein Grund für diese Entscheidung ist die modulare Bauweise der Datenbank DRAGOS. Die Module bekommen einen Einblick in das, ansonsten durch Funktion- und Datenabstraktion verborgene, Innenleben und können diese Information auch ausnutzen.

Für eine andere Datenbank benötigt man dann auch andere Lösungen um DRAGULA anzubinden.

7 Zusammenfassung

In diesem Kapitel wird eine Zusammenfassung, sowie einen Ausblick auf noch offene, weiterführende Fragen gegeben.

Das Ziel dieser Arbeit war eine erweiterbare anpassbare Basissprache zu entwickeln. Diese Sprache soll als die Grundlage für verschiedene Graphwerkzeuge dienen.

Die Anforderungen an die Sprache DRAGULA kamen aus unterschiedlichen Bereichen. DRAGULA soll gängige Graphtransformationssprachen abbilden können. Besondere Aufmerksamkeit wurde dabei der Fragestellung gewidmet, die Sprache nach Bedarf um zusätzliche Konstrukte zu erweitern, um sie an domänenspezfische Anwendungen und Graphersetzungssysteme adaptieren zu können. Ein weiterer wichtiger Punkt ist die Anwendungsunabhängigkeit und die Anpassungsfähigkeit von DRAGULA. Die Art und Weise, wie man die Sprache benutzt, soll nicht davon abhängen, welche Daten verarbeitet werden oder in welchem Kontext man arbeitet.

Um die Sprache in unterschiedlichen Graphwerkzeugen zu benutzen, wird ein allgemeines Graphschema benötigt. DRAGULA basiert auf dem Graphschema der Datenbank DRAGOS. Das Modell ist sehr allgemein definiert, strukturiert dennoch die Verhältnisse zwischen einzelnen Graphentitäten. Graphschemata aus Graphersetzungssystemen lassen sich semi-automatisch auf das Graphmodell von DRAGOS abbilden.

Die Sprache wurde entsprechend der Anforderungen entworfen. Die Transformationen werden mit Hilfe von Variablen, Bedingungen und Operationen beschrieben. Solche Trennung der Semantik innerhalb der Sprache bewirkt, dass in jedem Sprachobjekt nur ein Aspekt zur Beschreibung einer Transformation enthalten ist. Das erlaubt komplexe Objekte einfach zu beschreiben und erleichtert das Einführen von neuen Sprachteilen. Transformationen setzen sich aus Unteranfragen zusammen. Es gibt Möglichkeiten, die logischen Abhängigkeiten zwischen Unteranfragen zum Ausdruck zu bringen.

Anfragen und Transformationen werden mittels Kontrollstrukturen zu einer Transformationsvorschrift zusammengeschaltet. Die Transformationsvorschrift läuft in Abhängigkeit von der Ausführung einzelner Transformationen und erlaubt bedingte Verzweigungen und Schleifen, und bietet die Möglichkeit einige Entscheidungen mittels Backtracking rückgängig zu machen. Die Information, die während der Ausführung entsteht und an nachfolgende Regel übergeben werden muss, wird mittels eines orga-

nisiertes Datenflusses übertragen. Durch alle diese Sprachkonstrukte wird eine komplexe ausdrucksstarke Sprache definiert.

Die Ausdrücke in DRAGULA bestehen aus Graphen. Das bietet zum einen Vorteile bei der Darstellung der Ausdrücken, und zum anderen hat man die Vorteile, dass die Ausdrücke auch mit Hilfe von Transformationswerkzeugen automatisch modifiziert werden können.

Als Weiterentwicklung an diesem Punkt sind die Analyse- und Transformationswerkzeuge für die Sprache DRAGULA. Fast auf jeder Sprachschicht gibt es Aspekte, die unter Umständen relevant sein können. Es sind etwa Erfüllbarkeit aus dem Bereich model checking für Graphmuster, Konflikte zwischen Paaren von Transformationen oder Erreichbarkeit einzelner Regeln in einem Graphersetzungsprogramm.

Die Struktur des verwendeten Graphen wird in DRAGULA mittels Graphschema definiert. Eine Weiterentwicklung an der Schnittstelle zwischen der Datenbank und dem Auswertungsmodul sind die Regeln, die die Konsistenz der Daten aktiv überwachen und gegebenenfalls von der Datenbank angestoßen werden. Dadurch werden feingranularen Beziehungen zwischen Objekten in einem Graph zum Ausdruck gebracht.

Da die Sprache mit eventuell komplex aufgebauten Ausdrücken arbeitet, bedarf es einen passenden Editor. Der Editor soll für die Arbeit mit DRAGULA geeignet sein und gewissen Ausgleich zwischen der Detailliertheit der Darstellung und dem Gesamtüberblick schaffen. Dies geschieht entweder durch verschiedene Sichten auf die Spezifikation oder durch ein dynamisches Ausblenden von unnötigen Einzelheiten.

Literaturverzeichnis

[Ali08] Qasim Ali. Graph transformations based on relational databases. Master's thesis, RWTH Aachen, Lehrstuhl für Informatik 3, 2008.

[BGL05] Blech, Glesner, and Leitner. Formal verification of java code generation from UML models. *Proc. of 3rd intl. Fujaba Days*, 2005.

[BV] András Balogh and Dániek Varró. The VIATRA 2 model transformation framework users' guide.

[Böh04] Boris Böhlen. *Eine parametrisierbare Graph-Datenbank für Entwicklungswerkzeuge*. PhD thesis, Reinisch-Westfälische Technische Hochschule Aachen, 2004.

[CMW02] Katja Cremer, André Marburger, and Bernhard Westfechtel. Graph-based tools for re-engineering. *Journal of Software Maintenance*, 14(4):257–292, 2002.

[DHJ+06] Frank Drewes, Berthold Hoffmann, Dirk Janssens, Mark Minas, and Niels Van Eetvelde. Adaptive Star Grammars. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *ICGT*, volume 4178 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2006.

[FMRS07] C. Fuss, C. Mosler, U. Ranger, and E. Schultchen. The jury is still out: A comparison of agg, fujaba, and progres. *Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, vol. 6 of Elec. Communications of the EASST, 2007.

[GA05] Taentzer Gabriele and Rensink Arend. Ensuring structural constraints in graph-based models with type inheritance. *Lecture notes in computer science*, 2005.

[HM00] Berthold Hoffmann and Mark Minas. Towards generic rule-based visual programming. In *VL '00: Proceedings of the 2000 IEEE International*

Symposium on Visual Languages (VL'00), page 65, Washington, DC, USA, 2000. IEEE Computer Society.

[HM01] Berthold Hoffmann and Mark Minas. Transformation of shaped nested graphs and diagrams. *Electr. Notes Theor. Comput. Sci.*, 59(4), 2001.

[Hof00] Berthold Hoffmann. From graph transformation to rule-based programming with diagrams. In *AGTIVE '99: Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 165–180, London, UK, 2000. Springer-Verlag.

[HP] Annegret Habel and Detlef Plump. A core language for graph transformation (extended abstract).

[JSW99] Dirk Jäger, Ansgar Schleicher, and Bernhard Westfechtel. AHEAD: A Graph-Based System for Modeling and Managing Development Processes. In Manfred Nagl, Andy Schürr, and Manfred Münch, editors, *AGTIVE*, volume 1779 of *Lecture Notes in Computer Science*, pages 325–339. Springer, 1999.

[Kle04] Raimund Klein. Ein Interpreter für Diaplan, eine regelbasierte Sprache zum Programmieren mit Graphen und Diagrammen. Master's thesis, Universität Bremen, 2004.

[KNNZ99] Thomas Kleín, Ulrich A. Nickel, Jörg Niere, and Albert Zündorf. From uml to java and back again. Technical Report tr-ri-00-216, University of Paderborn, Paderborn, Germany, September 1999.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture ^{TM}: Practice and Promise*. Addison-Wesley Professional, 2003.

[LB93] Michael Löwe and Martin Beyer. AGG - an implementation of algebraic graph rewriting. In *RTA '93: Proceedings of the 5th International Conference on Rewriting Techniques and Applications*, pages 451–456, London, UK, 1993. Springer-Verlag.

[LS88] Claus Lewerentz and Andy Schürr. Gras, a management system for graph-like documents. In *JCDKB*, pages 19–31, 1988.

[MW02] André Marburger and Bernhard Westfechtel. Graph-based reengineering of telecommunication systems. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *ICGT*, volume 2505 of *Lecture Notes in Computer Science*, pages 270–285. Springer, 2002.

[Nag96] Manfred Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1996.

[PLGCGB06] Javier Pérez, Miguel A. Laguna, Yania Crespo González-Carvajal, and Bruno González-Baixauli. Requirements Variability Support Through MDA ™ and Graph Transformation. *Electr. Notes Theor. Comput. Sci.*, 152:161–173, 2006.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[Sch90] A. Schürr. Introduction to PROGRESS, an attribute graph grammar based specification language. In *WG '89: Proceedings of the fifteenth international workshop on Graph-theoretic concepts in computer science*, pages 151–165, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[SRB06] E. Schultchen, U. Ranger, and B. Böhlen. Graph-oriented storage for fujaba applications. *Proceedings of the Fujaba Days 2006*, vol. tr-ri-06-275 of Technical Report, 2006.

[SWZ96] Andreas Schürr, Andreas Winter, and Albert Zündorf. *Developing Tools with the PROGRES Environment*, chapter 3.6, pages 356–369. Volume 1170 of Nagl [Nag96], 1996.

[SWZ99] A. Schürr, A. J. Winter, and A. Zündorf. The PROGRES approach: language and environment. pages 487–550, 1999.

[Tae04] Gabriele Taentzer. Agg: A graph transformation environment for modeling and validation of software. pages 446–453. 2004.

[Thy05] Marcus Thyssen. Entwicklung einer Anfragesprache für die Graphdatenbank Gras/GXL. Master's thesis, RWTH Aachen, Lehrstuhl für Informatik 3, 2005.

[Tsa93] Edward Tsang, editor. *Foundation of Constraint Satisfaction*. Academic Press, London and San Diego, 1993.

[UML] OMG Unified Modeling Language. `http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/`.

[Vos94] Gottfried Vossen. *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme*. Addison-Wesley, 1994.

[Wei07a] Erhard Weinell. Adaptable support for queries and transformation in the DRAGOS graph database. *Applications of Graph Transformations with Industrial Relevance (AGTIVE'07)*, vol. 5088 of LNCS, 2007.

[Wei07b] Erhard Weinell. Extending graph query languages by reduction. *Graph Transformation and Visual Modeling Techniques (GT-VMT'08)*, vol. 10 of Elec. Communications of the EASST, 2007.

[Wei08] Erhard Weinell. Transformation-based operationalization of graph languages. *Doctoral Symposium of International Conference on Graph Transformation (ICGT'08)*, vol. 5214 of LNCS, Springer 2008, 2008.

[Zün95] A. Zündorf. *Eine Entwicklungsumgebung für PROgrammierte GRaphErsetzungsSysteme - Spezifikation, Implementierung und Verwendung*. PhD thesis, 1995.