# GReAT - Graph Rewriting And Transformation Language

Bastian Schoofs
227656

Betreut von Dipl.-Inf. Ulrike Ranger

## Zusammenfassung

GReAT ist ein Ansatz um Daten aus einer Domäne mit Hilfe von Graphregeln in eine andere Domäne zu transformieren. Um die Domänen und die Transformation zu definieren, ist GReAT als Plugin für ein visuelles Framework erstellt worden. Dies hat den Vorteil, dass die Transformation verständlich und übersichtlich definiert werden kann. Zusätzlich zum visuellen Framework bietet GReAT einen Debugger und die Möglichkeit C++ Quellcode zu erzeugen, der für Transformationen ohne das Framework benutzt werden kann.

# Inhaltsverzeichnis

# 1 Einleitung

Innerhalb des Seminars ‘Unterstützung modellgetriebener Entwicklungsprozesse’ wird GReAT als ein visuelles Werkzeug für Transformationen mittels Graph-Grammatiken vorgestellt. Ähnliche Ansätze werden innerhalb des Seminars in den Themen “AGG / GenGED” und “Tripel-Graph-Grammatiken” behandelt.
Mit Einführung der Model-Driven Architecture (MDA) werden Programme, mit deren Hilfe Datentransformationen durchgeführt werden, immer häufiger benutzt. Die Transformationen werden benutzt, um schon vorhandene Daten in andere Formate für die weitere Softwareentwicklung zu überführen. Den Daten liegt ein Modell zugrunde. Ein Modell ist eine abstrakte Repräsentation von Struktur, Funktion und Verhalten. Die Modelle werden in der Regel in UML definiert. Eine Idee von MDA ist es, den Softwareentwicklungprozess zu vereinfachen, indem bereits vorhandene Daten wiederverwendet werden. Diese Modelle der vorhandenen Daten sind jedoch meist so spezifisch, dass es nicht möglich ist, sie an anderen Stellen wiederzuverwenden. Angenommen die vorhandenen Daten sollen zum automatisierten Test der erstellten Software benutzt werden, dann müssen sie in Daten für die Testsoftware umgewandelt werden.
Um die Übersetzung der vorhandenen Daten in neue Daten durchzuführen, ist die Benutzung von Graphregeln einfacher und übersichtlicher, als für jede Transformation ein selbst geschriebenes Programm zu erstellen und mit diesem die Transformation durchzuführen. Graphregeln müssen zwar auch für jede Transformation erstellt werden, jedoch sind diese einfacher und schneller zu erstellen. da Graphregeln formal beschrieben sind und eine einfache Syntax und Semantik besitzen. Für den Fall, dass die den vorhandenen Daten zugrundeliegenden Modelle sich während des Softwareentwicklungsprozesses verändern, müssen die darauf basierenden Datentransformationen auch angepasst werden. Solche Anpassungen sind mit GReAT oft einfacher als mit selbst geschriebenen Transformationen.
In dieser Ausarbeitung wird als Hilfsmittel zur Graphtransformation eine Sprache mit dem Namen GReAT beschrieben. GReAT ist ein Akronym für Graph Rewriting and Transformation Language. GReAT erlaubt es den Benutzern, Graphregeln in visueller Form mit formaler und ausführbarer Semantik zu spezifizieren. Die Hauptaufgabe von GReAT ist es, Daten, die in einer definierten Domäne beschrieben sind, in Daten einer anderen Domäne zu überführen. Die verschiedenen Domänen werden mit Hilfe von UML Diagrammen definiert. Die UML Repräsentation der Domäne wird auch als Modell bezeichnet. Die Transformation wandelt Daten, die dem einen Modell entsprechen, in Daten eines anderen Modells um.
Um die Funktionsweise von GReAT beispielhaft vorzustellen, wird in Kapitel 2 ein durchgängiges Beispiel vorgestellt. In dem Beispiel werden vorhandene Studentendaten mit einem komplexen Modell in Studentendaten eines einfacheren

Modells transformiert. Mit das Wichtigste bei der Transformation von Graphen ist das Finden von Anwendungsstellen. Anwendungsstellen werden mit Hilfe von Mustern beschrieben. Die in GReAT verwendeten Muster werden in Kapitel 3 beschrieben. Das darauffolgende Kapitel 4 beschreibt, wie mittels GReAT eine Modelltransformation durchgeführt werden kann. Kapitel 5 zeigt die Umgebung, für die GReAT als Plugin erstellt wurde, sowie einige der Werkzeuge, die zur Verfügung stehen. Abschließend werden in Kapitel 6 die Vor- und Nachteile von GReAT dargestellt.

# 2 Beispiel

Als durchgängiges Beispiel wird eine Transformation von Studenteninformationen vorgestellt. Bei dieser Transformation sollen Studentendaten, die in einer Domäne vorliegen, in eine andere Domäne überführt werden. Die Transformation von der Startdomäne in die Zieldomäne soll dafür sorgen, dass aus der ausführlichen Darstellung der Studentendaten eine kompakte neue Darstellung erstellt wird. Damit eine Transformation der Daten mittels GReAT geschehen kann, müssen zunächst die jeweiligen Modelle der beiden Domänen erstellt werden [6]. GReAT selbst kann nicht zur Definition der Modelle benutzt werden, jedoch gibt es im Generic Modeling Environment (GME) die Möglichkeit, Modelle zu definieren [2]. GME wurde vom Institute for Software Integrated Systems an der Vanderbilt Universität erstellt. GME wurde entwickelt, um mit dessen Hilfe Domänen-spezifische Modelle zu realisieren [8]. GME bietet außerdem die Möglichkeit Plugins einzubinden. GReAT wurde als ein solches Plugin für GME erstellt.
Plugins, die in GME realisiert sind, werden *Paradigmen* genannt. Die für eine Transformation mit GReAT benötigten Modelle werden mit dem *Paradigma MetaGME* erstellt[6]. Das Modell der Startdomäne ist in Abbildung 1 dargestellt.Im Modell ist zu erkennen, dass die Studenteninformationen durch ein *Modell* mit dem Namen *Studenten* repräsentiert werden. Das Modell *Studenten* an sich darf nur *Atome* vom Typ *Student*, *Diplom* und *Vorlesung* sowie die beiden *Connection* Elemente *VorlesungsConnection* und *DiplomConnection* enthalten. Hierbei repräsentieren *Atome* Objekte mit Daten und die *Connection* Elemente werden benutzt, um Relationen zwischen zwei Objekten genauer zu beschreiben.
Das UML Element *Student* repräsentiert einen Studenten mit den Attributen *Name* und *Matrikelnummer*. *Diplom* wird genutzt, um festzuhalten, ob ein Student sein Diplom erworben hat oder nicht. Falls er das Diplom erworben hat, enthält das zugehörige Objekt die Note des Diploms und für den Fall, dass der Student das Diplom noch nicht erworben hat, wird die Anzahl der nicht erledigten Prüfungen

gespeichert. Hierzu beinhaltet das Element drei Attribute mit dem Namen *hatDiplom*, *DiplomNote* und *anzahlFehlenderPruefungen*. Um die Relation zwischen einem Diplom und dem Studenten darzustellen, gibt es ein Connection Element vom Typ *DiplomConnection*.

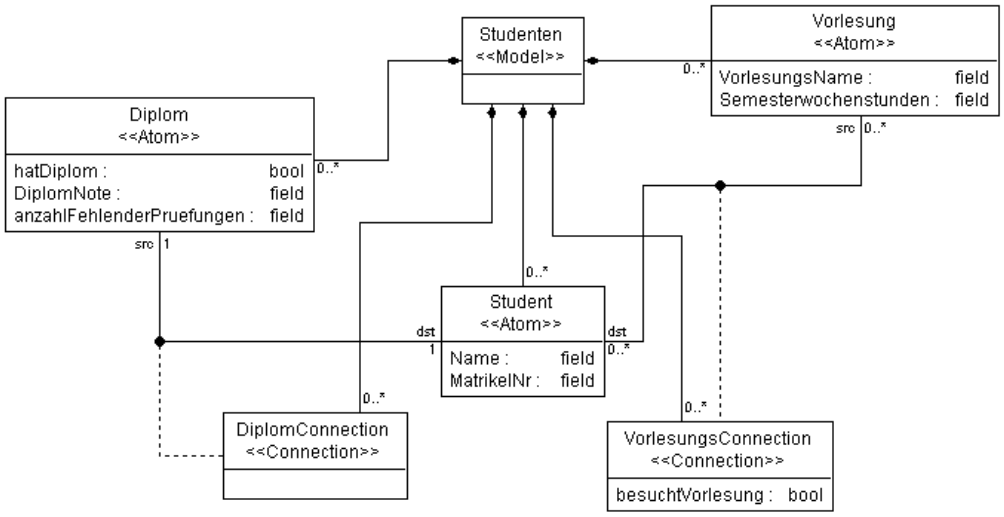


Abbildung 1: Modell der Startdomäne

Die Elemente vom Typ *Vorlesung* stellen jeweils eine Vorlesung und deren zugehörige Semesterwochenstunden dar. Hat ein Student diese Vorlesung besucht, existiert zwischen ihm und der Vorlesung ein *Connection* Element vom Typ *VorlesungsConnection*, dessen Attribut *besuchtVorlesung* mit *true* gefüllt ist.
Das Modell der Zieldomäne ist einfacher als das der Startdomäne. Es besteht lediglich aus 4 UML Klassen und ist in Abbildung 2 dargestellt. Das Modell der neuen Studenteninformationen wird durch eine Klasse mit dem Namen *neueStudentenDaten* dargestellt. Das Modell kann beliebig viele Objekte vom Typ *Student* beinhalten. *Student* beinhaltet die Attribute *Name*, *Vorlesungen*, *MatrikelNr* und *SemesterWochenstunden*. Im Attribut *Vorlesungen* werden alle Namen der Vorlesungen, die der Student besucht hat, gespeichert. *SemesterWochenstunden* beinhaltet entsprechend der besuchten Vorlesungen die Summe der dazugehörigen Semesterwochenstunden.
In Abhängigkeit davon, ob der Student in den alten Daten ein Diplom besitzt, wird die Klasse *StudentMitDiplom* zur Repräsentation der Daten benutzt. Hat er kein Diplom, wird die Klasse *StudentOhneDiplom* benutzt. *Student* selbst wird nicht in den Zieldaten benutzt, sondern die Objekte der Klassen *StudentOhneDiplom*

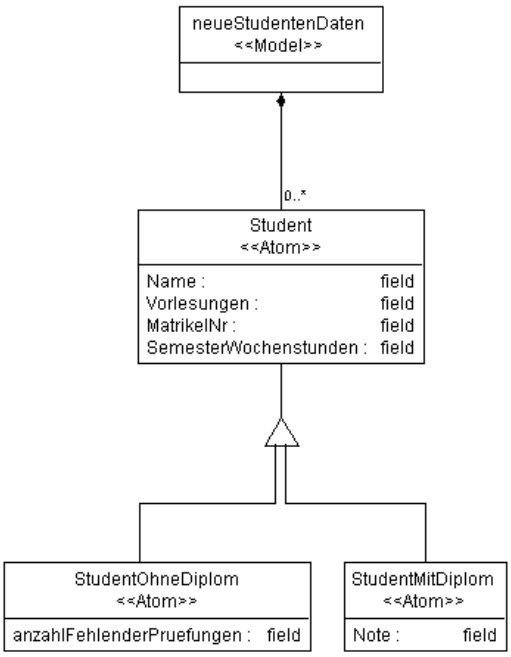


Abbildung 2: Modell der Zieldomäne

und *StudentMitDiplom*. Die beiden Klassen sind Spezialisierungen von der Klasse Student und erben daher die in der Klasse *Student* definierten Attribute. Zusätzlich zu den vererbten Attributen besitzt *StudentOhneDiplom* ein Attribut mit dem Namen *anzahlFehlenderPruefungen*, das benutzt wird, um die Anzahl der Prüfungen festzuhalten, die dem Studenten noch bis zum Erhalt seines Diploms fehlen. *StudentMitDiplom* besitzt zusätzlich ein Attribut *Note*, um die Note des Diploms zu speichern.
Nachdem das Modell der Startdomäne erstellt wurde, kann eine Instanz der Domäne erstellt werden. Die Daten der Startdomäne werden auch mit Hilfe von GME erstellt. Mit GME können Instanzen von Modellen über einen Editor, ähnlich dem zur Erstellung des Modells, erzeugt werden [6]. Die fertige Instanz der Startdomäne mit den Daten für die Beispieltransformation ist in Abbildung 3 dargestellt. Die Daten der Instanz sind in der Abbildung als Graph dargestellt, in dem nur die *Atome* des Modells als Knoten zu sehen sind. Die *Connection* Elemente des Modells werden durch die Kanten des Graphen dargestellt. Die Daten repräsentieren zwei Studenten, von denen einer sein Diplom besitzt, der andere nicht, sowie die Vorlesungen, die die jeweiligen Studenten besucht haben.
Um die Graphregeln der Studentendatentransformation zu erstellen, müssen zu-

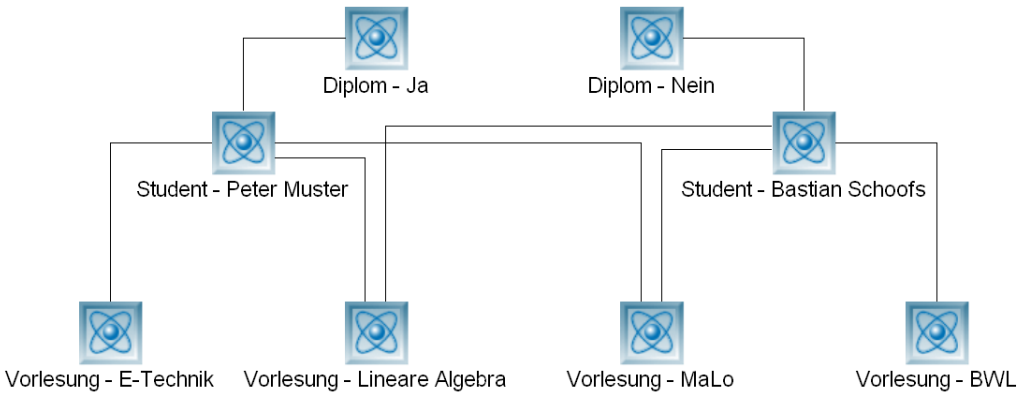


Abbildung 3: Studentendaten der Startdomäne

nächst Muster beschrieben werden. Muster sind die Hauptbestandteile der Graphregeln. Sie werden im nächsten Kapitel beschrieben.

# 3 Mustersuche

GReAT benutzt zur Transformation der Daten Graphregeln. Der wichtigste Bestandteil der Regeln sind Muster. Diese Muster werden benutzt, um zu bestimmen, ob eine Regel angewendet wird oder nicht. Jede Regel von GReAT benutzt das in ihr definierte Muster, um mit dessen Hilfe passende Anwendungsstellen im Graphen zu finden. Eine gefundene Anwendungsstelle wird immer durch eine Menge von Knoten und eine Menge von Kantenrelationen beschrieben[1]. Die Objekte aus den gefundenen Anwendungstellen des Instanzgraphen müssen mit den Klassen der Elemente des Musters übereinstimmen. Das bedeutet, ein Knoten von der Klasse *Vorlesung* findet auch nur Objekte vom Typ *Vorlesung* im Instanzgraph.
GReAT benutzt als Muster gerichtete attributierte Graphen, die mittels eines Diagramms aus Elementen des Modells dargestellt werden. Die Knoten des Graphen werden durch UML Objekte repräsentiert. Die UML Objekte beinhalten wiederum verschiedene Attribute. Die Kanten zwischen zwei Knoten repräsentieren auch eine UML Klasse mit verschiedenen Attributen, wobei die Kanten immer von einem Knoten des Graphen auf einen anderen Knoten des Graphen zeigen müssen. Um das Ziel zu erkennen, endet die Kante an diesem Element mit einer Raute.

## 3.1 Einfache Muster

Das einfachste Muster ist die exakte Darstellung des gesuchten Teilgraphen. Wenn ein Teilgraph gesucht wird, der darstellen soll, dass ein Student eine Vorlesung besucht hat und sein Diplom besitzt, kann durch die Darstellung der Klassen, der gesuchten Objekte, das Muster erstellt werden. Hierbei besteht der Muster aus den Klassen *Student*, *Vorlesung* und *Diplom*. Die beiden zuletzt genannten Klassen sind mit der Klasse *Student* verbunden. Bei der Mustersuche ist es wichtig, dass bei mehrfacher Suche eines Musters im selben Graphen immer die gleichen Anwendungsstellen zurückgegeben werden. Falls mit dem gesuchten Muster mehrere passende Anwendungsstellen gefunden werden, werden alle gefundenen Anwendungsstellen zurückgegeben. Eine zurückgegebene Anwendungsstelle besteht aus einer Menge von Knoten- und Kantenrelationen. Die Knotenrelationen bestehen jeweils aus einem Knoten des Musters und aus einem Knoten der Daten. Dasselbe gilt für die Kantenrelationen.
Wenn das Muster aus Abbildung 4 gesucht wird, um Anwendungsstellen in den Daten von Abbildung 5 zu suchen, werden zwei gültige Anwendungsstellen zurückgegeben. Die Beschriftung der Knoten und Kanten des Instanzgraphen in Abbildung 5 repräsentiert nicht den Inhalt der Objekte, sondern dient nur der Lesbarkeit. Die zurückgegebenen Anwendungsstellen sind: {{(*Student*, *Bastian Schoofs*), (*Vorlesung*, *Malo*), (*Diplom*, *Diplom*)}, {(*VorlesungsRelation*, *BS-Malo*), (*DiplomRelation*, *BS-Diplom*)}} und {{(*Student*, *Peter Muster*), (*Vorlesung*, *Malo*), (*Diplom*, *Diplom*)}, {(*VorlesungsRelation*, *PM-Malo*), (*DiplomRelation*, *PM-Diplom*)}}. Bei den gefundenen Anwendungsstellen werden zuerst die gefundenen Knotenrelationen und danach die gefundenen Kantenrelationen zurückgegeben. Welche der beiden Anwendungsstellen als erste zurückgegeben wird, hängt davon ab, wie die Suche nach ihnen implementiert ist.
Die Suche nach allen Anwendungsstellen in einem Graphen ist sehr zeitaufwendig und hat eine Zeitkomplexität von $O\left(C_h^{C_p}\right)$, wobei $C_h$ die Anzahl von Knoten des Graphen ist und $C_p$ die Anzahl der Knoten des Musters. Diese Laufzeit ist exponentiell und für die Suche nach passenden Anwendungsstellen in großen Datenmengen nicht praktikabel. Daher sollte die Komplexität der Suche reduziert werden. In GReAT geschieht dies dadurch, dass der Suche zusätzlich ein Kontext als Parameter übergeben wird. Dieser Kontext beinhaltet Knoten und Kanten der Anwendungsstelle und gibt vor, dass ein bestimmter Knoten des Graphen mit einem bestimmten Knoten des Musters übereinstimmt. Der Kontext ist so gesehen schon ein Teil des gesuchten Ergebnisses. Angewendet auf das Muster aus Abbildung 4 und die Daten aus Abbildung 5 sowie dem Kontext {(*Student*, *Bastian Schoofs*)} wird die Mustersuche nur noch das Ergebnis {{(*Student*, *Bastian Schoofs*), (*Vorlesung*, *Malo*), (*Diplom*, *Diplom*)}, {(*VorlesungsRelation*, *BS-*

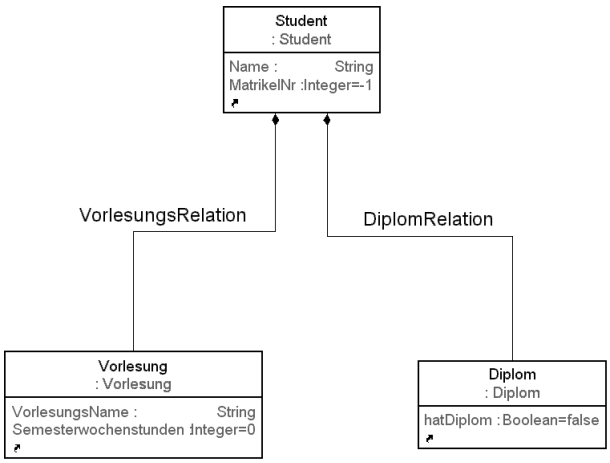


Abbildung 4: Einfaches Muster

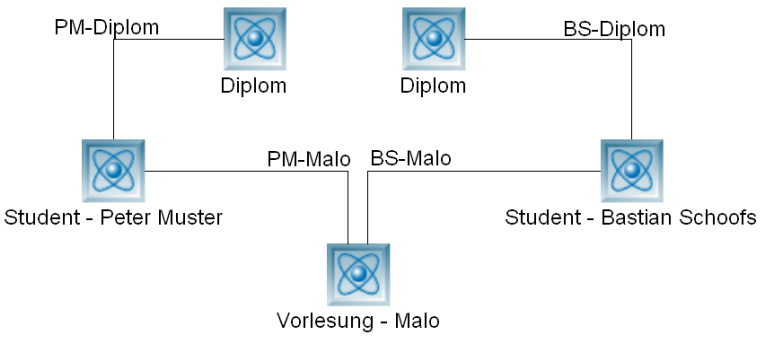


Abbildung 5: Einfacher Graph

*Malo*), (*DiplomRelation*, *BS-Diplom*)}} zurückgeben. Durch diesen Kontext verringert sich die Suchkomplexität in zwei Aspekten. Erstens muss ein Knoten des Musters weniger gesucht werden und zweitens müssen nur noch Knoten innerhalb einer Distanz $d$ ausgewertet werden, wobei $d$ die Länge des längsten Pfades im Muster ist.

## 3.2 Erweiterte Muster

Zusätzlich zu den eben beschriebenen einfachen Mustern gibt es noch einige andere Arten von Mustern, nach denen gesucht werden kann. Wird mit einem Muster in den Daten aus Abbildung 7 ein Student gesucht, der an zwei Vorlesungen

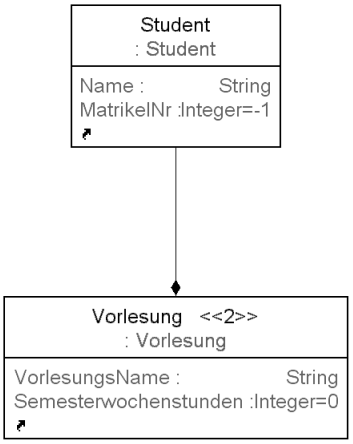


Abbildung 6: Erweitertes Muster mit Knotenanzahl

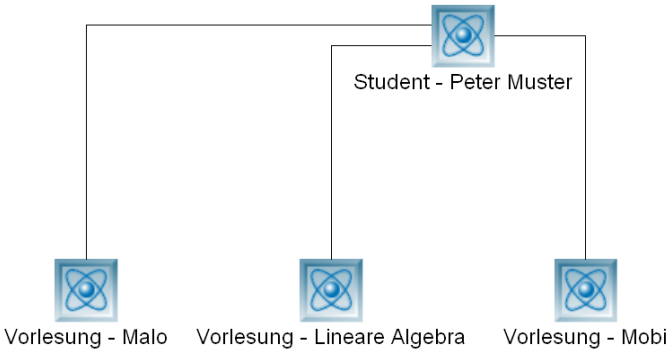


Abbildung 7: Instanzgraph mit drei Vorlesungen

teilgenommen hat, muss das Muster aus einem Studentenknoten und zwei Vorlesungsknoten bestehen. Um Suchmuster zu vereinfachen, in denen mehrere Objekte derselben Klasse gesucht werden, wird die Definition der Suchmuster erweitert. Die Knoten erhalten einen zusätzlichen Stereotypen, um darin die gesuchte Anzahl der Knoten anzugeben. Wenn der Stereotyp nicht explizit angegeben wird, ist die Anzahl 1. Das Suchen eines Studenten, der zwei Vorlesungen besucht, würde daraufhin mit dem Suchmuster aus Abbildung 6 geschehen. Die Anzahl der gesuchten Knoten ist in spitzen Klammern hinter dem Namen des Knotens zu erkennen. Das Muster in Abbildung 6 findet, angewendet auf den Graphen aus Abbildung 7, mehrere Ergebnisse. Das ist darauf zurückzuführen, dass alle möglichen Kombinationen der *Vorlesung* Objekte in den Ergebnissen zu finden sind.

Jedes der zurückgegebenen Ergebnisse würde sich in den zurückgegebenen *Vorlesung* Objekten unterscheiden.

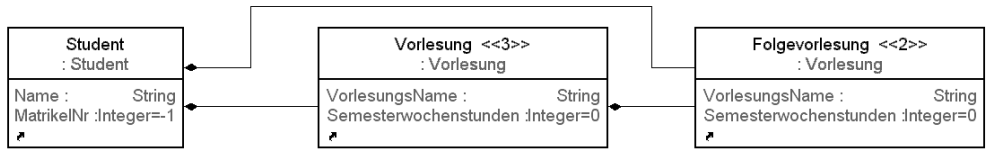


Abbildung 8: Erweitertes Muster mit Folgevorlesungen

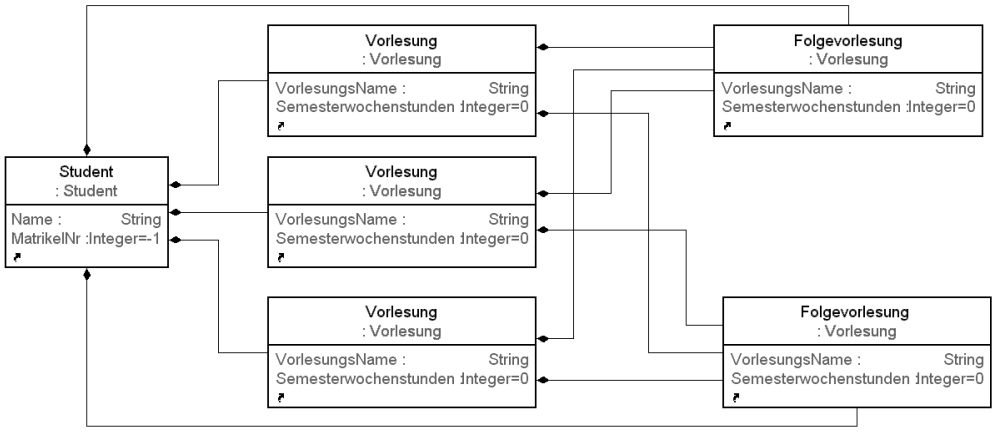


Abbildung 9: Beispiel Mengensemantik

Wird das Muster aus Abbildung 6 um einen Knoten vom Typ *Vorlesung* erweitert, wird die Darstellung dadurch mehrdeutig. Ein solch erweitertes Muster ist in Abbildung 8 dargestellt. Der hinzugefügte Knoten *Folgevorlesung* stellt den Sachverhalt dar, dass Vorlesungen existieren, die auf anderen Vorlesungen aufbauen. GReAT interpretiert ein solches Muster nach der Mengensemantik. Die Mengensemantik geht davon aus, dass mit einer Kante im Muster nur Kanten im Graphen gefunden werden, die mit den gleichen Datenobjekten verbunden sind. Angewendet auf das Beispiel bedeutet dies: Die Kante im Muster vom Knoten *Folgevorlesung* zum Knoten *Vorlesung* findet bei der Anwendungsstellensuche immer die gleichen Objekte vom Typ *Folgevorlesung*. Diese Interpretation des Musters führt dazu, dass sich das Muster genau so wie das Muster aus Abbildung 9 verhält.
Alternativ zur Interpretation des Musters mit der Mengensemantik existiert die Möglichkeit das Muster mit der Baumsemantik zu interpretieren. In der Interpre-

tation mit der Baumsemantik bedeutet es, dass der Knoten *Folgevorlesung* (Anzahl 2), der mit dem Knoten *Vorlesung* verbunden ist, für jedes gefundene Objekt vom Typ *Vorlesung* die gleichen beiden Objekte vom Typ *Folgevorlesung* findet. Für den Fall, dass das Muster aus Abbildung 8 mit der Baumsemantik interpretiert wird, werden die gleichen Anwendungsstellen wie mittels des Musters aus Abbildung 10 gefunden.

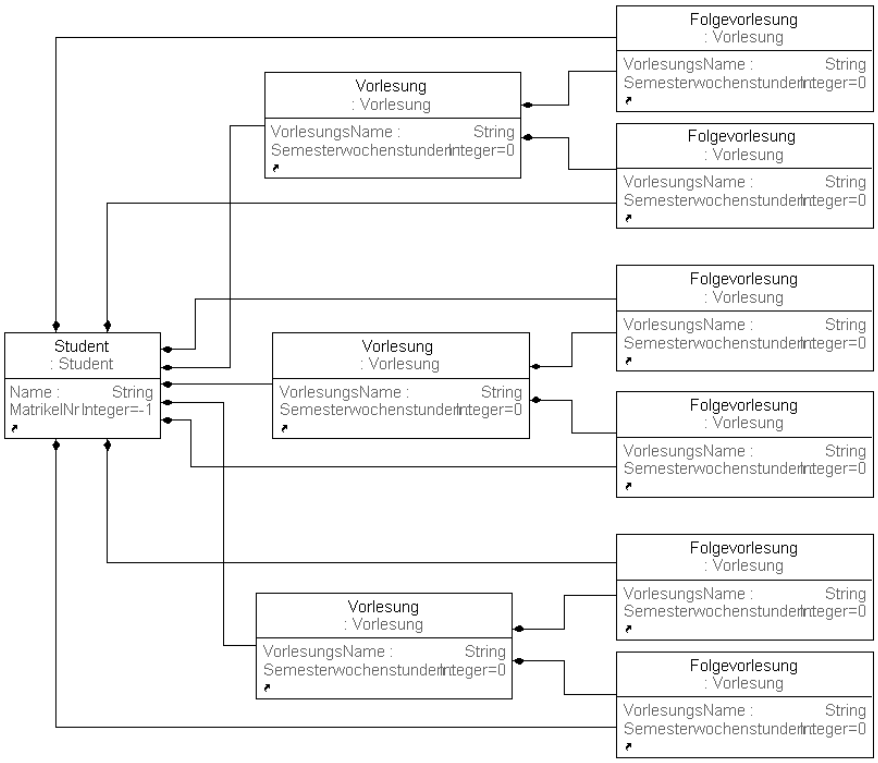


Abbildung 10: Beispiel Baumsemantik

Die Interpretationen des Musters aus Abbildung 8 finden unterschiedliche Anwendungsstellen. Die Interpretation des Musters mittels der Baumsemantik findet mehr Anwendungsstellen als die Interpretation mittels der Mengensemantik. Um das Muster aus Abbildung 10 ebenfalls in kompakter Form darstellen zu können, muss die Notation erweitert werden [4].
Hierzu wurde die erweiterte Mengensemantik entwickelt. In der erweiterten Mengensemantik werden die Muster um die Möglichkeit erweitert, Teilgraphen und Knotenmengen zu gruppieren. Vereinfacht kann dies mit Hilfe eines Strings veranschaulicht werden. Ein String “abcbcbcbc” kann auch durch folgendes Muster dargestellt werden: “a4(bc)”. Hierbei wird das Teilwort “bc” zusammengefasst und 4 mal wiederholt. Dasselbe Prinzip lässt sich auch auf Graphen anwenden.

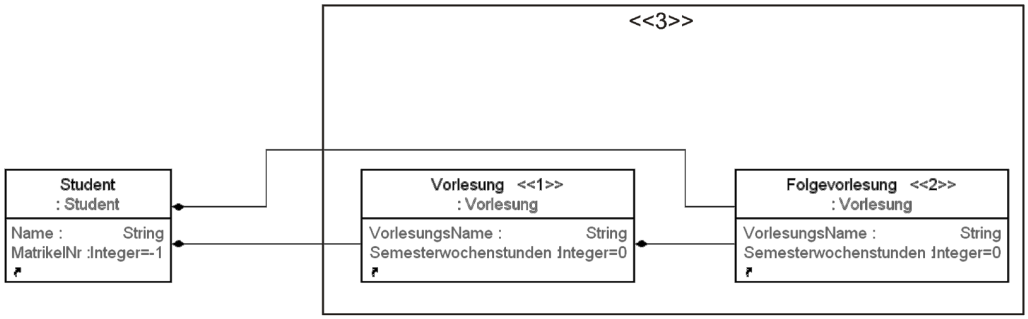


Abbildung 11: Beispiel erweiterte Mengensemantik

Wenn ein Muster einen Teilgraphen mit der Kardinalität 4 beinhaltet, muss dieser Teilgraph auch vier-mal im Graphen gefunden werden. Das Muster aus Abbildung 8 lässt sich mit der erweiterten Mengensemantik wie in Abbildung 11 darstellen. Das dargestellte Muster findet alle Anwendungsstellen des Musters aus Abbildung 10. Zur Erstellung eines Musters existiert leider kein Syntaxgesteuerter Editor, um die Mustererstellung zu erleichtern. Es existiert auch keine Möglichkeit, die erstellten Muster gegen die Modelle zu prüfen. Da eine solche Prüfung nicht existiert, ist es möglich, Muster zu erstellen, die zu keinem definierten Modell passen und daher auch keine Ergebnisse finden.
Analog zur Definition der Knotenanzahl im Muster ist es durch die Erweiterung der Kantendefinition möglich, zu definieren, wie viele Kanten zwischen denselben Knoten existieren sollen. Hierzu erhalten die Kanten einen neuen Stereotypen. Das Muster aus Abbildung 12 vereinfacht durch das Muster in Abbildung 13 dargestellt werden. Die sechs Kanten aus Abbildung 12 können vereinfacht durch drei Kanten dargestellt werden. Ebenso lassen sich die drei *Vorlesung* Knoten mittels der Knotenanzahl in einer kompakten Form darstellen. Dies vereinfacht die Lesbarkeit von Mustern mit mehreren Kanten zwischen denselben Knoten enorm.

Es kann jedoch auch Muster geben, bei denen die Anzahl der gesuchten Kanten und Knoten nicht feststeht. Wie bei regulären Ausdrücken existiert auch hier eine Möglichkeit, zu definieren, dass eine Kante oder ein Knoten beispielsweise dreibis fünfmal existieren soll. Um die Obere und Untere Grenze in der Knoten und Kanten festzulegen, kann die gesuchte Anzahl in der Form $(x..y)$ definiert werden. Wobei $x$ definiert, wie viele Kanten/Knoten mindestens vorhanden sein müssen und $y$ definiert, wie viele Kanten/Knoten maximal existieren dürfen. Als maximale Obergrenze kann auch $*$ eingesetzt werden, um darzustellen, dass es keine Obergrenze gibt. Genau so wie darstellbar ist, dass mindestens 4 Knoten/Kanten existieren müssen, kann dargestellt werden, dass bestimmte Kanten/Knoten optional sind. Dies wird durch das Benutzen von $(0..1)$ erreicht. Durch Einführen

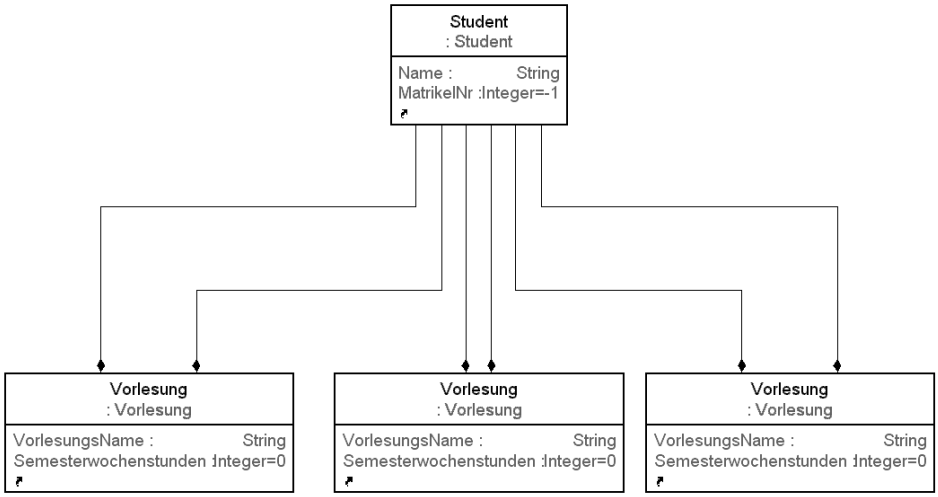


Abbildung 12: Muster ohne definierte Kanten- und Knotenanzahl

der variablen Kanten- und Knotenanzahl lassen sich sehr viele Anwendungsstellen durch Muster finden.
Die variable Kanten- und Knotenanzahl bringt jedoch auch neue Probleme mit sich. Soll zum Beispiel mit dem Muster aus Abbildung 14 in den Daten aus Abbildung 15 eine Anwendungsstelle gesucht werden, ist nicht klar, welche Ergebnisse zurückgeliefert werden. Soll das Programm nur den gesamten Graph zurückliefern oder alle möglichen Teilgraphen, die die minimale Knotenanzahl erfüllen? Die Suche wird in diesem Fall nur eine Anwendungsstelle zurückliefern, den gesamten Graphen, der eine maximale Anwendungsstelle ist. Die Anwendungsstellensuche liefert immer maximale Anwendungsstellen zurück und für den Fall, dass mehrere Anwendungsstellen möglich sind und zurückgegeben werden, darf keine der Anwendungsstellen eine Teilmenge einer anderen sein. Maximale Anwendungsstellen sind Anwendungsstellen, bei denen die größtmögliche Anzahl von Knoten und Kanten gemäß ihrer Anzahl im Muster gefunden wurde. Hierdurch ist gewährleistet, dass keine der Anwendungsstellen sich überschneidet und es doppelte Ergebnisse gibt.

## 3.3 Zusammenfassung

Zusammengefasst lassen sich Muster, die zur Suche von Anwendungsstellen in Graphen verwendet werden, wie folgt definieren. Ein Knoten eines Musters ist vom Typ einer Klasse aus einem Modell und besitzt einen Stereotyp zur Festle-

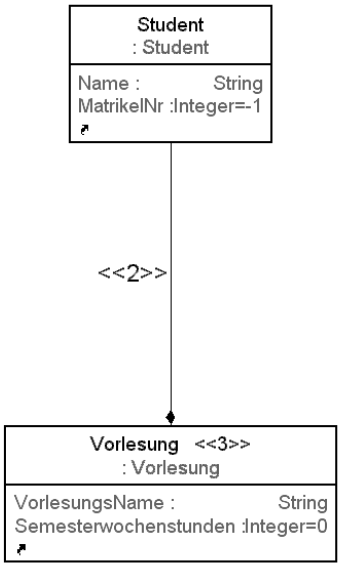


Abbildung 13: Muster mit definierter Kanten- und Knotenanzahl

gung der gesuchten Anzahl des Knotens. Die Klasse definiert alle Attribute, die der Knoten repräsentiert. Die Anzahl des Knotens wird durch eine obere und untere Grenze bestimmt, wobei beide positive natürliche Zahlen oder $*$ sein müssen. Eine Kante in einem Muster besteht aus einem 4-Tupel (Kantentyp, Startknoten, Endknoten, Anzahl). Der Start- und Endknoten der Kante muss aus der Knotenmenge des Musters stammen. Zusätzlich wird in der Kante auch der Stereotyp zur Festlegung der gesuchten Kantenanzahl gespeichert.
Die zurückgelieferten Anwendungsstellen, die bei der Suche von einem Muster auf einem Graphen zurückgegeben werden, bestehen jeweils aus einer Menge von Kanten- und Knotenrelationen. Die Kantenrelationen bestehen aus einer Kante des Musters und einer Menge von Kanten des Graphen, in dem gesucht worden ist. Dies gilt analog für die Knotenrelationen.

# 4 Graphtransformation

Die Hauptaufgabe von GReAT ist es, Daten einer Domäne in Daten einer anderen Domäne zu überführen. Die Daten der Startdomäne werden bei der Transformation nicht verändert, die Daten der Zieldomäne werden aus den Startdaten generiert. Es kann Fälle geben, bei denen es nicht direkt möglich ist, Daten von der Startdomäne in die entsprechende Zieldomäne zu überführen. In diesem Fall ist

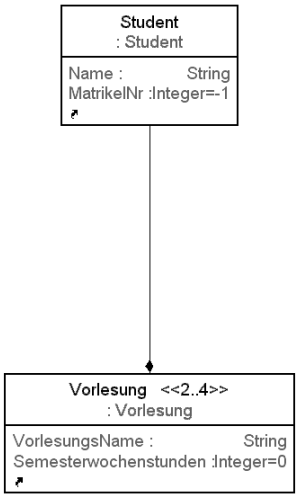


Abbildung 14: Muster mit variabler Knotenanzahl

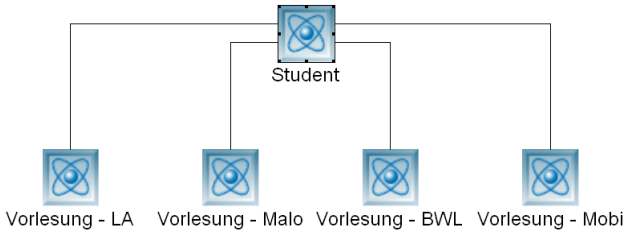


Abbildung 15: Instanzgraph eines Studenten mit vier Vorlesungen

es notwendig, erst die Daten von der Startdomäne in eine definierte Zwischendomäne zu überführen und diese dann in die Zieldomäne. Solche Zwischendomänen werden dann gebraucht, falls in der Transformation Objekte benötigt werden, die nicht Teil der Start- oder Zieldomäne sind. Eine Zwischendomäne würde in der Studententransformation gebraucht, falls die Summierung der Vorlesungen nicht mit den vorhandenen Elementen zu lösen ist.
Damit die Transformation kontrolliert ausgeführt werden kann, benutzt GReAT verschiedene Arten von Kontrollstrukturen. Diese können in GReAT zur direkten Transformation von Daten oder zur Steuerung des Transformationsprozesses genutzt werden. Daten werden mit Hilfe von *Regeln* transformiert. Auf den Aufbau von Regeln wird in Kapitel 4.1 eingegangen. Die *Entscheidungsregeln* werden verwendet, um Entscheidungen zu treffen, siehe Kapitel 4.2. Wenn in Abhängigkeit von Werten eines bestimmten Elementes eine bestimmte Regel ausgeführt werden soll, dann wird hierzu eine *Entscheidungsregel* verwendet. Als Kontrollstrukturen für die Strukturierung des Transformationsprozesses werden in GReAT *Block*, *ForBlock* und *Konditionale Regel* Elemente eingesetzt. Die *Graphregeln* können innerhalb von *Block* und *ForBlock* Elementen aufgerufen werden. Im Unterschied zu den *Graphregeln* können die *Block* und *ForBlock* Elemente sich gegenseitig aufrufen und beinhalten. Da die *Graphregeln* nur zur Transformation der Daten genutzt werden, enthalten sie keine *Block* und *ForBlock* Elemente. Der genaue Aufbau von *Block* und *ForBlock* wird in Kapitel 4.3 beschrieben.

## 4.1 Aufbau einer Regel

Alle Regeln, die für die Transformation in GReAT definiert werden, enthalten ein Suchmuster. Dieses Muster dient einerseits dazu, passende Anwendungsstellen in den Daten zu finden und andererseits um Daten zu löschen und/oder neu anzulegen. Um dies zu ermöglichen, existiert in jeder Kante und jedem Knoten aus dem Muster ein zusätzlicher Stereotyp, der definiert, was mit dem Objekt des Suchmusters geschehen soll. Der neue Stereotyp in den Kanten und Knoten der Muster kann drei verschiedene Werte enthalten [9]:

- *bind*: Das Objekt wird für die Suche der Anwendungsstellen benutzt.
- *delete*: Das Objekt wird für die Suche der Anwendungsstellen benutzt und anschließend gelöscht.
- *new*: Das Objekt wird nicht für die Suche benutzt. Falls die Anwendungsstelle gefunden wurde, wird dieses Objekt neu erstellt.

Objekte, die mit *delete* gekennzeichnet sind, werden in GReAT durch ein rotes "X" innerhalb des Musterknotens markiert und solche, die mit *new* gekennzeichnet sind, werden mit einem blauen Häckchen markiert. Die Objekte, die mit *bind* gekennzeichnet sind, werden nicht besonders dargestellt, da *bind* der Standard für die Objekte ist.
Die Anwendungsstellensuche sucht nun zuerst die Anwendungsstellen, die durch die *bind* und *delete* Objekte des Musters definiert werden. Nachdem diese Anwendungsstellen gefunden worden sind, werden alle Objekte, die mit *delete* markiert sind, gelöscht und diejenigen, die mit *new* markiert sind, erstellt. Dadurch verändert sich der Graph, durch den die Daten repräsentiert werden. Dieser veränderte Graph kann in der nächsten Regel weiter transformiert werden.

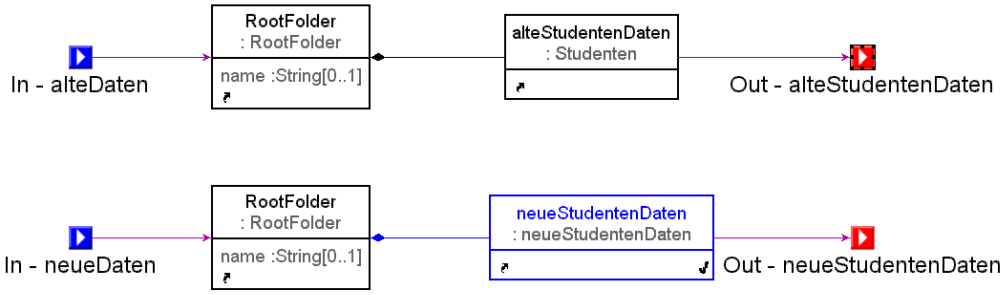


Abbildung 16: Regel zum Erstellen des Basismodelles

In der Transformation der Studentendaten wird der Wert *delete* nicht verwendet, jedoch die beiden anderen. Die erste Regel aus der Transformation ist in Abbildung 16 dargestellt. Die Regel dient dazu, das initiale Modell Objekt in den neuen Studentendaten zu erstellen. Bei der Suche nach den Anwendungsstellen wird wie folgt vorgegangen. In den alten Daten wird ein *RootFolder* Objekt gesucht, das mit einem *Studenten* Objekt verbunden ist. Ebenso wird in den neuen Daten das *RootFolder* Objekt gesucht. Das *RootFolder* Objekt ist die Basis eines jeden Modells und wird nicht explizit im Modell definiert. Jede Datendatei besitzt einen solchen *RootFolder*, er muss nicht erst angelegt werden. Nachdem die Anwendungsstelle gefunden wurde, wird ein neues Objekt *neueStudentenDaten* erstellt und mit dem gefundenen *RootFolder* der neuen Daten verknüpft. Somit wurde das erste Element der Zieldomäne in der Beispieltransformation erstellt.
Die Regeln von GReAT haben bestimmte Pflichtbestandteile und bestimmte optionale Elemente. Jede Regel muss sogenannte Ports besitzen. Ports sind die Schnittstellen zwischen den einzelnen Regeln. Es gibt IN-Ports und OUT-Ports. Die IN-Ports repräsentieren die Eingabeparameter für eine Regel, die OUT-Ports die Ausgabeparameter. Regeln müssen jedoch mindestens einen IN-Port enthalten, die OUT-Ports sind optional. Die Ein- und Ausgabeparameter der Regeln sind in GReAT der Suchkontext mit Elementen aus dem Datengraphen. Dieser Kontext wird wie in Kapitel 3.1 verwendet, um die Suche zu vereinfachen.
Wenn eine Regel die Parameter der IN-Ports auswerten soll, müssen sie an die passenden Knoten des Suchmusters gebunden werden. Im Beispiel aus Abbildung 16 ist der *IN* Port der alten Daten mit dem *RootFolder* aus der Startdomäne verbunden, der *IN* Port der neuen Daten mit dem *RootFolder* aus der Zieldomäne.
Mit Hilfe der IN Ports wird der Kontext, der für die Suche benutzt wird, an die Regel weitergegeben. Um die Klasse des Objektes, das weitergegeben werden soll, zu bestimmen, sind die OUT-Ports mit den jeweiligen Klassen im Muster verbunden. Es ist auch möglich, die Objekte der IN-Ports direkt an die OUT-Ports der Regel weiterzugeben. In diesem Fall werden die Objekte nicht betrachtet. Eine Regel ist so gesehen also nicht viel anders als eine Funktion, die mit Eingabe- und Ausgabeparametern aufgerufen wird. Findet eine Regel bei ihrer Ausführung mehrere Anwendungsstellen, kann es passieren, dass mehrere Ausgabe Objekte zu den OUT-Ports der Regel weitergeleitet werden. Jedes dieser Objekte wird auch als *Datenpaket* bezeichnet. Jede Regel arbeitet daher mit Datenpaketen von Eingabedaten. Nach ihrer Ausführung kann eine Menge von Ausgabepaketen erstellt werden.
Eine Transformation wird beendet, wenn eine Regel keine Pakete generiert hat, die an eine nachfolgende Regel weitergeleitet werden können und es ansonsten keine weiteren Pakete gibt, die bearbeitet werden müssen. Eine Regel generiert keine Pakete zum Weiterleiten, falls keine Anwendungsstelle gefunden wurde oder ein *Wächter* die Bearbeitung der Regel abgebrochen hat. Ebenso wird die Transfor-

mation beendet, wenn eine Regel ohne OUT-Ports aufgerufen wird. Falls die Objekte, mit der eine Regel aufgerufen wird, nicht mit den Objekten übereinstimmt, die im Muster definiert sind, wird die Regel und alle darauffolgenden nicht ausgeführt. Eine explizite Prüfung, ob die Regel mit den richtigen Objekten aufgerufen wird, existiert nicht.
Zu den optionalen Elementen einer Regel gehören *Wächter*. *Wächter* definieren Bedingungen, die nach dem Suchen der Anwendungsstelle überprüft werden. Diese Bedingungen werden vor dem Hinzufügen oder Löschen von Objekten der Daten überprüft. Nur wenn die Bedingungen erfüllt sind, werden die Daten gemäß dem Suchmuster bearbeitet. Die in den *Wächtern* enthaltenen Bedingungen werden in C++ geschrieben. Sollte der *Wächter* nicht wahr sein, wird die Bearbeitung der Regel abgebrochen, ohne dass die Daten verändert werden. Damit die Daten nicht verändert werden, wird der Wächter ausgeführt, nachdem mit dem Muster Anwendungsstellen gefunden worden sind und bevor Daten gelöscht oder angelegt werden. Sollen in der Studententransformation nur Studenten betrachtet werden, deren Matrikelnummer größer als 400000 ist, kann dies durch einen Wächter geschehen.
Ein weiteres optionales Element einer Regel ist die *Attributauswertung*. Die *Attributauswertung* wird verwendet, um neu angelegte oder vorhandene Elemente nach der Anwendungsstellensuche mit Werten aus anderen Elementen zu belegen. Die *Attributauswertung* beinhaltet genau wie die Wächter C++ Code. Innerhalb des Codes können die Attribute aller Knoten, die im Suchmuster der Regel verwendet wurden, mit Hilfsfunktionen erreicht werden. Die *Attributauswertung* wird nach dem *Wächter* und, nachdem neue Daten angelegt worden sind, ausgeführt. Dadurch kann in der *Attributauswertung* noch auf Daten zugegriffen werden, die anschließend gelöscht werden. In der Studententransformation wird die *Attributauswertung* zum Beispiel dazu verwendet, um den Namen und die Matrikelnummer der Studenten von den alten Daten auf die neuen Daten zu übertragen.
Eine Regel aus der Transformation der Studentendaten, die beide optionalen Elemente beinhaltet, ist in Abbildung 17 dargestellt. Diese Regel wird benutzt, um ein neues Objekt vom Typ *StudentMitDiplom* im Modell *neueStudentenDaten* zu erstellen. Die Ausführung dieser Regel geschieht folgendermaßen. Zuerst wird in den alten Daten passend zum übergebenen *StudentenDaten* Objekt das Muster ausgewertet. Im Muster der Abbildung ist ein *Connector*. Der *Connector* ist in der Abbildung als ein kleiner schwarzer Punkt zwischen den *Vorlesungen* und *StudentenDaten* Objekten zu erkennen. Dieses Element wird verwendet, wenn es notwendig ist, auf Attribute von Kanten im Datengraphen zuzugreifen. Mittels einer gestrichelten Linie ist der *Connector* auch mit dem Element verbunden, das die Kante des Datengraphen beschreibt. Das hierzu in der Abbildung verwendete Objekt *VorlesungsConnection* wird benötigt, da es ein Attribut *besuchtVorlesung* beinhaltet, welches vom *Wächter* ausgewertet werden soll. Der *Wächter* wird in

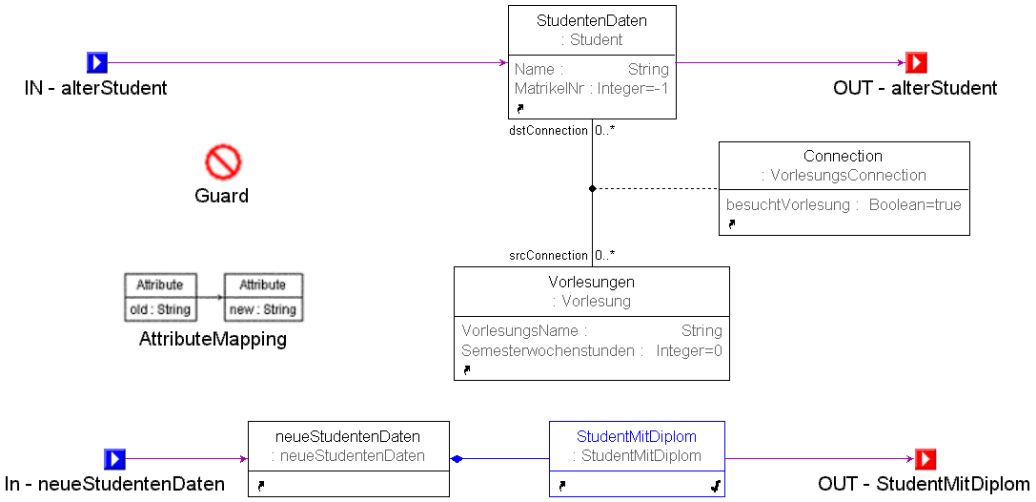


Abbildung 17: Regel zur Erstellung eines StudentMitDiplom Objektes

der Abbildung als *Wächter* dargestellt. Nachdem mit dem Muster die Objekte gefunden worden sind, prüft der *Wächter*, ob das Attribut *besuchtVorlesung* mit dem Wert *"true"* gefüllt ist. Ist dies der Fall, wird das neue *StudentMitDiplom* Objekt angelegt und mit dem *neueStudentenDaten* Objekt verbunden. Somit ist gewährleistet, dass in der Transformation nur Vorlesungen betrachtet werden, die der Student auch besucht hat. Der Quellcode des in der Abbildung verwendeten *Wächters* lautet wie folgt:

```
bool boolValue;
Connection.GetBoolValue("besuchtVorlesung",boolValue);
if(boolValue)
return true;
else
return false;
```

Nachdem der Wächter die zurückgegebenen Anwendungsstellen dahingehend gefiltert hat, dass nur solche in Betracht kommen, bei denen der Student eine Vorlesung besucht hat, wird für jede Anwendungsstelle ein neues *StudentMitDiplom* Objekt angelegt. Um die neuen Objekte mit Werten zu füllen, wird die *Attributauswerung* ausgeführt. Diese sorgt dafür, dass das neue *StudentMitDiplom* Objekt mit Werten gefüllt wird. In dieser Regel weist die *Attributauswerung* dem neuen Objekt *StudentMitDiplom* die Attribute *Name*, *Matrikelnummer* sowie die Note des Diploms zu. Der Code der *Attributauswertung* lautet wie folgt:

```
__int64 matNr;
StudentenDaten.GetIntValue("MatrikelNr",matNr);
string strName;
StudentenDaten.GetStrValue("Name",strName);
string strDiplomNote;
Diplom.GetStrValue("DiplomNote",strDiplomNote);

StudentMitDiplom.SetStrValue("Name",strName);
StudentMitDiplom.SetIntValue("MatrikelNr",matNr);
StudentMitDiplom.SetStrValue("Note",strDiplomNote);
```

Nachdem die *Attributauswertung* ausgeführt ist, werden alle Objekte der Anwendungsstelle, die gelöscht werden sollen, aus den Daten entfernt. Da in der Regel aus Abbildung 17 im Muster kein Element zum Löschen markiert ist, werden keine der Daten gelöscht.

## 4.2 Entscheidungsregeln

Eine wichtige Art der Kontrollstrukturen sind Entscheidungsregeln. Sie werden benutzt, um aufgrund von bestimmten Bedingungen, weitere Regeln oder Kontrollstrukturen aufzurufen oder die Transformation abzubrechen. Die Elemente, die in GReAT Entscheidungsregeln darstellen, werden *Test* genannt.

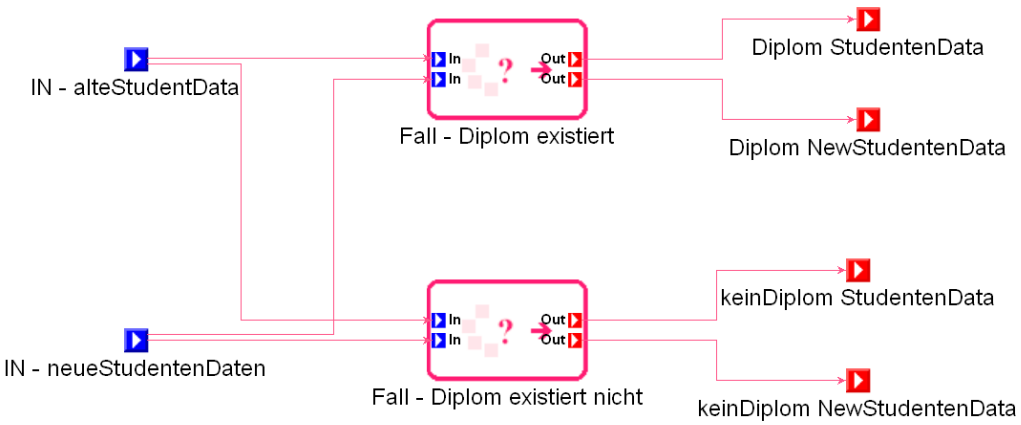


Abbildung 18: Bsp. Entscheidungsregel

Eine Entscheidungsregel besitzt nur IN- und OUT-Ports sowie *Case*-Regeln. *Case*-Regeln sind besondere Regeln, in denen es zum Beispiel keine *Attributauswer-*

*tung* gibt. Mit diesen Regeln können keine Daten verändert werden. Falls eine Entscheidungsregel mehrere *Case*-Regeln beinhaltet, müssen die IN-Ports mit allen *Case*-Regeln verbunden sein. Das standardmäßige Verhalten einer Entscheidungsregel entspricht einem “if”-Statement ohne “else”. Alle *Case*-Regeln werden ausgeführt. Wenn eine *Case*-Regel erfüllt ist, werden deren Ausgabeobjekte an die dazugehörigen OUT-Ports geleitet.
Eine Entscheidungsregel aus der Transformation der Studenten ist in Abbildung 18 dargestellt. Diese Regel wird gebraucht, um zu testen, ob der gefundene Student ein Diplom besitzt oder nicht. In Abhängigkeit davon, ob der Student ein Diplom besitzt oder nicht, muss eine andere Graphregel aufgerufen werden. Die Entscheidungregel aus der Abbildung besitzt 2 IN-Ports, 2 *Case*-Regeln und 4 OUT-Ports. Die *Case*-Regeln werden in GReAT durch ein Fragezeichen im Regelsymbol gekennzeichnet. Die IN-Ports sind mit jeweils beiden *Case*-Regeln verbunden. Die eine *Case*-Regel *Fall - Diplom existiert* prüft, ob der übergebene Student ein Diplom besitzt. Die andere Regel *Fall - Diplom existiert nicht* prüft den Fall, dass der Student kein Diplom hat. Wenn eine *Case*-Regel erfüllt ist, leitet diese die Parameter an die dazugehörigen OUT-Ports.
Falls nicht gewünscht ist, dass alle *Case*-Regeln in der Entscheidungsregel ausgeführt werden, sondern nur die erste passende, kann dieses Verhalten erzwungen werden. Dies geschieht dadurch, dass in allen *Case*-Regeln ein *Cut*-Element eingefügt wird. Das *Cut*-Element kann nur in *Case*-Regeln verwendet werden. Wenn das *Cut*-Element in der *Case*-Regel existiert und eine andere *Case*-Regel schon erfolgreich war, wird diese und alle weiteren *Case*-Regeln nicht ausgeführt. Wenn alle *Case*-Regeln ein *Cut*-Element besitzen, verhält sich die Konditionale Regel wie ein “if-elseif” Konstrukt. Falls eine *Case*-Regel kein *Cut*-Element besitzt und erfolgreich ist, wird die nächste *Case*-Regel ausgeführt. Das geschieht so lange, bis alle *Case*-Regeln abgearbeitet wurden, oder eine *Case*-Regel mit einem *Cut*-Element erfolgreich war. Jedoch ist bei diesem Konstrukt die Reihenfolge, in der die *Case*-Regeln abgearbeitet werden, entscheidend. GReAT bestimmt die Reihenfolge durch die Anordnung der Fälle in der Entscheidungsregel. Die *Case*-Regeln werden von oben nach unten ausgewertet und, falls eine Verbindung zwischen den *Case*-Regeln existiert, von links nach rechts. In diesem Fall müssen alle miteinander verbundenen *Case*-Regeln erfüllt sein, damit die Datenpakete zu den *OUT*-Ports gelangen. Die Positionierung der *Case*-Regeln innerhalb der Entscheidungsregel wird mit Hilfe des visuellen Editors gemacht.
Die *Case*-Regel *Fall - Diplom existiert* aus Abbildung 18 ist in Abbildung 19 dargestellt. Diese *Case*-Regel, sowie die andere *Case*-Regel, besitzt ein *Cut*-Element, das dafür sorgt, dass jeweils immer nur eine Regel erfolgreich ausgeführt wird. In der *Case*-Regel existiert ein *Wächter* Element, das prüft, ob das Attribut *hatDiplom* vom Diplom Objekt dem Wert *“true”* entspricht. Ist dies der Fall, wird das *Student* und *neueStudentenDaten* Objekt an die OUT-Ports weitergeleitet.

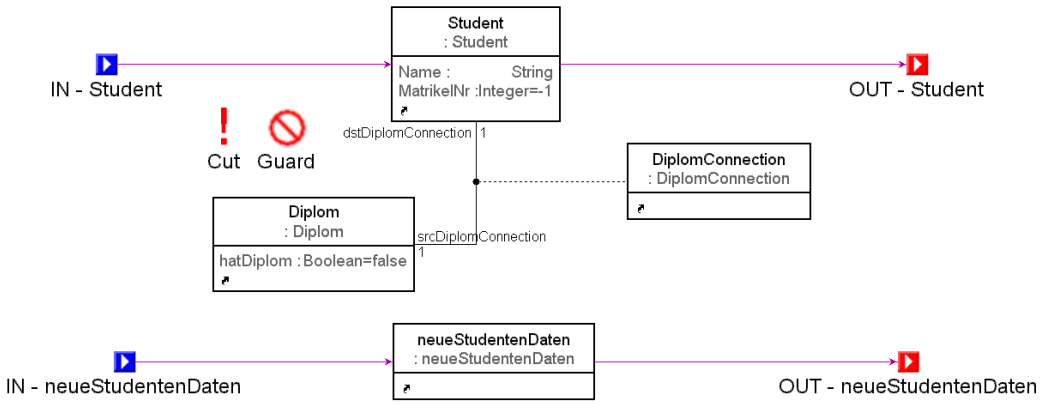


Abbildung 19: Beispiel einer *Case*-Regel

## 4.3 Block/ForBlock Regeln

GReAT besitzt eine Kontrollflusssprache, die es ermöglicht, strukturiert Regeln abzuarbeiten. Sonderstatus in dieser Sprache haben die Entscheidungsregeln, wie schon in Kapitel 4.2 beschrieben. Der Kontrollfluss einer Transformation in GReAT wird durch das Benutzen von Regeln erzeugt. In der Transformation von Studentendaten erzeugt die Regel *Suche Studenten* (Abbildung 20) für jedes Studenten Objekt in den Daten ein eigenes Ausgabepaket. Es werden daher 2 Ausgabepakete erstellt. Das eine Datenpaket enthält den Studenten Peter Muster und das andere den Studenten Bastian Schoofs. Zusätzlich enthalten beide Datenpakete das Objekt *neueStudentenDaten*.
Die in GReAT implementierte Kontrollflusssprache erfüllt die folgenden Eigenschaften [7]:

1. *Test/Case*: Es existieren Entscheidungsregeln, die in Abhängigkeit von Bedingungen verschiedene Regeln aufrufen

2. *Sequenziell*: Die einzelnen Regeln werden nacheinander abgearbeitet.

3. *Hierarchisch*: Block/ForBlock Regeln können andere Regeln beinhalten.

4. *Nicht-Determinismus*: Parallele Regeln werden nicht-deterministisch abgearbeitet.

5. *Wiederverwendbarkeit von Regeln*: Vorhandene Regeln können an unterschiedlichen Stellen wiederverwendet werden.

6. *Rekursion*: Regeln können sich direkt und indirekt wieder selbst aufrufen.

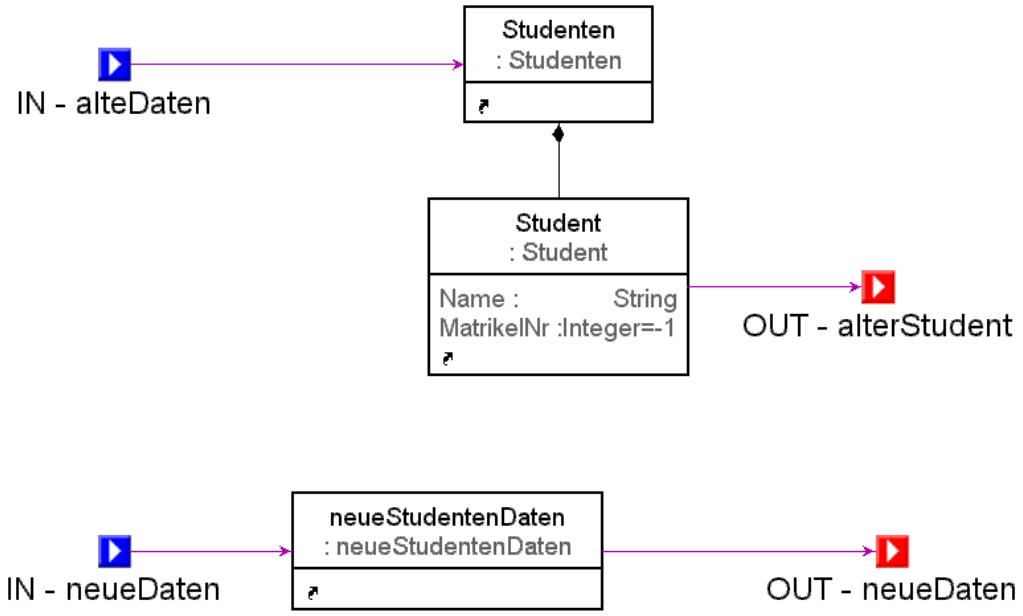


Abbildung 20: Suche Studenten Regel

Die *sequenzielle* Verknüpfung geschieht durch Verbinden der OUT-Ports einer Regel mit den IN-Ports einer anderen Regel. Durch die Verknüpfung der Regeln miteinander wird die Reihenfolge, in der sie abgearbeitet werden, vorgegeben. Zuerst wird die Regel abgearbeitet, die mit den IN-Ports der Regel verbunden ist, und danach die Regel, die mit den OUT-Ports der vorherigen Regel verbunden ist. Siehe hierzu Abbildung 21. Die Abbildung zeigt die *Start* Regel der Studentendatentransformation. Die Eingabepakete der Block-Regel werden zuerst in der Regel *Erstelle Basis Modell* bearbeitet, danach in der Regel *Suche Studenten* und zum Schluss in *Bearbeite Studenten*. Die Regel *Erstelle Basis Modell* ist in Abbildung 16 dargestellt. Sie wird benutzt, um in den neuen Daten ein Modell Objekt zu erzeugen, in das alle Studenten gespeichert werden sollen. Die Zweite Regel *Suche Studenten* ist in Abbildung 20 dargestellt und dient der Suche der Studenten Objekte in den alten Daten. Für jeden der gefundenen Studenten Objekte wird die dritte Regel *Bearbeite Studenten* aufgerufen, damit die vollständige Transformation abgeschlossen wird. Eine genaue Beschreibung dieser Regel wird später in diesem Kapitel gegeben. Die letzte Regel besitzt keine OUT-Ports, da nach ihrer Bearbeitung die Transformation beendet ist.

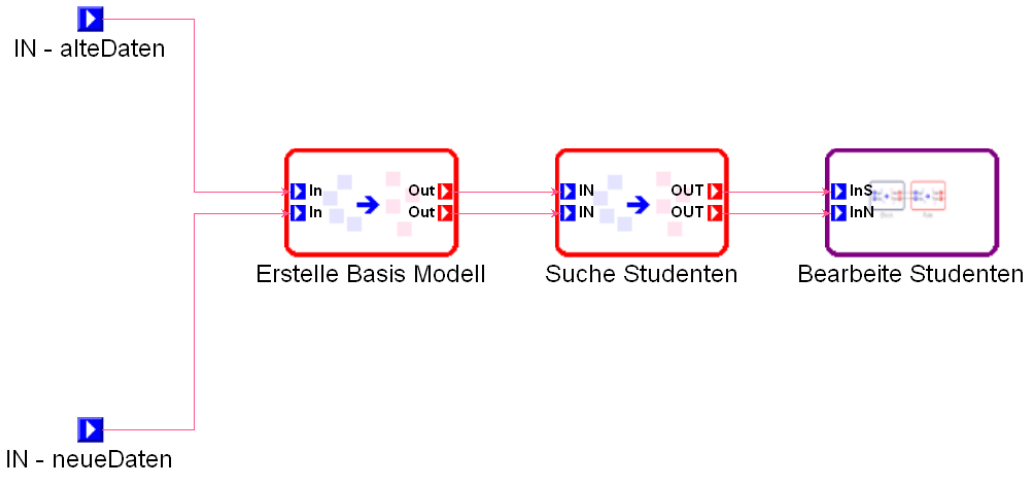


Abbildung 21: Block Regel aus der Beispieltransformation

### 4.3.1 Block Regeln

Es gibt zwei Arten von *hierarchischen* Regeln: *Block* und *ForBlock*. Diese *hierarchischen* Regeln werden dazu genutzt, um die Reihenfolge, in der die Regeln abgearbeitet werden sollen, zu strukturieren. Im Unterschied zu den Graphregeln können *hierarchische* Regeln keine Daten verändern. Um Daten zu verändern, müssen innerhalb der *hierarchischen* Regeln Graphregeln aufgerufen werden. *Block* und *ForBlock* unterscheiden sich nur in der Art, in der die Datenpakete bearbeitet werden. Bei Regeln, die in einer *Block*-Regel angeordnet sind, funktioniert die Ausführung folgendermaßen: Alle hereinkommenden Pakete werden zusammen an die erste Regel weitergeleitet und dort zusammen bearbeitet. Die Ausgaben der Regel werden an die OUT-Ports oder an eine andere interne Regel weitergeleitet. Die OUT-Ports dürfen nur mit einer anderen Regel verbunden sein. Sind in der Regel die IN-Ports direkt mit den OUT-Ports oder internen Regeln verbunden, dann werden die Pakete erst an die OUT-Ports übergeben, nachdem alle internen Regeln ausgewertet sind. Die Startregel aus Abbildung 21 ist eine *Block-Regel*. Dies ist anhand der Abbildung nicht zu erkennen. Ob eine Regel eine *Block* oder *ForBlock* Regel ist, ist nur in der übergeordneten aufrufenden Regel zu sehen. Die letzte Regel *Bearbeite Studenten*, die aufgerufen wird, hingegen ist eine *ForBlock-Regel*. Dies ist in GReAT daran erkennbar, dass die Regel mit einem lila Rahmen dargestellt ist.

### 4.3.2 ForBlock Regeln

Die *ForBlock*-Regeln verhalten sich dahingehend anders, dass sie nicht alle Pakete auf einmal an die erste Regel übergeben, sondern immer nur einzelne Pakete. Erst wenn die komplette Bearbeitung des ersten Paketes innerhalb der *ForBlock*-Regel abgeschlossen ist und die generierten Ausgabepakete erstellt, aber noch nicht freigegeben sind, wird das nächste Paket bearbeitet. Freigegeben werden die Pakete erst, wenn alle Pakete, mit denen die *ForBlock*-Regel aufgerufen worden ist, auch bearbeitet sind. Der Unterschied zu den *Block*-Regeln besteht darin, dass die Datenpakete nicht auf einmal abgearbeitet werden, sondern nacheinander. Bei der Studententransformation soll für jeden Studenten einzeln geprüft werden, ob er ein Diplom besitzt oder nicht. Für welchen konkreten Fall *Block* und *ForBlock* einen Unterschied in der Bearbeitung von Daten machen, wurde in der Literatur nicht beschrieben [1][4].

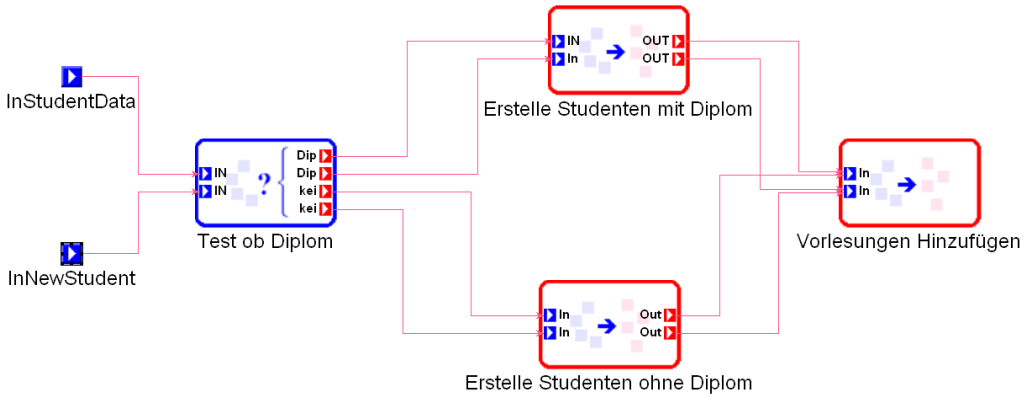


Abbildung 22: ForBlock Regel Bearbeite Studenten

Die Regel *Bearbeite Studenten* aus dem Beispiel ist in Abbildung 22 dargestellt. In dieser Regel wird zuerst mit Hilfe der Entscheidungsregel *Teste ob Diplom existiert* geprüft, ob der übergebene Student ein Diplom besitzt oder nicht. Da die Regel *Teste ob Diplom existiert*, so definiert ist, dass immer nur ein Fall gültig ist, wird entweder die Regel *Erstelle Studenten mit Diplom* oder *Erstelle Studenten ohne Diplom* aufgerufen. Diese Regeln legen entweder ein *StudentOhneDiplom* oder ein *StudentMitDiplom* Objekt an und fügen es in die neuen Daten ein. Die letzte Regel *Vorlesungen Hinzufügen* bearbeitet für jede Vorlesung, die der Student besucht hat, das neue Studenten Objekt. Diese Regel ist in Abbildung 23 dargestellt.
Die Regel *Vorlesungen Hinzufügen* aus Abbildung 23 besitzt keine OUT-Ports, da sie die letzte Regel ist, die für die Transformation der Studentendaten notwen-

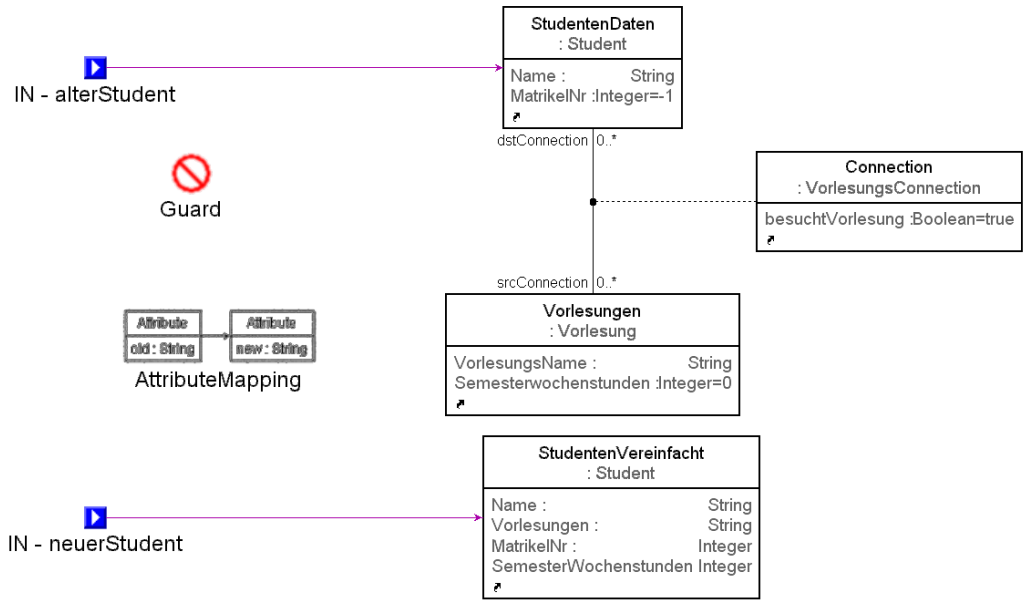


Abbildung 23: Regel zum Hinzufügen der Vorlesungen

dig ist. Sie ist in der Transformation der Studentendaten dafür zuständig, dass die Vorlesungen, die der Student besucht hat, zusammengefasst werden. Für jede Anwendungsstelle, die mit dem Muster gefunden wird, führt die Regel die *Attributauswertung* aus, um den Namen und die Semesterwochenstunden in das *StudentenVereinfacht* Objekt hinzuzufügen. Diese Regel wird sowohl für Studenten mit und ohne Diplom ausgeführt. Dies ist möglich, da das Objekt *StudentenVereinfacht* vom Typ *Student* ist, welcher die Oberklasse der beiden Typen *StudentOhneDiplom* und *StudentMitDiplom* darstellt.

### 4.3.3 Nicht-Determinismus von Regeln

In den *hierarchischen* Regeln ist es möglich, Regeln zu definieren, bei denen die IN-Ports mit mehreren anderen Regeln verbunden sind. Dies wird benötigt, falls 2 verschiedene Teile der Transformation mit denselben Datenpaketen aufgerufen werden sollen. Eine solche Notwendigkeit existiert in der Beispieltransformation nicht. Daher ist in Abbildung 24 beispielhaft eine solche Regel angegeben. In der Abbildung existieren zwei parallele Regeln *Lösche Daten* und *Erzeuge Daten*, die beide mit den IN-Ports verbunden sind. In diesem Fall ist es nicht möglich, die Reihenfolge, in der die beiden Regeln ausgeführt werden sollen, zu beeinflussen. Die Kontrollstrukturen werden zwar sequenziell abgearbeitet, jedoch für den Fall, dass mehrere Regeln mit denselben Ports verbunden sind, können die Re-

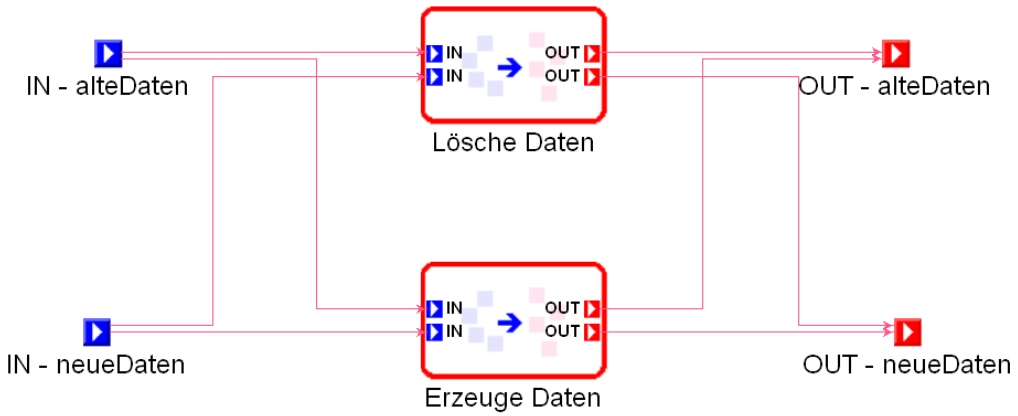


Abbildung 24: Bsp. Block-Regel mit parallen Regeln

geln nicht sequenziell abgearbeitet werden. Die Entscheidung, welche der Regeln zuerst ausgeführt wird, wird für den Fall, nicht-deterministisch getroffen. Nachdem die erste der parallelen Regeln ausgeführt wurde, werden erst die mit den OUT-Ports verbundene Regel und die darauffolgenden ausgeführt. Erst wenn die Bearbeitung der nachfolgenden Regeln komplett durchgeführt wurde, wird nicht-deterministisch die nächste parallele Regel ausgeführt.

### 4.3.4 Rekursion

Durch den modularen Aufbau der Regeln ist es möglich, Rekursionen zur Transformation zu nutzen. Eine rekursive Regel existiert nicht in der Beispieltransformation und ist daher nur beispielhaft angegeben. Bei der in Abbildung 25 dargestellten rekursiven Regel wird der rekursive Aufruf durch das *Rekursiver Aufruf* Objekt erreicht. Dass es sich bei dem Objekt um einen Rekursionsaufruf handelt, ist zusätzlich durch einen Pfeil in der Darstellung der Regel zu erkennen. Der Rekursionsabbruch wird durch eine Entscheidungsregel bestimmt. Es ist auch möglich, dass eine Regel sich nicht selbst aufruft, sondern von einer anderen Regel aufgerufen wird (indirekte Rekursion).

# 5 GReAT Laufzeitumgebung

Die Sprache GReAT wurde mit dem visuellen Framework GME (Generic Modelling Environment) implementiert [3]. GME bietet Möglichkeiten verschiedene

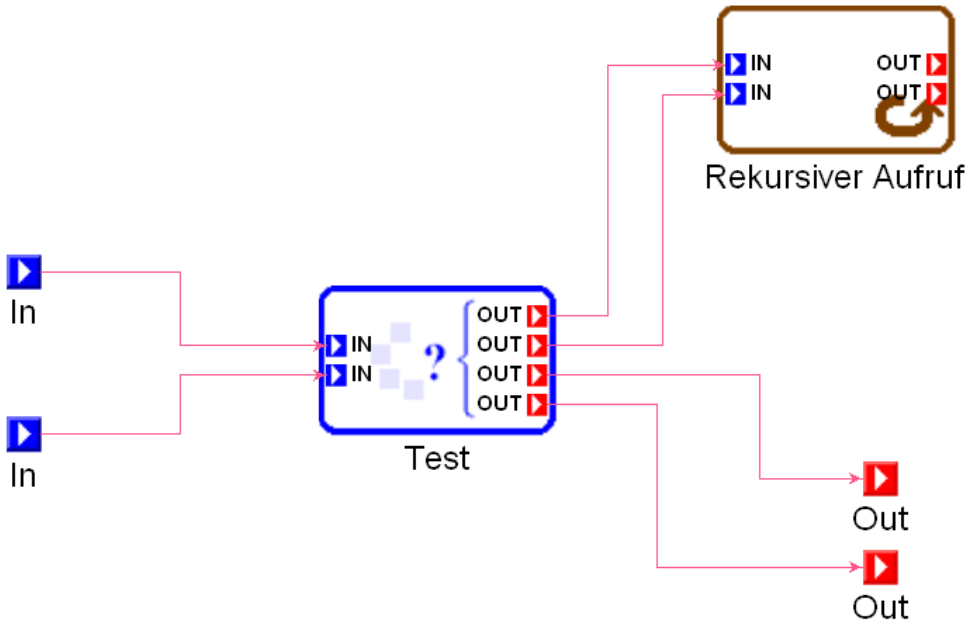


Abbildung 25: Block mit rekursiven Aufruf

Modelle visuell zu definieren und zu implementieren. GReAT ist ein Plugin für GME. GReAT macht GME zu einer visuellen Modellierungsumgebung zur Erstellung und Anpassung von Transformationen.
GReAT wurde mit Hilfe des Universal Data Modell (UDM) Paketes entwickelt. UDM ist eine Metaprogrammiersprache, die einen Entwicklungsprozess und Werkzeuge zur Erstellung von C++ Code bereitstellt. Mittels UDM werden aus den für die Transformation benötigten Modellen C++ Schnittstellen erstellt. Sie werden von GReAT benutzt, um auf die Daten während der Transformation zuzugreifen. Die erstellten C++ Schnittstellen basieren auf der Definition der Klassen im Modell. Über die durch UDM erstellten Schnittstellen ist es möglich, auf Daten in verschiedenen Speicherorten zuzugreifen. Die Daten können zum Beispiel als XML, in einer Datenbank oder als GME Datenmodell vorliegen. Da der Zugriff auf die Daten von UDM geregelt wird, existiert nur eine Schnittstelle, die unabhängig vom Speicherort benutzt werden kann. Außer den Domänen-abhängigen Schnittstellen existiert noch ein weiteres, das die Manipulation von Objekten erlaubt. Dadurch ist es möglich, Modelle eines bestimmten Modells zu bearbeiten oder zu erstellen.
Die Transformation der Daten wird mittels des GReAT Interpreters durchgeführt. Die Architektur des GReAT Interpreters 'GReAT Runtime Environment' (GRE)

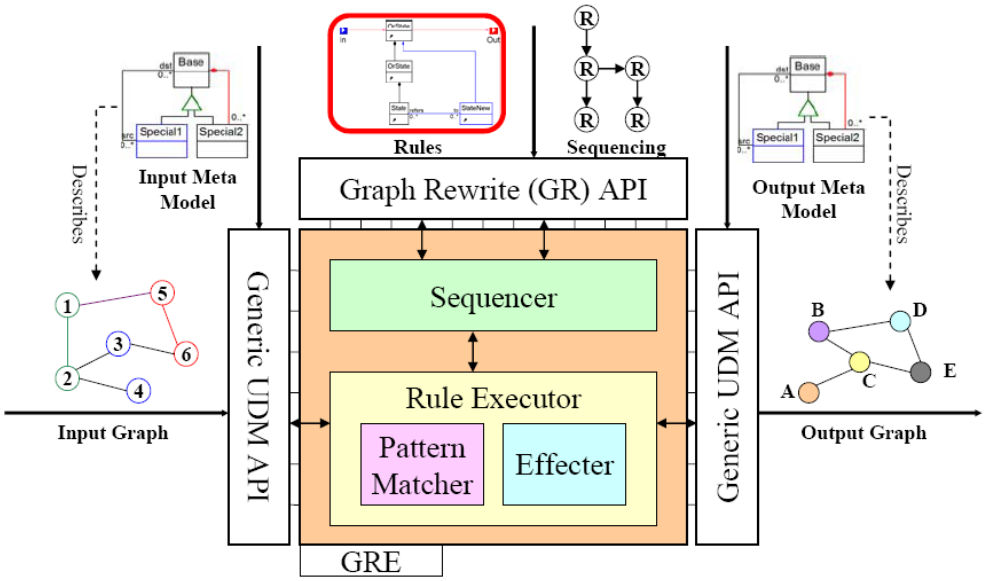


Abbildung 26: Der GReAT Interpreter

ist in Abbildung 26 grob dargestellt. Der Interpreter greift auf das Quell- und das Zielmodell mittels der erstellten UDM Schnittstellen zu. Da der Interpreter nicht direkt auf die Modelle zugreift, muss er auch nicht wissen, wie und wo die Modelle gespeichert sind. Er muss lediglich wissen, wie die Modelle aussehen und wie auf diese zugegriffen wird. In der Studententransformation wurde das *Input Meta Model* in Abbildung 1 definiert, der *Input Graph*, welcher der Struktur des Modells entspricht, ist in Abbildung 3 zu sehen. Passend zu dem Modell wird eine UDM Schnittstelle erstellt, die es dem GReAT Interpreter ermöglicht, auf die Daten im *Input Graph* zuzugreifen. Analoges gilt für das Modell der Zieldomäne aus Abbildung 2.
Der in der Abbildung zu sehende *Sequencing* Graph ist im Beispiel die Reihenfolge, in der die einzelnen Regeln abgearbeitet werden sollen. Diese Reihenfolge wird durch die verwendeten Block-, ForBlock- und Entscheidungsregeln bestimmt. Die Regeln, die für die Transformation zuständig sind, werden in der Abbildung durch das *Rules* Element dargestellt.
Der GReAT Interpreter ist in zwei Teile aufgeteilt, in den *Sequencer* und in den *Rule Executor*. Der *Sequencer* entscheidet, in welcher Reihenfolge die Regeln abgearbeitet werden sollen und ruft mit der zu bearbeitenden Regel den *Rule Executor* auf. Letzterer unterteilt sich wiederum in den *Pattern Matcher* und den *Effector*. Dieser benutzt den *Pattern Matcher*, um mittels des in der Regel definierten Musters und dem initialen Kontext, Anwendungsstellen zu finden. Sind diese

Anwendungsstellen gefunden und alle Vorbedingungen erfüllt, wird der *Effector* aufgerufen, um die Transformation durchzuführen. Wenn der *Rule Executor* mit der Ausführung fertig ist, entscheidet der *Sequencer*, welche Regel als nächstes ausgewertet werden soll.
Mit Hilfe von GME wurde ein Modellierungswerkzeug realisiert. Das Modellierungswerkzeug wird in Kapitel 5.1 beschrieben und dient dazu, die Transformation visuell zu definieren. Einige Werkzeuge von GReAT wurden ohne GME realisiert [5]. Damit diese jedoch aus dem Modellierungswerkzeug heraus aufgerufen werden können, wurden sie als Interpreter in GME implementiert. Eines dieser Werkzeuge ist die Ausführungseinheit, die in Kapitel 5.2 beschrieben wird. Sie wird benutzt, um die GReAT Transformation durchzuführen. Ein weiteres Werkzeug, die Quellcode Generierung, wird in Kapitel 5.3 beschrieben. Mit ihr ist es möglich, Quellcode zu erstellen, um Transformationen ohne GME durchzuführen. Als letztes Werkzeug wird in Kapitel 5.4 der GReAT Debugger näher beschrieben. Der Debugger ermöglicht es Transformationen genauer zu betrachten.

## 5.1 Modellierungswerkzeug

Das als Modellierungswerkzeug verwendete GME wird durch GReAT zu einer visuellen Entwicklungsumgebung für Graphtransformationen. Jede Transformation in GReAT benötigt eine Konfiguration. Die Konfiguration der Studentendaten Transformation ist in Abbildung 27 dargestellt.
Die Konfiguration definiert, wo die Daten gespeichert sind, die für die Transformation benutzt werden sollen. Die Daten der Startdomäne sowie die Daten der Zieldomäne benötigen jeweils ein eigenes *File* und *File Type* Objekt. Im *File* Objekt wird definiert, wo die Datendatei gespeichert ist bzw. gespeichert werden soll. Die *File Type* Objekte werden mit den IN-Ports der Startregel verbunden. Die Startregel der Studentendatentransformation ist in Abbildung 21 dargestellt. Hierdurch ist gewährleistet, dass das richtige Modell mit dem richtigen IN Port verbunden ist.
Die Abbildung 27 zeigt ein Bild der visuellen Entwicklungsumgebung in GME [2], in der die Konfiguration der Studententransformation zu sehen ist. Die Oberfläche vom GME hat für den Editor zur Erstellung der Modelle diesselben Aufteilungen. In der Mitte rechts befinden sich in einer Baumdarstellung alle Elemente, die innerhalb der Studententransformation erstellt wurden. Im großen Bereich in der Mitte werden alle Elemente (Graphregeln, Block Regeln, ForBlock Regeln, Konfiguration) editiert. Dieser Bereich wird als Editor bezeichnet. Welche Elemente innerhalb des gerade bearbeiteten Objektes hinzugefügt werden können, kann im *Part Browser* links unten gesehen werden. Wird ein neues Objekt benötigt, kann es per Drag & Drop aus dem *Part Browser* an die gewünschte Stelle

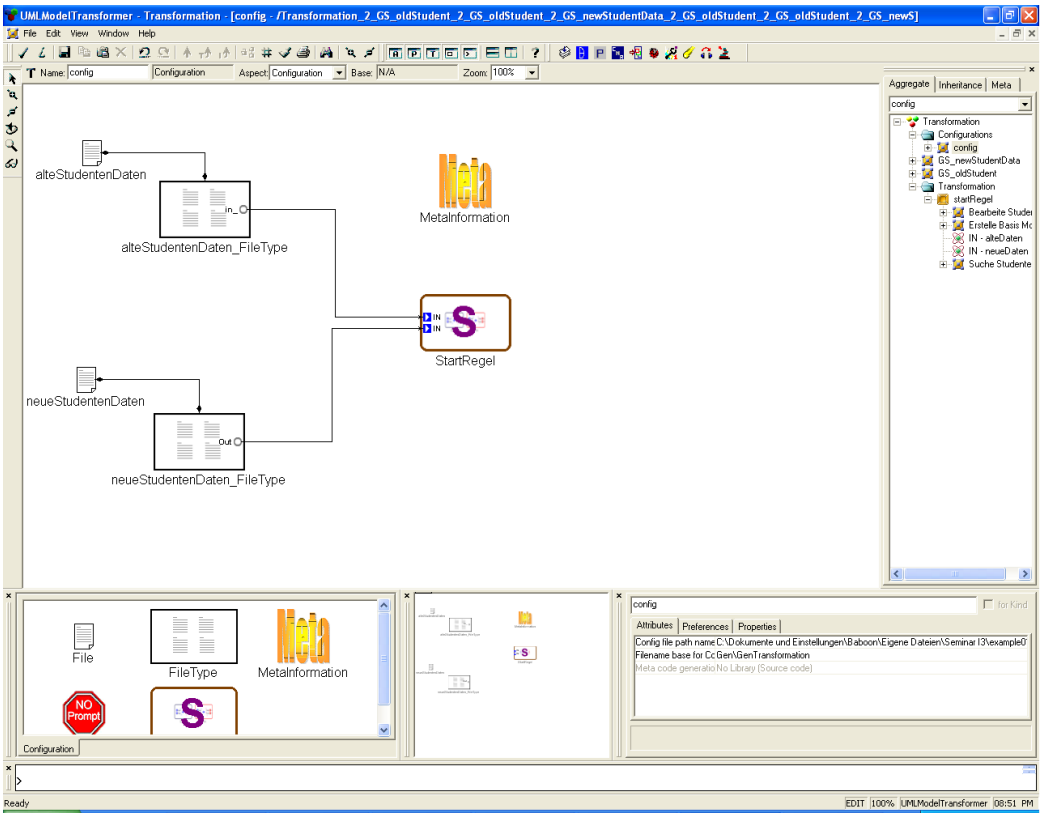


Abbildung 27: GReAT Projekt Konfiguration

gezogen werden. Rechts neben dem *Part Browser* befindet sich eine verkleinerte Darstellung des bearbeiteten Elements, das im Editor angezeigt wird. Dies ist besonders bei großen Regeln eine Hilfe, da erkennbar ist, welcher Bereich der Regel gerade im Editor betrachtet wird. Wiederum rechts daneben werden die Eigenschaften der Elemente im Editor und des *Part Browsers* dargestellt. In der Abbildung sind die Eigenschaften des *config* Elementes im *Part Browser* dargestellt. Alle Attribute des Elementes können hier angepasst werden. Unten in der Abbildung ist eine Zeile sichtbar, welche für Protokolle und Fehlermeldungen benutzt wird.

## 5.2 Ausführungseinheit

Die Ausführungseinheit führt die Transformation in drei Schritten durch. Die Reihenfolge der Schritte ist in Abbildung 28 dargestellt. Jeder der Schritte stellt ein eigenes Werkzeug dar, das automatisiert hintereinander aufgerufen wird. Als er-

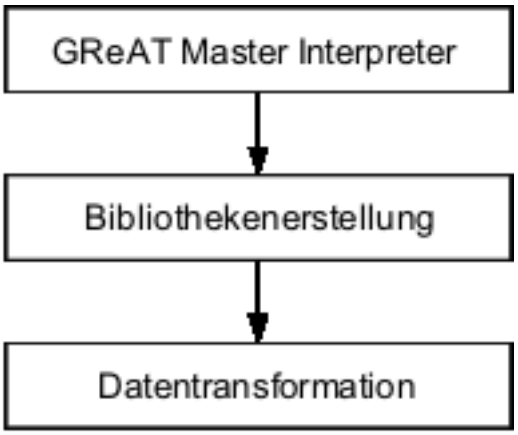


Abbildung 28: Ablauf der Ausführungseinheit

stes wird ein Werkzeug mit dem Namen GReAT Master Interpreter aufgerufen. Er ist dafür zuständig, die GReAT Konfiguration zu exportieren und benötigte Konfigurationen für die UDM Schnittstellen zu erstellen. Die fertigen Schnittstellen werden im zweiten Schritt erstellt. Die exportierte GReAT Konfiguration beinhaltet alle Informationen aus den in Abbildung 27 dargestellten Konfigurationswerten in einer für die Bibliothekenerstellung benötigten Form. Ohne diese exportierte Konfiguration ist es nicht möglich, die Transformation durchzuführen oder ein anderes Werkzeug aufzurufen. Hat sich etwas an der Konfiguration verändert, muss der GReAT Master Interpreter als eigenständiges Werkzeug erneut aufgerufen werden. Ansonsten werden die Änderungen nicht für die Transformation oder ein anderes Werkzeug benutzt. Die Beispieltransformation nutzt die in Abbildung 3 dargestellten Daten als Daten der Startdomäne.
Zum Ausführen einer Transformation aus GME heraus wird der Graphtransformationsinterpreter aufgerufen. Dieser Interpreter führt die eigentliche Transformation durch. Vor dem Start der Bibliothekenerstellung erscheint ein Fenster (Abbildung 29), in dem es möglich ist, die konfigurierten Basisdaten und Zieldatendateien auszutauschen. Wird der Interpreter zum ersten Mal oder nach Ausführen des *Master Interpreters* aufgerufen, werden zunächst die für die Transformation benötigten Bibliotheken erstellt. Diese Generierung geschieht automatisch. Sollten in den definierten *Wächtern* oder *Attributauswertungen* Syntaxfehler existieren, funktioniert die Generierung nicht und die Transformation wird nicht ausgeführt. Nach erfolgreicher Generierung der Bibliotheken startet die eigentliche Transformation der Daten. Die Beendung der Transformation wird durch das Erscheinen einer Meldung signalisiert. Die Meldung beinhaltet den Pfad, an dem die fertig transformierten Daten abgelegt wurden. Das Transformationsergebnis der Studentendaten ist in Abbildung 30 dargestellt. Es wurde ein *StudentMitDiplom* und ein

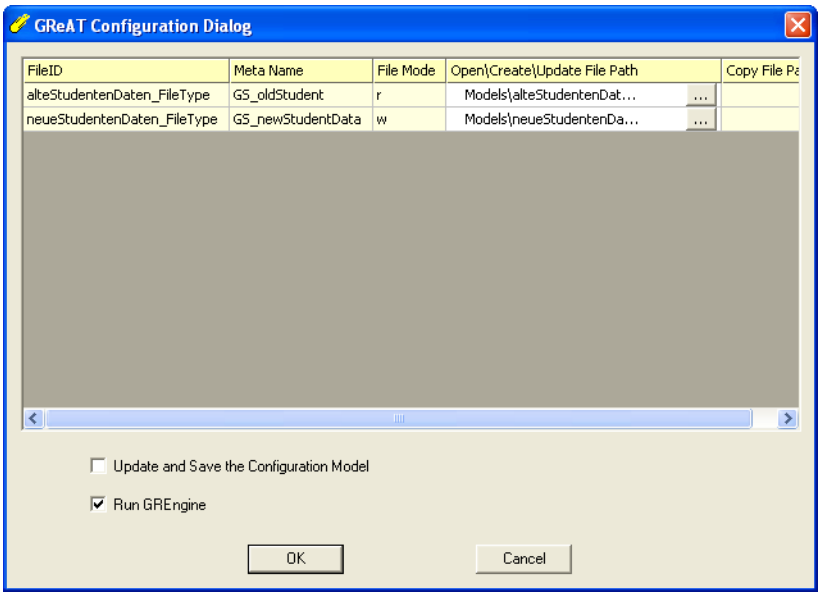


Abbildung 29: GReAT Konfigurations Dialog

*StudentOhneDiplom* Objekt erstellt. Die Attribute beider Objekte wurden gemäß der Transformation mit den Daten gefüllt. Exemplarisch sind in der Abbildung die Daten des *StudentOhneDiplom* Objekts dargestellt. Es ist mit der Ausführungseinheit nicht möglich, die Transformation schrittweise zu betrachten. Hierzu muss die Transformation mittels des Debuggers durchgeführt werden. Der Debugger ist in Kapitel 5.4 beschrieben.

## 5.3 Quellcode Generierung

Nachdem eine Transformation mittels GReAT im GME erstellt wurde, ist es möglich, zu der erstellten Transformation Quellcode zu generieren. Dies kann mittels des *Code Generator* Interpreters geschehen. Dieser benötigt als Vorbedingung eine fertig generierte GReAT Konfiguration. Ist diese nicht vorhanden, muss sie noch erstellt werden. Der Code Generator erstellt ein C++ Visual Studio Project mit allen nötigen Dateien, um eine ausführbare Datei zu generieren. Abbildung 31 zeigt das Visual Studio Projekt, das mittels des Code Generators aus der Beispieltransformation erstellt wurde. Quellcode in anderen Sprachen als C++ kann nicht generiert werden.

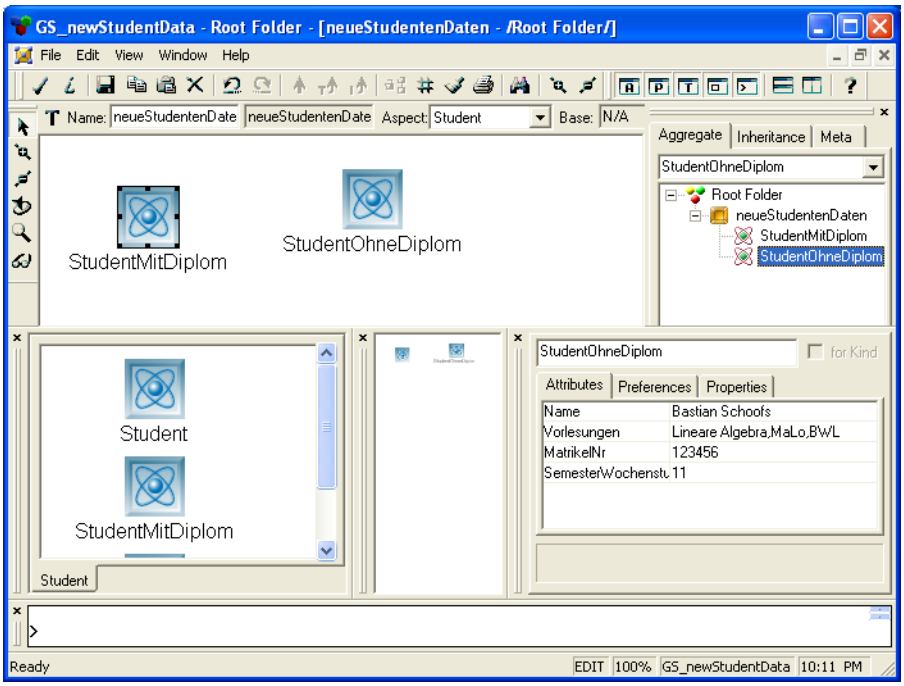


Abbildung 30: Ergebnis nach der Transformation

## 5.4 Debugger

Die Möglichkeit eine Transformation im GME zu debuggen, wurde durch eine zusätzliche Applikation bereitgestellt, die aus dem GME gestartet werden kann. Mit Hilfe des Debuggers können bei einer Transformation die Werte der IN und OUT Ports, sowie die gefundenen Anwendungsstellen überwacht werden. Der Debugger bietet zum jetzigen Zeitpunkt die Möglichkeit, Haltepunkte zu setzen. Wurde die Transformation an einem Haltepunkt angehalten, kann die Transformation schrittweise oder bis zum nächsten Haltepunkt fortgeführt werden. Abbildung 32 zeigt den GReAT Debugger beim Debuggen der Beispieltransformation. In der Abbildung sind alle möglichen Stellen für Haltepunkte zu erkennen. Zum Ende der *Suche Studenten* Regel und zu Beginn der *Erstelle Studenten ohne Diplom* Regel ist jeweils ein Haltepunkt markiert. Die gesetzten Haltepunkte sind durch rote Quadrate in der Abbildung markiert.
Während des Debuggens ist es nicht möglich, in den Ablauf einzugreifen, um Regelausführungen zu überspringen. Es ist auch nicht möglich, währenddessen Muster oder Parameter zu verändern, um durch ihren Inhalt Entscheidungsregeln oder die Anwendungsstellensuche zu beeinflussen.
Der Debugger startet immer mit den Daten die in der GReAT Konfiguration definiert sind. Soll die Datendatei, geändert werden, muss der Debugger geschlossen,

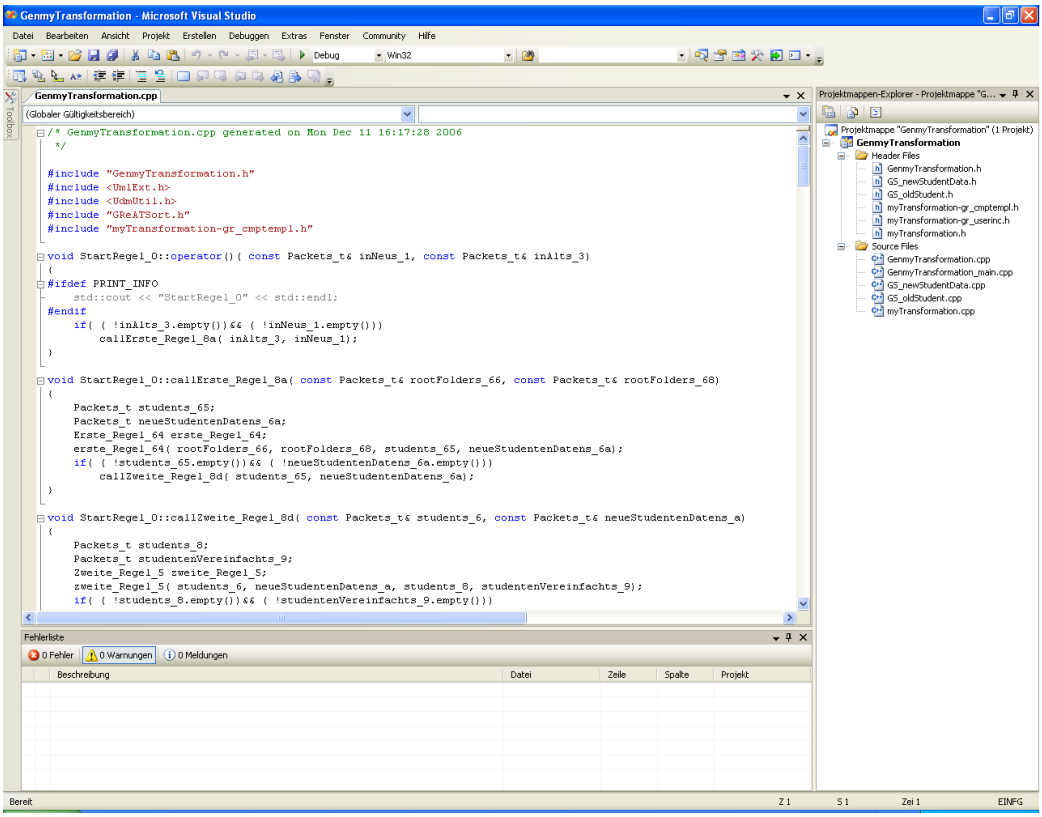


Abbildung 31: Generiertes Visual Studio Projekt

die Konfiguration angepasst und danach der Debugger neu gestartet werden. Eine ähnliche Möglichkeit die Datendateien vor dem Start des Debuggers zu wechseln, wie es die Ausführungseinheit bietet, existiert nicht.

# 6 Fazit

GReAT ist eine Sprache zur visuellen Definition von Graphtransformationen. Diese Transformationen werden benutzt, um Daten, die in einer definierten Domäne vorliegen, in Daten einer anderen Domäne umzuwandeln. Transformationen, die in GReAT erstellt sind, lassen sich durch ihren modularen Aufbau leicht verstehen und anpassen. Da die Transformation auf vielen kleinen Regeln basiert, die sich nahezu beliebig kapseln lassen, ist es möglich, die Transformation so aufzubauen, dass auch Außenstehende sich zurechtfinden und kleine Anpassungen vornehmen können. Ein großer Vorteil von GReAT ist der eingebaute Code Generator. Der durch den Code Generator erstellte C++ Code kann dazu benutzt

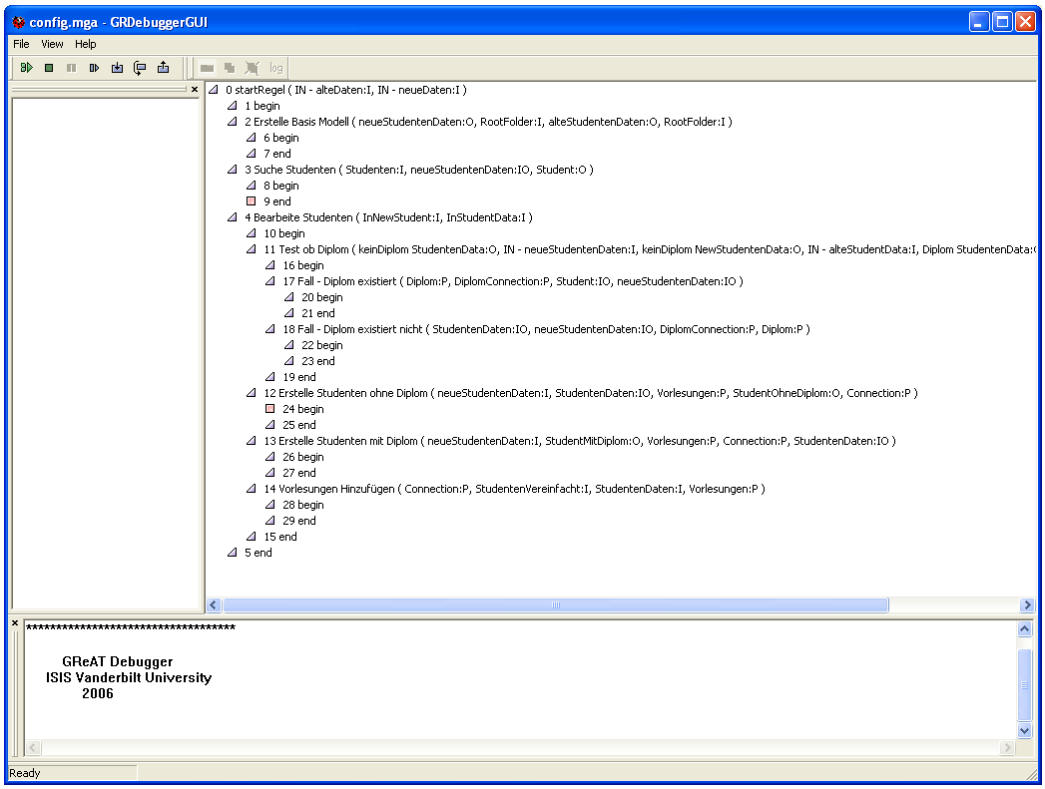


Abbildung 32: GReAT Debugger

werden, eine ausführbare Datei zu erstellen. Mittels dieser ist es möglich, eine Transformation ohne Benutzung des visuellen Editors durchzuführen. Es ist auch denkbar, den Code in vorhandene Applikationen einzubinden oder die ausführbare Datei während eines Softwareerstellungsprozesses wiederzuverwenden. Der vorhandene Code Generator macht die Ausführung von Transformationen auch ohne GReAT möglich.
Der Debugger ist bei der Fehlersuche von Vorteil, denn es kann zum Beispiel überprüft werden, warum eine Transformation an einer bestimmten Stelle abbricht oder warum bestimmte Regeln nicht ausgeführt werden. Jedoch ist es nicht möglich, Objekte der Transformation im Debugger zu verändern.
Zusammenfassend kann gesagt werden, dass GReAT ein interessanter Ansatz zur Graphtransformation ist. Die Definition von eigenen Modellen und ist nicht besonders intuitiv, jedoch nach kurzer Einarbeitungszeit leicht zu bewerkstelligen. Bei GReAT besteht, wie bei den meisten Applikationen, noch Verbesserungspotenzial. Eine Syntaxüberprüfung für die verwendeten Codeteile in den Wächtern und der Attributauswertung wäre hilfreich bei der Entwicklung. Da diese nicht vorhanden ist, passiert es häufig, dass das Kompilieren nicht funktioniert.

Der Debugger ist nur bedingt hilfreich, da er für einige Problemstellungen nicht genug Informationen bereithält. Die Auswertungen von Wächtern und Attributauswertungen werden im Debugger nicht dargestellt und lassen sich daher auch nicht betrachten. Ebenso wenig kann im Debugger erkannt werden, warum eine Regel trotz übergebener Objekte nicht ausgeführt wird. Dies ist der Fall, wenn die übergebenen Objekte nicht mit den erwarteten Objekten an den IN Ports übereinstimmen.
Ob eine Transformation mittels GReAT einen Geschwindigkeitsvorteil gegenüber selbst erstellten Transformationen bietet, kann ohne weitergehende Tests nicht gesagt werden. Gegenüber Transformationen mittels XSLT bietet GReAT den Vorteil, dass die Regeln von GReAT durch den visuellen Editor und den modularen Aufbau leichter verständlich sind. Außenstehende haben es daher einfacher vorhandene Transformationen, die in GReAT definiert sind, zu modifizieren als Transformationen, die mittels XSLT definiert sind.

# Literatur

[1] AGRAWAL, ADITYA: *A Formal Graph Transformation based language for Model-to-Model Transformations*. Visionenpapier, Vanderbilt University, 2004.

[2] AGRAWAL, ADITYA, ZSOLT KALMAR, GABOR KARSAI, FENG SHI und ATTILA VIZHANYO: *GReAT User Manual*. Institut für integrierte Softwaresysteme an der Vanderbilt University, 2003. `http://www.escherinstitute.org/Plone/tools/suites/mic/great/GReAT%20User%20Manual.pdf`.

[3] AGRAWAL, ADITYA, GABOR KARSAI und AKOS LEDECZI: *An end-to-end domain-driven software development framework*. In: *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Seiten 8–15, New York, NY, USA, 2003. ACM Press.

[4] AGRAWAL A., KARSAI G., SHI F.: *Graph Transformations on Domain-Specific Models*. Technischer Bericht, Institut für integrierte Softwaresysteme an der Vanderbilt University, 2003. ISIS-03-403.

[5] INSTITUT FÜR INTEGRIERTE SOFTWARESYSTEME AN DER VANDERBILT UNIVERSITY: *GReAT: Graph Rewriting and Transformations Model Transformation Environment for Model-Based of Systems1 Summary of*

*Features*, 2006. `http://www.escherinstitute.org/Plone/tools/suites/mic/great/GReAT-Sum.pdf`.

[6] INSTITUT FÜR INTEGRIERTE SOFTWARESYSTEME AN DER VANDERBILT UNIVERSITY: *Step-by-step GReAT Guide*, 2006. `http://www.escherinstitute.org/Plone/tools/suites/mic/great/Step-by-step%20GReAT%20Guide.pdf`.

[7] KARSAI, GABOR, ADITYA AGRAWAL, FENG SHI und JONATHAN SPRINKLE: *On the Use of Graph Transformation in the Formal Specification of Model Interpreters*. Journal of Universal Computer Science, Special issue on Formal Specification of computer-based systems, 9(11), 2003.

[8] LEDECZI, AKOS, MIKLOS MAROTI, ARPAD BAKAY, GABOR KARSAI, JASON GARRETT, CHARLES THOMASON, GREG NORDSTROM, JONATHAN SPRINKLE und PETER VOLGYESI: *The Generic Modeling Environment*. In: *Workshop on Intelligent Signal Processing*, Budapest, Hungary, 2001. IEEE.

[9] VIZHANYO, ATTILA, GABOR KARSAI, ADITYA AGRAWAL, FENG SHI, ZSOLT KALMAR und ANANTHA NARAYANAN: *Reusable Idioms and Patterns in Graph Transformation Languages*. In: *Workshop OOPSLA/GPCE: Best Practices for Model-Driven Software Development*, 2004.