Diplomarbeit

Entwurf und Implementierung eines Interpreters für amalgamierte Graphtransformationen

Bożena Karwan

Technische Universität Berlin
Fachbereich IV, Informatik
Institut für Software und Theoretische Informatik
Forschungsgruppe Formale Spezifikation

Betreuer:

Prof. Hartmut Ehrig
Dr. Gabriele Taentzer

Inhaltsverzeichnis

1. Einleitung

1.1 Ziel und Motivation

Das Hauptziel der Diplomarbeit ist die praktische Umsetzung eines Konzeptes für die parallelen Graphtransformationen und die Erweiterung des AGG-Systems für diesen Zweck. AGG ist ein Spezifikations- und Programmierwerkzeug, das auf den attributierten Graphgrammatiken basiert. Die graphische Darstellung von komplexen Softwaresystemen in der Form eines Graphen stellt ein mächtiges und ausdrucksreiches Werkzeug der Modellierung dar. Es gibt verschiedene Ansätze in diesem Gebiet. Die Aufgabe der Diplomarbeit umfasst die Spezifikation und Implementierung einer theoretischen Idee, beschrieben von Dr. Gabriele Taentzer in ihrer Dissertation [Taenz96]. Es handelt sich hier um einen der oben genannten Ansätze, um amalgamierte Graphtransformation.

Diese Art der Graphersetzung erlaubt die Ableitung mehrerer Produktionen einer Unterregel und mehrerer Erweiterungsregeln in der Form einer *amalgamierten* Regel. Für die Unterregel wird nur einmal ein Ansatz gefunden, wobei die Erweiterungsregeln so oft wie möglich, in der Abhängigkeit vom Kontext, abgeleitet werden können. Anders ausgedrückt ist die Amalgamierung eine sequentielle Graphtransformation im Bezug auf den variablen Kontext. Im Rahmen der Diplomarbeit wurde die amalgamierte Graphersetzung auf den attributierten Graphen mit der Verwendung von den negativen Anwendungsbedingungen (*NACs*) im *Single Pushout*–Konzept betrachtet. Diese Variante der Graphtransformation ist allgemeiner als *Doppelt Pushout*. Die umfangreiche Beschreibung zu diesem Thema ist im [Löw93] zu finden.

Die amalgamierten Graphersetzungskonzepte können eine Einsetzung in der Modellierung komplexer Operationen finden. Ein plausibles Beispiel dafür ist das *Refactoring*-Konzept. Die *Refactorings* sind Programmtransformationen welche die Softwarestruktur der objektorientierten Softwaresysteme verbessern, während das externe Verhalten erhalten bleibt. Dieser Aspekt wurde im [MeTaRu04] genauer beschrieben.

1.2 Strukturierung der Arbeit

Der folgende Überblick über einzelne Kapiteln beschreibt genauer, was der Leser in der schriftlichen Ausarbeitung der Diplomarbeit zu erwarten hat.

In ***Kapitel 2*** werden die Grundbegriffe der Graphtransformationen vorgestellt. Im ersten Abschnitt wird beschrieben, was unter den allgemeinen Begriffen wie zum Beispiel Graph, Regel und Graphmorphismus verstanden werden soll. Außerdem wird das Konzept der einfachen Graphtransformationen vorgestellt. Dagegen wird in dem zweiten Unterkapitel auf die Konzepte der amalgamierten Graphtransformationen eingegangen. Wie schon erwähnt stellt eine amalgamierte Transformation eine Möglichkeit der parallelen Ausführung mehrerer Regeln auf einem Graphen dar. Es wird erläutert wie die Grundkonstrukte wie Regelschema und Instanzschemata, welche die Regeln beschreiben, definiert werden.

Im ***Kapitel 3*** werden die Mittel zur Modellierung des Konzeptes und zur Implementierung vorgestellt. Die *Unified Modeling Language (UML)* wird als die entsprechende Modellierungssprache gewählt, weil sie sehr ausdrucksreich ist und ein breites Spektrum an verschiedenen Modellierungstechniken anbietet. Die einzelnen Methoden, die UML bietet und in der Diplomarbeit eingesetzt wurden, werden genauer beschrieben. Die Programmiersprache *Java* wird aus zahlreichen praktischen Gründen verwendet. Ein wichtiger Punkt ist die Tatsache, dass das ganze AGG – System auch in Java implementiert wurde. Andere Vorteile der Programmiersprache werden genauer in dem Unterkapitel *3.2* vorgestellt.

Das AGG–System wird im ***Kapitel 4*** vorgestellt. Dieses System, welches ständig erweitert wird, ist eine Implementierung, die von M. Löwe angeführt wurde. Die Anfangsmotivation für die Untersuchung des *Single Pushouts*-Konzeptes [Löw93] war die Entwicklung der Strukturen für eine Implementierung der algebraischen Graphersetzung. Die graphische Oberfläche des AGG - Systems bietet viele Möglichkeiten an. Die Beschreibung im ***Kapitel 4*** bezieht sich auf die GUI–Benutzung, wie das Erstellen einer Graphgrammatik, das Editieren einer Regel und die Morphismusspezifikation. Es wird auch genauer erklärt, wie die Attribute, Ausdrücke und Anwendungsbedingungen definiert werden sollen. Anschließend

wird die Ausführung eines Transformationsschritts vorgeführt.

Im ***Kapitel 5*** werden die Anforderungen vorgestellt, die ich an das entwickelte Konzept und die Implementierung gestellt habe. Hier erfolgt die Überlegung, welche Komponenten für das Amalgamierungskonzept zur Verfügung gestellt sein sollten und wie man die Idee überhaupt realisieren könnte. Es war auch notwendig die eventuellen Erweiterungen und Anpassungen in der graphischen Oberfläche von dem AGG-System für das Konzept zu spezifizieren.

Zunächst werden die Details zum Entwurf und zur Implementierung im ***Kapitel 6*** ausführlich dargestellt. Das während der Diplomarbeit entwickelte *Package* und seine Abhängigkeiten zum *xt_basis–Package* werden beschrieben und graphisch dargestellt. Jede Klasse aus dem implementierten *Package* wird mit genauer Beschreibung und graphischen Darstellung in der *UML*-Notation versehen. Während der Modellierungsphase sind einige Probleme aufgetreten, die zu bestimmten Einschränkungen geführt haben, die auch in diesem Kapitel beschrieben wurden.

Das nächste Unterkapitel stellt die Änderungen an der graphischen Oberfläche des AGG–Systems dar, die vorgenommen werden mussten. Die neu implementierten, erweiterten und existierenden Klassen werden differenziert beschrieben. Der zweite Teil vom *Kapitel 6* erläutert die Implementierung in der Programmiersprache *Java*. Unter anderem wird erwähnt, welche Version von *Java* benutzt wurde und welche Standardbibliotheken eingesetzt wurden.

Im ***Kapitel 7*** wird vorgestellt, wie man die erweiterte Benutzerschnittstelle verwenden kann. Die üblichen Aktionen, wie zum Beispiel das Laden oder Erstellen der AGT–Graphgrammatik und ihrer Komponente werden vorgeführt. Insbesondere konzentriert sich der Abschnitt auf die Unterschiede zu der alten Version des AGG–Systems.

Im Ausblick (***Kapitel 8***) wird beschrieben, wie das umgesetzte Konzept eventuell noch erweitern werden kann und speziell, wie die entwickelten Komponenten dafür verwendet werden können. Die Lösungsansätze werden teilweise vorgestellt. Außerdem wird auf die Beziehung zu anderen Konzepten, die im AGG–System eingesetzt wurden, eingegangen.

2. Einführung in die Graphtransformationen

Im folgenden Abschnitt wird das grundlegende Konzept der Graphtransformationen eingeführt und die dazu relevanten Begriffe einzeln beschrieben. Die Graphersetzungskonzepte werden genauer im [Ehr79, EPS73] beschrieben und die attributierten Graphtransformationen werden in [HMTW94] behandelt.

2.1 Allgemeine Begriffe

Die folgenden Definitionen sind informell und sind von innerhalb der Diplomarbeit eingeführt worden, um das ganze Konzept der Graphtransformationen besser nachvollziehen zu können. Auf die formale Beschreibung verweisen die Literaturreferenzen.

2.1.1 Graph

Ein Graph ist durch zwei disjunkte Mengen definiert: eine Knotenmenge und eine Kantenmenge. Kanten und Knoten eines Graphen werden als Graphobjekte bezeichnet. Jeder Knoten und jede Kante besitzt einen Typ. In einem Graphen kann es mehrere Graphobjekte geben, die den Typ repräsentieren. Bei einem gerichteten Graphen müssen die Kanten gerichtet sein. Das bedeutet, dass sie zwischen zwei Knoten verlaufen, wobei einer davon die Quelle der Kante und der andere das Ziel ist. Es ist auch möglich, dass die Quelle und das Ziel ein und derselbe Knoten ist. In dem Fall spricht man von einer Schlinge. Quell- und Zielknoten müssen immer definiert sein. Es ist nicht möglich, eine Kante zu erstellen, die keine Quelle oder kein Ziel besitzt, eine sogenannte hängende Kante. Es sind auch leere Graphen erlaubt. In dem Fall besitzt ein Graph keine Knoten und Kanten.

Die formale Beschreibung des Graphbegriffs ist in der Veröffentlichung [Ehr,Tae] zu finden.

2.1.2 Graphmorphismus

Es sind zwei Graphen G und H gegeben. Ein Graphmorphismus f : G → H zwischen diesen Graphen ist definiert durch ein Paar von partiellen Abbildungen von Graphobjekten des Graphen G auf die Graphobjekte des Graphen H. Dabei dürfen die Kanten nicht auf die Knoten oder umgekehrt abgebildet werden. Es gibt noch weitere Einschränkungen. Die Abbildungen müssen struktur- und typverträglich sein. Eine Abbildung ist dann strukturverträglich, wenn das Ziel (Quelle) einer Kante aus dem Graphen G genau auf das Ziel (Quelle) ihres Bildes in dem Graph H abgebildet ist. Sie ist dann typverträglich, wenn die Graphobjekte auf die des gleichen Typs abgebildet sind. Ein Graphmorphismus ist injektiv (surjektiv, bijektiv), wenn die Abbildungen injektiv (surjektiv, bijektiv) sind.

2.1.3 Attributierter Graph

Ein attributierter Graph lässt sich durch eine Algebra ausdrücken. Die Spezifikation zu der Algebra besteht aus der graphischen Spezifikation, die den Graphen beschreibt, aus der Spezifikation für die Attribute und aus der Zuordnung eines Graphobjektes zu einem Ausdruck. Kurz gefasst unterscheidet sich ein attributierter Graph von einem einfachen dadurch, dass die Graphobjekte attributiert sind. Das bedeutet, dass sie Attribute besitzen, die so ähnlich wie Variablen in konventionellen Programmiersprachen definiert sind. Man spezifiziert einen Namen und einen bestimmten Typ (wie z.B. *integer, String*) für ein bestimmtes Attribut. Jedes Attribut besitzt einen Wert. Der Wert kann eine Konstante, Variable oder ein Ausdruck sein. Ein Graphobjekt kann mehrere Attribute haben.

Die *Abbildung 1* zeigt ein Beispiel eines attributierten Graphen. Er besteht aus verschiedenen Knotentypen, wie: *Class, Variable, Method* und Kantentypen, wie *contains, access, update*. Die Knoten sind mit jeweils einem Attribut versehen. Der Wert von jedem Attribut ist vom Typ *String* und ist eine Konstante.

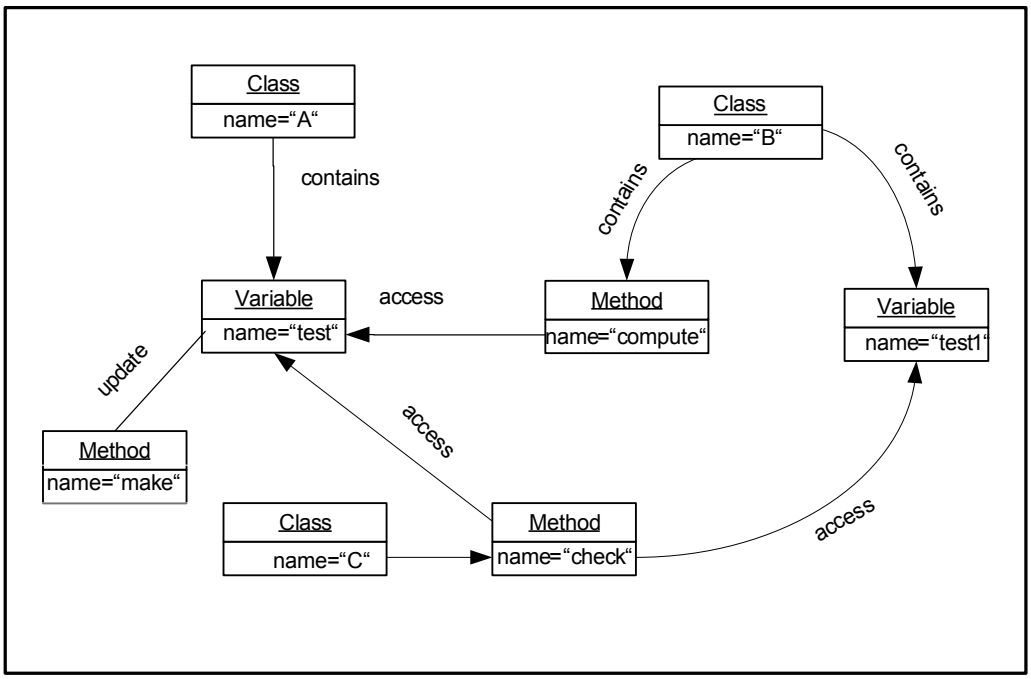

Abbildung 1. *Attributierter Graph G*

2.1.4 Graphmorphismus zwischen attributierten Graphen

Ein spezieller Begriff ist ein Graphmorphismus zwischen zwei attributierten Graphen. So ein Morphismus lässt sich mit der Hilfe des einfachen Graphmorhismus und des Homomorphismus zwischen Algebren, die jeweils den Quelle- und Zielgraphen definieren, beschreiben. Es gibt hier gewisse Unterschiede zwischen beiden Konzepten für Graphtransformationen. Im Fall des *SPO*–Ansatzes kann ein einfacher Graphmorhismus partiell sein, dagegen im *DPO*–Ansatz muss er total sein. In beiden Fällen muss der Homomorphismus zwischen den Algebren total sein.

2.1.5 Regel

Durch eine Regel wird definiert, wie ein Graph in einen neuen transformiert wird. Die hier vorgestellte Definition bezieht sich auf den *SPO*–Ansatz.

Eine Graphregel $r : L \rightarrow R$ besteht aus zwei Graphen und einem Morphismus zwischen denen. L ist der linke und R der rechte Regelgraph. Die Graphtransformation kann das Löschen, Hinzufügen, Ändern oder das Erhalten von Graphobjekten beinhalten.

Im einfachsten Fall ist eine Regel auf einen Graphen G anwendbar, wenn ihre linke Seite L ein Teilgraph von G ist, also dann, wenn es eine Abbildung auf

den Graphen G gibt, die über einen totalen, injektiven Graphmorphismus gefunden wird.

Die *Abbildung 2* zeigt eine Regel. Die zu einem *Class*-Knoten, der eine Variable enthält, zwei weitere Methoden hinzufügt.

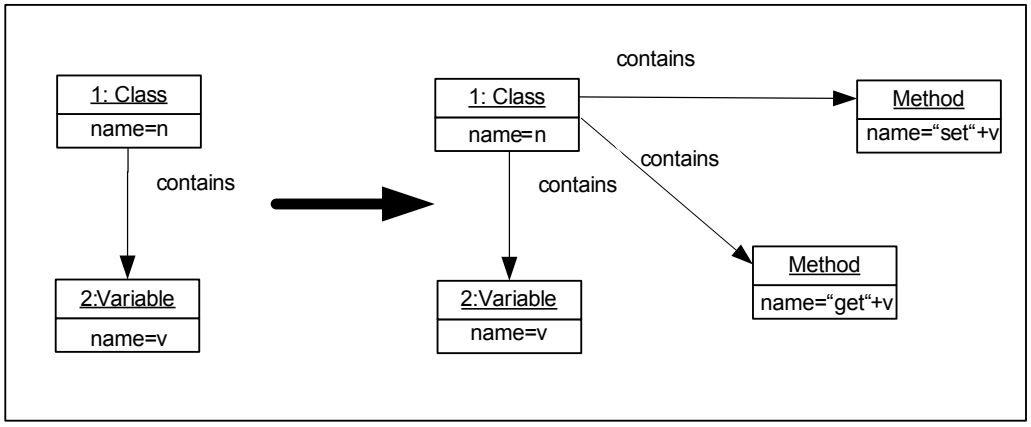

Abbildung 2. *Regel*

2.1.6 Ansatz einer Regel

Ein Ansatz einer Regel, auch ein Match genannt, identifiziert die Objekte der linken Regelseite mit den Objekten in dem Arbeitsgraphen G. Folglich beschreibt die Regel die Graphobjekte, die bei der Ableitung beteiligt werden. Bei dem Ansatz einer Regel in der *DPO*–Graphersetzung muss die Klebebedingung erfüllt werden, wobei das bei *SPO*–Graphersetzung keine Voraussetzung ist.

Man sollte noch erwähnen, dass in beiden Fällen ein Ansatz einer Regel $r : L \rightarrow R$ im Graph G ein totaler Graphmorphismus $m : L \rightarrow G$ zwischen der linken Seite der Regel und den Graphen G vorhanden sein muss.

2.1.7 NAC (Negative Application Condition)

Eine Regel kann eine oder mehrere negative Anwendungsbedingungen enthalten, welche die Situationen bezeichnen, die nicht vorkommen dürfen, damit eine Regel angewendet werden kann. Formal wird diese negative Bedingung mittels eines Graphen N und eines Graphmorphismus $l : L \rightarrow N$ beschrieben. Es gibt eine formale Definition der Semantik der NACs. Grundsätzlich besagt sie, dass eine Regel in Bezug auf einen gegebenen Match $m : L \rightarrow G$ angewendet werden kann, wenn kein totaler Morphismus $n : N \rightarrow G$

gefunden werden kann. Das heißt, dass es keinen totalen Morphismus n : N → G gibt, so dass n o l = m.

Die folgende Abbildung zeigt eine allgemeine formale Darstellung einer negativen Bedingung.

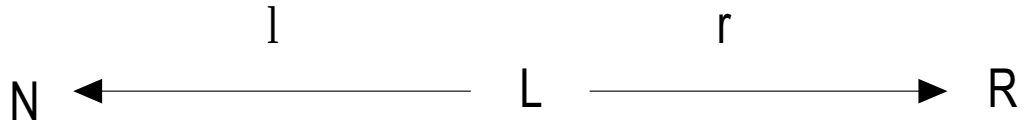

Abbildung 3. *NAC*

Die *Abbildung 4* zeigt ein konkretes Beispiel für eine Regel mit NAC. Die Zahl "1" zeigt die Abbildung des Objekts: *Class*. Die Bedingung verbietet die Anwendung der gegebenen Regel auf das Objekt von dem Typ *Class*, das mit einem Objekt des Typs *Method* in einer Beziehung steht und die Attributbedingung für den Variablenwert *s* ist wie folgt definiert: *s="set"+v .*

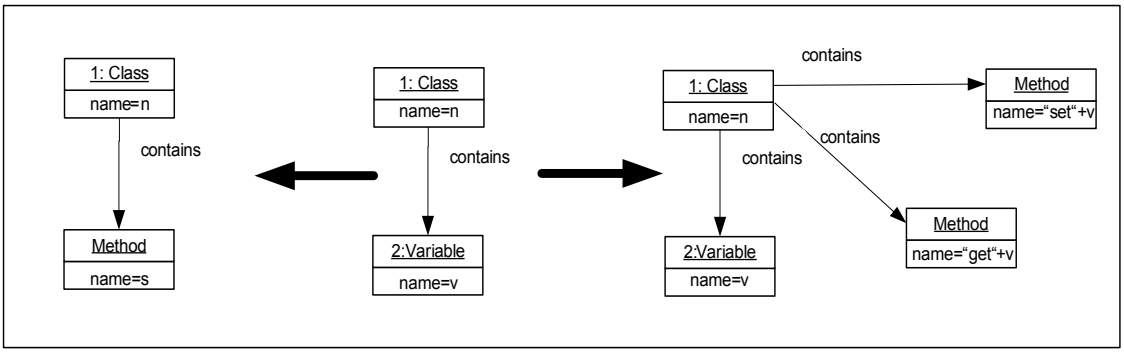

Abbildung 4. *NAC – Beispiel*

Die theoretischen Grundlagen für die Regeln mit Anwendungsbedingungen sind in [HHT96] zu finden.

2.1.8 Transformationsschritt

Das Ergebnis der Anwendung einer Graphregel r : L → R auf einem Arbeitsgraphen G an einem Match m : L → G ist ein transformierter Arbeitsgraph H mit zwei Graphmorphismen m* : R → H und r*: G → H.

Dieser Effekt wird durch einen *Single Pushout* im Bezug auf attributierte Graphen mit partiellen Graphmorphismen charakterisiert.

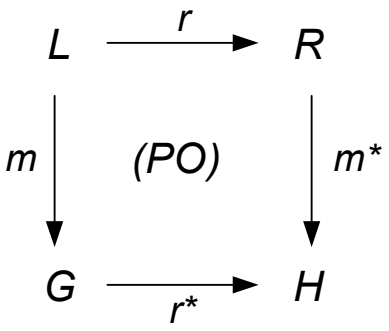

Abbildung 5. *Pushout*

Im Folgenden wird die Konstruktion informal beschrieben. Die Konstruktion ist ein Ausdruck für das Ergebnis des Transformationsschrittes. Die formale Beschreibung der Struktur und der *Single Pushout* – Eigenschaften kann man in [Löw93] finden.
Der Ergebnisgraph H wird in zwei Schritten konstruiert:

- **Erster Schritt: Verkleben und Hinzufügen**

Man betrachtet erstmal die Objekte aus dem Definitionsbereich *dom(r)* des Regelmorphismus *r*, also diejenigen Graphobjekte aus dem linken Regelgraphen, die tatsächlich Abbildungen auf die Graphobjekte aus dem rechten Regelgraphen über r besitzen. Diese Objekte sind wiederum im Arbeitsgraphen G über den Ansatz *m* abgebildet. Man fügt die Objekte, die nach der Regel erzeugt werden sollen, zwischen die entsprechenden Objekten in Arbeitsgraphen ein. Außerdem verklebt man die Objekte aus G die von dem Regelmorphismus erkannt werden. Man erhält ein Zwischenresultat: einen Graphen, in dem *G* als ein Teilgraph enthalten ist. Die Objekte aus G, die kein Urbild unter *m* in *dom(r)* haben, werden in den Graphen Z einfach unverändert übernommen. Bei diesem Schritt werden Objekte verklebt, wenn der Regelmorphismus r sie verklebt. Dagegen werden diejenigen Objekte hinzugefügt, die sich in dem rechten Regelgraphen befinden und über r nicht erreicht werden können.

- **Zweiter Schritt: Löschen**

Man löscht die Abbildungen aller Objekte aus *L,* die nicht in der Definitionsbereich *dom(r)* von *r* liegen, also diejenigen Objekte, die sich in dem rechten Regelgraphen befinden, aber in dem linken nicht vorkommen. Alle diese Objekte haben ihre Abbildungen im Graphen G über den Ansatz m. Die Objekte sind auch in dem Graphen Z zu finden, weil man in dem Zwischenschritt noch nichts gelöscht hat. Erst im nächsten Schritt werden sie gelöscht. Dabei können hängende Kanten entstehen, also solche, deren Ziel- oder Quellknoten gelöscht wurde. Die Kanten werden dann ebenfalls gelöscht. Sonst wäre das Ergebnis inkonsistent, weil nach der Definition eines Graphen, keine hängende Kanten erlaubt sind.

Die folgende Abbildung zeigt eine Graphtransformation in dem *Single Pushout* – Ansatz. Die in der *Abbildung 4* vorgestellte Regel wurde auf einem Beispielgraphen G angewandt.

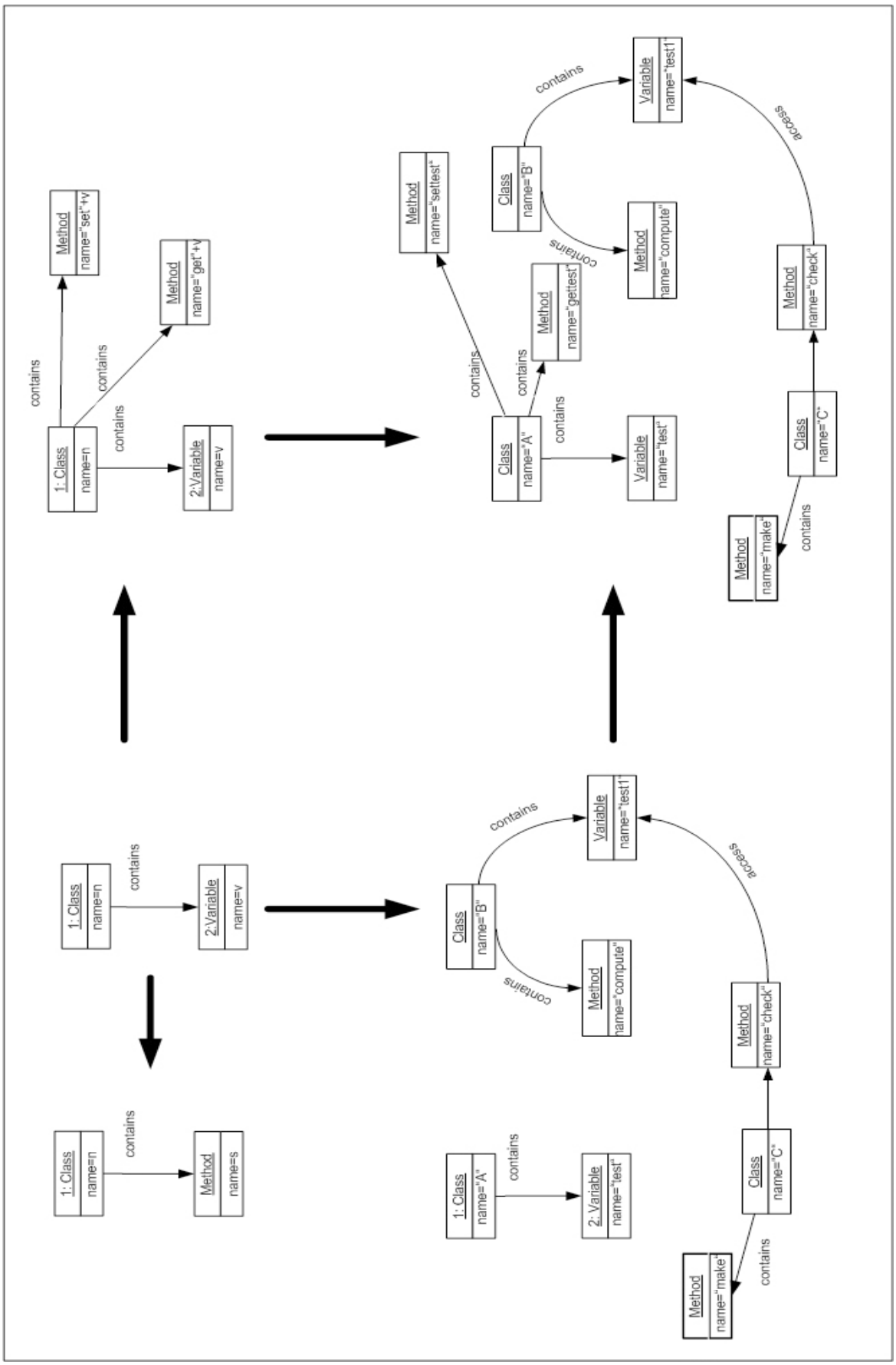

Abbildung 6. *Pushout - Beispiel*

2.1.9 Unabhängige Regeln

- **Parallele Unabhängigkeit**

Der einfachste Typ von parallelen Regeln sind so genannte *unabhängige Regeln*. Wenn sie auf verschiedene Objekten angewendet werden, können sie problemlos parallel ausgeführt werden. Die Anwendung der parallelen Regeln, welche die parallele Ausführung der nur unabhängigen Aktionen modellieren kann, ist äquivalent zu der Anwendung einer Regel in mehreren Abläufen. Im Folgenden werden nur Regeln ohne NACs definiert. Es seien zwei Regeln r1 und r2 mit den linken Regelseiten L1 und L2, sowie den rechten Regelseiten R1 und R2 gegeben. Man nennt die Ableitung eines Graphen G mit Regel r2 am Ansatz m1 unabhängig von r1, wenn sie kausal von ihr unabhängig ist, d.h. beliebig verzögert werden kann. Intuitiv bedeutet das, dass r2 nicht auf solchen Elementen von G operiert, die von r1 unter m1 gelöscht werden. Alternative Ausdruckweise: Die Elemente des Ausgangsgraphen, die im Ansatz der zweiten Ableitung liegen, dürfen nur im Kontext oder im bewahrten Teil der ersten Ableitung liegen. Zu beachten sind hier auch die bei der ersten Ableitung implizit gelöschten Kanten. Zwei Ableitungen, die wechselseitig schwach parallel unabhängig sind, werden (stark) parallel unabhängig genannt.
Neben der Unabhängigkeit von Ableitungen lässt sich auch die Unabhängigkeit von Regeln untersuchen: Zwei Regeln sind voneinander unabhängig, wenn alle möglichen Ableitungen (an beliebigen Graphen mit beliebigen Ansätzen) mit diesen Regeln voneinander unabhängig sind.

- **Sequentielle Unabhängigkeit**

Eine Ableitung ist (schwach) sequentiell unabhängig von einer anderen, wenn sie keine Elemente benutzt, die von anderen Ableitungen erst erzeugt werden. (Starke) sequentielle Unabhängigkeit und Unabhängigkeit von Regeln sind analog zum parallelen Fall definiert. Die formale Definition der sequentiellen und parallelen Unabhängigkeit findet sich in [KLTW93].

2.2 Amalgamierte Graphtransformation

In folgendem Abschnitt werden die relevanten Begriffe zur amalgamierten Transformation erläutert. Die formale Beschreibung zu der amalgamierten Graphtransformation ist in [Taenz96] und [Taen,Bey] zu finden. Die wichtigen Komponenten der amalgamierten Graphtransformation werden anhand eines konkreten Beispiels vorgestellt. Anschließend werden eine amalgamierte Regel und ihre Anwendung anhand eines Beispiels genauer betrachtet.

2.2.1 Synchronisierte Regel

Wenn die Aktionen nicht unabhängig von einander sind, können sie trotzdem parallel angewendet werden, wenn sie durch Unteraktionen synchronisiert werden können. Wenn zwei Aktionen das Löschen oder das Bilden des gleichen Knotens enthalten oder dasselbe Attribut geändert wird, kann die Operation in zwei separate Aktionen geteilt werden. Die Aktionen sind dann die Unterregeln der originalen Aktion (die elementare Regel). Die Anwendung der Regeln, die durch Unterregeln synchronisiert wird, wird durch das Zusammenkleben der elementaren Regel über ihre Unterregel durchgeführt, was zu der amalgamierten Regel führt.

2.2.2 Amalgamierung (informale Beschreibung)

Amalgamierte Graphtransformationen erweitern die algebraische Art von parallelen Graphtransformationen. Der Ansatz von parallelen Graphtransformationen ist eine Grundvariante einer Graphersetzung, wo der ganze Arbeitsgraph mit den Vorkommen (Ansätzen) der linken Seiten von Produktionen (Regeln) bedeckt sein muss. Die Schnittstellen von unterschiedlichen Ansätzen einer Regel müssen während der parallelen Graphersetzung konstant erhalten bleiben.

Diese Einschränkungen entfallen für amalgamierte Graphtransformationen. Dynamische Schnittstellen sind hier erlaubt, das bedeutet, dass die Schnittstelle während eines Transformationsschrittes sich ändern kann. Außerdem kann die Abdeckung mit den Ansätzen auf dem Arbeitsgraphen partiell sein. Nur der abgedeckte Teil wird ersetzt und der Kontext wird zu dem transformierten Arbeitsgraphen unverändert hinzugefügt. Wenn man komplexe Operationen bei

den Graphtransformationen beschreiben möchte, kann man sagen, dass all die Graphregeln, die beteiligt sind, in eine Regelmenge aufgefasst werden.

Mit Hilfe von Amalgamierungen hat man die Möglichkeit die Aktionen, die auf vielen Graphobjekten in den Graphtransformationen durchgeführt werden, durch ein Regelschema auszudrücken. Damit werden sequentielle und parallele Graphersetzungskonzepte kombiniert. Um eine amalgamierte Graphtransformation durchführen zu können, muss man eine amalgamierte Regel erstellen. Dies geschieht in mehreren Schritten.
Zuerst wird ein Regelschema spezifiziert, dann wird ein Instanzschemata aufgebaut und anschließend wird eine amalgamierte Regel erstellt. Dies geschieht in mehreren Schritten.

2.2.3 Regelschema RS

Ein Regelschema RS besteht aus einer Unterregel S und einer Menge von Erweiterungsregeln {E1, E2….Ei}. Die linke Seite der Unterregel L_s ist ein Teilgraph der linken Seite jeder Erweiterungsregel LE_i und die rechte Seite der Unterregel R_s ist ein Teilgraph der rechten Seite jeder Erweiterungsregel RE_i. Zwischen der Unterregel und jeder Erweiterungsregel gibt es jeweils eine Einbettung. Die ist ausgedrückt durch einen Graphmorphismus $el_i : L_s \rightarrow LE_i$ auf der linken Seite und $er_i : R_s \rightarrow RE_i$ auf der rechten Seite. Formal wird ein Regelschema in [ETB05] beschrieben.

Die folgende *Abbildung 7* zeigt ein Regelschema in einer allgemeinen Konstruktion nach dem *local all* – Konzept (siehe Definition in [Taenz96]). Auf dem Bild sieht man eine Unterregel S und Erweiterungsregeln von 1 bis i.

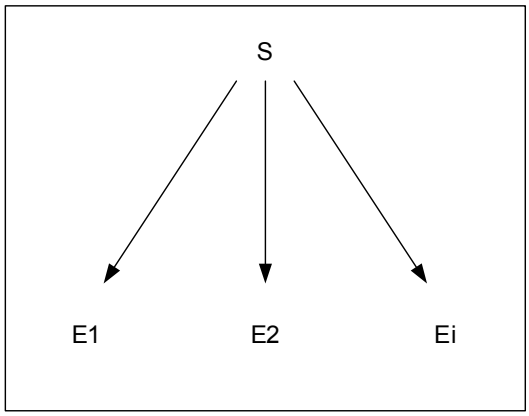

Abbildung 7. *Regelschema*

Die

Abbildung 8 zeigt ein Regelschema in einem konkreten Beispiel.

In dem vorgestellten Regelschema wird eine Unterregel S und zwei Erweiterungsregeln E1 und E2 dargestellt.

Die Unterregel S ist wie folgt definiert: "Hinzufügen der *'Set-Methode'* und *'Get-Methode'* zu der vorgegebenen Klasse".

Die Erweiterungsregel E1 beinhaltet zusätzlich eine *'Access-Methode'*, die den Zugriff auf die Variable die in der Klasse definiert ist, ändert. Ähnlich wird die Erweiterungsregel E2 definiert. Hier wird aber eine *'Update-Methode'* hinzugefügt, die nach der Anwendung der Regel die Variable über die *'Set-Methode'* ändert.

Die Regelmorphismen und die Einbettungsmorphismen werden durch die Zahlen in den Graphknoten dargestellt. Man kann die Einbettungen von der Unterregel an den Zahlen “1“ für *Class* und “2“ für *Variable* bezeichneten Knoten in beiden Erweiterungsregeln erkennen. Die Zahlen “3“ und “4“ stellen die Einbettungsmorphismen zwischen der Unterregel und Erweiterungsregeln, dar.

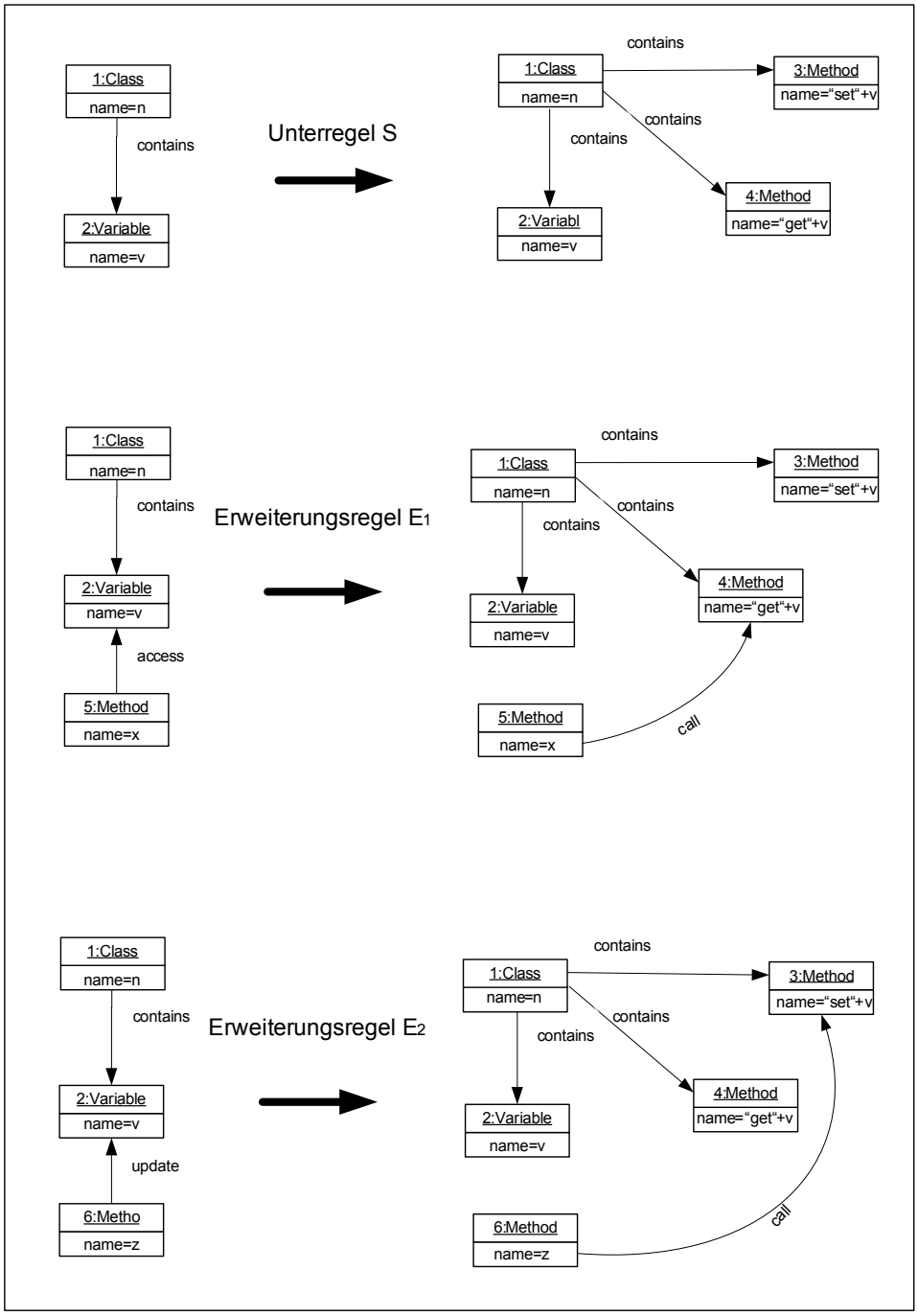

Abbildung 8. *Regelschema - Beispiel*

2.2.4 Instanzschema IRS

Ein Instanzschema IRS wird im Bezug auf ein Regelschema RS aufgebaut. Es besteht aus einer Unterregel S und mehreren Instanzen (I1, I2…Ij) von der Erweiterungsregeln (E_1, E_2,…Ei). Die Instanzen entstehen dadurch, dass jede

Erweiterungsregel durch jeden möglichen Ansatz vervielfältigt wird. Entsprechend müssen die Einbettungen zwischen der Unterregel und jeder Instanz beibehalten werden. Die formale Beschreibung ist auch in [Taenz96] zu finden.

Die *Abbildung 9* zeigt eine allgemeine Darstellung eines Instanzschemas. Man kann auf dem Bild eine Unterregel S mit Instanzen von 1 bis i von jeder Erweiterungsregel sehen.

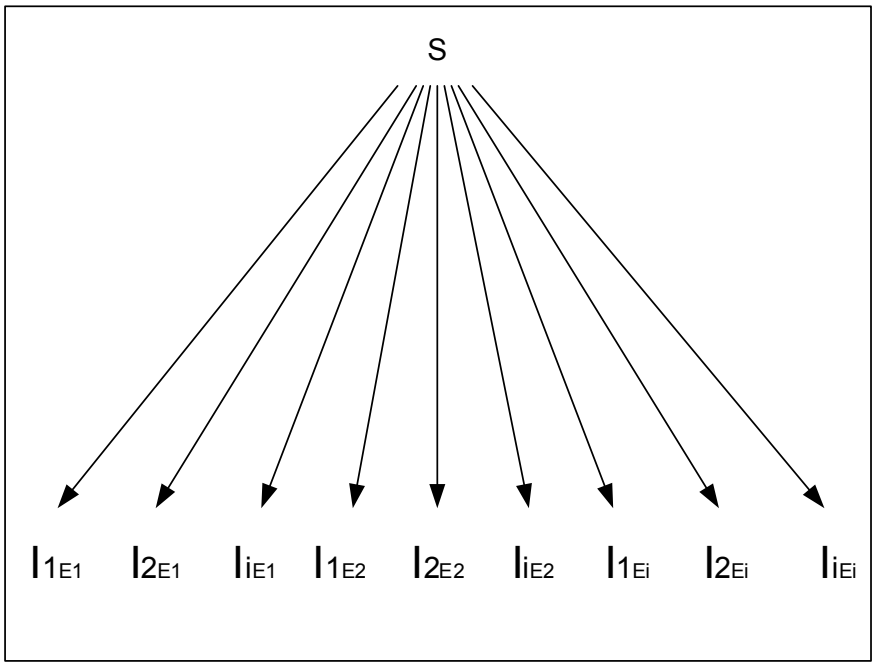

Abbildung 9. *Instanzschemata*

Dagegen wird in der *Abbildung 10* der Ansatz von IRS im Bezug auf das oben vorgestellte Regelschema, an einem konkreten Beispiel dargestellt.
Die Unterregel S bleibt unverändert. Die in vorherigem Abschnitt im Regelschema vorgestellte Erweiterungsregel E1 besitzt zwei Instanzen I1E1 und I2E1. Deswegen findet hier eine Variablenumbenennung statt, weil gleiche Variablennamen in den Instanzen einer Erweiterungsregeln nur im Unterregelbereich erlaubt werden. Die Erweiterungsregel E2 besitzt nur eine Instanz I1E2, hier wird die Variablenumbenennung nicht benötigt.

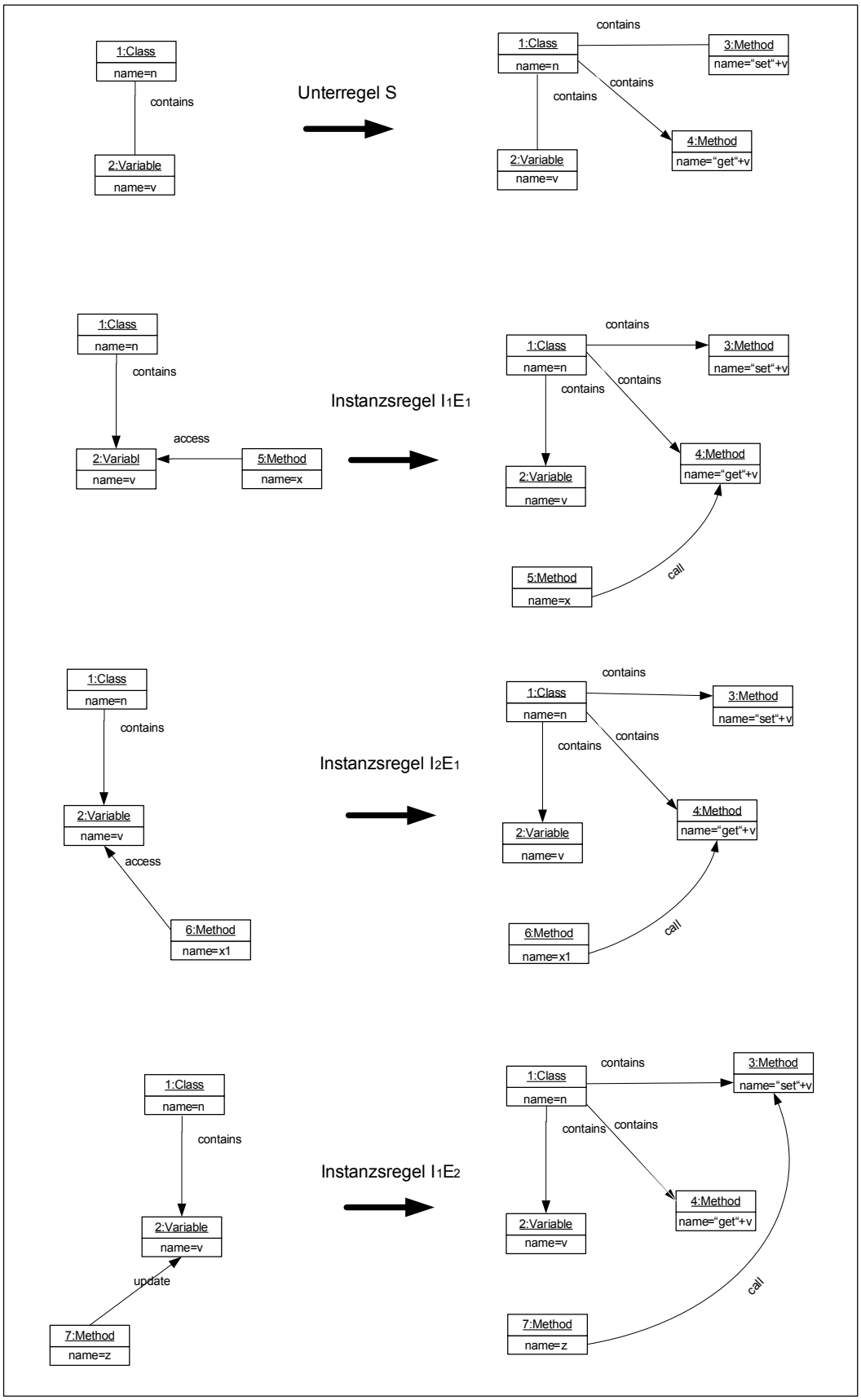

Abbildung 10. *Instanzschema - Beispiel*

2.2.5 Amalgamierte Regel

Eine amalgamierte Regel entsteht dadurch, dass die Komponenten von einem Instanzschema miteinander verklebt werden. Die Unterregel legt fest, welche Objekte der linken und der rechten Seiten der Instanzen miteinander verklebt werden. In anderen Worten, besteht die amalgamierte Regel aus einer Menge der Instanzen der Erweiterungsregeln, die sich in der Unterregel überlappen. Eine Amalgamierungsregel bezeichnet man als auf einen Graphen anwendbar, wenn mindestens ein Ansatz für ihre Unterregel existiert. Eine Anwendung der amalgamierten Regel auf einen Graphen G nennt man eine amalgamierte Graphtransformation.

Die *Abbildung 11* stellt die schematische Konstruktion einer amalgamierten Regel dar. Man kann den linken (LS) und den rechten Graphen (RS) der Unterregel S und die Instanzregel mit ihren Regelmorphismen genau erkennen. Der linke und der rechte Graph von der amalgamierten Regel werden mittels Kolimeskonstruktion berechnet (siehe *Kapitel 2.2.7*).

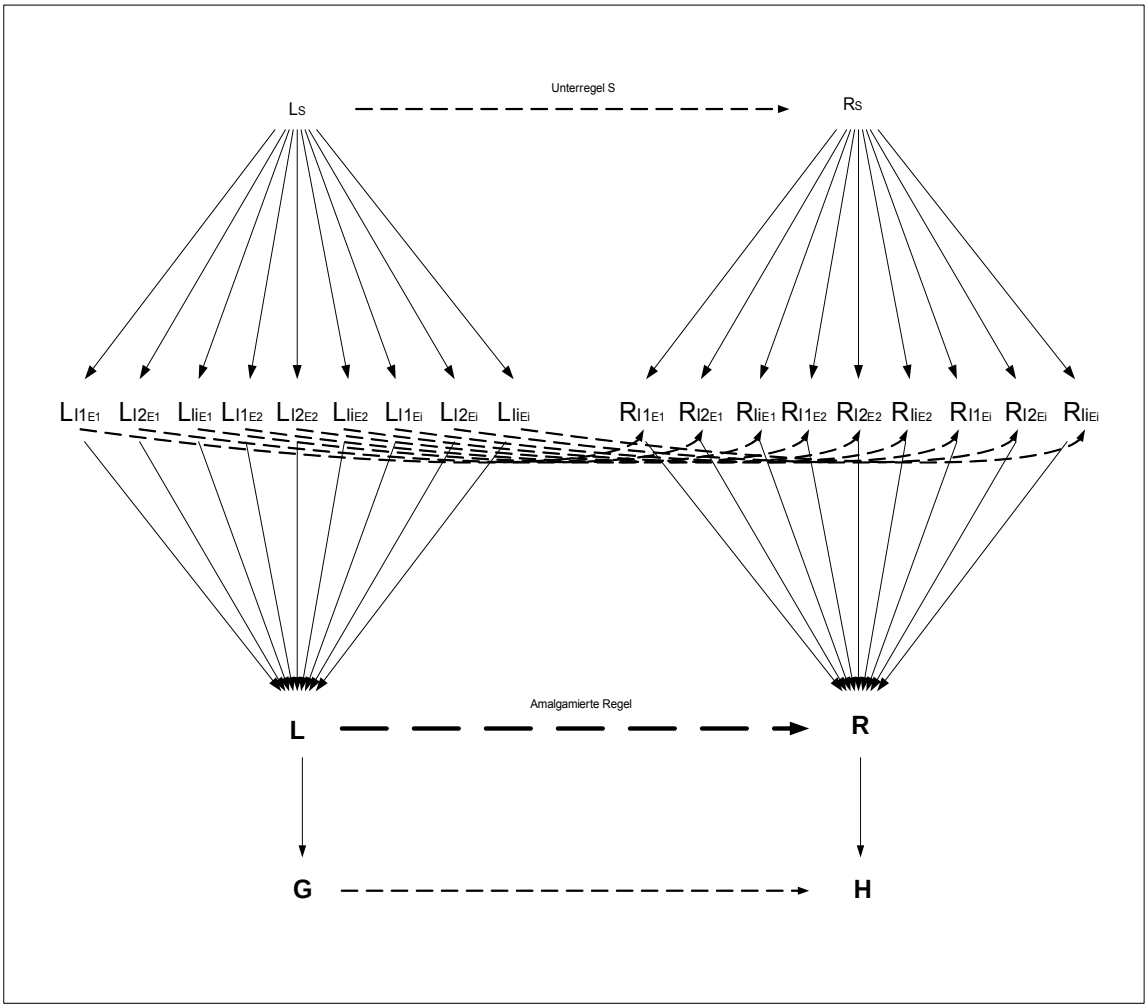

Abbildung 11. *Amalgamierte Regel*

2.2.6 Konstruktion der amalgamierten Regel

Um die amalgamierte Regel zu bilden und dann ihren Ansatz in Graphen G zu finden, muss man mehrere Morphismen und Matches berechnen.

Zuerst wird der Ansatz der Unterregel S gefunden. In dem entwickelten Algorithmus wird er *Match Ms* genannt. Sehr wichtig sind die o.g. Einbettungsmorphismen (*embedding morphism*) zwischen der Unterregel und den Erweiterungsregeln. Diese werden nicht berechnet, weil sie vorgegeben sind.

Danach wird für jede Erweiterungsregel ein Match ME erstellt. Match Ms wird als partieller Match übernommen. Im nächsten Schritt werden isomorphe Kopien von jeder Erweiterungsregel gebildet. Diese Kopie wird als eine Instanz

einer Erweiterungsregel genannt. Man bekommt so viele Kopien, wie Ansätze von der Erweiterungsregel in dem Graph G gefunden werden.

Der Ansatz von jeder Instanz in dem Graphen G wird als Match Mi bezeichnet.

Um den linken und rechten Graphen einer amalgamierten Regel zu berechnen, muss man die Einbettungsmorphismen zwischen der Unterregel und jeder Instanz finden. Sie werden durch die Basiseinbettungsmorphismen (embbeded morphism) berechnet.

Analog wird das ganze Verfahren auf der rechten Seite durchgeführt.

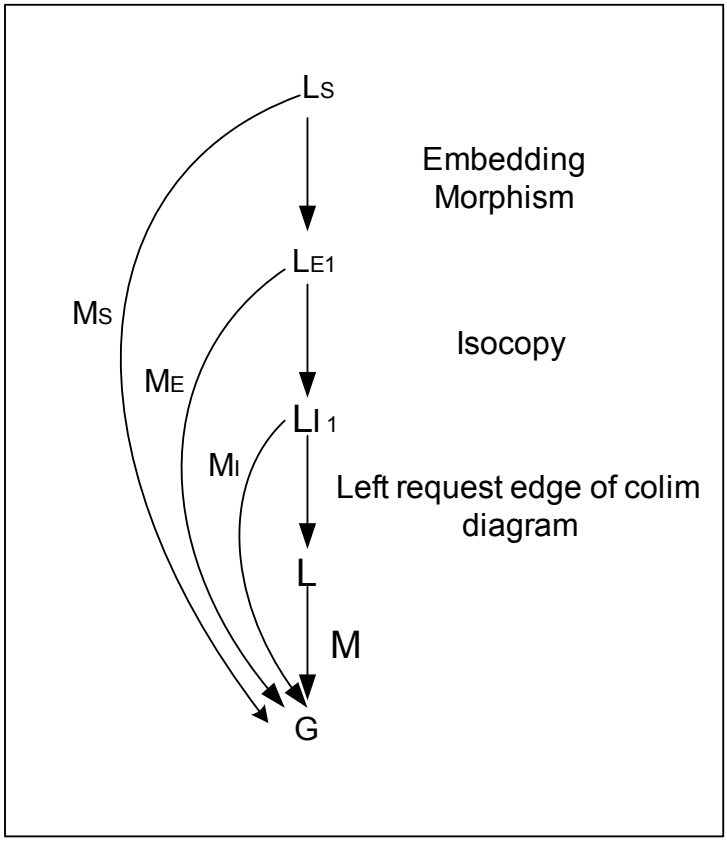

Abbildung 12. *Morphismen und Matches*

2.2.7 Kolimesdiagramme

Um den linken und rechten Graphen von einer amalgamierten Regel zu bekommen, werden zwei Kolimiten berechnet. Die formale Definitionen von einem *Diagramm* und einem *Kolimit* ist in der Dissertation [Wolz98] zu finden.

Als Resultat bekommt man die Morphismen zwischen jeder linken Seite der

Instanzregeln und dem linken Graphen der amalgamierten Regel (*left request edges*). (siehe *Abbildung 12*) Analog sieht die ganze Struktur auf der rechten Seite aus.

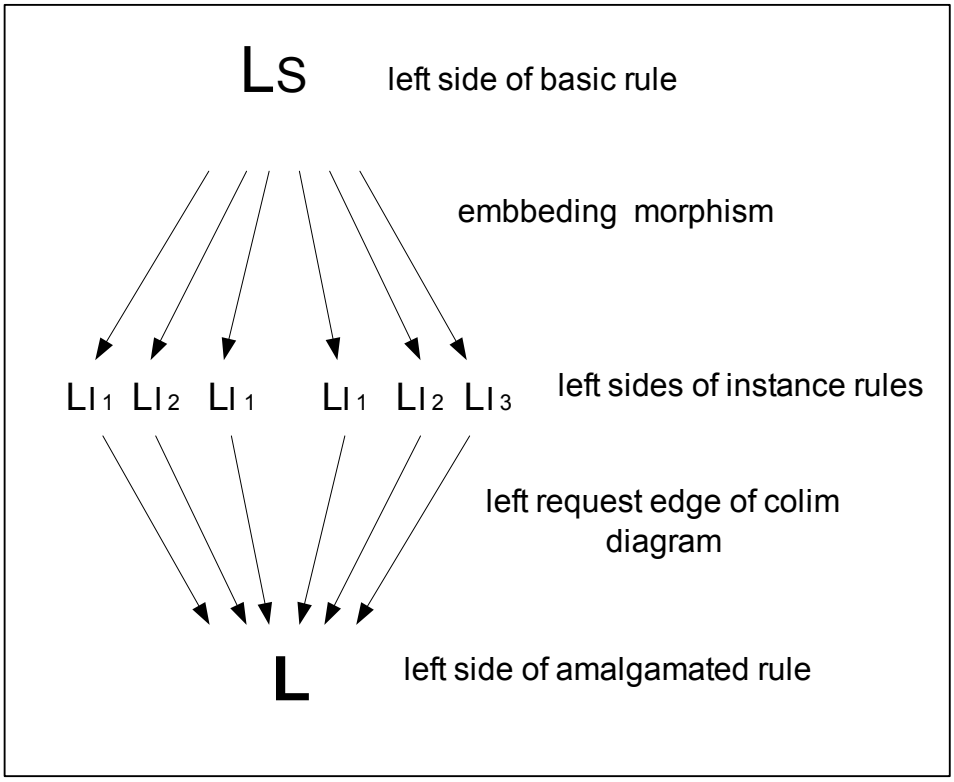

Abbildung 13. *Kolimesdiagramm*

2.2.8 Regelmorphismen

Von Anfang an werden Regelmorphismen von der Unterregel rs und jeder Erweiterungsregel rE vorgegeben. Nachdem man die linken und rechten Graphen aus den Instanzen erstellt hat, muss man für diese den Regelmorphismus ergänzen. Man findet Quell- und Zielgraphobjekte in dem Regelmorphismus für eine Instanzregel über den Definitionsbereich *dom(r)* ihrer Erweiterungsregel und über folgende Morphismen: zwischen dem linken Graphen der Erweiterungsregel und dem linken Graphen der Instanzregel LELI und über den Morphismus zwischen rechten Graphen der Erweiterungsregel

und rechten Graphen der Instanzregel $R_E R_I$ (siehe *Abbildung 14*).

Dabei werden die Attributbedingungen und Variablen von der Unterregel und den Erweiterungsregeln übernommen. Besitzt eine Erweiterungsregel mehrere Instanzen, werden die Variablen und die Attributsbedingungen in jeder Instanz umbenannt. Die Instanzregelmorphismen braucht man, um den Regelmorphismus von der amalgamierten Regel zu vervollständigen.

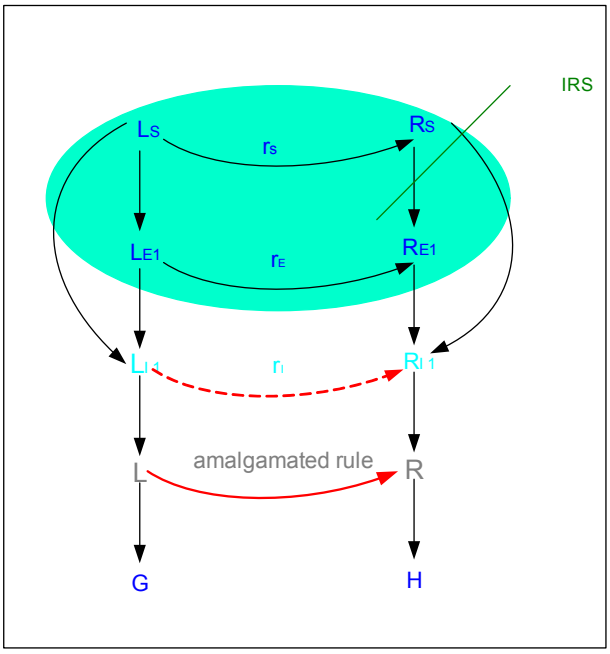

Abbildung 14. *Regelmorphismen*

2.2.9 NACs und Attributsbedingungen in AGT

Die negativen Anwendungsbedingungen werden von der Unterregel und den Erweiterungsregeln übernommen und zu der amalgamierten Regel hinzugefügt. Bei der Erweiterungsregel werden die NACs für jede Instanz kopiert. Im Endeffekt besitzt die amalgamierte Regel von jeder Instanz und von der Unterregel die NACs. Auch die Attributsbedingungen werden von der Unterregel und jeder Instanzregel übernommen und bei der amalgamierten Transformation berücksichtigt.

In der folgenden *Abbildung 15* wird aus einer Unterregel und Instanzregeln, die im Beispiel von dem Instanzschema vorgestellt wurden (siehe *Abbildung 10*),

eine amalgamierte Regel erstellt. Die NACs (NAC1, NAC2) und die Attributsbedingungen werden übernommen.

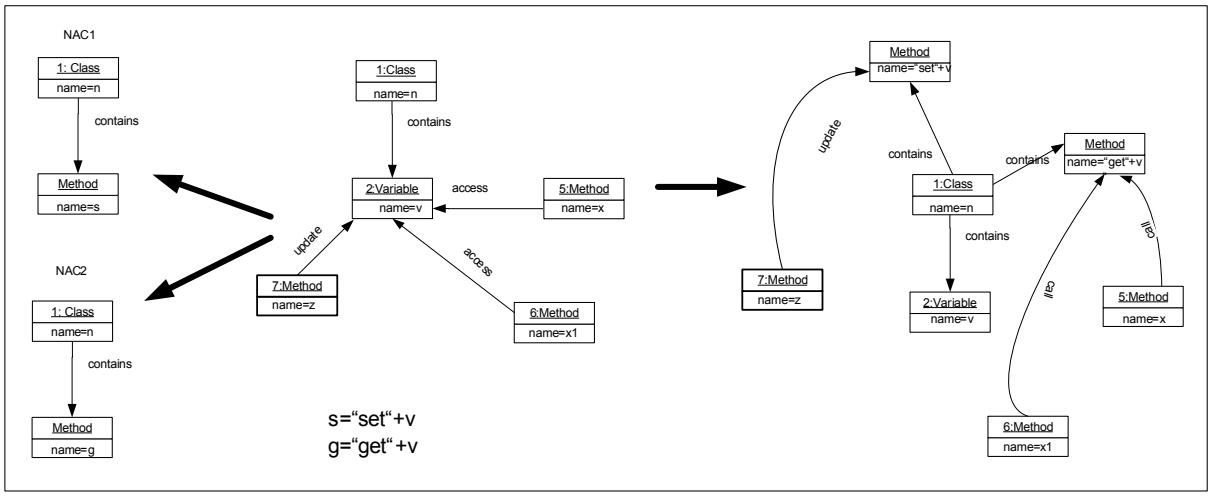

Abbildung 15. *Amalgamierte Regel - Beipiel*

2.2.10 Ansatz der amalgamierten Regel

Nach dem die amalgamierte Regel erstellt wurde, wird ihr Ansatz (*Match M*) im Graphen G berechnet. Match M wird durch die partiellen Morphismen die über left request edges und die Matches ME berechnet (siehe *Abbildung 12*). Danach wird die amalgamierte Regel wie eine einfache Regel angewendet.

Die folgende Abbildungen zeigen die Anwendung der amalgamierten Regeln auf einem Arbeitsgraphen G. Die *Abbildung 16* zeigt den Ansatz der amalgamierten Regel im Graph G. Das nächste Bild (*Abbildung 17*) zeigt den Arbeitsgraphen G unmittelbar nach der Ausführung des Transformationsschrittes.

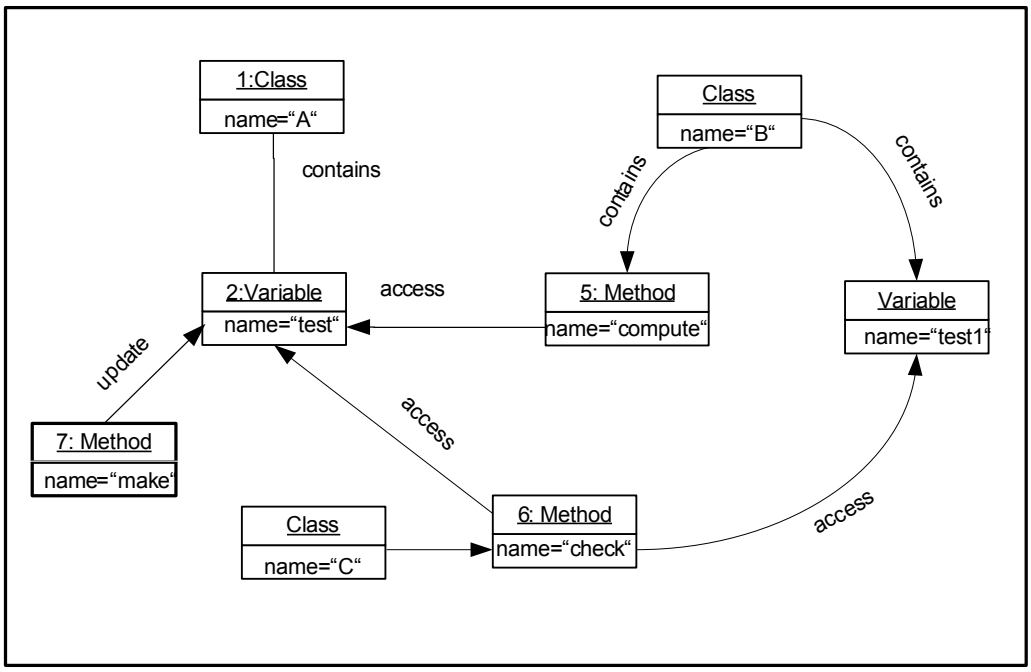

Abbildung 16. *Ansatz der amalgamierten Regel*

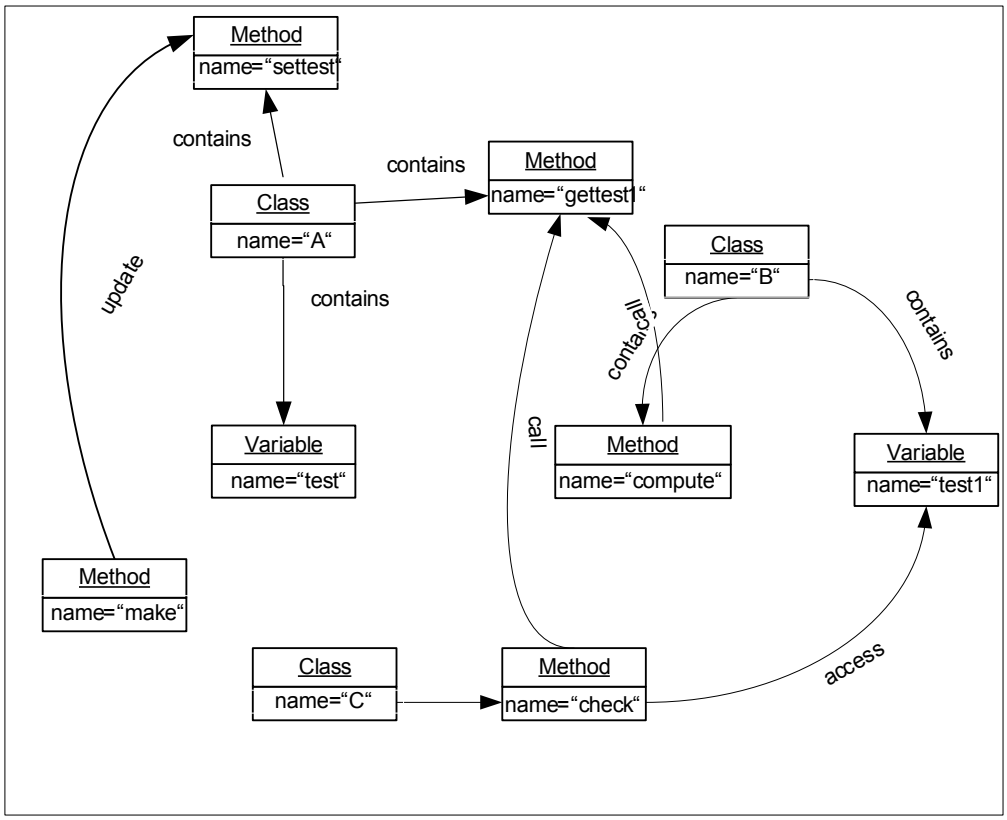

Abbildung 17. *Graph G nach der Transformation*

3. Methoden und Werkzeuge

In diesem Kapitel wird beschrieben, welche Methoden und Werkzeuge benutzt wurden, um die Implementierung des Amalgamierungskonzeptes zu verwirklichen. Der folgende Abschnitt beschreibt die verwendete Modellierungssprache UML. Ihre Möglichkeiten und ihr Beitrag zum optimalen Entwurf des Konzeptes werden geschildert. Zunächst wird die verwendete Modellierungssprache und ihre Vorteile vorgestellt.

3.1 Modellierungssprache UML

Als Modellierungssprache für das Software-Konzept wurde UML (*Unified Modelling Language*) gewählt. Insbesondere wurden die Klassen-abhängigkeiten im AGT-Package mit der Hilfe von Klassendiagrammen präsentiert. UML ist ein Standardwerkzeug, um umfassende Softwaresysteme, Geschäftsprozesse und allgemeine Systeme zu spezifizieren und zu visualisieren. UML bietet eine breite Palette von Darstellungsmethoden, die sich erfolgreich in der Modellierung von umfangreichen und komplexen Softwaresystemen bewiesen haben. Mittels formaler graphischer Notation können mögliche Entwürfe untersucht und die Softwarearchitektur validiert werden. Zu den wichtigen Aspekten von UML zählt auch die Unabhängigkeit von konkreten Programmiersprachen und Entwicklungsprozessen. UML bietet auch eine formale Grundlage zum Verständnis der Modellierungssprache und unterstützt übergeordnete Entwicklungskonzepte, wie Kollaborationen, Frameworks, Patterns und Komponenten.

Wie oben erwähnt wurde, wurden Klassen- und Sequenzdiagramme zur Modellierung der Software für die amalgamierte Graphtransformation angewendet. Das Klassendiagramm erlaubt es, die Struktur von den entwickelten AGT-Package zu spezifizieren und seine Abhängigkeiten von AGG–Packages zu visualisieren. Mit der Hilfe von den Aktivitätsdiagrammen wurde der Algorithmus des Amalgamierungsprozesses dargestellt. Eine genauere Beschreibung zu UML ist auf der Webseite [UML] zu finden.

3.1.1 Klassendiagramme

Die Klassendiagramme werden breit verwendet, um die Typen und die Relationen von Objekten zu beschreiben. Sie verdeutlichen die Struktur der Klassenmodelle und anderen Entwürfen, die Designelemente wie Klassen, *Packages* und Objekte benutzen. Klassendiagramme beschreiben drei verschiedene Perspektiven beim Systemdesign: die konzeptionelle, die Anforderungs- und die Implementierungsperspektive. Die Perspektiven werden beim Erstellen des Diagramms herauskristallisiert und helfen dadurch, das Design deutlich zu umschreiben.

Im Mittelpunkt der Klassendiagramme, stehen die Begriffe *Klasse, Schnittstelle* und *Package*. Eine Klasse (siehe *Abbildung 18*) ist eine Zusammenfassung gleichartiger Objekte, welche die Grundelemente einer Anwendung sind. In einer Klasse werden alle Eigenschaften, die für ein Objekt eine Rolle spielen, definiert. Dazu gehören sowohl dessen Daten (Attribute, Elemente) als auch die durchführbaren Operationen (Methoden). Man kann eine Klasse auch als abstrakten Datentyp oder die Umsetzung eines abstrakten Datentyps in einer Programmiersprache betrachten.
In der UML–Notation werden die Klassen durch Rechtecke dargestellt, die den Namen der Klasse, die Attribute und Operationen der Klasse enthalten, die voneinander durch eine horizontale Linie getrennt werden. Ein Klassenname ist immer im Singular und beginnt mit einem Großbuchstaben. Die Attribute werden durch ihren Namen, Typ und manchmal durch ihren Initialwert beschrieben. Operationen können ebenfalls durch Parameter, Namen, Initialwerte usw. beschrieben werden.

Schnittstellenklassen spezifizieren das externe Verhalten von Klassen und enthalten in abstrakter Form Signaturen und Beschreibungen von Operationen. Es sind abstrakte Klassen mit dem Stereotyp "interface". Klassen, die von einer bestimmten Schnittstellenklasse geforderten Operationen bereitstellen können, sind eine Umsetzung dieser Schnittstelle. Eine Klasse kann mehrere Schnittstellen implementieren.

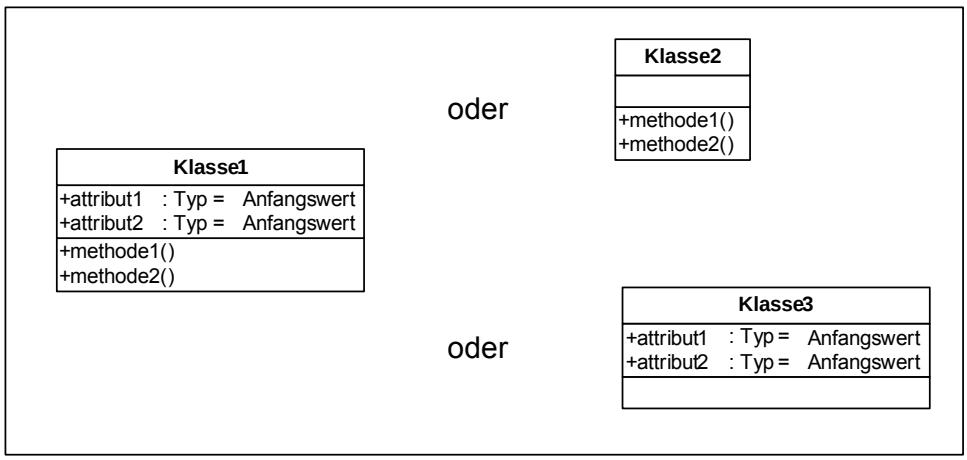

Abbildung 18. *Klassendarstellung*

Die *Abbildung 19* zeigt die Klassenelemente eines Klassendiagramms.

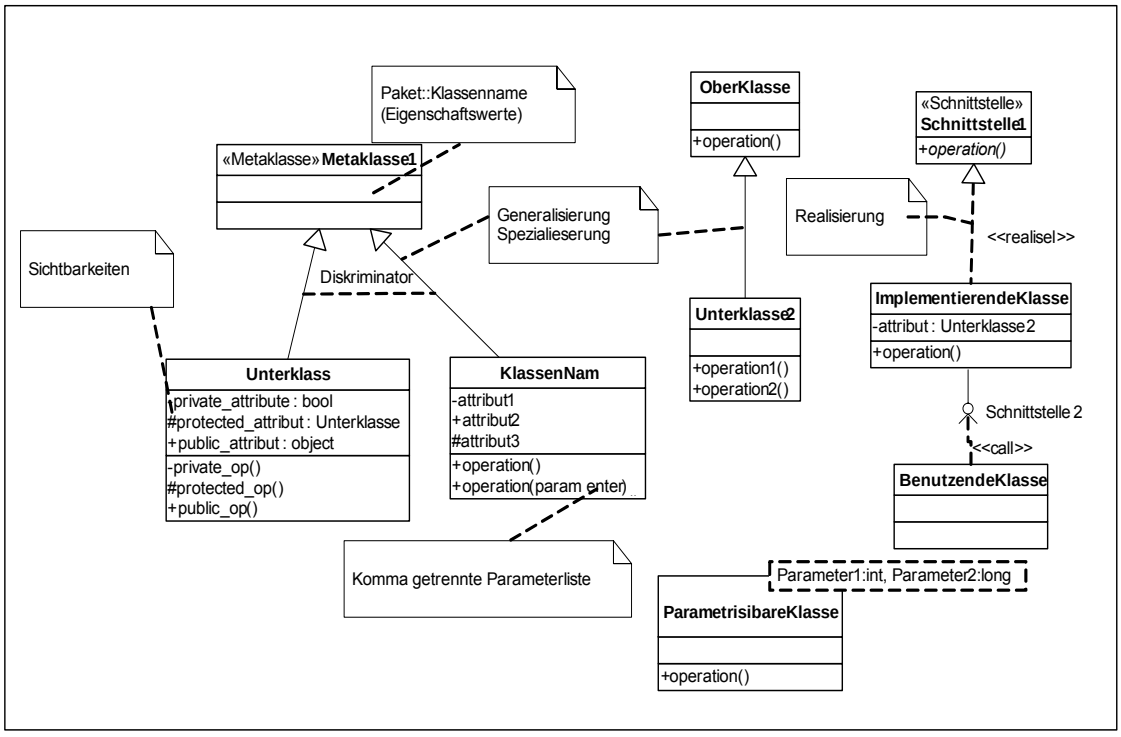

Abbildung 19. *Klassendiagramm*

3.1.2 Aktivitätsdiagramme

Die Aktivitätsdiagramme sind nützlich, wenn man komplizierte sequentielle Algorithmen und Modellierungsapplikationen mit parallelen Prozessen beschreiben möchte. Ein Aktivitätsdiagramm (siehe *Abbildung 20)* beschreibt die Zustände und die Zustandübergänge, die während des Programmablaufs ausgeführt werden.

Die graphische Darstellung von Aktivitätsdiagrammen wird von oben nach unten gelesen. Eine Aktivität wird durch ein abgerundetes Rechteck dargestellt, der eine Beschreibung der internen Aktion enthält. Aktivitäten werden durch Transitionen verbunden, die den Abschluss der internen Aktion und den Übergang zur nächsten Aktivität darstellen. Die Ablaufdiagramme können *branches* (Verzweigungen) und *forks* (Gabelungen) enthalten, um Bedingungen und parallele Aktivitäten zu beschreiben. *Fork* wird benutzt, wenn mehrere Aktionen gleichzeitig ausgeführt werden. *Branch* beschreibt, welche Aktionen in Abhängigkeit von Bedingungen ausgeführt werden sollen. In einem *merge* wird die Abbruchbedingung erkannt. Die parallelen Aktivitäten werden mit einem *join* kombiniert.

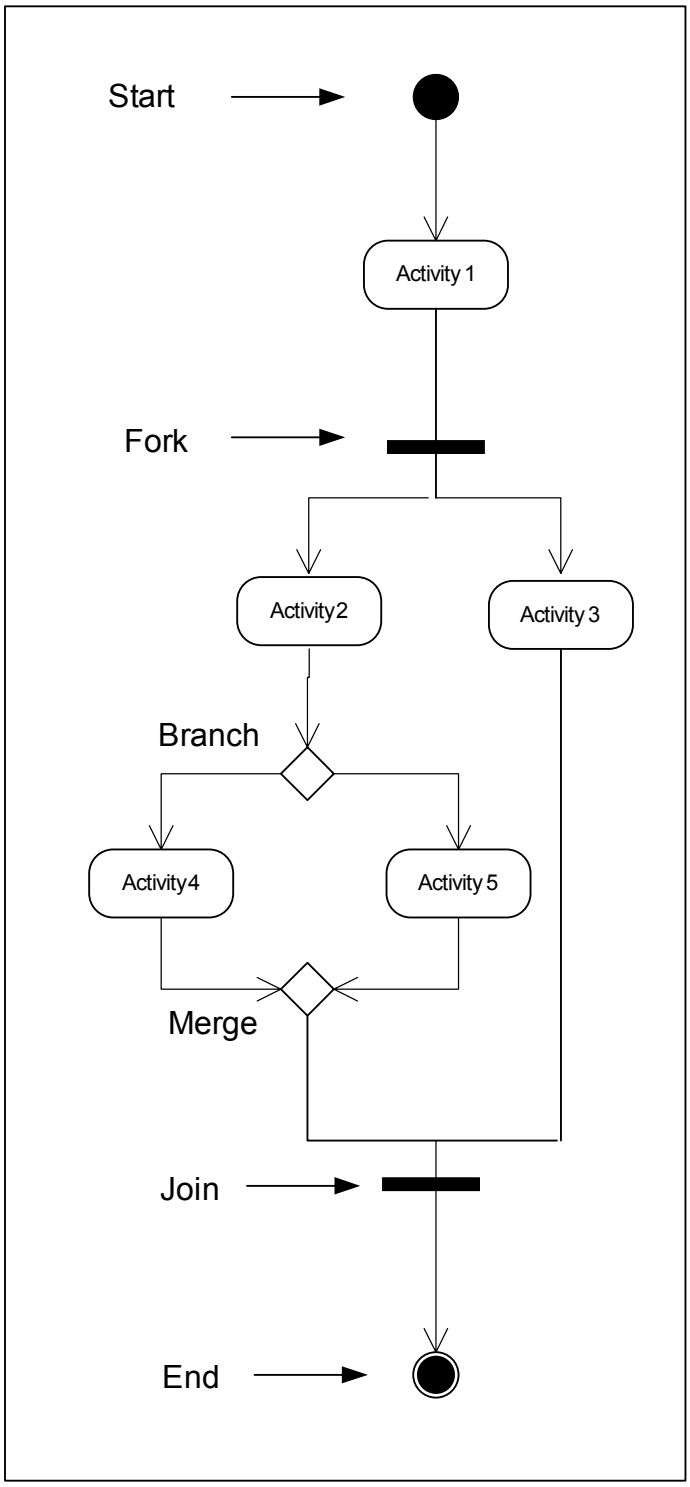

Abbildung 20. *Aktivitätsdiagramm*

3.2 Programmiersprache JAVA

Der Interpreter für die amalgamierte Graphtransformationen beschränkt sich auf die Erweiterung des existierenden AGG -Tools, das vollständig in Java

implementiert wurde. Aus diesem Grund wurde auch in der Diplomarbeit Java verwendet.

Bei Java handelt es sich um eine vollständig objektorientierte Programmiersprache. Außerdem hat Java mehrere Vorteile anderen Programmiersprachen gegenüber. Ausführliche Beschreibung zu diesem Thema ist auf der Webseite [Java] zu finden.

- Plattformunabhängigkeit und der hohe Grad an Portabilität sind der wichtigsten Vorteile der Java-Programmiersprache. Das wurde dadurch erreicht, dass ein Java Compiler Bytecode erzeugt wird, der auf allen Systemen und jeder Hardware identisch durch den Interpreter übersetzt wird. Im Prinzip sind Java – Programme auf jedem Betriebsystem lauffähig.
- Alle erzeugten Programme werden lediglich in einem festen Bytecode kompiliert. Erst bei der Ausführung wird dieser durch die JVM (Java Virtual Machine) interpretiert und in ein passendes Format des Zielsystems übersetzt. Somit laufen Interpretation und Kompilierung als paralleler Prozess ab.
- Die neuen Konzepte von Java erlauben es, viele Prozesse parallel ablaufen zu lassen, was ein besseres Laufzeitverhalten verursacht. Die Vorteile dieser Technologie sind allerdings auch stark vom zugrundeliegenden System abhängig.
- Jedes Konzept wird in Java ist in Klassen eingeschlossen. Externe Definitionen sind nicht zulässig. Desweiteren erlaubt Java als eine objektorientierte Sprache die Vererbung. Die Mehrfachvererbung wurde mit Hilfe von Interfaces gelöst.
- In Java werden die Wertebereiche der internen Datentypen fest vorgeschrieben. Dadurch ergeben sich keine Differenzen auf unterschiedlichen Systemen und die Applikationen werden in der Regel exakt übersetzt. Spezifische Details einer Umgebung können somit vernachlässigt werden.
- Sehr hilfreich ist die Typisierung von Java. Das bedeutet, dass alle Datentypen feste Werte und Bereiche besitzen, was gewisse Unabhängigkeit vom benutzen System verschafft. Aus diesem Grund können keine betriebsystemspezifische Differenzen auftreten und das Laufzeitverhalten negativ beeinflussen.

- Man kann Java als sehr stabil und robust bezeichnen. Viele Laufzeitfehler werden vermieden, weil sie alle Typkonvertierungen prüft. Java kann auch eine hohe Sicherheit zur Speicherverwaltung nachweisen.
- Java eignet sich auch zur Benutzerschnittstellenimplementierung. Die Standardbibliotheken wie *AWT* oder *swing* bieten dem Entwickler ganze Reihe von *GUI*–Komponenten an.

4. AGG System

Der folgende Abschnitt beschreibt, was das AGG-System (Version 124) vor meiner Erweiterung anbieten und leisten konnte. Es wurde genau erklärt, wie man eine Graphgrammatik laden oder neu erstellen konnte und ihre Komponenten, wie einen Graphen und Regeln definieren konnte. Für mehr Informationen zu dem AGG-System ist die AGG-*Homepage* [AGG] und [ErSchul, RuTaen99] zu empfehlen.

4.1 AGG – Graphische Benutzerschnittstelle

Die AGG-Umgebung wurde als ein Tool gestaltet, damit man die Graphgrammatiken definieren kann. Sie ermöglicht auch das Editieren von einzelnen Komponenten der Graphgrammatik. Die AGG-Graphgrammatik ist durch einen Arbeitsgraphen und eine Menge von Transformationsregeln definiert. Außerdem wird die visuelle Interpretation und das Debugging der attributierten Graphgrammatiken unterstützt. Das Debugging bedeutet die Darstellung der direkten Transformationsschritten für die von dem Benutzer ausgewählte Regel im *Single-Pushout* Konzept der Graphtransformationen. Eine gesamte Transformationssequenz ist in dem Interpretationsmodus ausführbar.

Die Benutzerschnittstelle des AGG-Systems bietet die umfassende Funktionalität für die Eingabe und Modifikation der attributierten Graphgrammatiken. Es gibt vier verschiedene Editoren, zwei graphische Editoren: für Graphen und Regeln, und einen Texteditor. Zusätzlich gibt es ein Fenster, um Grammatiken in Form eines Baums zu editieren. Der Texteditor dient dazu, dass man die Attribute in Form von Java-Ausdrücken, die in die graphische Oberfläche integriert werden, definiert. Der Regeleditor zeigt den linken und den rechten Graphen, den Regelmorphismus und soweit vorhanden auch die NAC mit den NAC-Morphismen von einer Regel die vom Benutzer selektiert wurde. Die Regel kann auf einen Arbeitsgraphen angewendet werden.

4.2 Anwendung der Benutzerschnittstelle

Die *Abbildung 21* stellt die graphische Benutzerschnittstelle des AGG-Systems dar. Links sieht man ein Fenster mit den aktuellen Graphgrammatiken. Es ist möglich mehrere Graphgrammatiken zu erstellen oder zu laden. Jede Graphgrammatik ist als ein Baum mit den zugehörigen Komponenten, wie der Arbeitsgraph und die Regel, dargestellt. Man kann die Namen der Grammatik, ihrer Regeln und ihres Graphen manuell ändern.

Das Popupmenü "*File*" besteht aus mehreren Menüeinträgen, um eine Graphgrammatik zu laden oder zu speichern, ihre Komponenten zu erstellen oder löschen und um das System zu beenden. Das Popupmenü "*Edit*" erlaubt Graphen, Regeln und Attribute zu editieren. Das Popupmenü "*Transform*" dient der Interpretierung und Debugging von der Graphtransformation. Hier gibt es die Möglichkeit, ein Match zu definieren und einen Transformationsschritt oder eine Transformationssequenz auszuführen.

Wenn der Benutzer eine neue Graphgrammatik erstellen möchte, kann er das, wie schon oben gesagt, über das Popupmenü "*File*" erreichen oder durch das Drücken des entsprechenden Buttons. Es wird automatisch eine Grammatik mit einem leereren Graphen und einer leeren Regel (*Rule0*) erstellt. Nachdem die Grammatik erzeugt wurde, kann man den Graphen und die miterstellte Regel vervollständigen oder auch die neuen Regeln definieren. Es ist auch möglich, die automatisch erzeugten Namen der Komponenten zu ändern.

Man kann eine Regel vervollständigen, in dem man in dem linken Fenster den Namen, der diese Komponente repräsentiert, anklickt. Rechts werden dann zwei Editoren angezeigt, einer für die linke und einer für die rechte Seite der Regel.

Der graphische Editor bietet eine Menge von graphischen Darstellungen an, um Knoten und Kanten zu repräsentieren. Bei einem Knoten kann man die Rechtecke, Rechtecke mit runden Ecken, Ovale und Kreise wählen.

Dagegen können die Kanten als gepunktete, gestrichelte oder durchgezogene Linie dargestellt werden. Sowohl für die Knoten als auch für die Kanten kann man verschiedene Farben auswählen. Man kann auch einen bestimmten Typnamen für die Identifizierung eines Objektes definieren, aber es ist nicht notwendig.

4.3 Editieren von Graphen

Wenn der Benutzer einen Arbeitsgraphen zeichnen möchte, soll er, nachdem die Kanten- und Knotenarten definiert wurden, in den Zeichenmodus wechseln. Das kann erfolgen, in dem man den Menüeintrag "*Draw*" in dem Popupmenü auswählt. Es gibt noch eine weitere Möglichkeit um in den Zeichenmodus zu gelangen. Man kann direkt im Grapheditor mit der rechten Maustaste "*Edit Mode*" öffnen und da den Menüeintrag "*Draw*" auswählen. Danach ist es möglich die Knoten zu zeichnen in dem die linke Maustaste angeklickt wird. Um eine Kante darstellen zu können, wird erst mal ein Knoten als die Quelle und dann der andere als das Ziel angeklickt.

In dem Selektionsmodus kann man ein oder mehrere Objekte gleichzeitig verschieben, löschen, duplizieren oder mit den Attributen versehen.

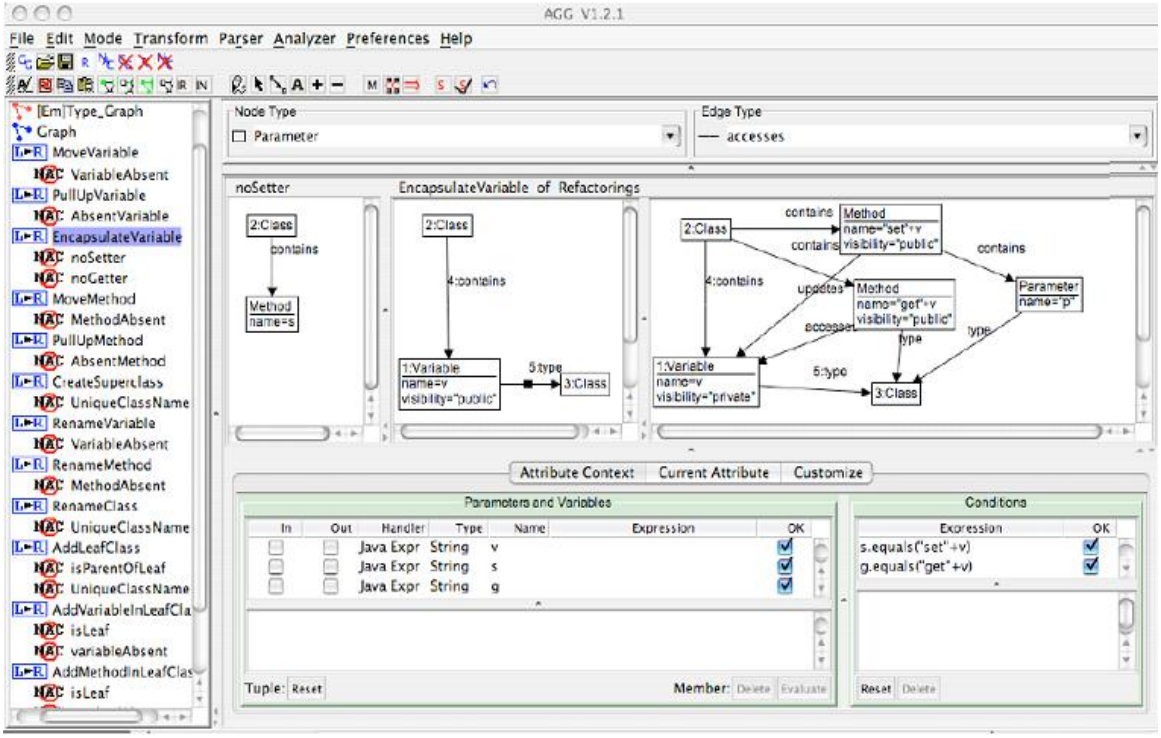

Abbildung 21. *AGG - Tool*

4.4 Editieren einer Regel

Beim Editieren einer Regel ist es empfehlenswert zuerst die Objekte von der linken Seite der Regel zu zeichnen und dann den Menüeintrag "*Identic-*

Rule“ im “*Edit*“ Menü auszuwählen. Auf dieser Weise werden die Objekte von der linken Seite in die rechte Seite der Regel kopiert und ein identischer Graphmorphismus wird generiert. Das kann man daran erkennen, dass jedes Objekt auf der linken Seite und seine Abbildung auf der rechten Seite mit der gleichen Zahl gekennzeichnet wird. Wenn man die Objekte löscht oder ändert, wird auch der Graphmorphismus geändert. Für das gelöschte oder geänderte Objekt wird auch das *Mapping* gelöscht. Wenn man neue Objekte dazu erzeugen möchte, kann man auch manuell die *Mappings* setzen. Es gibt zugleich auch die Möglichkeit, den ganzen Graphmorphismus benutzerdefiniert zu vervollständigen. Das geschieht im *Map*-Modus. Die Objekte, die man abbilden möchte, müssen einzeln zuerst auf der linken Seite und dann ihr Abbildungen auf der rechten Seite anklickt werden.

Analog kann eine negative Anwendungsbedingung erzeugt werden. Zuerst muss man im Strukturbaum auf der linken Seite den Namen einer Regel zu der man einen NAC definieren möchte mit der rechten Maustaste anklicken. In dem Menü, welches dann erscheinen wird, selektiert man den Menüeintrag: “*New NAC*“. Zu der Regel wird eine neue NAC hinzugefügt. Wenn sie angeklickt wird, wird sie in dem NAC Editor geladen.

4.5 Attributdefinition in einem Graphobjekt

Wenn man ein Attribut für ein Objekt definiert, muss man einen Typen, einen Namen und einen Wert festlegen. Die Attribute in dem Startgraph müssen konstante Werte besitzen.

In einer Regel können die Attributwerte als Konstanten, Variablen oder Ausdrücke definiert werden. Mit den Ausdrücken ist gemeint, dass man Methodenaufrufe auf einem Objekt oder Funktionen über dem Attributswert, wie zum Beispiel: *nextYear = x +1*, definiert. Allerdings darf man auf der linken Seite der Regel keine Ausdrücke definieren. Hier werden nur Konstanten und Variablen erlaubt. Jede Variable in einer Regel ist global definiert. In der rechten Seite einer Regel sind auch *Java*-Ausdrücke erlaubt. Sie können auch die Variablen enthalten, die in der linken Seite der Regel deklariert wurden. In der rechten Seite der Regel kann es zwei spezielle Arten von Variablen geben: Input- und Outputparameter.

In zwei Fällen werden die Attribute gebraucht: bei den Regeln und bei dem

Startgraphen. Der Attributseditor erlaubt einen Typen, einen Namen und einen Wert für ein Attribut zu deklarieren. Man kann die Sichtbarkeit eines Attributes bestimmen. Wenn man das Kontrollkästchen "*Shown*" wählt, werden alle deklarierten Attribute in der graphischen Darstellung einer Regel oder des Startgraphen angezeigt.

Ein Graphobjekt kann mehrere Attribute besitzen. Die Attributbehandlungsroutine interpretiert die Ausdrücke in *Java*-Syntax. Die Attributstypen können entweder als primitive *Java*-Typen (wie zum Beispiel: *byte*, *short*, *char*) oder *Java*-Objekte deklariert werden.
Abgesehen von der Java Standardbibliotheken kann man weitere benutzerdefinierte Klassen für die Attributierung benutzen. In dem Fall ist es nötig die Klassen in den *CLASSPATH* aufzunehmen. Wenn man ein *Package* benutzt muss man das zu dem passenden Editor hinzufügen.

In AGG erfolgt die Attributierung von Objekten in einem speziellen Editor. Es gibt mehrere Möglichkeiten diesen Editor aufzurufen. Die einfachste Methode ist es, den gewünschten Knoten oder die gewünschte Kante anzuklicken und den Menüeintrag "*Attributes*" auszuwählen. Eine andere Möglichkeit ist das gewünschte Graphobjekt zu selektieren und im Popupmenü "*Edit*" den Menüeintrag "*Attributes*" zu wählen. Die letzte Möglichkeit, den Editor aufzurufen, ist es zu dem Modus in "*Mode*" zu wechseln und das Objekt, dessen Attribute editiert werden sollen, anzuklicken.

Der Attributeditor ist geteilt in drei Editoren unterteilt: "*Attribute Context*", "*Current Attribute*", "*Customize*", welche selektiert werden können, in dem man auf die entsprechende Registerkarte anklickt. Wenn man das erste Mal den Attributseditor startet, wird "*Current Attribute*" Editor selektiert, welcher in zwei Teile getrennt wird. Oben gibt es eine Tabelle, mit den eingetragenen Attributen. Ein Attribut ist ein *Member* und mehrere Attribute bilden ein *Tupel*.

4.6 Interpretation und Debugging

In diesem Abschnitt werden verschiedene Modi für die Regelapplikation erläutert.

Der erste Modus zur Anwendung einer Regel wird *Debug*-Modus genannt. Hier wird eine selektierte Regel exakt einmal auf dem Startgraphen angewandt.

Der *Matchmorphismus* im *Debug*-Modus kann vom Benutzer analog zum Regelmorphismus definiert werden. Aus dem Popupmenü "*Transform*" wählt man den Menüeintrag "*MatchDe*f" und definiert einen *Matchmorphismus,* in dem man zuerst die Objekte von der linken Seite der Regel und dann ihre Abbildungen in dem Arbeitsgraphen anklickt. Natürlich müssen die entsprechenden Graphobjekte den gleichen Typ haben und es muss eine konsistente Abbildung auf die Attribute vorhanden sein. Andere Möglichkeiten, um eine Abbildung zu definieren, ist es einfach die *Map*-Option aus dem "*Mode*"–Menü für ein Graphobjekt auszuwählen und seine Abbildung in dem Startgraphen anzuklicken. Das manuelle Erstellen der Abbildungen kann mühsam werden. Aus dem Grunde erlaubt das AGG–System, einen partiellen Match zu vervollständigen. Um das zu erreichen wählt man den Menüeintrag "*NextCompletion*" in dem "*Transform*"-Menü. Der partielle Match wird dann zu einem totalen ergänzt.

Wenn es mehrere Möglichkeiten gibt, den Match zu vervollständigen, wird eine von denen zufällig ausgewählt. Wenn man "*NextCompletion*" nochmal aufruft, wird eine weitere Ergänzung gefunden, falls sie vorhanden ist. Auf diese Weise werden alle Ansätze einer Regel gefunden und angezeigt.

Wenn man den ganzen Match definiert hat, wählt man den Menüeintrag "*Step*" im "*Transform*"-Menü um die Regel einmal auf dem Arbeitsgrafen auszuführen. Der transformierte Graph wird in dem Grapheditor angezeigt. Danach kann man eine andere Regel auswählen und den ganzen Prozess auf dem neuen Graphen wiederholen.

Außerdem ist es möglich, Matchbedingungen festzulegen: zum Beispiel injektiv oder nicht injektiv, mit NACs oder ohne u.s.w.. Diese Optionen erlauben es die Matchstrategie an die Graphgrammatik anzupassen.

5. Anforderungen

Das AGG-System stellt die Umgebung dar, die im Rahmen der Diplomarbeit angepasst und erweitert werden sollte, um das Konzept der Amalgamierung umsetzten zu können. Einige Einschränkungen werden im Folgenden erläutert.

Covering stellt eine Konstruktion in der amalgamierten Graphtransformation dar und besteht aus einem Instanzschemata (IRS) und einer Menge von Matches von allen Produktionen in IRS. Es gibt verschieden Methoden zur Beschreibung des *Coverings.* Es gibt mehrere Ansätze der *Covering*–Konstruktion, die wichtigsten sind *local all* und *fully synchronized Covering*. Eine genaue Beschreibung der Definitionen ist in [Taen04] zu finden.

Nur der erste Ansatz soll realisiert werden. Die Umsetzung der Idee basiert auf der im zweiten Kapitel vorgestellten Konstruktion der amalgamierten Regel, also einer Konstruktion mit einer Unteregel und mehreren Erweiterungsregeln.

5.1 Basiskomponente für die Amalgamierung

Die Konstruktion der amalgamierten Regel, sowie die Durchführung der amalgamierten Graphtransformation muss innerhalb der Basiskomponente zur Verfügung gestellt werden.

Die Regelschemen sollen die Abhängigkeiten zwischen Erweiterungsregeln und einer Unterregel abbilden. Die Abhängigkeiten werden durch Morphismen ausgedrückt. Die Voraussetzung für die Regelschemakonstruktion ist, dass ein Regelschema wenigstens eine Unterregel besitzen muss, deswegen sollte mit der Erstellung von *RuleScheme*–Instanz eine Instanz von der *BasicRule*-Klasse miterzeugt werden. Es ist jedoch nicht notwendig, dass ein Regelschema eine Erweiterungsregel hat. In dem Fall, wo keine Erweiterungsregeln definiert wurden, muss die Unterregel wie ganz einfache Regel ausgeführt werden können.

Die Implementierung sollte auch eine AGT-Graphgrammatik definieren, die eine Art Behälter für eine Menge von Regelschemen darstellt. Das ist ein Hauptunterschied zu einer einfachen Graphgrammatik die aus Regeln besteht.

5.2 GUI – Erweiterung

Aufgrund der konzeptionellen Erweiterungen muss die bestehende graphische Oberfläche des AGG-Systems erweitert werden. Die neue Oberfläche sollte die Möglichkeit haben, die neuen AGT–Graphgrammatiken zu erstellen und sie zu laden. Ein Navigationsbaum innerhalb einer Grammatik muss das Hinzufügen, Modifizieren und Löschen von Regelschemen ermöglichen.

Das System muss eine Reihe von kritischen Fällen, wie zum Beispiel das Löschen einer Unterregel, identifizieren und abfangen können.
In dem genannten Beispiel, dem Löschen einer Unterregel werden viele umgebene Elemente beeinflusst. Sowohl bei einer Änderung als auch beim Entwerfen einer Unterregel werden die dazugehörigen Erweiterungsregeln zerstört. Ein Reparierungsmechanismus muss bei einer Modifikation durch den Benutzer definiert werden. Jedes Mal wenn die Unterregel geändert wurde, sollen erfolgen, die existierenden Erweiterungsregeln anpasst werden.

Wie im *Kapitel 1* erwähnt, gehören amalgamierte Graphtransformationen zu den parallelen Graphersetzungskonzepten, die auf dem *Doppelt Pushout - Ansatz (DPO)* basieren. DPO wird genauer in [Ehr79, EPS73] beschrieben. Die Voraussetzung für DPO ist die *dangling-condition*.
Die entsprechenden Einstellungen müssen am Anfang gesetzt werden, um den Amalgamierungsprozess überhaupt durchführen zu können. Bei der Benutzung der jetzigen Benutzeroberfläche wird die *dangling–condition* (Klebebedingung) als *default* gesetzt.

Dangling verbietet die Entstehung von hängenden Kanten. Wenn man diese Bedingung benutzen würde, könnte man den Amalgamierungsprozess nicht durchführen, weil bei der Erstellung von einer amalgamierten Regel durchaus hängenden Kanten vorkommen können. Aus diesem Grund muss die *dangling*–Einstellung in der GUI erst zum späteren Zeitpunkt, kurz vor dem Anwenden der amalgamierten Graphtransformation gesetzt werden.

Es gibt noch einige Anforderungen, die sich auf die GUI beziehen und, die man unbedingt beachten sollte. Die erste Einschränkung bezieht sich auf die amalgamierte Regel selbst. Die amalgamierte Regel darf nur einmal angewendet werden, weil ihre Erstellung streng von mehreren Ansätzen der Erweiterungsregeln in dem Arbeitsgraphen abhängt und die weitere

Anwendung zu einem inkonsistenten Zustand führen würde. Die amalgamierte Regel soll nach einer Transformation gelöscht werden.

Eine weitere Einschränkung bezieht sich auf partielle Matches für den Ansatz der amalgamierten Regel. Ein solcher Match soll ausschließlich auf der Unterregel manuell gesetzt werden können.

6. Entwurf und Implementierung

Der Entwurf und die Implementierung des *AGT-Package* beziehen sich auf die Erweiterung des existierendem *Package*: *xt_basis*. *AGT-Package* stellt die Basis für die amalgamierten Graphtransformationen dar.
Das *xt_basis* Package umfasst die Grundklassen des AGG–Systems. Hier werden solche Ansätze wie Knoten, Kanten, Graphen, Regeln, Graphmorphismen, etc. implementiert. Die neu zu definierenden Klassen vererben entweder die Klassen, die in *xt_basis* definiert sind, oder benutzen diese. Die genauen Beziehungen (Assoziationen, Vererbungen) sind in der *Abbildung 22* dargestellt.

6.1 Entwurf

Im Folgenden wird jede Klasse aus dem während der Diplomarbeit erweiterten AGT–Package genau beschrieben. Die Beziehungen innerhalb des *Packages* und zum *xt_basis*–Package, werden vorgestellt.

In dem *Unterkapitel 5.2* werden die Maßnamen zur GUI–Erweiterung vorgestellt. Die vorhandene Oberfläche musste für die amalgamierten Graphtransformationen musste entsprechend anpasst werden. Jede Änderung in schon existierenden AGG–Klassen wurde dokumentiert. Natürlich musste man auch neue Klassen dazu implementieren und damit *Editor*- und *GUI*– Package erweitern. Die neuen Klassen, die entwickelt wurden, sind: *EdRuleScheme.java* und *RuleSchemePopupMenu.java*.

6.1.1 Package AGT

Das *Package* AGT besteht aus sieben Klassen: AGTFactory, AGTGraGra, AGTGraTra, BasicRule, ExtendedRule, Covering und RuleScheme. (siehe *Abbildung 22*). Im Folgenden wird jede genau beschrieben.

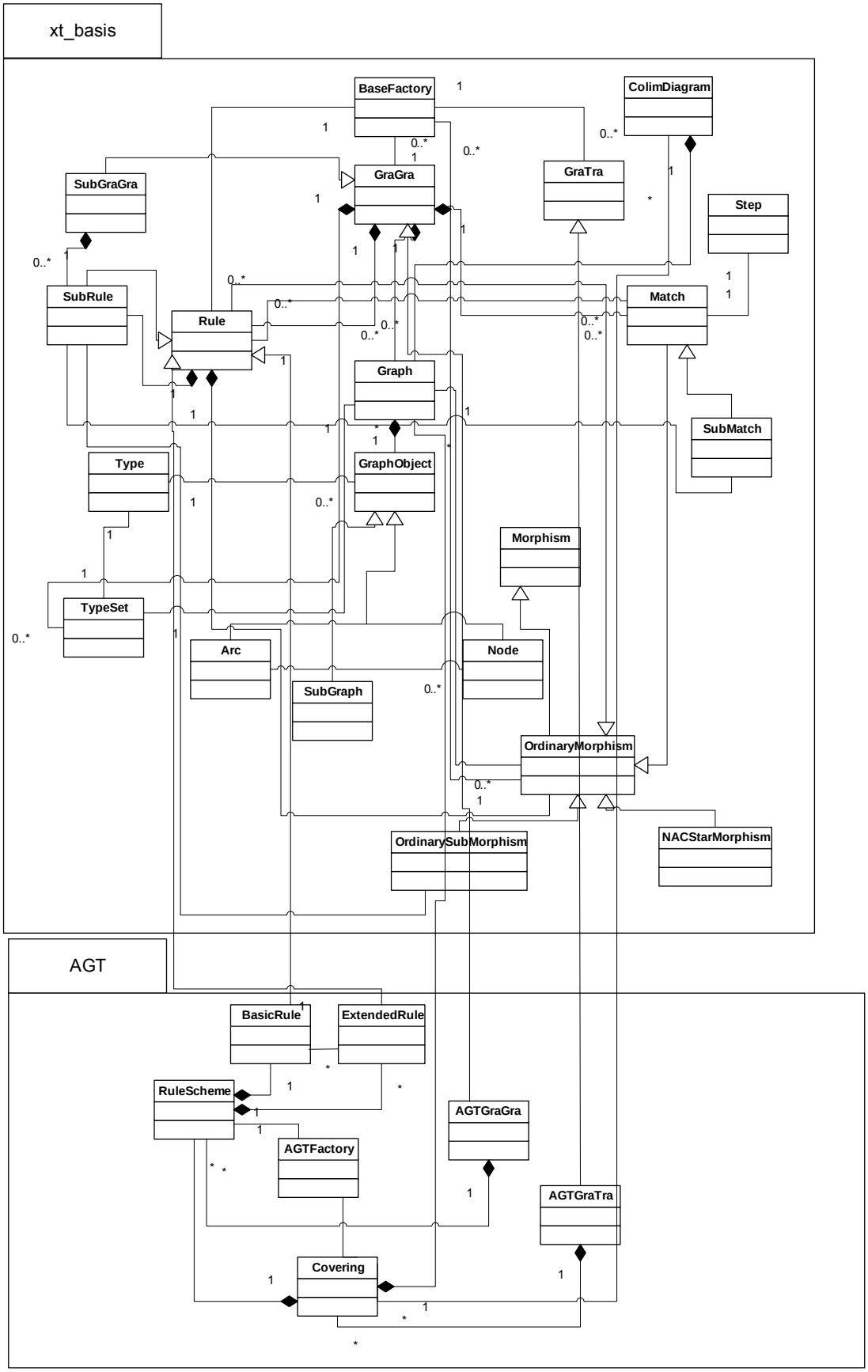

Abbildung 22. *AGT und xt_basis Packages*

BasicRule

BasicRule definiert die Unterregel eines Regelschemas. Sie erbt von der *Rule*-Klasse, die im *Package AGG* implementiert ist. Sie implementiert auch die *Observer*-Schnittstelle. Außer Methoden, die von der Klasse *Rule* geerbt wurden, stellt *BasicRule* die Methoden zur Verfügung, die eine Beziehung zu ihrem eigenem Regelschema (*Rulescheme*) herstellen. Die implementierte *Observer*-Schnittstelle erlaubt jede Änderung der Unterregel festzustellen, was hilfreich ist, um die Einbettungsmorphismen nachzubilden falls die Unterregel von dem Benutzer geändert wurde, nachdem die Erweiterungsregel bereits erstellt wurden.

setRuleScheme() – Diese Methode baut die Verbindung zwischen der Unterregel und ihrem Regelschema auf.

Die Methode *getRuleScheme()* ist hilfreich, wenn man dieses Regelschema abfragen möchte.

hasChanged() wird immer dann auf *true* gesetzt, wenn die Unterregel geändert wurde und wenigstens eine Unterregel vorher erstellt wurde. Wenn so ein Fall auftritt, wird die Methode updateExtendedRules() in der *RuleScheme*–Klasse aufgerufen, um die Erweiterungsregel zu reparieren und die Einbettungen wiederherzustellen.

setChanged() – Diese Methode erlaubt auch außerhalb der *BasicRule*- Klasse auf die *isChanged*–Variable zuzugreifen. Es ist dann nützlich, wenn man die AGT- Grammatik laden möchte und verhindern möchte, dass die Einbettungsmorphismen neu erstellt werden. Die *Abbildung 23* zeigt die *BasicRule*–Klasse mit ihren Attributen und Methoden.

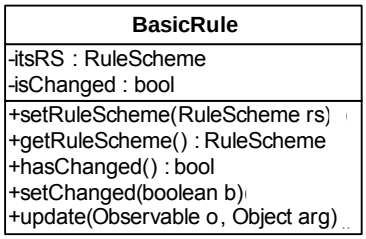

Abbildung 23. *BasicRule*

ExtendedRule

ExtendedRule definiert eine Erweiterungsregel. Sie erbt auch von der *Rule*-Klasse.

Die Klasse definiert die linken und rechten Einbettungsmorphismen. Diese werden schon beim Erstellen in der RuleScheme-Klasse vervollständigt. Die direkte Verbindung zu dem eigenen Regelschema gibt es nicht. Sie wird über die *BasicRule*-Instanz gewonnen.

Die *ExtendedRule*-Klasse bietet auch die Behälter an, die definiert wurden, um die eigenen Objekte unabhängig von den Objekten, die durch die Einbettungsmorphismen mit der Unterregel in der Beziehung stehen, zu unterscheiden. Diese werden gebraucht, falls die Einbettungsmorphismen bei der Änderung der *BasicRule* zerstört werden. In dem Fall wird die *updateEmbedding()*–Methode aufgerufen. Sie ruft wiederum drei weitere Methoden auf: *updatePairs()*, *markMappings()*, *updateExtRule().* Die erste checkt ab ob alle Objektpaaren in beiden Vektoren: *objectPairsLeft* und *objectPairsRight* eingetragen wurden, falls eine Paar fehlt, wird sie hinzugefügt. Die erwähnten Vektoren besitzen Objekte der Klasse: *Pair.* Sie definiert Paare von Objekten der Einbettungsmorphismen. Detaillierter gesagt wird ein Paar für jedes Graphobjekt aus einer Erweiterungsregel mit seinem Urabbild gebildet, falls solche Urabbildung nicht existiert, wird das Graphobjekt mit *null* verbunden. So kann man unterscheiden, welche Graphobjekte zu der Einbettung gehören und welche nicht.

markMappings() wurde definiert um sich die *Mappings* zwischen den eigenen Objekten in der Unterregel zu merken um sie dann nachbilden zu können.

In der Methode *updateExtRule()* werden die eigenen Graphobjekte gemerkt, bevor alle Graphobjekte aus dem linken und rechten Graphen der Erweiterungsregel gelöscht werden. Außerdem werden hier die Methoden *makeIsoCopy()* und *createNodes*() implementiert. Die erste Methode fügt isomorphe Kopien von der geänderten Unterregel zu der Erweiterungsregel. In diesem Moment ist die Erweiterungsregel mit der Unterregel identisch und neue Einbettungsmorphismen sind vorhanden.

In dem nächsten Schritt bildet die *createNodes*()–Methode, die vorhin gemerkte erweiterungsregeleigene Knoten hinzugefügt und danach auch die Kanten. Zuletzt wird *reconstuctMappings()* aufgerufen. Diese Methode braucht man, um

die *Mappings* zwischen eigenen Graphobjekten zu rekonstruieren. Es ist kompliziert für die Objekte die sich nicht in den Einbettungsmorphismen befinden. Diese werden erstmal mit ihren neuen Instanzen als Paare in den Vektoren *oldNewLeft* und *oldNewRight* eingetragen. Dies ist notwendig, wenn man fehlende Kanten nachtragen möchte. Man findet das Urabbild des alten Objektes in der Unterregel und dann die Abbildung davon in der Erweiterungsregel, um den Ziel- oder Quellknoten für die entsprechende Kante zu finden. Die beiden Vektoren sind auch hilfreich, wenn man die NAC–Morphismen reparieren möchte, was in der Methode *reconstructNacMappings()* geschieht. Wenn während des Anpassungsprozesses, die linke und rechte Graphen der Erweiterungsregeln, erst mal ganz gelöscht werden, gehen die Instanzen verloren und genauso werden die NAC–Morphismen zerstört. Weil wir die alten und neuen Objekte gemerkt haben, ist es möglich, die Morphismen nachzubilden.

Es gibt noch eine Reihe von Methoden, die auf die Klassenattribute zugreifen, sie abfragen oder ändern, wie zum Beispiel *getBasicRule()*, die die Unterregel liefert die zu dem gleichen Regelschema gehört.

Es gibt auch Möglichkeit, die Einbettungsmorphismen zu setzen. Dazu dienen zwei Methoden: *setEmbbededMorphLeft()* und *setEmbbededMorphRight().* Die werden zum Beispiel verwendet, wenn man eine Erweiterungsregel erstellt oder wenn man die Erweiterungsregeln nach der Änderung der Unteregel anpassen möchte.

getOldNewObjects() erlaubt die Abfrage von alten und neuen Objekten die als Paare nach der Anpassung von Einbettungsmorphismen, eingetragen wurden. Die *Abbildung 24* zeigt die Definition von der *ExtendedRule*–Klasse:

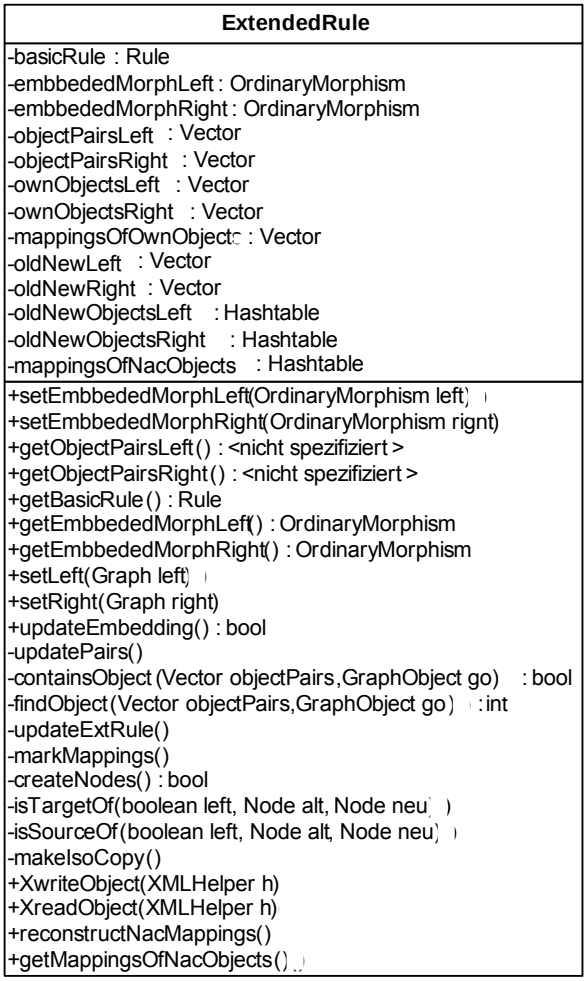

Abbildung 24. *ExtendedRule*

RuleScheme

In der Klasse *RuleScheme* wird die Struktur eines Regelschemas dargestellt. Beim Anlegen eines Objektes dieser Klasse wird automatisch ein Objekt der *BasicRule* Klasse erstellt. Danach können weitere Objekte der *ExtendedRule*-Klasse hinzugefügt werden. Beim Erstellen der Erweiterungsregel in der *createExtendedRule()* werden die Einbettungs-morphismen erzeugt. Das erfolgt, in dem die *isomorphicCopy()*–Methode auf linken und rechten Graphen der Unterregel aufgerufen wird. Diese Methode ist in der *Graph*–Klasse implementiert, die in der *AGG*–Bibliothek definiert ist. *isomorphicCopy()* erzeugt eine genaue Kopie eines Graphen und einen isomorphen Morphismus zwischen den Original und der Kopie.

Nachdem der linken und der rechten Graph der Erweiterungsregel erstellt wurde, muss man noch der Regelmorphismus nachbildet werden.

updateExtendedRules() wird im Fall der Änderung von der Unterregel ausgeführt und ruft für jede Erweiterungsregel die in *ExtendedRule*–Klasse definierte *updateEmbedding()*-Methode auf.

Die neu erstellte amalgamierte Regel wird der *RuleScheme*-Klasse zugewiesen. Hier gibt es die Möglichkeit sie und ihr Match mit den Methoden: *destroyAmalgamatedRule()* und *destroyAmalgamatedMatch*() zu löschen.

Außerdem ist es möglich, mit dem folgenden Methoden die beiden Attribute abzufragen und zu setzen: *getAmalgamatedRule(), setAmalgamatedRule(), getAmalgamatedMatch(), setAmalgamatedMatch().*

Mit der *getBasicRule()* Methode kann man auf die Unterregel dieses Regelschemas zugreifen. Die analoge *getExtendedRules()*–Methode liefert alle Erweiterungsregeln, die zu diesem Regelschema gehören.

Die *isValid()*-Methode fragt ab, ob das Regelschema gültig zum Ausführen des Amalgamierungsprozesses ist.

Die *XwriteObject()* und *XreadObject()* sind zum Speichern und Laden eines Objektes der RuleScheme–Klasse definiert. Die *Abbildung 25* zeigt die *RuleScheme*–Klasse in der UML–Notation.

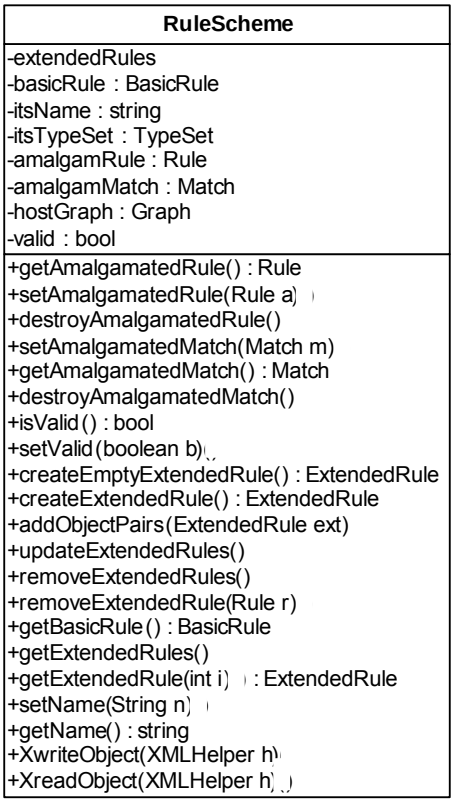

Abbildung 25. *RuleScheme*

Covering

Die eigentliche Amalgamierung findet in der Klasse *Covering* statt.
Diese Klasse startet mit einem *RuleScheme* und einem Arbeitsgraphen, auf dem die Anwendung der amalgamierten Regel erfolgen soll. Der Arbeitsgraph ist eine Instanz der *Graph*–Klasse, die im *xt_basic* Package definiert ist.
In der *Covering*–Klasse wird ein Instanzschema aufgebaut und die Matches und Morphismen werden erzeugt. (siehe *Abbildung 12*)
Die nächste Abbildung zeig die Covering–Klasse mit ihren Attributen und Methoden in der *UML*–Notation.

Abbildung 26. *Covering*

Der Amalgamierungsprozess beginnt mit dem Aufruf der *amalgamate()* Methode. Sie wiederum führt eine Reihe von weiteren Methoden aus. Der Ablauf wird in dem folgenden Aktivitätsdiagramm (*Abbildung 27*) dargestellt:

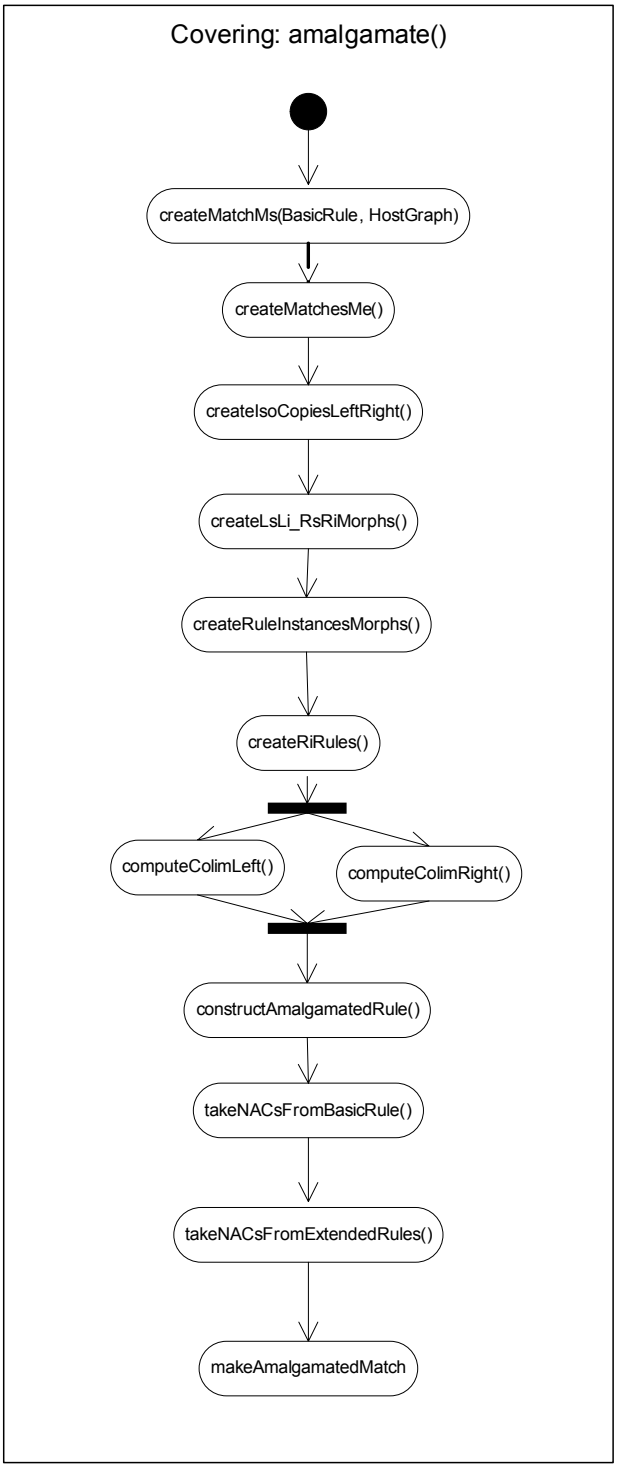

Abbildung 27. *Covering: amalgamate()*

Zuerst wird die *createIsoCopiesLeftRight()*–Methode aufgerufen, die dann zwei andere Methoden *createMatchMs()* und *createMatchesMe()* aufruft.

Die *createMatchMs( )*–Methode erstellt den *Match Ms* (siehe *Abbildung 28*). Einfacher ausgedrückt, findet sie einen Ansatz der Unterregel.

In der *createMatchesMe()*–Methode werden alle Ansätze aller Erweiterungsregeln gefunden, so genannten *Matches ME* (siehe auch *Abbildung 12*).

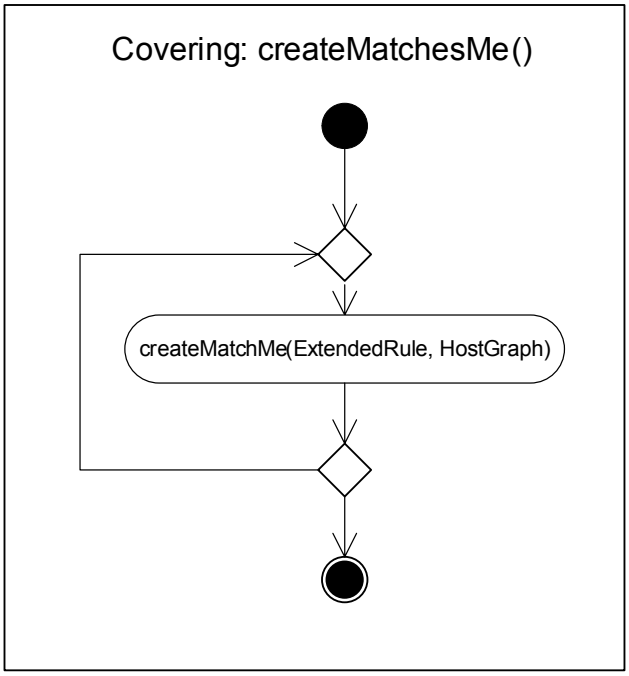

Abbildung 28. *Covering: createMatchesMe()*

Nachdem die beiden Methoden erfolgreich ausgeführt wurden, werden in der *createIsoCopiesLeftRight()*-Methode isomorphe Kopien von dem linken und rechten Graphen jeder Erweiterungsregeln erzeugt, welche die linke und rechte Seite der Instanzregeln darstellen. Von jeder Erweiterungsregel werden so viele Kopien gemacht, wie viele Ansätze im Arbeitsgraphen vorhanden sind.

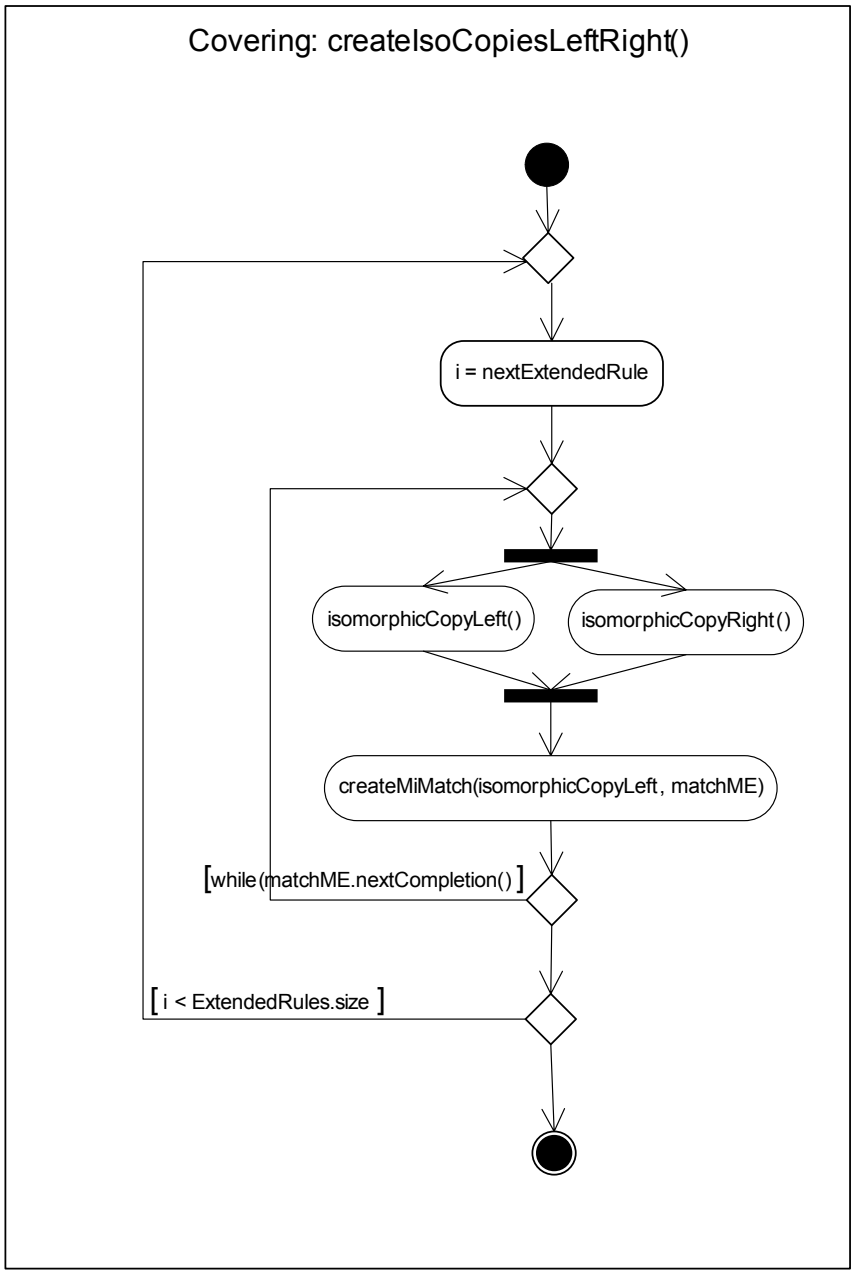

Abbildung 29. *Covering: createIsoCopiesLeftRight()*

Die *createLsLi_RsRiMorphs()*–Methode bildet die Einbettungsmorphismen zwischen der Unterregel und jeder Instanzregel. Diese werden später zur Berechnung von COLIM-Diagrammen benötigt. Das folgende Aktivitäts-diagramm (siehe *Abbildung 30*) zeigt den Ablauf der Methode.

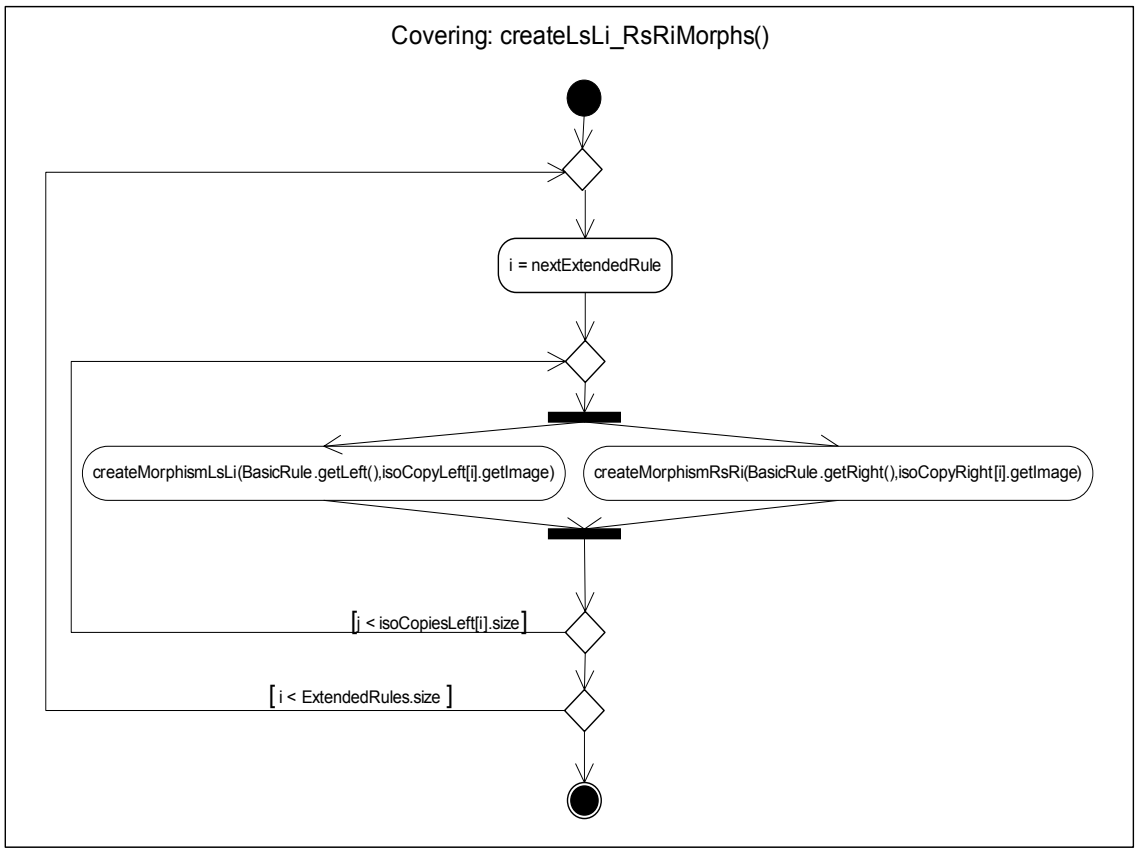

Abbildung 30. *Covering: createLsLi_RsRiMorphs()*

Sehr wichtig ist die nächste Methode: *createRuleInstancesMorphs()*. Sie erstellt die Regelmorhismen für die Instanzregeln. Hier werden auch Attributbedingungen, Variablen, Konstanten, Expressions für jede Instanzregel von der Unterregel und ihrer Erweiterungsregel übernommen. Hier erfolgt auch die Umbenennung von Variablennamen. Wenn eine Erweiterungsregel nur eine Instanz besitzt, ist die Umbenennung nicht notwendig. Für jede weitere Instanz wird die Variable umbenannt, indem zu ihrem Namen ein Index hinzugefügt wird. Der Index fängt mit eins an und wird inkrementiert. Das nächste Aktivitätsdiagramm (*siehe Abbildung 31*) stellt die Methode graphisch dar.

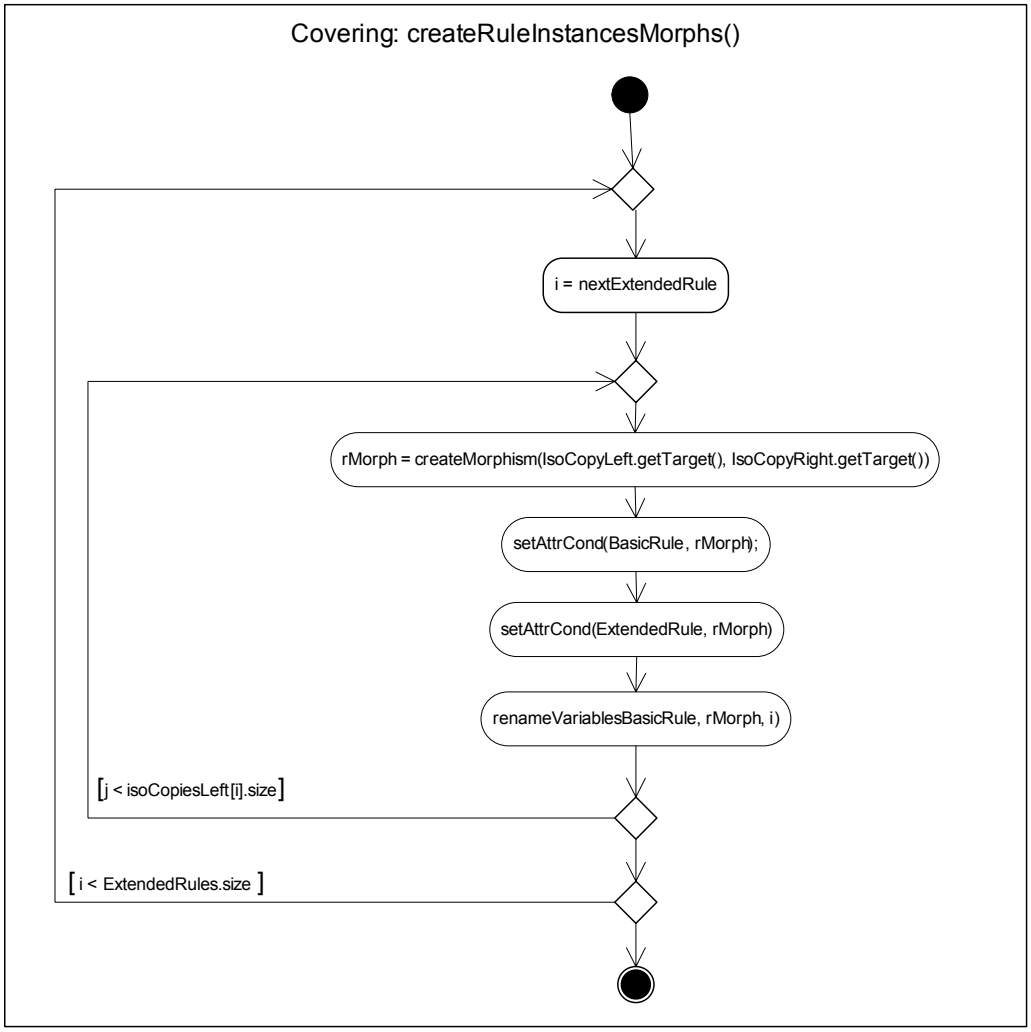

Abbildung 31. *Covering: createRuleInstancesMorphs()*

In der *createRiRules()*-Methode werden aus bereits erstellten Instanzregelmorphismen, mit Hilfe der *constructRuleFromMorph()*–Methode die Instanzregeln gebildet. Die *createRiRules()*-Methode wurde in der *Abbildung 32* dargestellt.

Die *constructRuleFromMorph()*–Methode ist in der *BaseFactory*–Klasse im *xt_basic-Package* definiert.

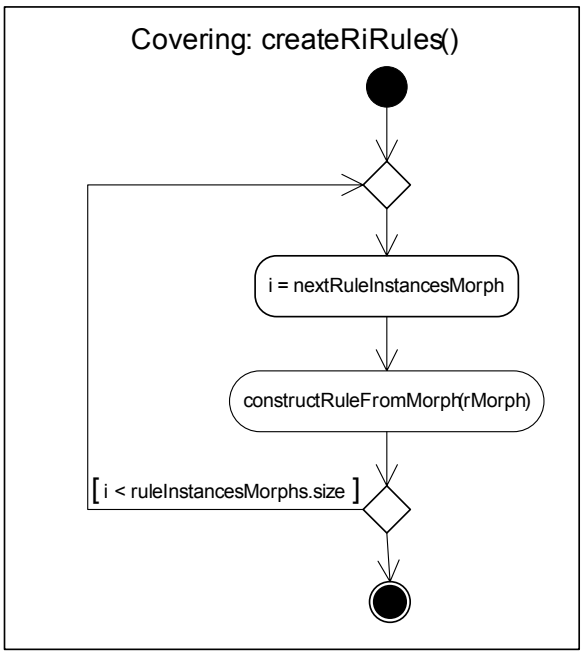

Abbildung 32. *Covering: createRiRules()*

Covering stellt auch die Methoden zu Verfügung, die den linken (*computeColimLeft()*) und rechten (*computeColimRight()*) Graphen der amalgamierten Regel berechnen (siehe *Abbildung 27*). Die beiden Methoden benutzen die im *xt_basis–Package* definierte Klasse: *ColimDiagram*. Die Klasse benutzt wiederum das *COLIM–Package*, welches zur Berechnung von Kolimitdiagrammen implementiert wurde.

Um Kolimitdiagramme zu berechnen, braucht man *COLIM*–Knoten und Kanten als Inputparameter. Bei der Berechnung von dem linken Graphen der amalgamierten Regel im Amalgamierungskonzept wurden zu dem COLIM–Diagram (eine Instanz von *ColimDiagram*–Klasse) durch die in der Klasse definierte *addNode()*-Methode folgende Knoten hinzugefügt: ein leerer Graph, der linke Graph von der Unterregel und die linken Graphen von jeder Instanzregel. Der leere Graph wird nach der Ausführung der *computeColimit()*-Methode zu dem resultierenden linken Graphen der amalgamierten Regel vervollständigt. Mit der Hilfe von der *addEdge(Graph g)*-Methode, die ebenso in der *ColimDiagram*–Klasse definiert wurde, addiert man die linken

Einbettungsmorphismen zwischen der Unterregel und jeder Instanzregel.

Nachdem die beiden Graphen der amalgamierten Regel erzeugt wurden, wird in der *constructAmalgamatedRule()*–Methode, der Regelmorphismus für die amalgamierte Regel erstellt. Die Mappings und der Attributkontext von jeder Instanzregel werden übernommen. Anschließend wird die amalgamierte Regel von dem erstellten Regelmorphismus erzeugt. Die nächste Graphik (siehe *Abbildung 33*) zeigt den Verlauf dieser Methode.

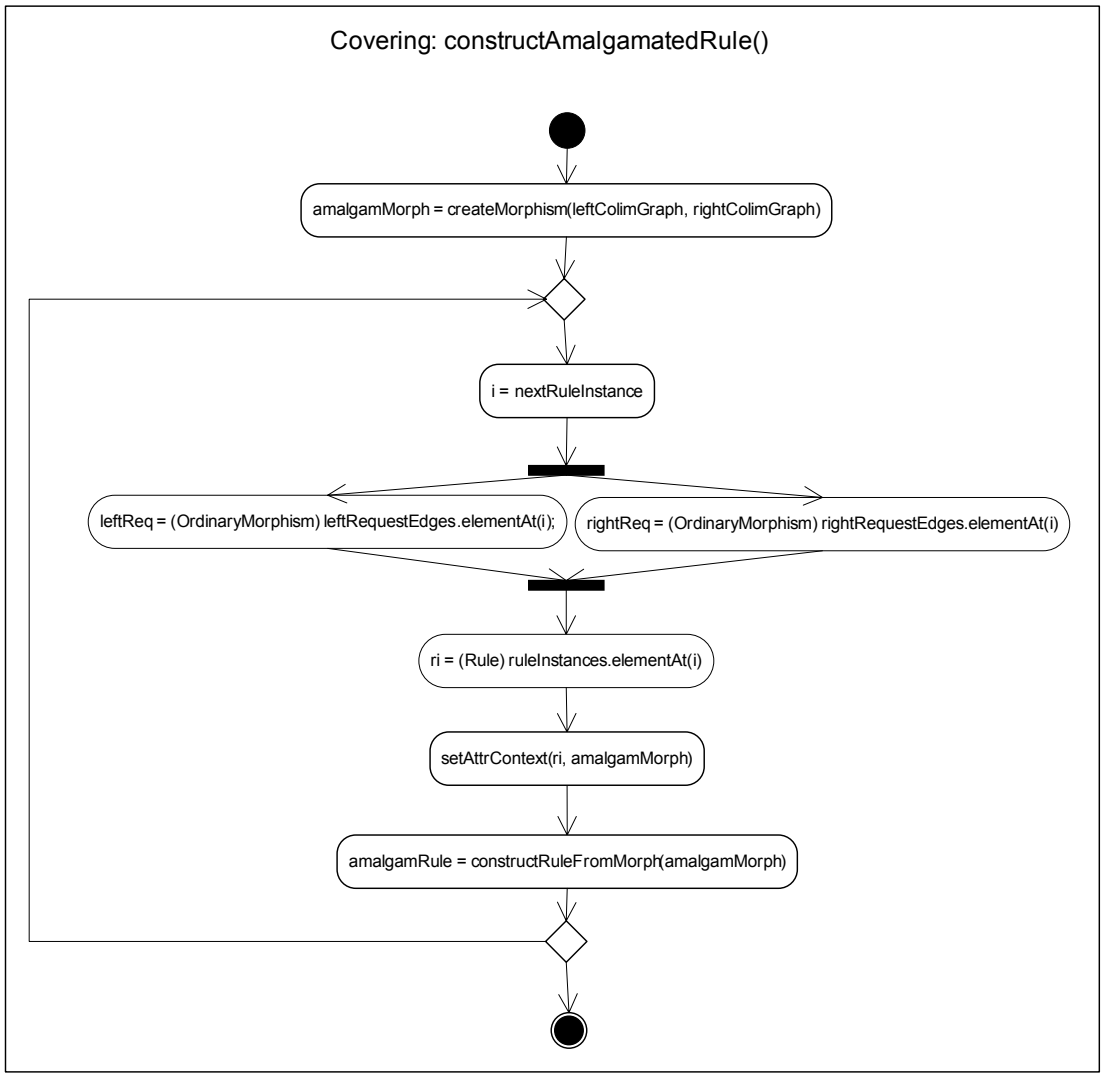

Abbildung 33. *Covering: constructAmalgamatedRule()*

Weitere Methoden: Die *takeNACsFromBasicRule()*-Methode und die *takeNACsFromExtendedRules()*-Methode übernehmen die NACs von der Unterregel und den Erweiterungsregeln. Das Hinzufügen der NACs von der

Unterregel zu der amalgamierten Regel ist ziemlich einfach. Es wird ein wenig komplizierter wenn die NACs von den Erweiterungsregeln übernommen werden sollen. Wie das genauer aussieht, wurde in dem *Unterkapitel 27* beschrieben und in den beiden Aktivitätsdiagrammen (*Abbildung 34, Abbildung 35*) dargestellt:

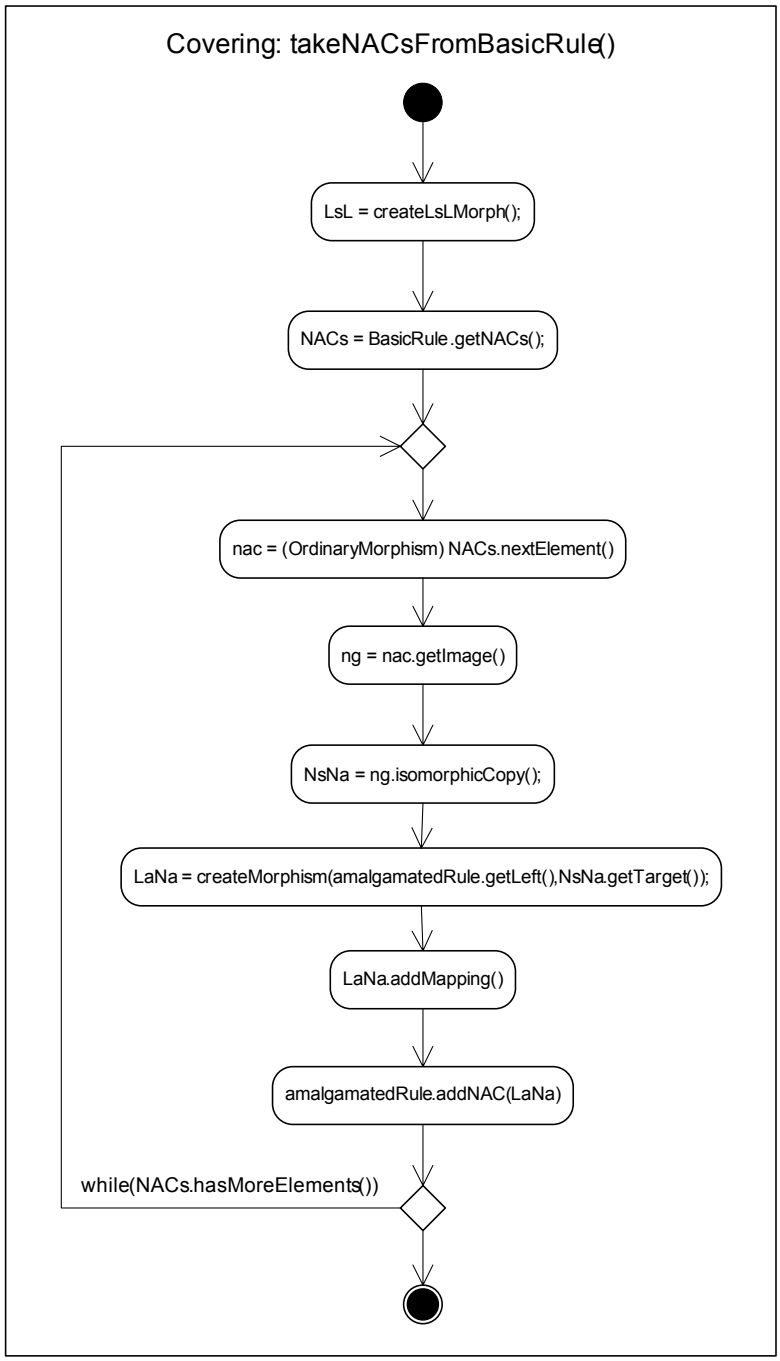

Abbildung 34. *Covering: takeNACsFromBasicRule()*

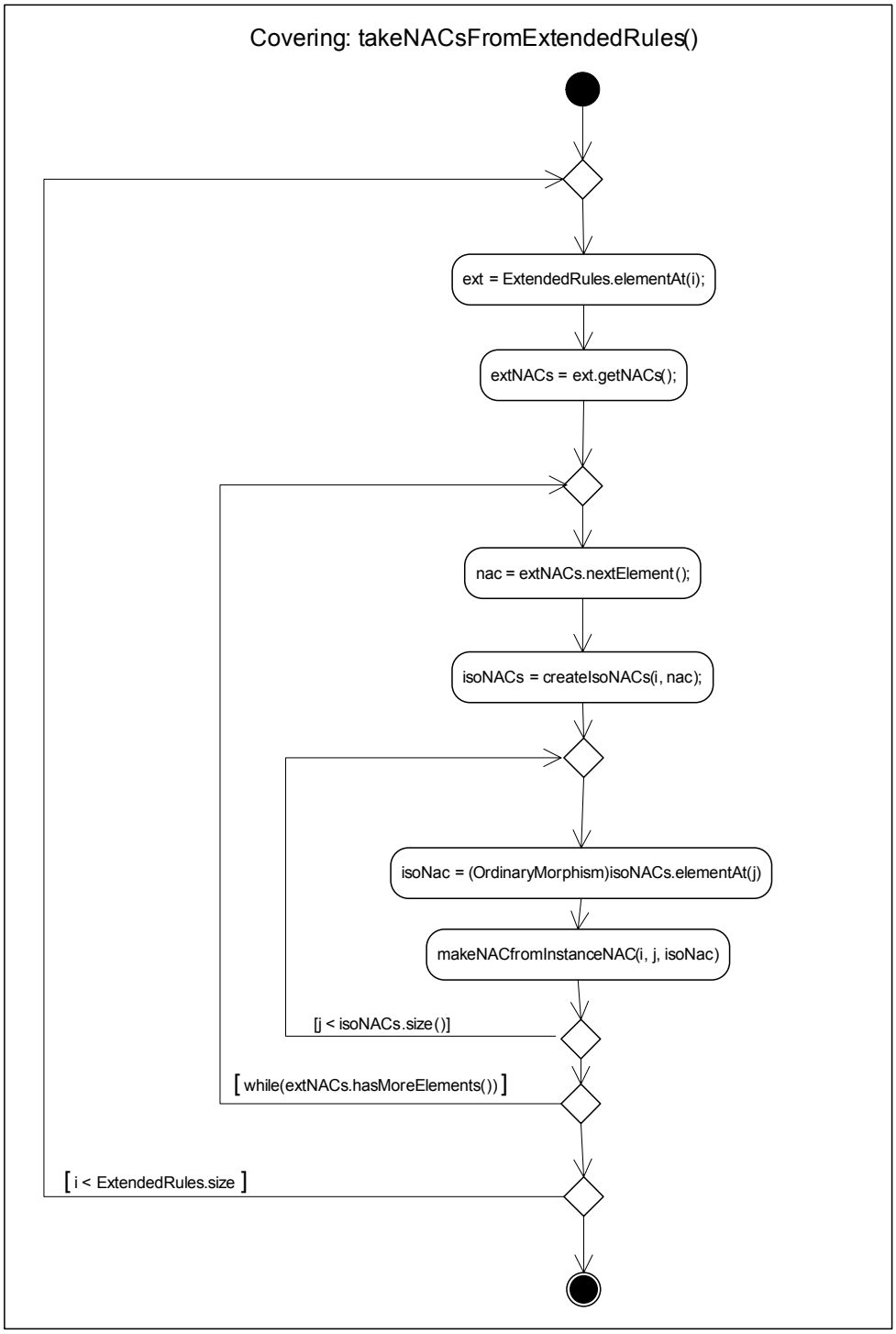

Abbildung 35. *Covering: takeNACsFromExtendedRules()*

Anschließend wird der Ansatz der amalgamierten Regel im Arbeitsgraphen berechnet. Dies passiert in der *makeAmalgamatedMatch()*-Methode. Hier wird für jede Instanzregel über den *Match Mi* : $Li \rightarrow G$ und *left reqest edge* (siehe *Abbildung 12*) die *makeDiagram()*-Methode aufgerufen. Sie ist in der *OrdinaryMorphism*–Klasse im *xt_basis–Package* definiert.

In der Covering Klasse wird die *dangling*–Bedingung in der graphischen Oberfläche am Anfang von *amalgamate()* zurückgesetzt und dann kurz vor der Ableitung der amalgamierten Regel wieder gesetzt. Es ist notwendig, um überhaupt den Amalgamierungsprozess durchführen zu können (siehe *Kapitel 5*).

Die *makeBasicRuleAsAmalgamatedRule()*-Methode wird aufgerufen, falls das Regelschema keine Erweiterungsregel hat oder diese nicht anwendbar sind. In dem Fall wird die *BasicRule*-Instanz der amalgamierten Regel zugewiesen und so ausgeführt. Dies hat sich als problematisch rausgestellt, weil die *GUI*-Klassen, die nicht als einfache Regel sehen konnten, was zu einer falschen Anzeige im Navigationsbaum geführt hat.

AGTGraGra

AGTGraGra stellt die Grammatik für die amalgamierte Graphtransformation dar. Sie erbt von der *GraGra*–Klasse, die im *xt_basis–Package* implementiert ist.

Die Superklasse verwaltet Regeln, dagegen ist meine *AGTGraGra*-Klasse eine Komposition von Regelschemen. Das bedeutet, dass zu der definierten Grammatik, keine Regeln hinzugefügt werden, sondern Regelschemen und zu denen erst die Regeln, genauer: eine Unterregel und Erweiterungsregeln.

Wenn eine Instanz der *AGTGraGra* – Klasse erzeugt wird, wird auch eine Instanz von der *RuleScheme*–Klasse miterzeugt. Die Instanz definiert ein Regelschema, dass in der GUI von dem Benutzer erweitert werden kann.

Die *AGTGraGra*–Klasse definiert Methoden um auf eigene Regelschema zu zugreifen, wie zum Beispiel: *getRuleSchemes()* oder um ein Regelschema zu erstellen: *createRuleScheme().*

Es werden auch die Methoden zum Speichern (*XwriteObject()*) und Laden (*XreadObject()*)der Grammatik und mit ihren Komponenten zur Verfügung gestellt.

Die folgende *Abbildung 36* zeit die *AGTGraGra*–Klasse mit ihren Attributen und Methoden.

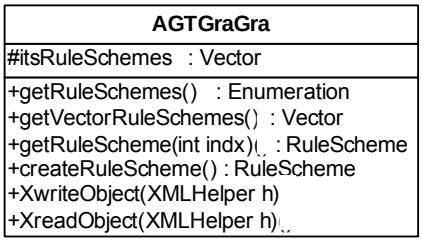

Abbildung 36. *AGTGraGra*

AGTGraTra

AGTGraTra erbt von der *GraTra*–Klasse, die ebenso im *xt_basis–Package* definiert ist. Die Hauptaufgabe dieser Klasse ist, den amalgamierten Match zu erstellen, anzuwenden oder zu löschen. Das passiert in der *apply()*–Methode.

Die *createMatch()*-Methode erstellt einen Match zu der amalgamierten Regel, die als Inputparameter angegeben wird.

Die UML–Notation dieser Klasse ist in der *Abbildung* **37** zu sehen.

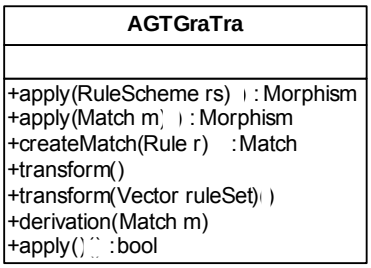

Abbildung 37. *AGTGraTra*

AGTFactory

AGTFactory ist als eine Hilfsklasse definiert um den direkten Zugriff auf bestimmte Methoden und Variablen aus anderen AGT-Klassen zu vermeiden.

Diese Klasse ermöglicht, ein Regelschema (*createRuleScheme()*) oder eine Instanz von der *Covering*–Klasse (*createCovering()*) zu erstellen.

Mit der *createAGTGraGra()*-Methode kann man eine AGT–Graphgrammatik

erzeugen. Mit der *destroyAGTGraGra()*–Methode ist es möglich, eine AGT-Graphgrammatik zu löschen.

Wenn man alle erstellten Graphgrammatiken abfragen möchte, gibt es die *getAGTGraGras()*-Methode.

Die *notify()*-Methode fügt eine Graphgrammatik hinzu, vorausgesetzt, dass sie noch nicht vorhanden ist. Um die Existenz der Graphgrammatik zu überprüfen, gibt es eine *isElement()*–Methode, die den Wahrheitswert auf *true* setzt, wenn die Graphgrammatik bereits existiert.

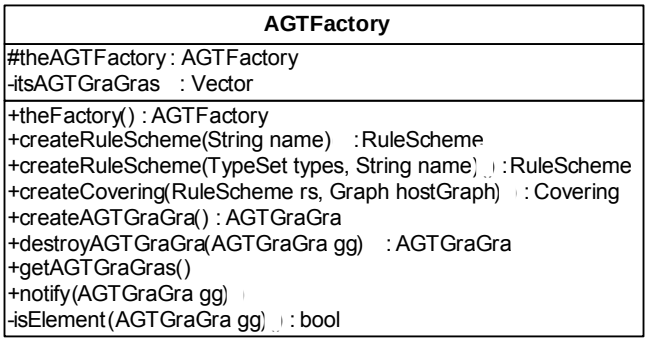

Abbildung 38. *AGTFactory*

6.1.2 Benutzerschnittstelle

Die Erweiterung der Benutzeroberfläche bezieht sich auf die Implementierung von zusätzlichen GUI–Klassen und die Anpassung existierender Klassen, die in AGG–Bibliothek definiert wurden. Im Folgenden werden die neu definierten und erweiterten Klassen einzeln genauer beschrieben.

EdRulescheme

Die von mir definierte Klasse *EdRuleScheme.java* definiert ein graphisches Layout für die *RuleScheme*–Klasse. Das Layout für die *BasicRule* und *ExtendedRule* wurde von dem vorhandenem Layout: *EdRule* von der Superklasse *Rule* übernommen. *EdRuleScheme* implementiert zwei Schnittstellen: *Serializable* und *XMLObject.* Ihre Komponenten sind: das Layout für die Unterregel (BasicRule), und ein Vektor mit Layout für die Objekte der *ExtendedRule*–Klasse, also Layout für die Erweiterungsregeln. Die wichtigste

Komponente ist aber Regelschema (RuleScheme), weiterhin von mir als Basisregelschema genannt, zu dem das Layout gehört. *getEdBasicRule()*-Methode greift auf das Objekt von *EdRule* Klasse zu. Das Objekt ist das Layout für die Unterregel, die zu dem Basisregelschema gehört. Die *getBasisRuleScheme()*–Methode liefert das Basisregelschema und *getEdExtendedRules()*–Methode gibt den Vektor mit Layouts für die Erweiterungsregeln zurück. Es ist auch möglich, mit der *getGraGra()*-Methode auf das Layout von der zugehörigen Graphgrammatik (AGTGraGra) zuzugreifen oder eine Graphgrammatik mit *setGraGra()* zu setzen.

createExtendedRules() erstellt die Layouts für alle, im Basisregelschema vorhandenen Erweiterungsregeln.

createExtendedRule() erzeugt dagegen nur ein Layout für die neue Erweiterungsregel. Ein Objekt von der *ExtendedRule*–Klasse wird im Basisregelschema miterzeugt und zu dem Vektor für die Erweiterungsregel hinzugefügt.

removeExtendedRule() löscht ein entsprechendes Layout für die Erweiterungsregel, soweit es vorhanden ist.

addAmalgamatedRule() fügt das Layout für die berechnete amalgamierte Regel und *removeAmalgamatedRule()* löscht sie.

Mit der Hilfe von der *updateExtendedRules()*–Methode, werden die Erweiterungsregel im graphischen Editor aktualisiert, nachdem die Unterregel durch den Benutzer geändert wurde.

updateExtendedRules() Methode ist analogisch zu der gleichen Methode in der *RuleScheme*–Klasse und wird mit ihr parallel ausgeführt wenn die Unterregel geändert wurde. Aber die Methode in der *EdRuleScheme*–Klasse passt die Erweiterungsregeln auf der Layoutebene an. Hier wird die Methode *getOldNewObjects()*-Methode aus der *ExtendedRule*–Klasse benutzt, um das Layout der linken und rechten Graphen der Regel behalten zu können.

Die Methoden *XwriteObject()* und *XreadObject()* sind definiert, um das Objekt speichern und laden.

EdRule

Die Klasse wurde ursprünglich als Layout für einfache Regeln benutzt. In der neuen Version stellt sie das graphische Layout für die Unterregel und Erweiterungsregeln dar. Ein paar Methoden mussten dazu implementiert werden um die Anpassung auf neue Strukturen möglich zu machen.
setBasisRule()–Methode setzt die entsprechende Regel zu diesem Layout.
Die *updateLayout()*-Methode wird vom *EdRuleScheme* aufgerufen und repariert das Layout für die Erweiterungsregeln. Die Layouts für NACs werden angepasst.

EdGraGra

Die *EdGraGra*–Klasse wurde für die neue AGT–Graphgrammatik angepasst. Im Konstruktor und weiteren Methoden wird nicht mehr die Basisklasse *GraGra* verwendet, sondern die neue *AGTGraGra.* Es wird nicht mehr auf den Instanzen von der EdRule–Klasse gearbeitet. Diese wurden durch die Instanzen der *EdRuleScheme*-Klasse ersetzt.
getRuleSchemes() liefert einen Vektor mit den *EdRuleSchemes,* die zu der *EdGraGra*–Klasse gehören.
getRuleScheme() ermöglicht den Zugriff auf die Instanz eines *EdRuleSchemes,* das zu der entsprechenden *BasicRule*–Instanz gehört. Die Methode wird in der *GraGraEditor*–Klasse gebraucht, wenn man die Layouts für die Erweiterungsregeln, die zu dem *EdRuleScheme* gehören, reparieren möchte.
createRuleScheme() erstellt ein neues Layout für ein Regelschema. Als Inputparameter wird ein Name übergeben. Es gibt alternativ die gleiche Methode die das Layout zu einer übergebenen Instanz von der *RuleScheme* – Klasse erzeugt.
createRuleSchemes() erstellt einen Vektor von Layouts für die eingegebene Instanzen von der RuleScheme–Klasse.
removeRuleScheme() entfernt aus dem Vektor die übergebene *EdRuleScheme*–Instanz. Die Methode wird verwendet, wenn man im GUI ein Regelschema löschen möchte.
XwriteObject() wird beim Speichern einer Grammatik aufgerufen und schreibt eine AGT Graphgrammatik mit ihren Regelschemen und weiteren Komponenten aus. Wenn die amalgamierte Regel bereits erstellt wurde, wird sie allerdings

nicht mitgespeichert.
Mit der Hilfe von *XreadObject()*–Methode wird eine gespeicherte *AGT*–Graphgrammatik geladen.

RuleSchemePopupMenu

Die *RuleSchemePopupMenu*-Klasse wird von GraGraTreeView verwendet. Die Klasse erbt von der *JPopupMenu*–Klasse, die in der Java Standardbibliothek: *javax.swing* definiert ist.
In dem *RuleSchemePopupMenu* wird "*New Extended Rule*"–Menüeintrag definiert um eine neue Erweiterungsregel zu dem entsprechenden Regelschema hinzuzufügen.
"*deleteRuleScheme*"–Menüeintrag führt eine Methode aus, die das Regelschema löscht und "*Amalgamate*" löst den Amalgamierungsprozess aus.

GraGraTreeView

Die Klasse definiert den Navigationsbaum für die Graphgrammatiken und ihren Komponenten und benutzt dafür die in der Java–Bibliothek vordefinierte *JTree* - Klasse. *GraGraTreeView* koordiniert das Löschen, Hinzufügen oder das Anzeigen entsprechender Komponenten in den graphischen Editoren.
Zu dem Navigationsbaum wurde das neue Regelschema-Menü hinzugefügt, deswegen mussten mehrere Methoden an die neue Struktur angepasst werden.
Die Methode *addGraGra()* addiert eine neue *AGT*-Graphgrammatik mit einem leeren Regelschema zu dem Navigationsbaum. In der früheren Version wurde alte Graphgrammatik mit einer leeren Regel hinzugefügt.
addRule() addiert neue Erweiterungsregel zu dem selektierten Regelschema. In der alten Version hat die Methode eine einfache Regel direkt zu der Graphgrammatik hinzugefügt.
addRuleScheme() addiert ein Regeschema zum selektierten *AGTGraGra*–Knoten in dem Navigationsbaum.
getRuleScheme() liefert eine EdRuleScheme–Instanz des gegebenen *DefaultMutableTreeNode*.
deleteRule() löscht eine Erweiterungsregel aus ihrem zugehörigen Regelschema. Das Löschen der Unterregel ist nicht erlaubt und wird abgefragt.

Falls solch eine Situation zutrifft, wird eine Nachricht ausgesendet.

deleteRuleScheme() löscht das selektierte Regelschema mit seinen Komponenten aus dem Navigationsbaum.

loadGraGra() lädt die in einer GXX-Datei gespeicherte AGT–Graphgrammatik mit ihren Regelschemen.

saveAsGraGra() speichert eine AGT–Graphgrammatik und ihre Komponenten in einer GXX– Datei.

Die neue Methode *addRuleScheme()* fügt ein neues Regelschema zu der selektierten Graphgrammatik hinzu.

createAmalgamatedRule() löst den Amalgamierungsprozess aus und fügt die amalgamierte Regel und ihre NACs zu dem Navigationsbaum.

In allen Methoden, die Events behandeln, wurden die Abfragen bezüglich des Regelschemas eingebaut.

EditPopupMenu und EditSelPopupMenu

Die beiden Klassen haben die gleiche Funktionalität, aber die *EditPopupMenu* ist für ein selektiertes Objekt zuständig und *EditSelPopupMenu* dient für mehrere gleichzeitig selektierte Objekte.

Hier wurde ein Mechanismus eingebaut, der die Graphobjekte der Erweiterungsregeln, die mit der Unterregel über einen Einbettungsmorphismus verbunden sind, nicht zu löschen oder modifizieren erlaubt. Wenn der Benutzer das versucht, wird eine Nachricht ausgeschickt.

TransformDebug

TransformDebug ist eine Klasse, die Graphtransformationsschritte per Buttondruck ausführt.

Die Methoden *getMatch(), nextCompletion(), step()* wurden auf Amalgamierungstransformationsschritte erweitert.

getMatch() setzt ein partielles Match auf den Arbeitsgraphen, allerdings bezüglich der Unterregel. Wurde vom Benutzer eine andere Regel, amalgamierte oder Erweiterungsregel, selektiert, ist die Ausführung dieser Methode nicht erlaubt und löst eine Warnungsnachricht aus.

nextCompletion() vervollständigt den aktuellen Match und ist auch nur dann

anwendbar wenn die Unterregel selektiert wurde.

step() führt den Transformationsschritt mit dem aktuellen Match aus. Die Methode kann nur dann angewendet werden, wenn die amalgamierte Regel bereits erstellt wurde und entweder sie oder die Unterregel selektiert wurde. Falls die amalgamierte Regel noch nicht existiert und Unterregel selektiert wurde, wird dar ganze Amalgamierungsschritt auf einmal durchgeführt ohne, dass die amalgamierte Regel in dem Navigationsbaum angezeigt wird. Das Match wird dann willkürlich ausgewählt.

GraGraLoad

Die *GraGraLoad*–Klasse lädt die Graphgrammatik aus einer GGX–Datei. Sie wurde so angepasst, dass die AGT–Graphgrammatik mit den Regelschemata geladen werden können. Die *getAGTGraGra()*–Methode wurde hier implementiert, um auf die geladene AGT–Graphgrammatik zugreifen zu können.

GraGraSave

Analog zu der *GraGraLoad*–Klasse wurde *GraGraSave* geändert. Ihre Methoden wurden an *AGT*–Graphgrammatiken angepasst.

GraGraTreeCellRenderer

Die *getTreeCellRendererComponent()*-Methode setzt das Icon für die entsprechende Komponente und wurde so angepasst, dass das Icon für das Regelschema und die amalgamierte Regel in dem Navigationsbaum angezeigt werden.

GraGraTreeModel

Die *GraGraTreeModel*-Klasse wurde so angepasst, dass das Icon für ein Regelschema und eine amalgamierte Regel in dem Navigationsbaum angezeigt werden. *GraGraTreeModel* erweitert die *DefaultTreeModel*–Klasse, die im *javax.swing.tree – Package* definiert ist.

GraGraTreeNodeData

Die *GraGraTreeNodeData*–Klasse erstellt mit der *GraGraTreeNodeData()*-Methode die Knoten des Navigationsbaums für Graphgrammatikkomponenten. Die Methode kriegt eine Komponente als ein Inputparameter und fügt diesen hinzu. Die *setData()*-Methode bekommt als Inputparameter ein Objekt, das einen Knoten im Navigationsbaum repräsentiert und erkennt, um welche Komponente es sich handelt. Diese Methoden wurden um eine weitere Komponente das Regelschema, erweitert.
Die *getRuleScheme()*-Methode liefert das Layout des Regelschema.

GraGraEditor

GraGraEditor ist eine Klasse, die einen Editor definiert um eine Graphgrammatik mit ihren Komponenten anzuzeigen. Der Editor besteht aus dem Typ-, Regel- und Grapheditor.
Die Änderung dieser Klasse bezieht sich auf die Behandlung der Regelschemata. Im Editor wird erkannt ob die Unterregel geändert wurde und sobald wir andere Komponenten in dem Navigationsbaum, selektieren, werden eine Reihe von Methoden ausgeführt die Erweiterungsregeln und Einbettungsmorphismen entsprechend reparieren können.

6.2 Implementierung

Wie schon im *Kapitel 3.2* erwähnt, wurde das Amalgamierungskonzept in Java der Version 1.4.2 implementiert. Der Kern der Implementierung bildet die *Covering*–Klasse, die im *Unterkapitel 6.1.1* ausführlich beschrieben wurde. Um den Algorithmus des Amalgamierungsprozesses dynamisch zu definieren, wurden die in Java vordefinierten Klassen, wie *Vector* und *Array* benutzt. Unter Dynamik wird hier die Möglichkeit der Erstellung einer amalgamierten Regel unabhängig von der Anzahl der Erweiterungsregeln verstanden. Die *Covering*–Klasse hat als Input ein *RuleScheme*-Objekt, dessen Komponente eine Unterregel und mehrere Erweiterungsregel sind, die in einem *Vector* zusammengefasst sind. Wenn man Instanzen jeder Erweiterungsregel bilden möchte, werden sie in einem Array, dessen einzelne Elemente Vektoren sind, aufgefasst. Die Länge des Arrays ist gleich der Anzahl der Erweiterungsregeln. Auf diese Weise kann man die Instanzen jeder Erweiterungsregel von einander trennen, was hilfreich ist, wenn die Einbettungsmorphismen einer Unterregel in

jeder Instanz oder die Regelmorphismen für Instanzen berechnet werden müssen. Die Struktur verschafft einen klaren Überblick. Für die Berechnung von Kolimesdiagrammen werden die *Arrays* von Vektoren wiederum aus Bequemlichkeit in einen Vector zusammengefasst. Diese Arrays bilden eine Zusammenfassung für die Einbettungsmorphismen von Instanzregeln. Die Konstruktion der *ColimDiagram*–Klasse, die zur Berechnung von Kolimesdiagrammen benutzt werden, fügt die Kanten (Einbettungsmorphismen) und Knoten (Graphen von Instanzen) der Reihe nach, ein. Die Unterscheidung von Erweiterungsregeln ist nicht mehr erforderlich.

Zur Erweiterung von der Benuteroberfläche wurden die Standardbibliotheken von Java wie *javax.swing* verwendet, insbesondere für die Implementierung von Menüs und Ausbau von dem Navigationsbaum. Ein sehr wichtiger Aspekt ist, dass die wichtigsten Komponenten wie: *RuleScheme*, AGTGraGra die *XMLObject*–Schnittstelle implementieren, was das Speichern und Laden dieser Strukturen ermöglicht.

7. Benutzung der Amalgamierung in AGG

Die Anwendung des AGG Tools hat sich in der neuen Version nicht grundsätzlich geändert. Das Grundkonzept wurde beibehalten. Im Folgenden werde ich schrittweise die Durchführung der amalgamierten Graphtransformation im AGG–System vorgestellt.

7.1 Erstellen der AGT-Graphgrammatik

Das Erstellen der Graphgrammatik erfolgt weiterhin entweder mit der Betätigung des "*NewGraGra*"–Buttons oder mit der Selektierung von "*NewGraGra*"–Menüeintrag in dem "*FileMenu*". Im Unterschied zu vorher wird eine neue AGT–Graphgrammatik erstellt. Im Navigationsbaum erscheint dann die Grammatik und ein Regelschema mit einer Unterregel. Soll eine Erweiterungsregel erzeugt werden, selektiert man den *RuleScheme*–Eintrag im Navigationsbaum und ruft mit der rechten Maustaste das zugehörige Popupmenu auf und wählt den Menüeintrag "*New Extended Rule*" aus. Jede Regel kann editiert werden, indem sie im Navigationsbaum angeklickt wird. Ähnlich werden weitere Regelschemata erzeugt. Der Eintrag der Graphgrammatik wird im Navigationsbaum selektiert. Durch einen Klick mit der rechten Maustaste wird die Möglichkeit gegeben, das Popupmenu erscheinen zu lassen. In dem Menu wird der Eintrag: "*New Rule Scheme*" ausgewählt. Für jedes neue Regelschema kann man mehrere Erweiterungsregeln erzeugen, aber es ist nicht möglich, neue Unterregeln zu erstellen. Wie schon oben erwähnt, wird diese mit jedem Regelschema automatisch miterstellt. Sie kann vervollständigt werden, eventuell modifiziert, aber es ist nicht gestattet, sie zu löschen. Für jede Regel ist es aber möglich, eine oder mehrere negative Anwendungsbedingungen zu definieren, indem die gewünschte Regel angeklickt wird und mit der rechten Maustaste das Popupmenü geöffnet wird. Zuletzt muss nur noch der "*New NAC*"–Eintrag ausgewählt werden.

Es ist auch möglich, eine gespeicherte Graphgrammatik zu laden, indem der "*Open*"–Button betätigt wird oder unter Menü "File" der Menüeintrag "*Open*" ausgewählt wird.

Die *Abbildung 39* zeigt das Erstellen eines Regelschemas.

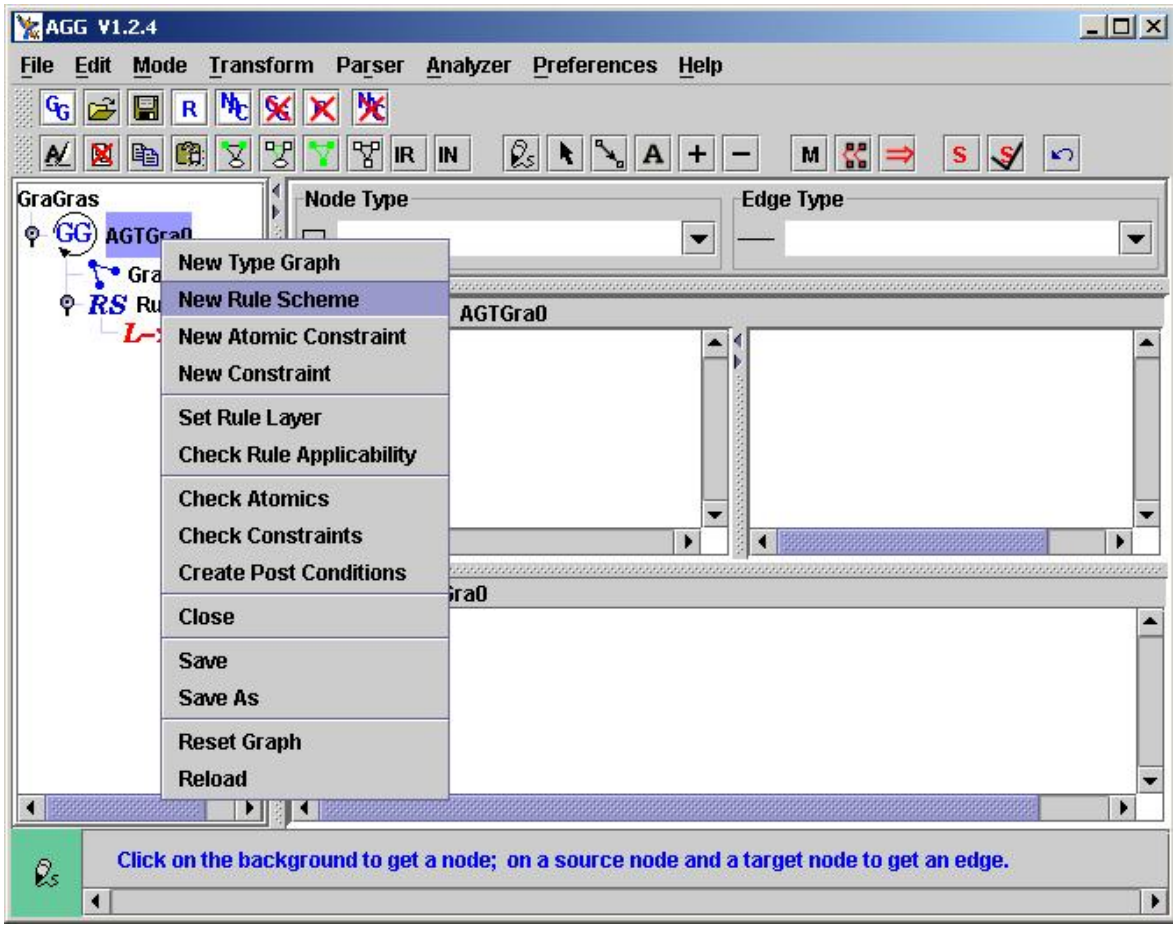

Abbildung 39. *Create rule scheme*

7.2 Erzeugen der amalgamierten Regel

Nach dem ein Regelschema erstellt wurde, die Unterregel und Erweiterungsregel definiert wurden und ein Arbeitsgraph gezeichnet wurde, kann der Menüeintrag "*Amalgamate*" im Popupmenu, zu Regelschema, ausgewählt werden, damit die amalgamierte Regel und ein Ansatz in dem Arbeitsgraphen erstellt werden können. Die amalgamierte Regel wird zum Navigationsbaum hinzugefügt. Falls für die Unterregel oder Erweiterungsregeln negative Anwendungsbedingungen definiert wurden, werden sie mitübernommen und in dem Baum an die amalgamierte Regel angehängt. Den amalgamierten Match kann man an der Nummerierung der Graphobjekten im Arbeitsgraphen erkennen.

Die *Abbildung 40* zeigt eine erzeugte amalgamierte Regel.

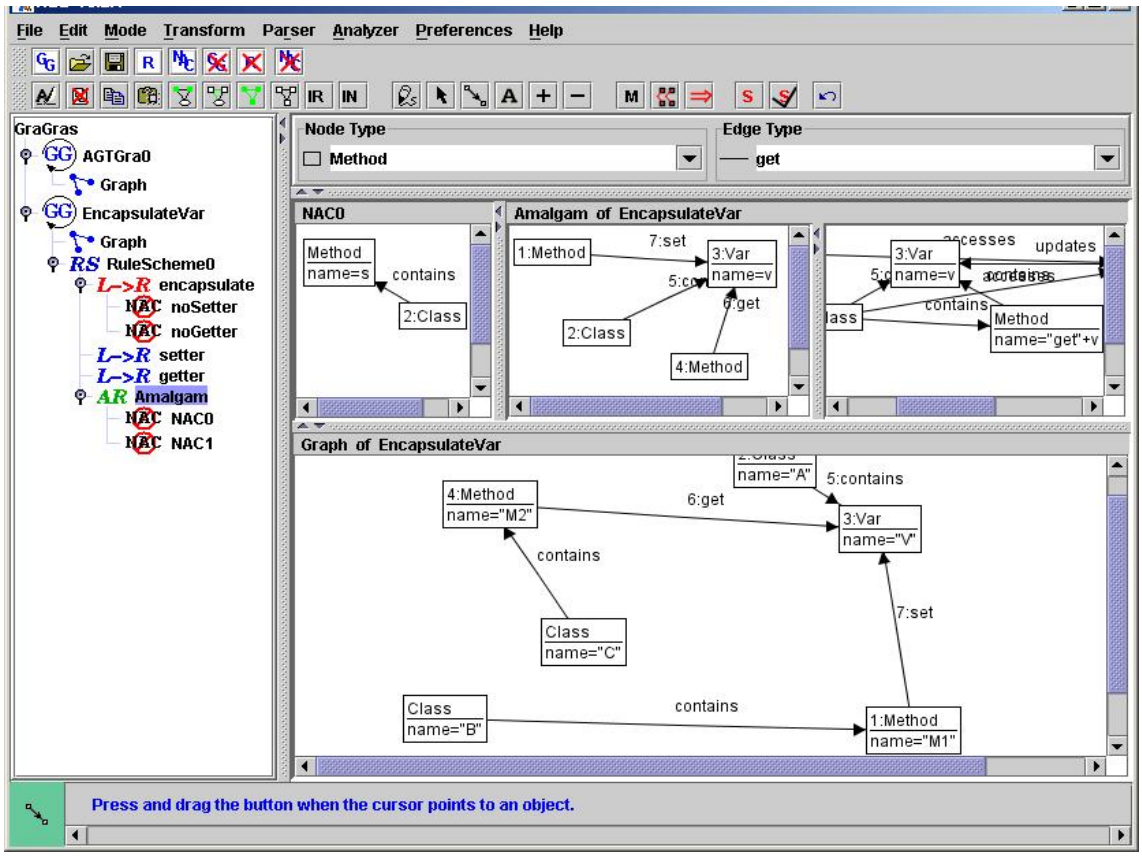

Abbildung 40. *Amalgamated rule.*

7.3 Attribute und Attributbedingungen

Die Attributierung der Objekte erfolgt genauso wie in der alten Benutzerschnittstelle. Entweder wird auf ein erwünschtes Objekt mit der rechten Maustaste angeklickt und in dem Popupmenü, welches erscheint, wählt man den Menüeintrag "*Attributes*" oder wenn man in dem Menü "Edit" auch "*Attributes*" auswählt, nachdem das Graphobjekt selektiert wurde. Im Editor erscheint ein Panel, in dem Attribute und Attributsbedingungen definiert werden können

Man kann für die Attributwerte Konstanten, Variablen oder Ausdrücke definieren. Als Attributstypen bietet hier das *AGG*–System die *Java*-Standardtypen, wie zum Beispiel *int*, oder *String* an. Wenn die Variablen oder Ausdrücke für die Graphobjekte in einer Erweiterungsregel definiert wurden und die Erweiterungsregel mehrere Instanzen besitzt, wird in jeder Instanz eine Umbenennung stattfinden. Ähnlich wird es für die Attributsbedingungen gemacht.

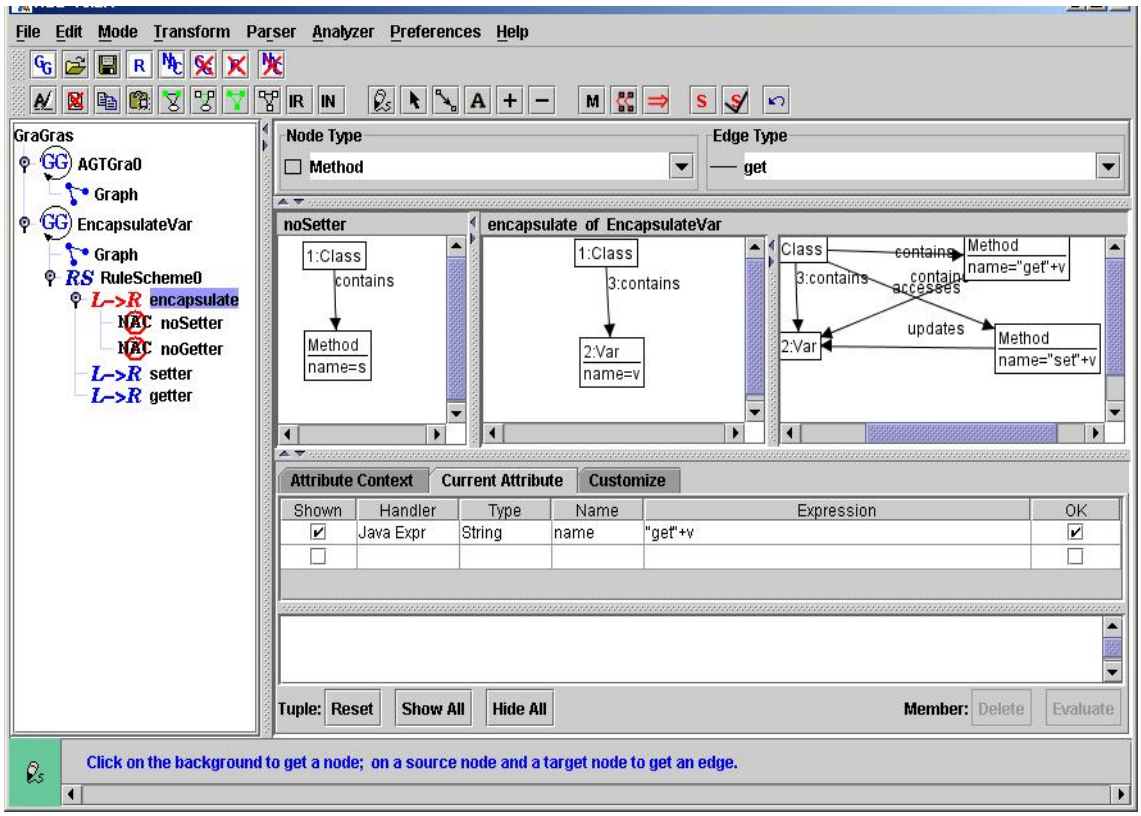

Abbildung 41. *Attribute settings*

7.4 Reparieren von Einbettungsmorphismen

Jedes Mal, wenn die Unterregel geändert wurde und wenigstens eine Erweiterungsregel bereits existiert, wird der Benutzer gefragt, ob er die Erweiterungsregeln und die Einbettungsmorphismen an die geänderte Unterregel anpassen möchte. Dies ist notwendig, um eine amalgamierte Transformation durchführen zu können. Wenn der Benutzer die Anfrage mit "*ok*" bestätigt, werden die Erweiterungsregeln entsprechend angepasst und Einbettungsmorphismen werden repariert.

Die Graphobjekte der Erweiterungsregeln, die Bilder von Einbettungsmorphismen sind, dürfen nicht modifiziert oder gelöscht werden. Es können nur die Graphobjekte gelöscht oder geändert werden, die keine Urabbildung in der Unterregel haben.

7.5 Partieller Ansatz der Unterregel

Es ist möglich einen partiellen Ansatz für die Unterregel anzugeben. Der

"*Interactive Match Mode*"–Button wird gedrückt und ein Objekt wird aus dem linken Graphen der Unterregel ausgewählt und dann wird eine gewünschte Abbildung des Objektes in dem Arbeitsgraphen angeklickt. Wenn die Aktion erfolgreich war, sieht man die gleiche Zahl in beiden Graphobjekten. Wenn der Ansatz vervollständigt werden soll, muss der "*Next Copmletion*" Button betätigt werden. Das wird solange gemacht bis der gewünschte Ansatz erreicht wird. Die gleichen Aktionen können durchgeführt werden, indem das Menü "*Transform*" benutzt wird.

7.6 Transformationsschritt

Wenn der Transformationsschritt mit einer amalgamierten Regel ausgeführt werden soll, drückt man den "*Transformation Step*"–Button oder wählt die Option im "*Transform*"–Menü aus. Wenn die Transformation erfolgreich war, wird im Grapheditor der transformierte Arbeitsgraph erscheinen.

Anzumerken ist, dass die Ausführung des Transformationsschrittes auch möglich ist, wenn die amalgamierte Regel noch nicht erzeugt wurde. Der Amalgamierungsprozess und die Transformation werden dann in einem Schritt ausgeführt und die amalgamierte Regel wird nicht in dem Navigationsbaum angezeigt, obwohl sie generiert und benutzt wurde. Dies ist nur eine andere, "schnellere" Möglichkeit, die amalgamierte Graphersetzung zu einem Regelschema durchzuführen.

8. Zusammenfassung und Ausblick

In dieser Arbeit wurden die praktische Umsetzung der Theorie der amalgamierten Graphtransformationen und ihre Beschreibung mit den Lösungen und Lösungsansätzen präsentiert. Die Aufgabe bestand in der Erweiterung des schon existierenden AGG–Systems. Das Ziel war, die graphische Oberfläche dem neuen Konzept so anzupassen, dass die alte Funktionalität beibehalten werden konnte. [0]

Der Erfolg des Vorhabens ist dadurch ermöglicht worden, dass die Imlementierung auf der AGG-Bibliothek basiert und mit dieser verzahnt ist. [0]Mehrere Klassen des entwickelten AGT-Packages erben von den Klassen aus dem xt_basis-Package. Wenn man nur eine Regel auf einem Arbeitsgraphen anwenden möchte, kann man ein Regelschema definieren, das keine Erweiterungsregel, [0]jedoch die Unterregel besitzt. Die Unterregel wird dann als eine einfache Regel abgeleitet.

Die Abhängigkeiten von den existierenden AGG–Klassen, hatte aber auch viele unerwartete und ungewollte Effekte, was erst während der Testphase zum Vorschein kam. Zum Beispiel musste die *dangling*–Bedingung berücksichtigt werden, was genauer im *Kapitel 5* beschrieben wurde.

Ein weiteres Problem ist bei der Anpassung der Einbettungsmorphismen aufgetreten. Es hat sich herausgestellt, dass nach der Änderung der Erweiterungsregel anhand der Modifikationen, die auf einer Unterregel angewandt wurden, zwar die Instanzen der Erweiterungsregeln tatsächlich angepasst wurden, aber nicht ihre Layouts. Deswegen verwies die Anzeige im Regeleditor auf inkonsistente Ergebnisse. Das Problem war, dass bei der Änderung einer Erweiterungsregel ihre linke und rechte Seite ersetzt wurden. Leider waren die Graphen auf der Layoutebene und die zugehörigen Graphobjekte noch nicht angepasst. Die Lösung war, die alten und neuen Graphobjekte sich zu merken und die Layouts zu übernehmen.

Problematisch war auch die Anwendung von verschiedenen Aktionen, die von der graphischen Oberfläche ausführbar sind und in dem alten AGG–System schon vorhanden waren. Diese mussten mit der neuen Version kompatibel gemacht werden. Zum Beispiel sollte man den partiellen Ansatz für die Unterregel setzten können, deswegen musste immer die Abfrage erfolgen,

ob in der Tat in dem Navigationsbaum eine Unterregel selektiert wurde.
Ähnlichen Abfragen mussten in vielen anderen Methoden eingebaut werden, um die unkorrekten Ableitungen zu vermeiden.
Das Hauptziel, der Aufgabenstellung wurde erreicht. Man sollte aber im Auge behalten, dass das Gebiet noch viele Erweiterungen erfordert. Das Konzept konzentriert sich nur auf dem einfacheren *local all* -Ansatz von dem *Covering*. Eine echte Herausforderung bietet die Idee, das Konzept auf das *fully synchronized Covering* zu erweitern, die im *Kapitel 5* erklärt wurde. Diese Umsetzung ist flexibler und dynamischer und erlaubt es, komplexere Strukturen auf einen Arbeitsgraphen abzuleiten. Die Programmkomponenten sind so aufgebaut, dass eine Erweiterung denkbar und durchaus möglich wäre. Vieles könnte übernommen werden. Natürlich würde auch das neue Konzept auch eine Menge von Anpassungen erfordern. Die Klasse, die das Regelschema repräsentiert, müsste mehrere Unterregeln erlauben. Der Algorithmus, der in der *Covering*-Klasse definiert ist und den Amalgamierungsprozess definiert, müsste jetzt auf neue Strukturen angepasst werden. Die Klassen wie: *AGTGraGra, AGTGraTra* und *AGTFactory* können mit der minimalen Modifizierung übernommen werden.

Es ist noch wichtig zu erwähnen, dass das alte AGG–System einige weitere Konzepte im Bezug auf die Ableitung von den Graphregeln unterstützt. Das AGG *Graphparser*–Konzept kann überprüfen, ob ein gegebener Graph zu einer bestimmten Graphsprache im Bezug auf eine Graphgrammatik gehört. In der formalen Sprachtheorie ist das Problem als *membership problem* bekannt. Es ist unentscheidbar für die Graphgrammatiken im allgemein, aber für bestimmte Klassen von Graphgrammatiken ist es mehr oder weniger effizient lösbar. Das AGG–System bietet verschiede Parseralgorithmen an, die alle auf dem *back tracking* basieren. Es gibt andere Parservarianten für die Regeln, wie zum Beispiel *critical pair analysis*. Die Variante kann benutzt werden, um das Parsen von Graphen effizienter zu gestallten. Die Entscheidung zwischen zwei Regeln, die im Konflikt zueinander stehen, wird so lange verzögert wie möglich. Das bedeutet, dass erst die Regel abgeleitet wird, die keine Konflikte verursacht und der Arbeitsgraph soweit möglich maximal reduziert wird. Das AGG unterstützt auch *Konsistenzbedingungen.* Dieser Kontrollmechanismus ist in der Lage zu untersuchen, ob ein Graph bestimmte Bedingungen erfüllt, die für eine

Graphgrammatik spezifiziert sind.

Diese Konzepte sind bisher für einfache Regeln formuliert und implementiert worden. Es bleibt offen, wie diese Konzepte auf amalgamierte Graphtransformationen übertragen werden können.

Literaturverzeichnis

[AGG] tfs.cs.tu-berlin.de/

[Ehr79] H. Ehrig. Introduction to the algebraic approach of graph grammars. Pages 3-14, *1987*

[EPS73] H. Ehrig, M. Pfender, H. J. Schneider. Graph grammars: an algebraic approach. *In 14th Annual Symposium on Switching and Automata Theory,* pages 167-180, *1973*

[ErSchul] C. Ermel, T. Schulzke. The AGG Environment: A Short Manual.

[ETB05] C. Ermel, G. Taentzer, R. Bardohl. Simulating Algebraic High-Level Nets by Parallel Attributed Graph Transformation. *2005*

[HHT96] A. Habel, R. Heckel, G. Taentzer. Graph Grammars with Negative Application Conditions. *Special issue of Fundamenta Informaticae, 26(3,4), 1996.*

[HMTW94] R. Heckel, J. Müller, G. Taentzer, A. Wagner. Attributed Graph Transformations with Controlled Application of Rules. *1994*

[Java] www.programmersbase.net/Content/Java/Content/Tutorial/Java/

[KLTW93] M. Korf, M. Löwe, G. Taentzer, A. Wagner. Algebraische Graph-grammatiken - zwischen Theorie und Praxis - Skript zur Lehrveranstaltung "Graphtransformation und Petrinetze"

[Löw93] M. Löwe. Algebraic approach to single-pushhout graph transformations. *TCS, 109: 181-224, 1993*

[MeTaRu04] T. Mens, G. Taentzer, O. Runge. Detecting Structual Refactoring Conflicts Using Critical Pair Analysis. *2004*

[RuTaen99] M. Rudolf, G. Taentzer. Introduction to the Language Concepts of AGG. *1999*.

[TaenBey] G. Taentzer, M. Beyer. Amalgamated Graph Transformations and Their Use for Specifying AGG – an Algebraic Graph Grammar System

[Taenz96] G. Taentzer. Parallel und Distributed Graph Transformation: Formal Description und Application to Communication-Based Systems. Dissertation, TU Berlin, 1996

[UML] pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/

[Wolz98] D. Wolz. Colimit Library for Graph Transformations and Algebraic Development Techniques. Dissertation, TU Berlin, 1998

Abbildungsverzeichnis

Anhang A

AGT-Package Dokumentation

Der folgende Abschnitt enthält eine Zusammenstellung der Dokumentation zu den Klassen aus dem AGT-Package

- ## Class BasicRule

```
java.lang.Object
  |
  +--java.util.Observable
        |
        +--agg.util.ExtObservable
              |
              +--agg.xt_basis.OrdinaryMorphism
                    |
                    +--agg.xt_basis.Rule
                          |
                          +--agg.agt.BasicRule
```

All Implemented Interfaces

agg.util.Disposable, agg.xt_basis.Morphism, java.util.Observer, java.io.Serializable, agg.util.XMLObject

Constructor Summary

`BasicRule()`

Constructor of the ExtendedRule class.

Method Detail

setRuleScheme

`public void setRuleScheme(agg.agt.RuleScheme rs)`

Set own rule scheme.

getRuleScheme

```
public agg.agt.RuleScheme getRuleScheme()
```

Return own rule scheme.

Returns:

RuleScheme

hasChanged

```
public boolean hasChanged()
```

Return true if basic rule has changed.

Overrides:

`hasChanged` in class `java.util.Observable`

setChanged

```
public void setChanged(boolean b)
```

Set changed variable.

update

```
public final void update(java.util.Observable o,
                         java.lang.Object arg)
```

Notice if the basic rule has changed.

Specified by:

`update` in interface `java.util.Observer`

• Class ExtendedRule

```
java.lang.Object
  |
  +--java.util.Observable
        |
        +--agg.util.ExtObservable
              |
              +--agg.xt_basis.OrdinaryMorphism
                    |
                    +--agg.xt_basis.Rule
                          |
                          +--agg.agt.ExtendedRule
```

All Implemented Interfaces

agg.util.Disposable, agg.xt_basis.Morphism, java.io.Serializable, agg.util.XMLObject

Constructor Summary

`ExtendedRule``(agg.xt_basis.Rule basicRule)`

Constructor of the ExtendedRule class

Method Detail

setEmbbededMorphLeft

`public void` **`setEmbbededMorphLeft`**`(agg.xt_basis.OrdinaryMorphism left)`

Set embedded morphism left.

Parameters:

`left` -

setEmbbededMorphRight

`public void` **`setEmbbededMorphRight`**`(agg.xt_basis.OrdinaryMorphism right)`

Set embedded morphism right.

Parameters:

`right` -

getObjectPairsLeft

`public java.util.Vector` **`getObjectPairsLeft`**`()`

Return the pbject pairs of the left embedded morphism.

getObjectPairsRight

`public java.util.Vector` **`getObjectPairsRight`**`()`

Return the pbject pairs of the right embedded morphism.

getOldNewObjects

`public com.objectspace.jgl.Pair` **`getOldNewObjects`**`()`

Return the object pair of old and new objects.

getMappingsOfNacObjects

`public java.util.Hashtable` **`getMappingsOfNacObjects`**`()`

Return the hashtable with the mappigs of NAC objects.

getBasicRule

`public agg.xt_basis.Rule` **`getBasicRule`**`()`

Return the basic rule.

getEmbbededMorphLeft

`public agg.xt_basis.OrdinaryMorphism` **`getEmbbededMorphLeft`**`()`

Return the embedded morphism left.

getEmbbededMorphRight

`public agg.xt_basis.OrdinaryMorphism` **`getEmbbededMorphRight`**`()`

Return the embedded morphism right.

setLeft

`public final void` **`setLeft`**`(agg.xt_basis.Graph left)`

Set the left hand side of the extended rule.

setRight

`public final void` **`setRight`**`(agg.xt_basis.Graph right)`

Set the right hand side of the extended rule.

updateEmbedding

`public boolean` **`updateEmbedding`**`()`

Update embedded morphism.

XwriteObject

`public void` **`XwriteObject`**`(agg.util.XMLHelper h)`

Specified by:

`XwriteObject` in interface `agg.util.XMLObject`

Overrides:

`XwriteObject` in class `agg.xt_basis.Rule`

XreadObject

`public void` **`XreadObject`**`(agg.util.XMLHelper h)`

Load the extended rule.

Specified by:

`XreadObject` in interface `agg.util.XMLObject`

Overrides:

`XreadObject` in class `agg.xt_basis.Rule`

• Class RuleScheme

```
java.lang.Object
  |
  +--agg.agt.RuleScheme
```

All Implemented Interfaces

`agg.util.XMLObject`

Constructor Summary

`RuleScheme``()`

Constructor of the RuleScheme class.

`RuleScheme``(java.lang.String name)`

Constructor of the RuleScheme class.

`RuleScheme``(agg.xt_basis.TypeSet typeSet,java.lang.String name)`

Constructor of the RuleScheme class.

Method Detail

getAmalgamatedRule

`public agg.xt_basis.Rule` **`getAmalgamatedRule`**`()`

Return amalgamated rule.

Returns:

Rule

setAmalgamatedRule

`public void **setAmalgamatedRule**(agg.xt_basis.Rule a)`

Set amalgamated rule.

destroyAmalgamatedRule

`public void **destroyAmalgamatedRule**()`

Despose amalgamated rule.

setAmalgamatedMatch

`public void **setAmalgamatedMatch**(agg.xt_basis.Match m)`

Set amalgamated match.

getAmalgamatedMatch

`public agg.xt_basis.Match **getAmalgamatedMatch**()`

Return amalgamated match.

Returns:

Match

destroyAmalgamatedMatch

`public void **destroyAmalgamatedMatch**()`

Despose amalgamated match.

isValid

`public boolean **isValid**()`

Return true if the rule scheme is valid.

setValid

`public void **setValid**(boolean b)`

Set valid on true.

createEmptyExtendedRule

`public agg.agt.ExtendedRule **createEmptyExtendedRule**()`

Create an extended rule without the embedded morphism.

createExtendedRule

`public agg.agt.ExtendedRule` **`createExtendedRule`**`()`

Create an extended rule with the embedded morphism.

addObjectPairs

`public void` **`addObjectPairs`**`(agg.agt.ExtendedRule ext)`

Create pairs of object of the embedded morphismus.

updateExtendedRules

`public void` **`updateExtendedRules`**`()`

Update extended rules.

removeExtendedRules

`public void` **`removeExtendedRules`**`()`

Remove all own extended rules.

removeExtendedRule

`public void` **`removeExtendedRule`**`(agg.xt_basis.Rule r)`

Remove the specified extended rule.

getBasicRule

`public agg.agt.BasicRule` **`getBasicRule`**`()`

Return basic rule.

Returns:

BasicRule

getExtendedRules

`public java.util.Vector` **`getExtendedRules`**`()`

Return extended rules.

Returns:

Vector

getExtendedRule

`public agg.agt.ExtendedRule` **`getExtendedRule`**`(int i)`

Return extended rule at the specified index.

Returns:

ExtendedRule

setName

```
public final void setName(java.lang.String n)
```

Set a name of the rule scheme.

getName

```
public java.lang.String getName()
```

return the name of the rule scheme.

Returns:

String

XwriteObject

```
public void XwriteObject(agg.util.XMLHelper h)
```

Save the rule scheme.

Specified by:

`XwriteObject` in interface `agg.util.XMLObject`

XreadObject

```
public void XreadObject(agg.util.XMLHelper h)
```

Load the rule scheme.

Specified by:

`XreadObject` in interface `agg.util.XMLObject`

- # Class Covering

```
java.lang.Object
  |
  +--agg.agt.Covering
```

Field Detail

match

```
protected static agg.xt_basis.Match match
```

amalgamated match

Constructor Summary

```
Covering(agg.agt.RuleScheme rs,agg.xt_basis.Graph hostGraph)
```

Constructor of the Covering class

Method Detail

amalgamate

```
public boolean amalgamate()
```

Start of the amalgamation prozess, return true if successful.

getRuleScheme

```
public agg.agt.RuleScheme getRuleScheme()
```

Return the rule scheme.

getAmalgamatedRule

```
public agg.xt_basis.Rule getAmalgamatedRule()
```

Return the amalgamated rule.

getAmalgamatedMatch

```
public static agg.xt_basis.Match getAmalgamatedMatch()
```

Return the amalgamated match.

makeAmalgamatedMatch

```
private agg.xt_basis.Match makeAmalgamatedMatch()
```

Make a amalgamated match.

makeBasicRuleAsAmalgamatedRule

```
private agg.xt_basis.Rule makeBasicRuleAsAmalgamatedRule()
```

Define the basic rule as the amalgamated rule.

makeBasicMatchAsAmalgamatedMatch

```
private agg.xt_basis.Match makeBasicMatchAsAmalgamatedMatch()
```

Define the match of basic rule as the amalgamated match.

constructAmalgamatedRule

```
private agg.xt_basis.Rule constructAmalgamatedRule()
```

Construct the amalgamated rule.

createMatchMs

```
private boolean createMatchMs(agg.agt.BasicRule bRule, agg.xt_basis.Graph graph)
```

Create match of the basic rule.

createMatchesMe

```
private boolean createMatchesMe()
```

Create matches of the extended rules.

createMiMatch

```
private agg.xt_basis.OrdinaryMorphism createMiMatch(OrdinaryMorphism LeLi, OrdinaryMorphism matchME)
```

Create matches of the instance rules.

createLsLMorph

```
private agg.xt_basis.OrdinaryMorphism createLsLMorph()
```

Create morphism between left hand side of the basic rule and left colim diagram..

takeNACsFromExtendedRules

```
private boolean takeNACsFromExtendedRules()
```

Take NACs from the extended rules.

takeNACsFromBasicRule

```
private void takeNACsFromBasicRule()
```

Take NACs from the basic rule.

createIsoCopiesLeftRight

```
private boolean createIsoCopiesLeftRight()
```

Create isocopies of the extended rules.

computeColimLeft

```
private void computeColimLeft()
```

Compute left colim diagram.

computeColimRight

```
private void computeColimRight()
```

Compute right colim diagram.

createRuleInstancesMorphs

```
private void createRuleInstancesMorphs()
```

Create rule morphism for the instance rules.

createRiRules

```
private void createRiRules()
```

Create rule instance rules.

constructAmalgamatedMatch

```
private void constructAmalgamatedMatch()
```

Create amalgamated match.

• Class AGTGraGra

```
java.lang.Object
  |
  +--agg.xt_basis.GraGra
        |
        +--agg.agt.AGTGraGra
```

All Implemented Interfaces

agg.util.Disposable, java.io.Serializable, agg.util.XMLObject

Field Detail

itsRuleSchemes

```
protected java.util.Vector itsRuleSchemes
```

Constructor Summary

AGTGraGra()

Constructor of the AGTGraGra class

Method Detail

getRuleSchemes

```
public final java.util.Enumeration getRuleSchemes()
```

Return all rule schemes as an enumeration.

getVectorRuleSchemes

```
public final java.util.Vector getVectorRuleSchemes()
```

Return all rule schemes as a vector.

getRuleScheme

```
public agg.agt.RuleScheme getRuleScheme(int indx)
```

Return a rule scheme at the specified index.

Parameters:

indx -

createRuleScheme

```
public agg.agt.RuleScheme createRuleScheme()
```

Create a new rule scheme instance.

destroyRuleScheme

```
public void destroyRuleScheme(agg.agt.RuleScheme ruleScheme)
```

Despose the specified rule scheme instance.

Parameters:

`ruleScheme` -

XwriteObject

```
public void XwriteObject(agg.util.XMLHelper h)
```

Save AGTGraGra in a ggx file .

Specified by:

`XwriteObject` in interface `agg.util.XMLObject`

Overrides:

`XwriteObject` in class `agg.xt_basis.GraGra`

XreadObject

```
public void XreadObject(agg.util.XMLHelper h)
```

Specified by:

`XreadObject` in interface `agg.util.XMLObject`

Overrides:

`XreadObject` in class `agg.xt_basis.GraGra`

• Class AGTGraTra

```
java.lang.Object
  |
  +--agg.xt_basis.GraTra
        |
        +--agg.agt.AGTGraTra
```

Constructor Summary

`AGTGraTra``()`

Constructor of the AGTGraTra class.

Method Detail

apply

```
public agg.xt_basis.Morphism apply(agg.agt.RuleScheme rs)
```

Apply a rule scheme and make amalgameted match.

createMatch

```
public agg.xt_basis.Match createMatch(agg.xt_basis.Rule r)
```

Create match of the specified rule.

Overrides:

`createMatch` in class `agg.xt_basis.GraTra`

transform

```
public void transform(java.util.Vector ruleSet)
```

Not yet implemented.

Specified by:

`transform` in class `agg.xt_basis.GraTra`

• Class AGTFactory

```
java.lang.Object
  |
  +--agg.agt.AGTFactory
```

Field Detail

theAGTFactory

```
protected static agg.agt.AGTFactory theAGTFactory
```

Constructor Summary

`AGTFactory``()`

Constructor of the AGTGraGra class

Method Detail

theFactory

```
public static agg.agt.AGTFactory theFactory()
```

createRuleScheme

```
public agg.agt.RuleScheme createRuleScheme(java.lang.String name)
```

Create rule scheme.

Parameters:

`name` -

createRuleScheme

```
public agg.agt.RuleScheme createRuleScheme(agg.xt_basis.TypeSet types,
                                           java.lang.String name)
```

Create rule scheme.

Returns:

RuleScheme

createCovering

```
public agg.agt.Covering createCovering(agg.agt.RuleScheme rs,
                                       agg.xt_basis.Graph hostGraph)
```

Create covering.

Returns:

Covering

createAGTGraGra

```
public agg.agt.AGTGraGra createAGTGraGra()
```

Create AGTGraGra.

Returns:

AGTGraGra

destroyAGTGraGra

```
public void destroyAGTGraGra(agg.agt.AGTGraGra gg)
```

Dispose the specified gragra

Die selbständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, den 10.03.2005

_____

Unterschrift