Institut für Softwaretechnik

UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Abbildung von Ecore nach grUML

Bachelorarbeit
zur Erlangung des Grades eines Bachelor of Science im Studiengang Informatik

vorgelegt von
Kristina Heckelmann

Betreuer:
Professor Jürgen Ebert, Institut für Softwaretechnik
Dr. Volker Riediger, Institut für Softwaretechnik
Dipl.-Inform. Daniel Bildhauer, Institut für Softwaretechnik
Dipl.-Inform. Tassilo Horn, Institut für Softwaretechnik

Erstgutachter:
Professor Jürgen Ebert, Institut für Softwaretechnik

Zweitgutachter:
Dipl.-Inform. Tassilo Horn, Institut für Softwaretechnik

Koblenz, im Oktober 2010

Erklärung

Hiermit versichere ich gemäß § 17 Abs. 6 der gemeinsamen Prüfungsordnung für Studierende der Bachelor- und Masterstudiengänge des Fachbereichs Informatik an der Universität Koblenz-Landau in der Fassung vom 15.03.2007, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsausschuss vorgelegen.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung der Arbeit im Internet stimme ich zu.

Koblenz, den 05. Oktober 2010

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit der Abbildung von Ecore nach grUML. Dabei wird sowohl die Transformation von in Ecore modellierten Modellen nach Graphen in grUML, als auch die Konvertierung von Ecore-Metamodellen nach grUML-Schemas betrachtet.
Zunächst werden die beiden Modellierungssprachen Ecore und grUML einzeln beschrieben. Dabei werden die Metamodelle der Sprachen erklärt und die Programmierschnittstellen vorgestellt. Im Anschluss werden Unterschiede und Gemeinsamkeiten von grUML und Ecore erläutert und auf dieser Grundlage eine Abbildung definiert. Außerdem werden Anforderungen an die Implementierung der Transformation festgehalten. Danach folgen verschiedene Details zur Realisierung. Zum Schluss wird mit Hilfe einiger Beispiele gezeigt, welche Ergebnisse durch die Transformation entstehen.

Abstract

This thesis deals with the mapping from Ecore to grUML. Thereby the transformation of models in Ecore to graphs in grUML is as well considered as the conversion of Ecore metamodels to grUML schemas.
At first the modeling languages Ecore and grUML are descriped separatly. Thereby the metamodels of the languages are explained and the API is presented. Subsequent differences and similarities of grUML and Ecore are exemplified and based on this a mapping is defined. Also requirements for the implementation of the transformation are noted. After that details of the realisation follow. Finally the results of the transformation are shown with the help of some examples.

Inhaltsverzeichnis

1 Einführung

In diesem Kapitel wird zunächst die Motivation für diese Arbeit erläutert. Anschließend werden die verfolgten Ziele aufgeführt und erklärt.

1.1 Motivation

Die Arbeitsgruppe Ebert nutzt eine mächtige Klasse von *Graphen zur Repräsentation von Daten* in verschiedenen Anwendungsbereichen. Diese Klasse der *TGraphen* besitzt besondere Eigenschaften. Eine davon ist, dass alle *Knoten und Kanten typisiert* sind. Die möglichen Knoten- und Kantentypen, die ein Graph enthalten darf, werden in einem *Schema* beschrieben. Jedes Schema beschreibt eine spezielle Klasse von Graphen. Beispiele dafür sind Syntaxgraphen von Softwaresystemen, Architektur-Modelle oder Straßenkarten. Zur Modellierung von Graphenschemas wird die Modellierungssprache grUML (Graph UML) verwendet.

grUML ist eine Teilmenge der Klassendiagramme aus der Modellierungssprache UML mit zusätzlicher Graphensemantik. In grUML wird ein Schema als Klassendiagramm dargestellt, wobei die Knotenklassen des Graphen durch Klassen und die Kantenklassen durch Assoziationen beziehungsweise Assoziationsklassen repräsentiert werden. Zusätzlich muss jedes Graphenschema genau eine Klasse mit dem Stereotyp `<<graphclass>>` enthalten, die keinerlei Assoziationen besitzt. Diese Klasse definiert den Namen und die Attribute der Graphklasse [BHR+10].

Die Graphenbibliothek *JGraLab* ermöglicht es, TGraphenschemas zu erstellen und zu verändern. Außerdem bietet JGraLab die Möglichkeit, zu einem Schema korrespondierende Java-Klassen zu generieren. Damit können zu den Graphklassen passende Graphen erstellt und bearbeitet werden.

Es lassen sich aber auch Modelle und ihre Metamodelle aus anderen Modellierungssprachen als Graphen und ihre Graphenschemas interpretieren. Ein Beispiel für eine solche Modellierungssprache ist Ecore. *Ecore* ist ein Metametamodell, das innerhalb des Eclipse Modeling Framework genutzt wird, um Metamodelle zu repräsentieren.

Modelle in Ecore bestehen aus *Objekten und ihren Verflechtungen untereinander*. Werden die einzelnen Objekte als Knoten und die Beziehungen zwischen ihnen als Kanten interpretiert, lassen sich Ecore-Modelle auch als Graphen betrachten.

Metamodelle in Ecore sind in der Struktur ebenfalls eine Teilmenge von UML-Klassendiagrammen. Die Sprache enthält Klassen mit Attributen und Referenzen zwischen Klassen, doch einige Konstrukte, wie zum Beispiel die Assoziationsklassen, fehlen [SBPM08]. Ecore-Metamodelle beschreiben ihre Modelle in gleicher Weise, wie grUML-Schemas ihre Graphen. Als Graphenschema betrachtet, würden Klassen die Knotentypen darstellen, während die Referenzen als Kantentypen interpretiert werden können.

Werden Ecore-Modelle als grUML-Graphen interpretiert, können Graphenalgorithmen darauf angewendet werden. Um die vorhanden Funktionalität aus JGraLab nutzen zu können, ist eine Transformation nötig, die Ecore-Metamodelle in grUML-Schemas und Ecore-Modelle in grUML-Graphen umwandelt.

1.2 Zielsetzung

Zunächst soll eine *Abbildung* gefunden werden, die es ermöglicht ein Ecore-Metamodell in ein grUML-Schema zu transformieren. Dazu müssen beide Modellierungssprachen untersucht und die *Unterschiede* zwischen ihnen festgehalten werden. Diese beruhen hauptsächlich darauf, dass grUML-Schemas Graphensemantik besitzen, während Ecore-Metamodelle den Aufbau von Objektgeflechten beschreiben. Anschließend können *Probleme* erkannt und der Umgang damit bestimmt werden. Eine Schwierigkeit stellt beispielsweise die Tatsache da, dass bei Ecore-Modellen Kanten nur Referenzen und keine eigenständigen Objekte sind [HE10].

Anschließend soll eine *Software entworfen und implementiert* werden, welche die gefundene Abbildung realisiert. Sie soll dazu in der Lage sein, aus Ecore-Metamodellen vergleichbare grUML-Schemas zu generieren. Außerdem soll sie die passenden Ecore-Modelle in grUML-Graphen umwandeln können. In uneindeutigen Situationen sollen möglichst *gute Heuristiken* verwendet werden, um eine *automatische Transformation* zu ermöglichen. Zusätzlich soll dem *Benutzer* die Möglichkeit geboten werden, durch Angabe zusätzlicher Informationen in definierter Weise *Einfluss auf die Transformation* zu nehmen.

2 grUML

Die *Graph Unified Modeling Language (grUML)* stellt eine Teilmenge der UML-Klassendiagramme dar. Sie wurde von der AG Softwaretechnik an der Universität Koblenz entwickelt und wird dazu verwendet TGraphenschemas zu modellieren.

2.1 TGraphen

TGraphen sind eine besonders allgemeine Art von gerichteten Graphen mit besonderen Eigenschaften. Sowohl Knoten als auch Kanten sind *typisiert* und können *Attribute* besitzen. Die Menge der möglichen Knoten- und Kantentypen sowie ihre Attribute werden in TGraphenschemas beschrieben. Das Typsystem erlaubt auch *Mehrfachvererbung* bei Knoten und Kanten.

Knoten wie auch Kanten sind unabhängig voneinander identifizierbare *"First-Class-Objects"* [BHR+10]. Zudem erlaubt die Implementierung in JGraLab, die gerichteten Kanten in beide Richtungen zu durchlaufen, was es einfacher macht, sich innerhalb des Graphen zu bewegen.

Außerdem ist auf den Mengen der Knoten und Kanten eine *Ordnung* definiert. Ebenfalls geordnet sind für jeden Knoten die ein- und auslaufenden Kanten.

Alle diese Eigenschaften ermöglichen es, TGraphen als *vielseitige Datenstruktur* in unterschiedlichen Bereichen einzusetzen [BHR+10].

2.2 grUML-Schemas

In diesem Abschnitt wird der Aufbau von *TGraphenschemas* in grUML anhand eines einfachen Beispiels beschrieben. Das Beispiel-Schema, welches behandelt werden soll, wird in Abbildung 1 dargestellt.

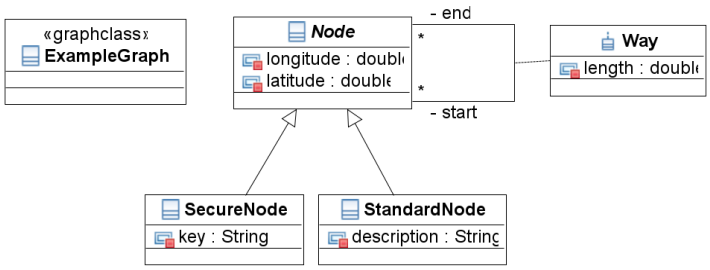

Abbildung 1: *Beispiel: Einfaches grUML-Graphenschema*

Jedes zu einem Schema gehörende grUML-Diagramm muss genau eine Klasse mit dem Stereotyp `<<graphclass>>` besitzen. Diese darf keine Assoziationen zu anderen Klassen haben, kann aber Attribute besitzen. Sie definiert den Namen der Graphklasse, die durch das Schema beschrieben wird [BHR+10]. In diesem Beispiel lautet der Name der Graphklasse `ExampleGraph`.

Alle anderen Klassen in einem grUML-Diagramm repräsentieren *Knotentypen*. Sie unterstützen Generalisierung, wobei auch Mehrfachvererbung möglich ist. Außerdem besitzen sie Attribute und können als `<<abstract>>` markiert werden [BHR+10].

Das Beispiel-Schema besteht aus einer abstrakten Knotenklasse `Node`, die als Attribute Längen- und Breitengrad vom Typ `double` besitzt. Spezialisierungen von `Node` sind die Klassen `SecureNode` und `StandardNode`. Zusätzlich zu den Attributen der Basisklasse besitzt die Klasse `SecureNode` noch ein Attribut `key` und die Klasse `StandardNode` ein Attribut `description`, beide vom Typ `String`. Das sind alle Knotenklassen, die das Beispiel-Schema bereit stellt.

Assoziationen beziehungsweise Assoziationsklassen in einem grUML-Diagramm definieren *Kantenklassen* im grUML-Schema. Ebenso wie Knotenklassen, unterstützen sie Generalisierung, können Attribute besitzen und als `<<abstract>>` markiert werden. An den Assoziationsenden werden zusätzlich Multiplizitäten und Rollennamen notiert, ihre Bedeutung ist die gleiche, wie in regulären UML Diagrammen [BHR+10].

Im Beispiel-Schema existiert eine Assoziationsklasse `Way`, die zwei `Node`-Instanzen verbindet und zusätzlich die Länge des Weges abspeichert. `Way`-Instanzen sind gerichtet, sie haben einen `start` und einen `end` Punkt. Durch die Multiplizitäten wird ausgedrückt, dass es beliebig viele Wege geben darf, die zu einem Knoten führen oder von dort kommen. Es kann auch sein, dass ein Knoten gar nicht erreichbar ist.

Für die Domains der Attribute bietet grUML verschiedene Datentypen an, die in Abschnitt 2.3 noch erläutert werden. Zwei sollen aber hier schon genannt werden, da sie im Diagramm modelliert werden.

In grUML besteht die Möglichkeit einen *Enumeration-Datentyp* zu definieren. Dazu wird wie in UML eine Klasse mit dem Stereotype `<<enumeration>>` verwendet.

Mit dem Stereotype `<<record>>` kann ein Container-Datentyp erstellt werden, der Komponenten zusammenfasst. Diese Komponenten sind Attribute mit verschiedenen Datentypen. Zyklische Beziehungen zwischen *Records* sind nicht erlaubt [BHR+10].

Ein Beispiel für einen Graph, der zu dem Beispiel-Schema passt, zeigt Abbildung 2.

Zu sehen ist, dass dieser Graph vier Knoten besitzt, wovon einer vom Typ `SecureNode` und drei vom Typ `StandardNode` sind. Alle Instanzen weisen den von `Node` geerbten Attributen `latitude` und `longitude` konkrete Werte zu. Die `SecureNode`-Instanz `v1` besitzt zusätzlich ein Schlüsselwort, das `"SecretPassword"` lautet. Die `StandardNode`-Instanzen haben als zusätzliches Attribut Beschreibungen, worum es sich bei dem Knoten handelt. So ist `v2` eine Tankstelle, `v3` ein Supermarkt und `v4` eine Schule.

Außerdem gibt es vier Kanten vom Typ `Way`. `e1` repräsentiert einen Weg mit fester Länge von `v1` zu `v2`. Zusätzlich gibt es noch einen Weg zwischen Tankstelle und Supermarkt, zwischen Supermarkt und Schule sowie zwischen Tankstelle und Schule.

Die Instanzen der Kantenklassen werden als vollwertige Objekte betrachtet, sie “kennen” jeweils ihren Start- und Endpunkt, genauso wie die verbundenen Knoten die Kante identifizieren können. Die Implementierung in JGraLab ermöglicht es dann, sich im Graph entlang der Kantenrichtung - aber auch entgegengesetzt zu bewegen.

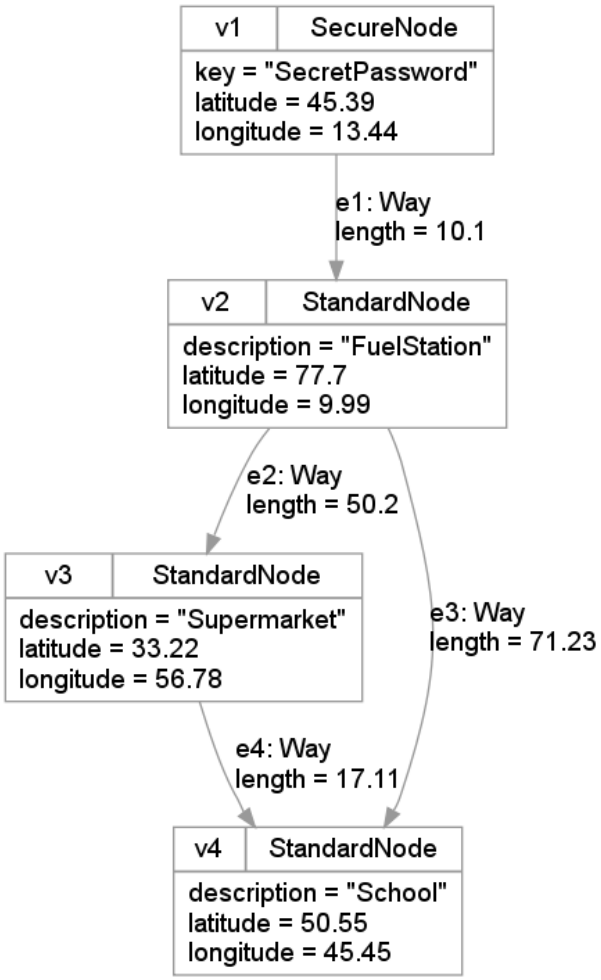

Abbildung 2: *Beispiel: Graph zum Schema*

2.3 Das grUML-Metaschema

Alle grUML-Graphenschemas werden durch das *grUML-Metaschema* spezifiziert, das aus einem “Structure”-Paket und einem “Domain”-Paket besteht. Im “Structure”-Paket, in Abbildung 3 zu sehen, werden die Zusammenhänge zwischen Knoten- und Kantenklassen sowie ihre Anordnung in Paketen beschrieben. Im “Domain”-Paket, in Abbildung 4 dargestellt, sind alle zur Verfügung stehenden Datentypen und ihre Beziehungen zueinander enthalten. Das grUML-Metaschema ist auch sein eigenes Metamodell.

Ein grUML-Schema besitzt einen `name` und ein `packagePrefix`. Es definiert genau eine `GraphClass` und besitzt genau ein `Package` als Default-Paket. Dieses ist dann automatisch das root-`Package` der Paketstruktur. `VertexClass` und `EdgeClass` erben beide von der Klasse `GraphElementClasses` und müssen daher genau einem Paket zugeordnet sein.

Jede `EdgeClass` hat einen Start- und einen Endpunkt. An beiden Enden müssen die Rollennamen, die Multiplizitäten und die Aggregationsart angegeben sein. Für die Aggregationsart stehen die Literale `NONE`, `SHARED` und `COMPOSITE` zur Auswahl. Bei normalen Kanten, wird die Aggregationsart auf beiden Seiten auf `NONE` gesetzt. Wird `SHARED` oder `COMPOSITE` gewählt,

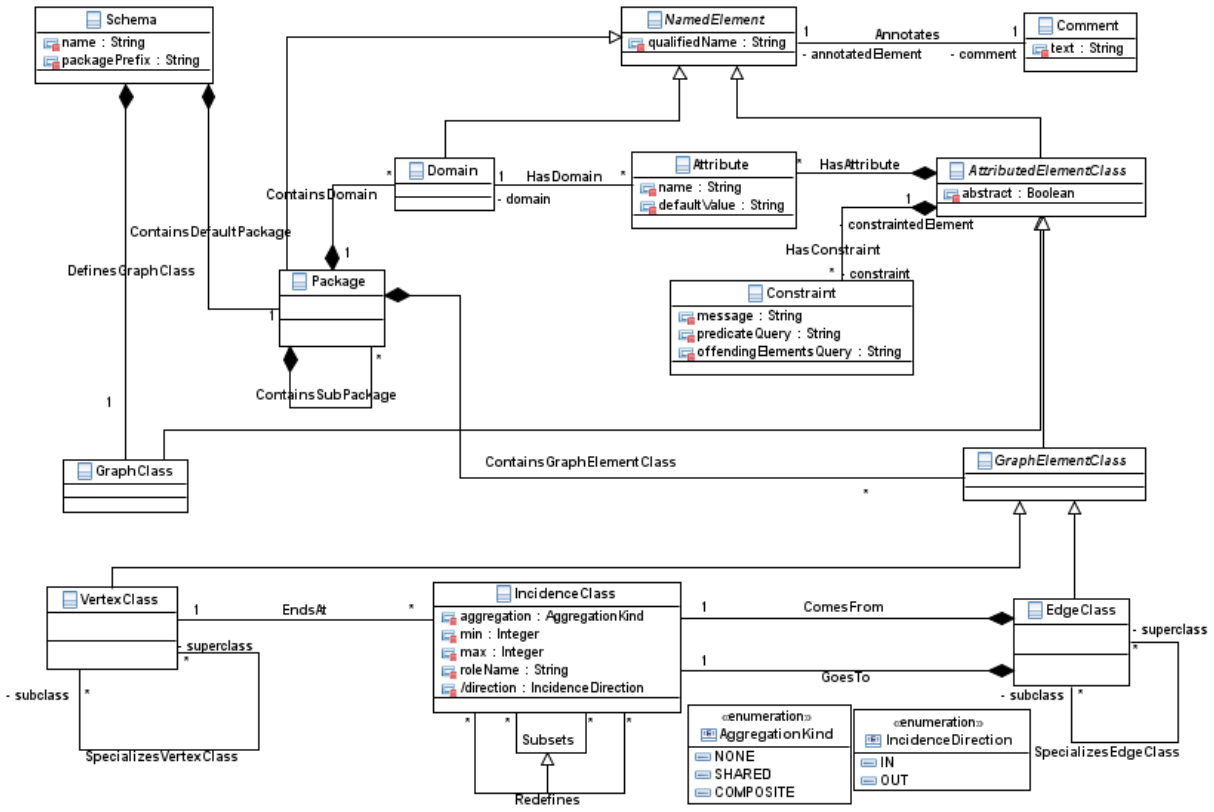

Abbildung 3: “Structure” Teil des grUML-Metaschema

modelliert das eine Teil-Ganzes Beziehung, wobei die gewählte Option auf der Teil-Seite gesetzt wird. Diese Start- und Endpunkte einer Kante werden im grUML-Metaschema durch `IncidenceClasses` repräsentiert. Eine `IncidenceClass` gehört genau zu einer `VertexClass`. Zusätzlich zu den Rollennamen wird für jede `IncidenceClass` noch gespeichert, ob sie ein Kantenende für eine ein- oder auslaufende Kante ist. Dazu existiert das abgeleitete Attribut `/direction`, das entweder `IN` oder `OUT` sein kann.

`GraphElementClass` und `GraphClass` sind `AttributedElementClasses`, können also beliebig viele `Attribute` besitzen. Jedes Attribut hat eine `Domain`. Welche `Domains` in grUML vorhanden sind, zeigt Abbildung 4.

Die Basistypen String, Integer, Long, Double und Boolean werden selbstverständlich unterstützt. Zusätzlich gibt es Enumerations, Collections wie Lists und Sets, Maps sowie zusammengesetzte Datentypen, genannt `Record`.

Außerdem bietet grUML die Möglichkeit, Kantenklassen, Knotenklassen und die Graphklasse mit `Constraints` zu versehen. `Constraints` dienen dazu, Regeln aufzustellen, die sich durch die normale Diagramm-Syntax nicht ausdrücken lassen. Um diese Regeln exakt zu definieren, kann in grUML GReQL verwendet werden. Das soll hier aber nicht näher erläutert werden. Mehr Informationen dazu finden sich unter [EB10].

Zusätzlich können `Comments` an Paketen, Datentypen, die Graphklasse und die Graphelemente angehängt werden. Diese haben die selbe Semantik wie in UML und können Text jeglicher Art enthalten.

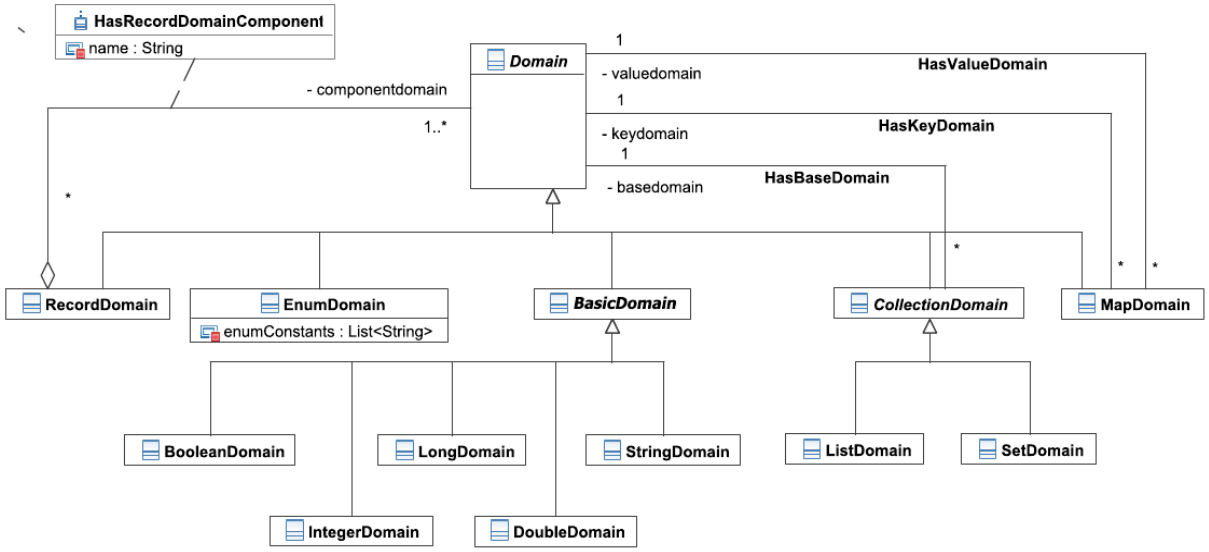

Abbildung 4: “Domain” Teil des grUML-Metaschema

Abbildung 5 zeigt das Beispiel-Schema als Schema-Graph.

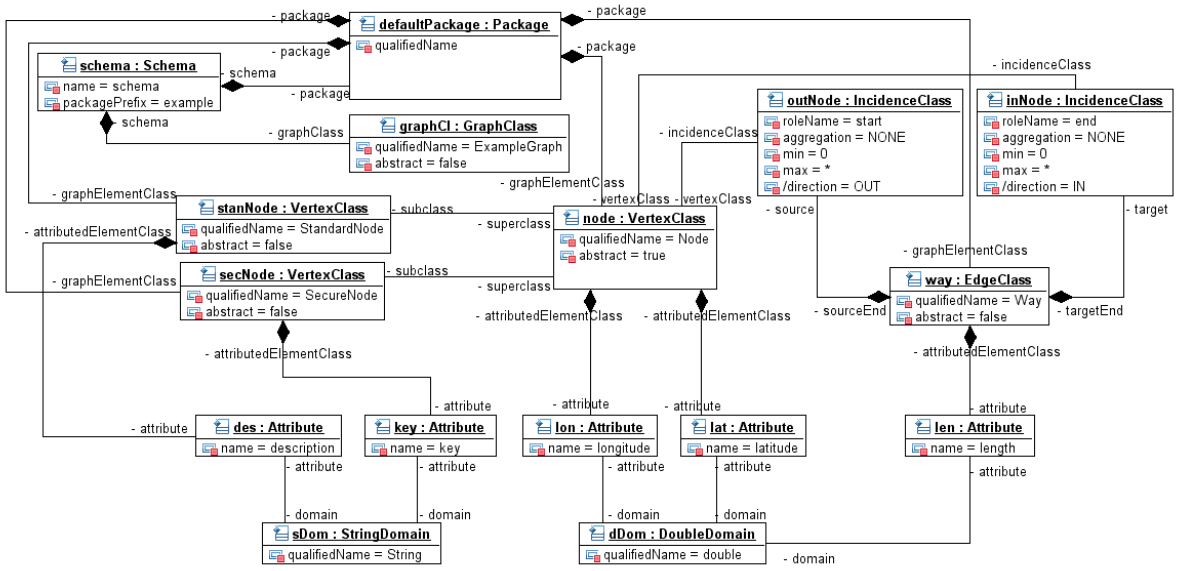

Abbildung 5: Beispiel-Schema als Instanz des grUML-Metaschemas

Die Kantenklasse `Way` mit ihrem Attribut `length` wird hier durch ein `EdgeClass`-Objekt, zwei `IncidenceClass`-Objekte und ein `Attribute`-Objekt mit zugehörigem `DoubleDomain`-Objekt repräsentiert. Bei größeren Diagrammen mit vielen Assoziationen lässt sich ein solches Objektdiagramm nicht mehr gut visuell darstellen.

2.4 TG-Dateiformat

grUML-Schemas und grUML-Graphen werden im *TG-Dateiformat* gespeichert. Dabei besteht eine TG-Datei entweder nur aus dem Schema oder aus dem Schema und einem passenden Graphen.

Im Folgenden wird kurz der Aufbau einer TG-Datei anhand des Beispiel-Schemas erläutert. Eine genaue Definition in EBNF findet sich unter [BHR+10].

Der Inhalt der TG-Datei für das Beispiel-Schema aus Abbildung 1 sieht folgendermaßen aus:

```
1 TGraph 2;
2 Schema example.schema.ExampleSchema;
3 GraphClass ExampleGraph;
4 abstract VertexClass Node { latitude: Double, longitude: Double };
5 VertexClass SecureNode: Node { key: String };
6 VertexClass StandardNode: Node { description: String };
7 EdgeClass Way from Node (0,*) role start
8             to Node (0,*) role end { length: Double = "0" };
```

Jede TG-Datei besitzt einen *Header*, der aus dem Literal `TGraph` und der Versionsnummer besteht. Die aktuelle Versionsnummer ist 2.

Nach dem Header wird zunächst der Name des Schemas und der der Graphklasse genannt. Direkt danach werden die verwendeten *Pakete* aufgeführt. Sub-Pakete werden definiert, indem das übergeordnete Paket durch einen Punkt getrennt in den Namen des Sub-Paketes mit einfließt. Existiert zum Beispiel ein Paket `vehicles` und darin ein Paket `cars`, dann wird diese Struktur so beschrieben: `Package vehicles; Package vehicles.cars;`.

Das Beispiel-Schema hat keine Paketstruktur, daher befinden sich alle Elemente im Default-Paket. Sollte sich jedoch ein Datentyp, eine Kanten- oder Knotenklasse in einem Paket befinden, so muss, wie bei Sub-Paketen, der Name des Paketes durch einen Punkt abgetrennt vor den Namen des Elementes gesetzt werden. Eine Knotenklasse `Node` im Paket `structure` würde demnach die Bezeichnung `structure.Node` tragen.

Neben den Paketen werden eigene Datentypenklassen wie Enumerations und Records definiert. Da im Beispiel-Schema kein neuer Datentyp definiert wurde, fehlen diesbezügliche Einträge hier.

Knotenklassen werden mit ihren Bezeichnungen und Attributen beschrieben. Generalisierungen können notiert werden, indem die Superklassen durch einen Doppelpunkt getrennt hinter dem Namen der spezialisierten Klasse genannt werden. Mehrere Superklassen werden voneinander mit Komma getrennt.

Im Beispiel-Schema sind die Klassen `SecureNode` und `StandardNode` von der abstrakten Superklasse `Node` abgeleitet. Aus diesem Grund lauten die Bezeichnungen `SecureNode: Node` und `StandardNode: Node`. Die Abstraktheit der Superklasse `Node` wird durch das Schlüsselwort `abstract` vor dem Elementtyp, hier `VertexClass`, definiert.

Kantenklassen werden ebenfalls aufgeführt. Neben Bezeichnung und Attributen wird hier zusätzlich genannt, welche Knotenklassen verbunden werden, sowie Multiplizitäten und Rollennamen. Falls eine Kantenklasse eine Teil-Ganzes Beziehung modelliert, wird dies durch den Zusatz `aggregation shared` oder entsprechend `aggregation composite` ausgedrückt.

Das Beispiel-Schema enthält eine Kantenklasse mit Namen `Way`. Genau wie im Diagramm, kann hier nun spezifiziert werden, dass es beliebig viele `Ways` zwischen `Nodes` gibt und dass ein `Way` immer `start` und `end` verbindet.

Zusätzlich können noch *Kommentare* zu den vorher beschriebenen Elementen angefügt werden. Diese Möglichkeit wird im Beispiel-Schema jedoch nicht genutzt.

Mit diesen Komponenten lässt sich ein Graphenschema vollständig beschreiben.

Wird ein Graph mitgespeichert, ändert sich der Anfang der TG-Datei nicht. Der Graph wird einfach am Ende angefügt. Das Beispiel-Schema mit Beispiel-Graph sieht dann folgendermaßen aus:

```
1  TGraph 2;
2  Schema example.schema.ExampleSchema;
3  GraphClass ExampleGraph;
4  abstract VertexClass Node { latitude: Double, longitude: Double };
5  VertexClass SecureNode: Node { key: String };
6  VertexClass StandardNode: Node { description: String };
7  EdgeClass Way from Node (0,*) role start
8                 to Node (0,*) role end { length: Double = "0" };
9  Graph "SmallGraph" 36 ExampleGraph (1000 1000 4 4);
10 Package ;
11 1 SecureNode <1> "SecretPassword" 456.39 83.44;
12 2 StandardNode <-1 2 3> "Fuel station" 770.7 9999.99;
13 3 StandardNode <-2 4> "Supermarket" 33.22 56.78;
14 4 StandardNode <-3 -4> "School" 12345.6 6433.33;
15 1 Way 9921.529855753093;
16 2 Way 9970.521644051529;
17 3 Way 12111.9516827636;
18 4 Way 13865.608214820582;
```

Der *Header* des Graphen beginnt mit dem Schlüsselwort `Graph` gefolgt von einer eindeutigen Graph-ID. Anschließend kommt die Graph-Version und der Name der Graphklasse. Dann wird spezifiziert, wie viele Knoten und Kanten der Graph maximal enthalten darf und wie viele aktuell enthalten sind. Falls die Graphklasse Attribute besitzt, werden sie im Anschluss aufgeführt.

Im Falle unseres Beispiel-Graphen lautet die ID `SmallGraph`, die Version `36` und der Name der Graphklasse `ExampleGraph`. Es darf maximal `1000` Knoten und `1000` Kanten geben, bisher hat der gespeicherte Graph jeweils `4` davon. Die maximale Anzahl ist jedoch keine feste Grenze, sondern kann dynamisch angepasst werden.

Nach dem Header werden zuerst die *Knoten* beschrieben. Die Schreibweise sieht hier vor, zunächst die Nummer und dann den Knotentyp zu speichern. Darauf folgen die Verweise auf ein- oder auslaufende Kanten und zum Schluss die Attributwerte. Das Minus vor dem Kantenverweis definiert, dass es sich hier um eine einlaufende Kante handelt.

Im Anschluss werden dann die *Kanten*, ebenfalls nummeriert, mit ihren Attributwerten aufgeschrieben.

Für unseren Beispiel-Graphen wird notiert, dass Knoten `2` vom Typ `StandardNode` ist, wobei das `description` Attribut mit dem Wert `"FuelStation"` gefüllt ist, während die geerbten

Attribute `latitude` und `longitude` die Werte `770.7` und `9999.99` haben. Weiterhin ist erkennbar, dass der Knoten `2` eine eingehende und zwei ausgehende Kanten hat. Die eingehende Kante ist die mit der Nummer `1`, die von dem Knoten `1` kommt. Die ausgehenden Kanten sind `2` und `3`, die zu den Knoten `3` und `4` führen.

2.5 JGraLab für Schemas

Um grUML-Graphenschemas und Graphen zu verarbeiten, wird die *Graphenbibliothek JGraLab* benutzt. Sie bietet vielfältige Funktionalität zum Erstellen, Verändern, Laden und Speichern von Schemas und Graphen. In diesem Abschnitt sollen nun die wichtigsten Aspekte im Umgang mit Schemas gezeigt werden.

2.5.1 Erstellen eines Schemas

Eine Möglichkeit, ein Graphenschema als *Objektmodell* im Programmcode zur Verfügung zu stellen, ist das direkte Erzeugen der gewünschten Knoten- und Kantenklassen. Im Folgenden soll dies anhand des Beispiel-Schemas kurz erläutert werden.

Zunächst wird ein `Schema`-Objekt erzeugt. Dem Konstruktor wird der Name des Schemas, in diesem Fall “ExampleSchema” und das Paket-Präfix, hier “example.schema“, übergeben.

```
1 //Generating a new Schema Object
2 Schema schema = new SchemaImpl("ExampleSchema", "example.schema");
```

Anschließend wird zu diesem Schema die `GraphClass` instanziiert. Als Parameter ist hier der Name, der für das Beispiel-Schema ”ExampleGraph“ lautet, nötig.

```
1 //Generating a GraphClass Object for the new Schema Object
2 GraphClass graphclass = schema.createGraphClass("ExampleGraph");
```

Nun können die Knoten- und Kantenklassen hinzugefügt werden. Für unser Beispiel-Schema sieht die Erzeugung des abstrakten Knotens `Node` mit den Attributen `latitude` und `longitude` so aus:

```
1  //Generating the VertexClass "Node"
2  //--- First, create an empty VertexClass with the Name "Node"
3  VertexClass node = graphclass.createVertexClass("Node");
4  //--- Declare the "Node" as abstract
5  node.setAbstract(true);
6  //--- Add an attribute with the name "latitude",
7  //--- the domain "Double" and no default value
8  node.addAttribute("latitude", schema.getDoubleDomain(), null);
9  //--- Add an attribute with the name "longitude",
10 //--- the domain "Double" and no default value
11 node.addAttribute("longitude", schema.getDoubleDomain(), null);
```

Zuerst wird ein leerer Knoten mit dem Namen `Node` erstellt und als `abstract` definiert. Im Anschluss werden die Attribute hinzugefügt. Für jedes Attribut muss eine Bezeichnung, der Datentyp und ein Default-Wert angegeben werden, wobei letzterer auch `null` sein kann.

Die von `Node` abgeleiteten Knoten `SecureNode` und `StandardNode` werden wie folgt angelegt:

```
1  //Generating the VertexClass "SecureNode" that is a specialized "Node"
2  //--- First, create an empty VertexClass with the Name "SecureNode"
3  VertexClass secNode = graphclass.createVertexClass("SecureNode");
4  //--- Add "Node" as SuperClass to "SecureNode"
5  secNode.addSuperClass(node);
6  //--- Add an attribute with the name "key",
7  //--- the domain "String" and no default value
8  secNode.addAttribute("key", schema.getStringDomain(), null);
9
10 //Generating the VertexClass "StandardNode" that is a specialized "Node"
11 //--- First, create an empty VertexClass with the Name "StandardNode"
12 VertexClass stanNode = graphclass.createVertexClass("StandardNode");
13 //--- Add "Node" as SuperClass to "StandardNode"
14 stanNode.addSuperClass(node);
15 //--- Add an attribute with the name "description",
16 //--- the domain "String" and no default value
17 stanNode.addAttribute("description", schema.getStringDomain(), null);
```

Durch die Methode `addSuperClass(VertexClass superClass)` kann die zuvor instanziierte `VertexClass Node` als Superklasse der neuen `VertexClasses SecureNode` und `StandardNode` deklariert werden. Zusätzliche Attribute können dann wie zuvor hinzugefügt werden.

Im Anschluss wird die Kantenklasse `Way` erzeugt.

```
1  //Generating the Edge "Way"
2  //--- First, create an empty EdgeClass with the Name "Way"
3  EdgeClass way = graphclass.createEdgeClass("Way",
4                  //from node with rolename "start" with the multiplicity 0..*
5                  node, 0, Integer.MAX_VALUE, "start", AggregationKind.NONE,
6                  //to node with rolename "end" with the multiplicity 0..*
7                  node, 0, Integer.MAX_VALUE, "end", AggregationKind.NONE);
8  //--- Add an attribute with the name length,
9  //--- the domain "Double" and the default value 0
10 way.addAttribute("length", schema.getDoubleDomain(), "0");
```

Zur Erzeugung eines `EdgeClass`-Objektes sind eine Reihe von Parametern nötig. Diese sind:

- Ein eindeutiger Name zur Identifikation
- Die Knotenklasse von der die Kante starten soll
- Die untere Grenze der Multiplizität am Startpunkt der Kante
- Die obere Grenze der Multiplizität am Startpunkt der Kante
- Der Rollenname für den Knotentyp am Startpunkt der Kante
- Inwieweit der Knotentyp am Startpunkt der Kante Teil in einer Teil-Ganzes Beziehung ist
- Die Knotenklasse, an der die Kante enden soll
- Die untere Grenze der Multiplizität am Endpunkt der Kante

- Die obere Grenze der Multiplizität am Endpunkt der Kante
- Der Rollenname für den Knotentyp am Endpunkte der Kante
- Inwieweit der Knotentyp am Endpunkt der Kante Teil in einer Teil-Ganzes Beziehung ist

Anschließend kann das neue `EdgeClass`-Objekt noch Attribute erhalten, wie `VertexClasses` auch. In diesem Beispiel wird noch ein Attribut mit Namen `length` vom Typ Double mit dem Default-Wert `0` hinzugefügt.

Damit wird das Beispiel-Schema als Objektmodell im Speicher gehalten.

2.5.2 Laden eines Schemas

Eine andere Möglichkeit, das Objektmodell im Programm zur Verfügung zu haben, besteht darin, das Schema aus der TG-Datei zu laden.

```
1 //Loading a schema from File
2 Schema schema = GraphIO.loadSchemaFromFile("ExampleSchema.tg");
```

Damit lässt sich sofort das komplette Schema in den Speicher laden und mit den Methoden `getVertexClassesInTopologicalOrder()` und `getEdgeClassesInTopologicalOrder()` kann auf die Knoten- und Kantenklassen zugegriffen werden. Als Parameter wird nur der Dateiname benötigt.

2.5.3 Speichern eines Schemas

Speichern lässt sich ein Graphenschema sehr ähnlich:

```
1 //Save the Example Schema as TG-File
2 GraphIO.saveSchemaToFile("SimpleExample.tg", schema);
```

Als Parameter werden der gewünschte Dateiname und das `Schema`-Objekt übergeben.

2.6 JGraLab für Graphen

Auf der Ebene der Graphen bietet JGraLab die Möglichkeit, einen Graph passend zu einem Schema zu erstellen, ihn zu speichern und anschließend wieder zu laden. Im diesem Abschnitt soll erläutert werden, wie JGraLab dazu genutzt werden kann.

2.6.1 Schema-spezifische Erstellung von Graphen

JGraLab ist dazu in der Lage, zu einem Schema *Javaklassen* zu generieren, mit denen dann Graph-Instanzen angelegt werden können. Die einfachste Möglichkeit ist, sich die Klassen zunächst abzuspeichern. Mit folgender Codezeile lassen sich Javaklassen zu unserem Beispiel-Schema erstellen:

```
1 //Generate Java code
2 schema.commit("GeneratedCode/SimpleExample", CodeGeneratorConfiguration.FULL);
```

Die generierten Klassen liegen nun im angegebenen Ordner.

Anschließend können die erstellten Klassen zu einem Projekt hinzugefügt und genutzt werden. Wie der Beispiel-Graph aus Abbildung 2 erstellt werden kann, sollen die nächsten Abschnitte beschreiben.

Zunächst muss ein leerer Graph der Graphklasse `ExampleGraph` instanziiert werden.

```
1 //Creating a new Graph
2 ExampleGraph graph = ExampleSchema.instance().createExampleGraph("SmallGraph");
```

Im Anschluss können die Knoten erzeugt werden:

```
1
2 //Creating a SecureNode
3 SecureNode sec1 = graph.createSecureNode();
4 //--- Filling Attributes with values
5 sec1.set_latitude(45.39);
6 sec1.set_longitude(13.44);
7 sec1.set_key("SecretPassword");
8
9 //Creating a StandardNode
10 StandardNode stan1 = graph.createStandardNode();
11 //--- Filling Attributes with values
12 stan1.set_latitude(77.70);
13 stan1.set_longitude(9.99);
14 stan1.set_description("FuelStation");
15
16 //Creating another StandardNode
17 StandardNode stan2 = graph.createStandardNode();
18 //--- Filling Attributes with values
19 stan2.set_latitude(33.22);
20 stan2.set_longitude(56.78);
21 stan2.set_description("Supermarket");
22
23 //Creating yet another StandardNode
24 StandardNode stan3 = graph.createStandardNode();
25 //--- Filling Attributes with values
26 stan3.set_latitude(50.55);
27 stan3.set_longitude(45.45);
28 stan3.set_description("School");
```

Für jeden Knoten wird zuerst ein entsprechendes leeres Knoten-Objekt instanziiert. Anschließend können die Attribute besetzt werden. Jede Knotenklasse besitzt für ihre jeweiligen Attribute passende Getter- und Setter-Methoden.

Die Kanten werden ähnlich erzeugt:

```
1
2 //Creating a Way between the SecretNode and the "FuelStation"
3 Way w1 = graph.createWay(sec1, stan1);
```

```
4  //—— Setting the length of the way
5  w1.set_length(10.1);
6
7  //Creating a Way between the "FuelStation" and the "Supermarket"
8  Way w2 = graph.createWay(stan1, stan2);
9  //—— Setting the length of the way
10 w2.set_length(50.2);
11
12 //Creating a Way between the "FuelStation" and the "School"
13 Way w3 = graph.createWay(stan1, stan3);
14 //—— Setting the length of the way
15 w3.set_length(71.23);
16
17
18 //Creating a Way between the "Supermarket" and the "School"
19 Way w4 = graph.createWay(stan2, stan3);
20 //—— Setting the length of the way
21 w4.set_length(17.11);
```

Als erstes wird ein `Way`-Objekt erstellt, das ein `Node`-Objekt als Start- und eines als Endpunkt besitzt. Danach kann das Längen-Attribut `length` mit dem gewünschten Wert gefüllt werden.

2.6.2 Generische Erstellung von Graphen

Auch im Hinblick auf das Ziel dieser Arbeit, ist es nicht immer sinnvoll, sich die Javaklassen als Dateien erstellen zu lassen. Falls ein Schema erzeugt wird und im gleichen Projekt ein passender Graph instanziiert werden soll, ergibt sich das Problem, dass die frisch erzeugten Klassen zunächst kompiliert und geladen werden müssen, bevor Objekte von ihnen erzeugt werden können. Um dies ohne große Schwierigkeiten zu ermöglichen, bietet JGraLab entsprechende Funktionalität an.

Der folgende Befehl erzeugt die Javaklassen zu dem `Schema`-Objekt, kompiliert sie intern und hält sie dann im Speicher.

```
1 schema.compile(CodeGeneratorConfiguration.FULL);
```

Anschließend stehen die Klassen zur Verfügung. Da jedoch erst zur Laufzeit entschieden wird, welche Klassen erzeugt werden, kann jetzt im Falle des Beispiel-Schemas nicht einfach, so wie zuvor, eine Instanz von `ExampleGraph` erstellt werden. Stattdessen bietet die `Schema`-Klasse an, eine Methode zu liefern, mit der ein passender Graph zum Schema instanziiert werden kann.

```
1 //Creating a graph
2 Method graphCreateMethod = schema.getGraphCreateMethod(false);
3 Object [] a = {"ExampleGraph", 40, 50};
4 Graph graph = (Graph)graphCreateMethod.invoke(null, a);
```

Diese Methode benötigt als Parameter, den Namen der Graphklasse, sowie die maximale Anzahl Kanten und Knoten, die im Graph erlaubt sind. Da die Methode statisch ist, ist kein

Objekt nötig, um sie auszuführen, stattdessen wird einfach `null` übergeben. Der Rückgabetyp der Methode ist `Graph`, im Falle des Beispiel-Schemas wird allerdings ein `ExampleGraph`-Objekt erstellt, da die Klasse `ExampleGraph` eine Spezialisierung von `Graph` ist.

Nun können die Knoten erzeugt werden. Dazu bietet die Klasse `Graph` die Methode mit dem Namen `CreateVertex` an, welche als Parameter die zu instanziierende Klasse erhält.

```
1 //Creating a SecureNode
2 Vertex sec1 = graph.createVertex(secNode.getM1Class());
```

In unserem Beispiel soll ein `SecureNode` erstellt werden. Übergeben wird hier die Klasse, welche aus dem `secNode`-Objekt erstellt wurde. `secNode` ist die Instanz von `VertexClass`, welche bei der Schema-Erzeugung instanziiert wurde, um den "SecureNode" zu modellieren. Anschließend können die Attribute des `Vertex`-Objektes mit Werten gefüllt werden:

```
1 sec1.setAttribute("latitude", 45.39);
2 sec1.setAttribute("longitude", 13.44);
3 sec1.setAttribute("key", "SecretPassword");
```

Da wiederum nicht bekannt ist, welchen spezialisierten Typ die `Vertex`-Instanz hat, müssen die Namen der Attribute zusammen mit der gewünschten Wertebelegung übergeben werden.

Als nächstes werden die `StandardNodes` erstellt. Da die Codezeilen analog zum `SecureNode` sind, muss dies hier nicht noch einmal näher erläutert werden.

```
1
2 //Creating a StandardNode
3 Vertex stan1 = graph.createVertex(stanNode.getM1Class());
4 stan1.setAttribute("latitude", 77.70);
5 stan1.setAttribute("longitude", 9.99);
6 stan1.setAttribute("description", "FuelStation");
7
8 //Creating another StandardNode
9 Vertex stan2 = graph.createVertex(stanNode.getM1Class());
10 stan2.setAttribute("latitude", 33.22);
11 stan2.setAttribute("longitude", 56.78);
12 stan2.setAttribute("description", "Supermarket");
13
14 //Creating another StandardNode
15 Vertex stan3 = graph.createVertex(stanNode.getM1Class());
16 stan3.setAttribute("latitude", 50.55);
17 stan3.setAttribute("longitude", 45.45);
18 stan3.setAttribute("description", "School");
```

Um Kanten zu erstellen, wird die Methode `createEdge` der Klasse `Graph` genutzt. Mit Hilfe des Kantenklassen-Objektes wird so eine Instanz erzeugt.

```
1 //Creating a Way between sec1 and stan1
2 Edge w1 = graph.createEdge(way.getM1Class(), sec1, stan1);
3 w1.setAttribute("length", 10.1);
```

Im Beispiel wird das `way`-Objekt als Instanz von `EdgeClass` benötigt. Als Parameter verlangt die Methode die Klasse, welche instanziiert werden soll und zwei `Vertex`-Objekte, den

Startknoten sowie den Endknoten der Kante. Anschließend können die Attribute des `Edge`-Objektes belegt werden.

Die restlichen `Way`-Instanzen werden analog erzeugt.

```
1  //Creating a Way between stan1 and stan2
2  Edge w2 = graph.createEdge(way.getM1Class(), stan1, stan2);
3  w2.setAttribute("length", 50.2);
4
5  //Creating a Way between stan1 and stan3
6  Edge w3 = graph.createEdge(way.getM1Class(), stan1, stan3);
7  w3.setAttribute("length", 71.23);
8
9  //Creating a Way between stan2 and stan3
10 Edge w4 = graph.createEdge(way.getM1Class(), stan2, stan3);
11 w4.setAttribute("length", 17.11);
```

Damit besteht die Möglichkeit, dynamisch zur Laufzeit Knoten zu instanziieren, auch wenn das passende Schema zur Compilezeit noch nicht bekannt ist.

2.6.3 Speichern eines Graphen

Soll der erstellte Graph nun gespeichert werden, so geschieht das mit Hilfe der `GraphIO` Klasse. Als Parameter werden hier der Dateiname und das `Graph`-Objekt übergeben.

```
1  //Saving the graph into a TG-File
2  GraphIO.saveGraphToFile("SimpleExample_WithGraph.tg",
3                                  graph, new ProgressFunctionImpl());
```

2.6.4 Laden eines Graphen

Das Laden funktioniert dann entsprechend. Als Parameter wird der Dateiname erwartet.

```
1  //Loading a graph from a TG-File
2  Graph g = GraphIO.loadGraphFromFile("SimpleExample_WithGraph.tg",
3                                              new ProgressFunctionImpl());
```

2.7 Zusammenfassung

Die Graph Unified Modeling Language ermöglicht es Graphenschemas auf einfache Weise mit Hilfe von UML-Klassendiagrammen zu modellieren. Welche UML-Konstrukte zur Beschreibung von Graphen erlaubt sind und welche Semantik sie in diesem Fall besitzen, wird durch das grUML-Metaschema ausgedrückt. Zu den resultierenden Graphenschemas können verschiedene Graph-Instanzen erzeugt werden. Gespeichert werden sowohl Graphenschemas als auch die dazu passenden Graphen in einer `"*.tg"`-Datei, wobei ein selbst entworfenes Datei-Format genutzt wird. Um Graphen und ihre Schemas programmatisch zu erstellen, zu modifizieren oder zu traversieren steht die Graphenbibliothek JGraLab zur Verfügung.

3 Ecore

Ecore ist ein *Metametamodell* für eine Teilmenge von UML-Klassendiagrammen. Es ist Bestandteil des Eclipse Modeling Frameworks und wird dort dazu genutzt, Metamodelle zu repräsentieren. Ecore ist, ebenso wie das grUML-Metaschema, sein eigenes Metamodell.

3.1 EMF

Das Eclipse Modeling Framework bietet die Möglichkeit UML Klassendiagramme, XML-Dateien und Javacode zu vereinen.

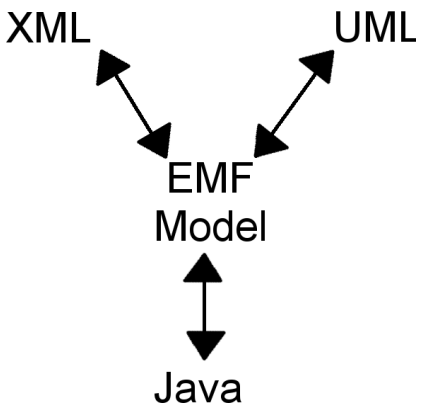

Abbildung 6: *EMF vereinigt Java, XML und UML [SBPM08]*

Alle drei Sprachen ermöglichen die *Beschreibung eines Metamodells* für ein gewähltes Szenario. Ausgehend von einer der drei Varianten kann mittels EMF ein Ecore-Metamodell erstellt werden und daraus dann die Metamodelle für die anderen beiden Sprachen.

Angenommen es soll eine Bibliothek modelliert werden. Sie soll einen Namen und eine Anschrift haben, sowie eine Sammlung von Büchern. Ein Buch soll einen Titel, einen Autor und eine ISBN-Nummer besitzen. Dann ergeben sich für die jeweiligen Sprachen folgende Metamodelle.

In *UML* kann das Beispiel Szenario als einfaches Klassendiagramm mit den Klassen `Library` und `Book` modelliert werden.

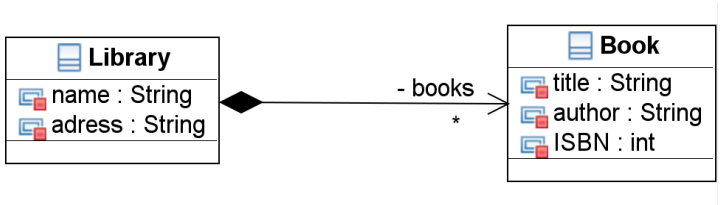

Abbildung 7: *UML Modell [SBPM08]*

In *Java* lassen sich zwei Klassen entwerfen.

```java
public class Library{

    String getName();
    void setName(String value);

    String getAdress();
    void setAdress(String value);

    List<Book> getBooks();
}

public class Book{

    String getTitle();
    void setTitle(String value);

    String getAuthor();
    void setAuthor(String value);

    int getISBN();
    void setISBN(int value);
}
```

Ein entsprechendes *XML-Schema* könnte so aussehen:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema = xmlns:xsd="http://www.w3.org/2001/XMLSchema"
              xmlns:lib="http://www.example.com/SimpleLIB"
              targetNamespace="http://www.example.com/SimpleLIB">
   <xsd:complexType name = "Library">
      <xsd:sequence>
         <xsd:element name="name" type="xsd:string"/>
         <xsd:element name="adress" type="xsd:string"/>
         <xsd:element name="books" type="lib:Book"
                      minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="Book">
      <xsd:sequence>
         <xsd:element name="title" type="xsd:string"/>
         <xsd:element name="author" type="xsd:string"/>
         <xsd:element name="ISBN" type="xsd:int"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:schema>
```

Prinzipiell kann jede der drei Formen als Ausgangsbasis für ein Ecore-Metamodell gewählt werden. Sollen die Java-Interfaces genutzt werden, sind allerdings noch *Annotationen* nötig. Jede Klasse und Methode, die Teil des Modells sein soll, muss mit `@model` annotiert werden. Das ist nötig, da EMF die Möglichkeit bietet, zwischen dem Ecore-Modell und der Java-Implementation zu wechseln. So beeinflussen nachträgliche Änderungen am Modell und anschließendes Neugenerieren der Javaklassen keine vom Benutzer erzeugten Methoden. Der

neu erzeugte Javacode und die alten Klassen mit den Benutzeränderungen werden mit Hilfe der Annotationen zusammengeführt [SBPM08].

3.2 Ecore-Metamodelle

Ecore-Metamodelle können mit den Eclipse Modeling Tools entworfen werden. Abbildung 8 zeigt ein Bibliothekensystem als Beispiel für ein Ecore-Metamodell.

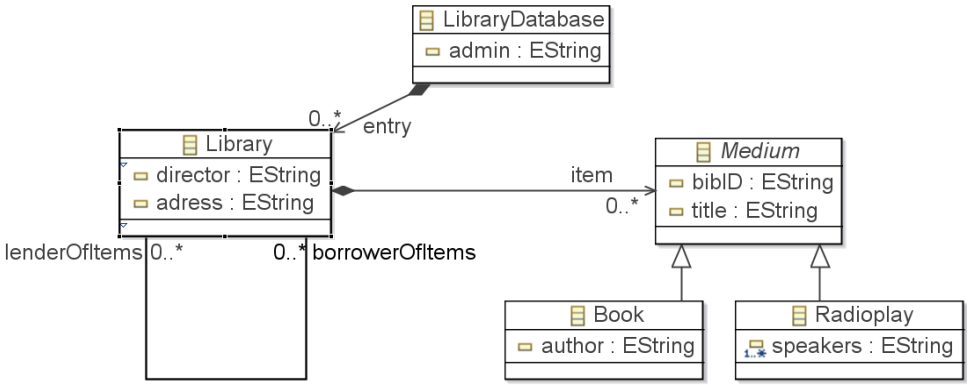

Abbildung 8: *Beispiel für ein Ecore-Metamodell*

Das Bibliothekensystem besteht aus einer `LibraryDatabase` in der `Libraries` als Einträge verzeichnet sind. Die Komposition signalisiert, dass die Bibliotheken in der Datenbank enthalten sind.

`Libraries` besitzen einen `director` und eine `adress` vom Typ `EString` als Attribute. Außerdem enthalten sie `items` vom abstrakten Typ `Medium`. Jedes `Medium` besitzt eine eindeutig `bibID` unter der es aufgeführt ist und einen `title`. Zusätzlich können `Libraries` `Mediums` an andere `Libraries` verleihen.

Konkrete Spezialisierungen von `Medium` sind `Book` und `Radioplay`. Ein `Book` besitzt einen `author` während ein `Radioplay` beliebig viele `speakers` haben darf.

3.3 Das Ecore-Metametamodell

Alle Ecore-Metamodelle werden durch das Ecore-Metametamodell beschrieben. Abbildung 9 zeigt die Struktur.

Im Zentrum des Modells stehen die `EClassifier`. Das können entweder `EClasses` oder `EDataTypes` sein. `EClassifier` müssen immer genau in einem `EPackage` enthalten sein. `EPackages` können Subpackages haben, sie besitzen eine URI um eindeutig identifiziert werden zu können sowie ein `EFactory` Objekt als `eFactoryInstance`. Damit können später, wenn Java Code zu dem Modell generiert wurde, Instanzen der Klassen erstellt werden, die in dem `EPackage` enthalten sind.

`EClasses` modellieren Klassen und Interfaces. Sie können als `abstract` definiert werden und eine `EClass` kann die Superklasse einer anderen `EClass` sein. `EClasses` besitzen `EAttributes`

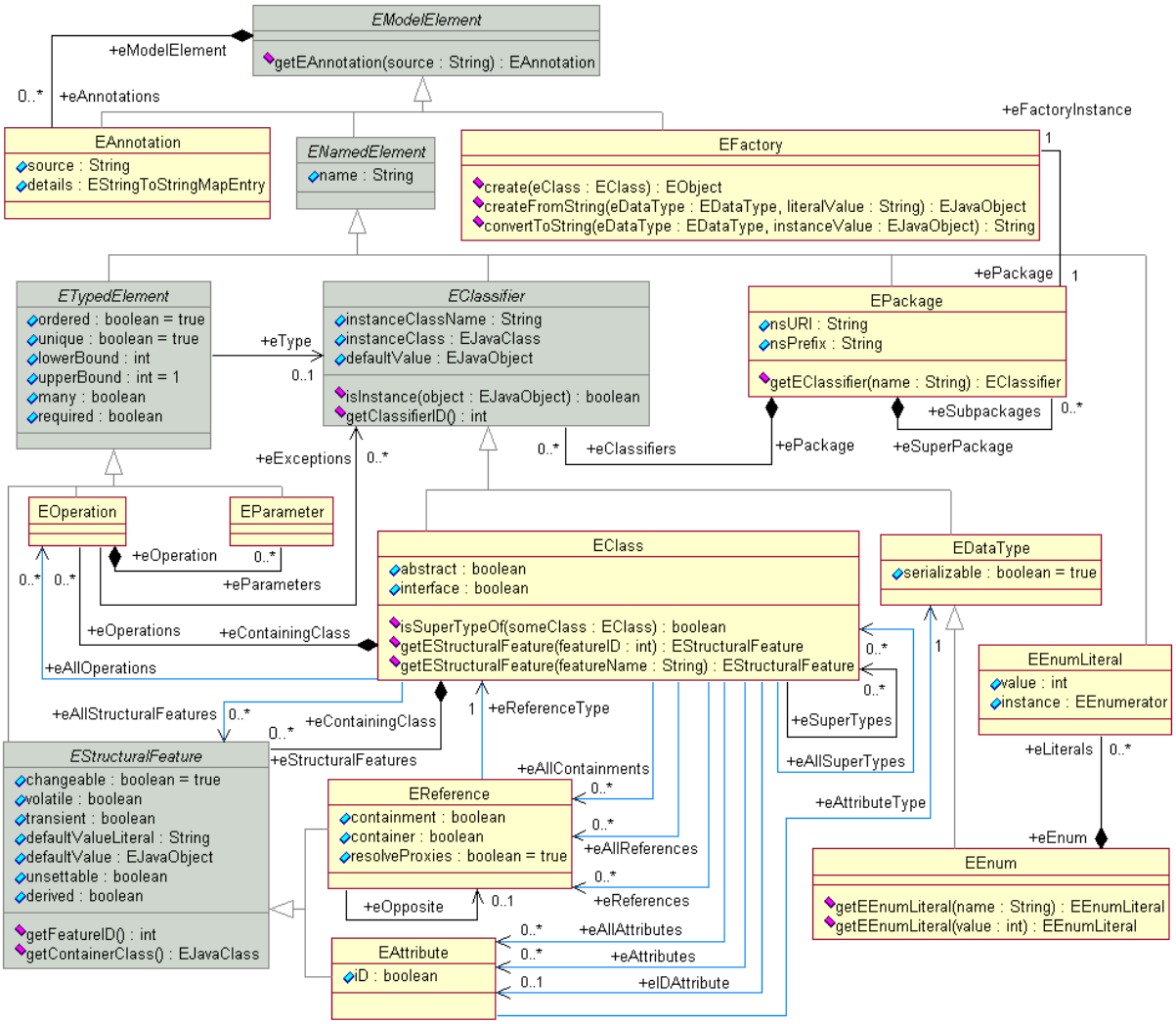

Abbildung 9: *Das Metametamodell Ecore* [lit]

und `EReferences`, die zu `EStructuralFeatures` zusammengefasst werden können. Außerdem können `EClasses` noch `EOperations` besitzen.

`EAttributes` haben einen Namen und einen `EDataType`. Auffällig ist hier, dass der Typ eines `EAttributes` keine `EClass` sein kann. Solche Beziehungen müssen also zwangsläufig als `EReference` modelliert werden.

Eine `EReference` gehört zu einer `EClass` und zeigt auf eine `EClass`. Durch die boolsche Variable `containment` kann ausgedrückt werden, ob die EReference eine Teil-Ganzes Beziehung modelliert. Ist die Teil-Ganzes Beziehung bidirektional, gilt für die `EReference`, die von dem Teil Part auf den Ganzes Part zeigt, dass die `container`-Variable `true` ist. Desweiteren kann eine `EReference` eine andere `EReference` als `eOpposite` referenzieren. Diese Möglichkeit erlaubt es, eine bidirektionale Beziehung durch zwei `EReferences` auszudrücken, wobei jede die jeweils andere als `eOpposite` speichert.

`EAttributes` und `EReferences` besitzen zusätzlich noch folgende boolsche Attribute ihrer Superklasse `EStructuralFeature`:

- `changeable`: Spezifiziert, ob der Wert des `EStructuralFeatures` von extern gesetzt werden kann.
- `volatile`: Ist gesetzt, wenn das `EStructuralFeature` nicht direkt mit einer Speicherstelle assoziiert wird.
- `transient`: Spezifiziert, ob das `EStructuralFeature` bei der Serialisierung weggelassen werden kann.
- `unsettable`: Spezifiziert, ob das `EStructuralFeature` einen zusätzlichen Zustand, `unset` genannt, haben darf. Was dieser Zustand bedeutet kann im Literaturpunkt [SBPM08] nachgelesen werden.
- `derived`: Spezifiziert, ob das `EStructuralFeature` aus anderen `EStructuralFeatures` berechnet wird.

`EOperations` modellieren Methoden, sie haben einen Namen, `EParameters` und gehören zu einer `EClass`.

Die Klassen `EStructuralFeature`, `EOperation` und `EParameter` besitzen eine gemeinsame Superklasse `ETypedElement`. Daher können für alle drei Klassen Multiplizitäten angegeben werden. Die Attribute `many` und `required` lassen sich aus den Werten der jeweiligen Multiplizitäten ableiten: `many` ist `true` wenn `upperBound` größer als eins ist, `required` ist `true` wenn `lowerBound` ungleich 0 ist.

`EDataTypes` modellieren einzelne Datentypen. Sie werden dazu genutzt, um Javas primitive Datentypen und standardisierte Klassen im Modell zur Verfügung zu stellen, ohne sie im Detail zu modellieren. Abbildung 10 zeigt, wie die Java-Datentypen als `EDataType` gekapselt werden, Abbildung 11 zeigt externe Datentypen, die Ecore ebenfalls unterstützt.

Zusätzlich können `EDataTypes` aber auch dazu genutzt werden, eigene Datentypen, die in Javacode vorliegen aber nicht ausmodelliert werden sollen, als gekapselte Einheit zu repräsentieren.

`EEnum` ist ein spezieller `EDataType`, der Enumerations modelliert. Zu jedem `EEnum` gehören beliebig viele `EEnumLiterals`, die jeweils aus einem Literal und einem Wert bestehen. Ihre Semantik ist dieselbe, wie in UML auch.

Außerdem bietet das Ecore-Metametamodell noch die Möglichkeit, alle `ModelElements` mit `EAnnotations` zu versehen. Wie oben bereits angedeutet, nutzt das Eclipse Modeling Framework diese Möglichkeit zum Beispiel um auszudrücken, dass eine Klasse oder Methode Teil des Modells ist.

Nicht angesprochene Elemente des Ecore-Metametamodells sind als für diese Arbeit wenig relevant eingestuft worden und können unter der Literaturquelle [SBPM08] genauer nachgelesen werden.

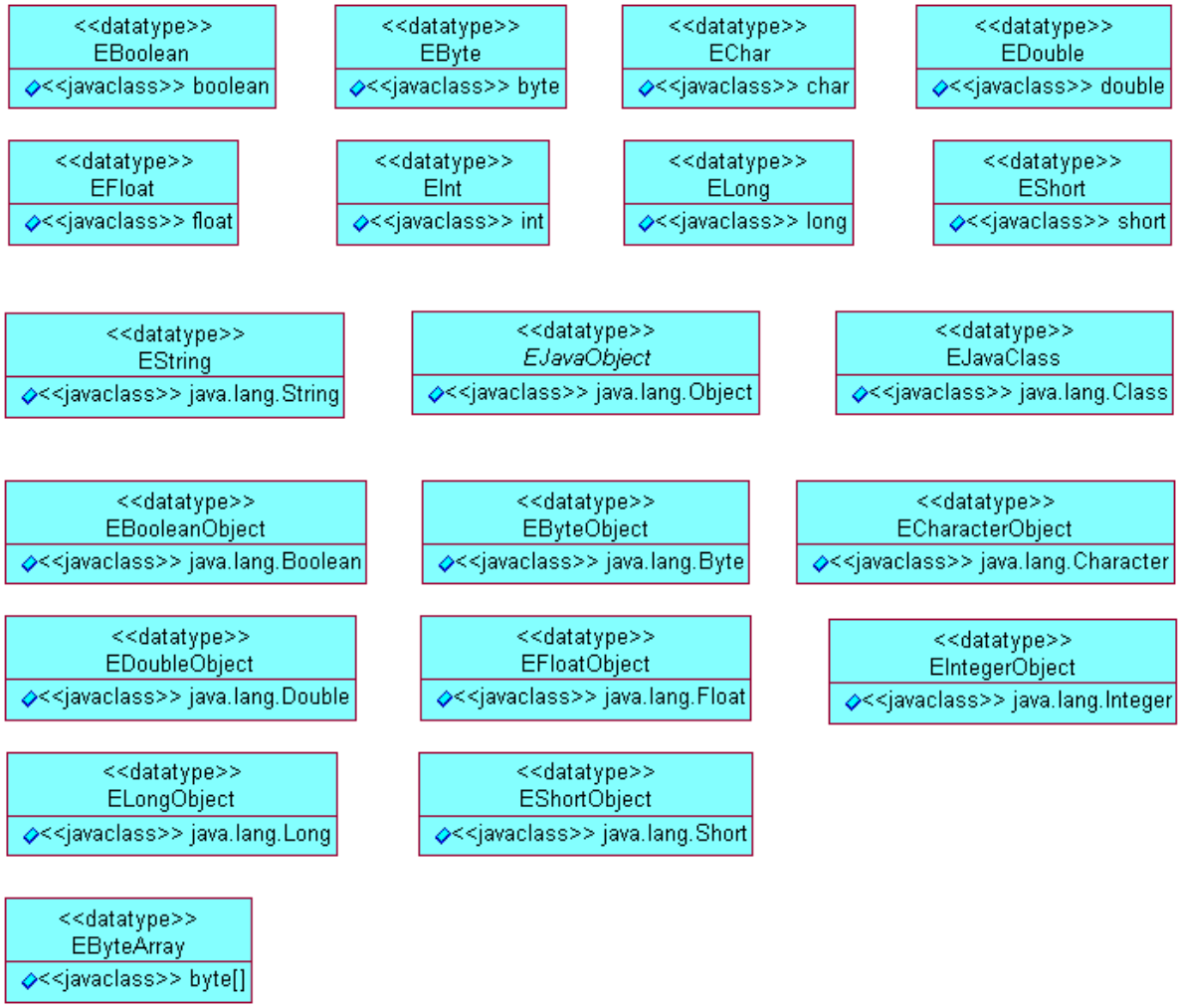

Abbildung 10: *Ecore Kapselung von Java Datentypen* [lit]

3.4 Serialisierung mit XMI

Gespeichert werden Ecore-Metamodelle im XMI-Format. Es wird eine Baumstruktur aufgebaut, wobei jede `containment`-Beziehung aus dem Metametamodell mittels Kindelementen ausgedrückt wird. Das Bibliothekensystem Beispiel wird wie folgt serialisiert:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ecore:EPackage xmi:version="2.0"
3      xmlns:xmi="http://www.omg.org/XMI"
4      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5      xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="librarysystem"
6      nsURI="http://librarysystem/1.0" nsPrefix="librarysystem">
```

Zunächst einmal müssen alle Klassen und Datentypen in einem `EPackage` enthalten sein. Aus diesem Grund wird das Wurzel-Paket zum root-Element der XMI-Datei. Die Elementattribute von `EPackage` geben Informationen über verwendete Namensräume.

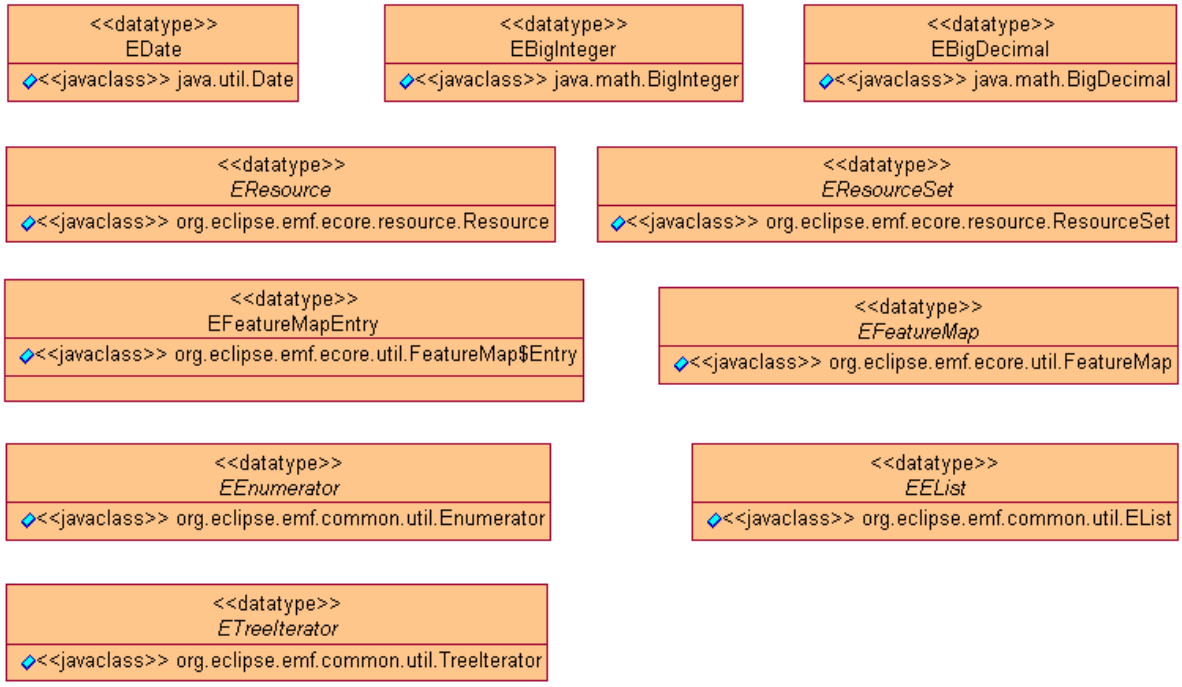

Abbildung 11: *Ecore Kapselung von externen Datentypen* [lit]

Als Kindelemente können nun Subpackages und `EClassifiers`, also `EClasses` und `EDataTypes` folgen. Hier wird als erstes die Klasse `Library` definiert.

```
1  <eClassifiers xsi:type="ecore:EClass" name="Library">
```

Anschließend können deren Kindelemente, die `EStructuralFeatures`, beschrieben werden.

```
1      <eStructuralFeatures xsi:type="ecore:EReference"
2          name="item" upperBound="-1" eType="#//Medium"
3          containment="true"/>
```

Dieses Element sagt aus, dass es von der `EClass Library` aus eine `EReference` gibt, welche auf `Medium` zeigt. Diese Referenz modelliert eine Teil-Ganzes Beziehung. Das Elementattribut `upperBound` legt in diesem Fall fest, dass eine Bibliothek beliebig viele Medien besitzen darf.

```
1      <eStructuralFeatures xsi:type="ecore:EAttribute"
2          name="director" eType= "ecore:EDataType
3          http://www.eclipse.org/emf/2002/Ecore#//EString"/>
```

Attribute von `Library` werden ebenfalls durch ein `eStructuralFeatures`-Element dargestellt. Das Attribut `director` vom Typ `EString` wird mit seinem Namen und seinem Typ als Elementattribute des `eStructuralFeatures` Elementes repräsentiert. Für die anderen Attribute gilt dasselbe.

```
1      <eStructuralFeatures xsi:type="ecore:EAttribute"
2          name="adress" eType="ecore:EDataType
3          http://www.eclipse.org/emf/2002/Ecore#//EString"/>
4      <eStructuralFeatures xsi:type="ecore:EReference"
```

```
5         name="lenderOfItems" upperBound="-1"
6         eType="#//Library" eOpposite="#//Library/borrowerOfItems"/>
7     <eStructuralFeatures xsi:type="ecore:EReference"
8         name="borrowerOfItems" upperBound="-1"
9         eType="#//Library" eOpposite="#//Library/lenderOfItems"/>
```

Ist die `EClass` komplett beschrieben, kann das `eClassifiers`-Element wieder geschlossen werden.

```
1   </eClassifiers>
```

Die `EClass Medium` wird dann analog beschrieben. Neu ist hier nur die Zuweisung eines Elementattributes, mit der `Medium` als abstrakte Klasse definiert wird.

```
1   <eClassifiers xsi:type="ecore:EClass"
2       name="Medium" abstract="true">
3     <eStructuralFeatures xsi:type="ecore:EAttribute"
4         name="bibID" eType="ecore:EDataType
5         http://www.eclipse.org/emf/2002/Ecore#//EString"/>
6     <eStructuralFeatures xsi:type="ecore:EAttribute"
7         name="title" eType="ecore:EDataType
8         http://www.eclipse.org/emf/2002/Ecore#//EString"/>
9   </eClassifiers>
```

Um die beiden Spezialisierungen von `Medium` zu beschreiben, wird noch ein weiteres Elementattribut von `eClassifiers` benötigt. Mit `eSuperTypes` wird ausgedrückt, dass `Book` und `Radioplay` die Superklasse `Medium` besitzen.

```
1   <eClassifiers xsi:type="ecore:EClass" name="Book" eSuperTypes="#//Medium">
2     <eStructuralFeatures xsi:type="ecore:EAttribute"
3         name="author" eType="ecore:EDataType
4         http://www.eclipse.org/emf/2002/Ecore#//EString"/>
5   </eClassifiers>
6   <eClassifiers xsi:type="ecore:EClass" name="Radioplay" eSuperTypes="#//Medium">
7     <eStructuralFeatures xsi:type="ecore:EAttribute"
8         name="speakers"
9         lowerBound="1" upperBound="-1"
10        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
11  </eClassifiers>
```

Als letztes wird noch die `EClass LibraryDatabase` beschrieben.

```
1   <eClassifiers xsi:type="ecore:EClass" name="LibraryDatabase">
2     <eStructuralFeatures xsi:type="ecore:EReference"
3         name="entry" upperBound="-1"
4         eType="#//Library" containment="true"/>
5     <eStructuralFeatures xsi:type="ecore:EAttribute"
6         name="admin" eType="ecore:EDataType
7         http://www.eclipse.org/emf/2002/Ecore#//EString"/>
8   </eClassifiers>
```

Sind alle Klassen, Datentypen und Subpackages definiert, kann das root-Element `EPackage` geschlossen werden.

```
</ecore:EPackage>
```

Diese XMI-Datei trägt dann die Endung `.ecore`.

Modelle werden separat in weiteren XMI-Dateien gespeichert. Ihre Endung entspricht dem `ns-prefix` des Metamodells. Da die XML-Datei eine Baumstruktur bilden muss, ist es gut, wenn es ein Objekt gibt, welches alle anderen als Kindelemente hat. Dieses Objekt wird dann zum Wurzelelement. Existiert ein solches Element nicht, wird ein künstliches Wurzelelement mit Namen `xmi:XMI` erzeugt.

Für das Beispiel-Modell ist dieses Wurzelelement eine LibraryDatabase Instanz.

```
<?xml version="1.0" encoding="UTF-8"?>
<librarysystem:LibraryDatabase
      xmi:version="2.0"
      xmlns:xmi="http://www.omg.org/XMI"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:librarysystem="http://librarysystem/1.0"
      admin="Com Puter">
```

Neben dem modellierten Attribut `admin` werden hier auch XML-Namensräume definiert.

Eine Bibliotheken-Datenbank besteht aus Einträgen. Es folgen also jetzt die Bibliotheken, die in der Datenbank enthalten sind mit ihren Attributen. In unserem Beispiel-Modell existiert eine Bibliothek, die von `"Hans Lovebook"` geleitet wird und deren Adresse `"Goodroute 4, Phantasia"` ist.

```
  <entry director="Hans Lovebook"
    adress="Goodroute 4, Phantasia"
```

Diese Bibliothek verleiht ihre Medien an zwei weitere Bibliotheken, die mit den Indexen 1 und 2. Von der Bibliothek mit dem Index 1 hat sie auch selbst Bücher und Hörspiele geliehen.

```
    lenderOfItems="//@entry.1 //@entry.2"
    borrowerOfItems="//@entry.1">
```

Kindelemente der `Library`-Instanz sind die Medien, welche die Bibliothek besitzt. Im Falle dieser Bibliothek ist das zunächst einmal ein Buch mit `bibID`, Titel und Autor.

```
    <item xsi:type="librarysystem:Book"
          bibID="B_J_Row_1"
          title="Harry Potter and the Philosopher's Stone"
          author="Joanne K. Rowling"/>
```

Auffallend ist hier, dass mit `librarysystem:Book` für `item` eine Typdefinition enthalten ist. Bei den `entry`-Elementen fehlt eine solche explizite Definition. Der Grund ist, dass ein `entry` immer eine Bibliothek ist, während ein `item` entweder ein Buch oder ein Hörspiel sein kann.

Außerdem besitzt die Bibliothek noch ein Hörspiel. Die Besonderheit ist hier, dass die `EClass Radioplay` ein Attribut `speakers` hat, welches beliebig oft vorkommen darf. Aus diesem Grund wird dieses Attribut nicht wie bisher als Elementattribut beschrieben. Stattdessen wird aus `speakers` ein eigenes Kindelement von `item`.

```
    <item xsi:type="librarysystem:Radioplay"
          bibID="R_SL_Tom_1"
          title="Crazy World">
          <speakers>Tom Tomsen</speakers>
          <speakers>Karl Karlsen</speakers>
          <speakers>Hugo Hugosen</speakers>
    </item>
  </entry>
```

Damit ist eine Bibliothek komplett beschrieben. Die zweite `Library`-Instanz ist analog aufgebaut.

```
  <entry director="Alice Wonderland"
    adress="Readstreet 4, Fictivia"
    lenderOfItems="//@entry.0 //@entry.2"
    borrowerOfItems="//@entry.0">
    <item xsi:type="librarysystem:Book"
          bibID="B_SL_Tol_1"
          title="Der Herr der Ringe"
          author="J. R. R. Tolkien"/>
    <item xsi:type="librarysystem:Book"
          bibID="B_SL_Tol_2"
          title="Der kleine Hobbit"
          author="J. R. R. Tolkien"/>
  </entry>
```

Die dritte Bibliothek hat die Besonderheit, dass sie selbst keine Medien verleiht, dafür aber von beiden anderen `Library`-Instanzen welche erhält. Aus diesem Grund fehlt in der XMI-Beschreibung das Attribut `lenderOfItems`.

```
  <entry director="James Mybooks"
    adress="Totalroute 5, Darktown"
    borrowerOfItems="//@entry.1 //@entry.0">
    <item xsi:type="librarysystem:Book"
          bibID="B_J_Roe_1"
          title="Robin Hood"
          author="Tilman Roehrig"/>
    <item xsi:type="librarysystem:Book"
          bibID="B_SL_Pra_1"
          title="The Unseen Academicals"
          author="Terry Pratchett"/>
  </entry>
</librarysystem:LibraryDatabase>
```

Ein Ecore-Modell kann in einer einzigen XMI-Datei gespeichert werden, es besteht aber auch die Möglichkeit es über mehrere Dateien aufzuteilen. In diesem Fall werden Objekte in anderen Dateien über ihre Dateipfade mit `href` referenziert.

3.5 Ecore API für Metamodelle

Um Ecore-Metamodelle zu verarbeiten und Instanzen davon zu erstellen, existiert die Ecore API. In diesem Abschnitt soll nun beschrieben werden, wie Metamodelle unter der Nutzung der Ecore API erzeugt, geladen und gespeichert werden können.

3.5.1 Laden eines Metamodells

Liegt die `.ecore` Date vor, kann das Ecore-Metamodell als Objektmodell in ein Java-Programm geladen werden.

```
1 //Load the Metamodel
2 //--- URI from the .ecore File
3 URI fileURI = URI.createFileURI("LibrarySystem.ecore");
```

Zunächst muss eine neue `XMIResource` erstellt werden, welche die zu ladende Datei repräsentiert.

```
1 //--- Generate a new Resource for the URI
2 XMIResource xmiResource = new XMIResourceImpl(fileURI);
```

Danach kann der Inhalt der Datei in die `Resource` geladen werden.

```
1 // Load the Resource
2 xmiResource.load(xmiResource.getDefaultLoadOptions());
```

Zum Schluss wird das root-Paket des Metamodells aus der Resource genommen, es befindet sich immer an der Stelle `0`.

```
1 // Take the first argument - it ist the root package
2 EPackage package = (EPackage) xmiResourceLoad.getContents().get(0);
```

Mit Hilfe des `EPackages` kann nun mit der Methode `getEClassifiers()` auf alle darin enthaltenen `EClasses` und `EDataTypes` zugegriffen werden.

3.5.2 Erstellen eines Metamodells

Zur *Erstellung eines Metamodells* gibt es zwei Möglichkeiten. Es können entweder die Eclipse Modeling Tools genutzt oder das Metamodell im Javacode direkt erzeugt werden. Die Eclipse Modeling Tools bieten eine grafische Oberfläche innerhalb von Eclipse, die es ermöglicht ein Metamodell mit wenigen Mausklicks zu erstellen.

Soll das Ecore-Metamodell direkt im Programm erstellt werden, wird das `EcoreFactory` benötigt. Im Folgenden wird die programmatische Erstellung eines Ecore-Metamodells mit Hilfe des Beispiel-Bibliothekensystems erläutert.

Zunächst einmal muss ein root-Paket definiert werden, das später alle Subpackages, Klassen und Datentypen aufnehmen kann.

```
1 //Defining the root package
2 EPackage rootPackage = EcoreFactory.eINSTANCE.createEPackage();
```

Anschließend kann damit begonnen werden, die EClasses zu erzeugen. Für die Erstellung der Klasse `LibraryDatabase`, wird zuerst eine leere `EClass`-Instanz benötigt.

```
1 //Creating the LibraryDatabase EClass
2 //-- Creating an empty EClass
3 EClass libBase = EcoreFactory.eINSTANCE.createEClass();
```

Danach kann der Name der neuen `EClass` auf `LibraryDatabase` gesetzt werden.

```
1 //-- Setting the name to "LibraryDatabase"
2 libBase.setName("LibraryDatabase");
```

Die `EClass LibraryDatabase` besitzt ein Attribut `admin` vom Typ `EString`.

```
1 //-- Creating the admin Attribute of the LibraryDatabase EClass
2 EAttribute adminAttribute = EcoreFactory.eINSTANCE.createEAttribute();
3 adminAttribute.setName("admin");
4 adminAttribute.setEType(EcorePackage.eINSTANCE.getEString());
```

Nachdem es erstellt wurde, kann es zur `LibraryDatabase` hinzugefügt werden.

```
1 //-- Adding the admin Attribute to the LibraryDatabase EClass
2 libBase.getEStructuralFeatures().add(adminAttribute);
```

Die `EClass Library` wird analog erstellt.

```
1  //Creating the Library EClass
2  //-- Creating an empty EClass
3  EClass lib = EcoreFactory.eINSTANCE.createEClass();
4  //-- Setting the name to "Library"
5  lib.setName("Library");
6  //-- Creating the director Attribute of the Library EClass
7  EAttribute directorAttribute = EcoreFactory.eINSTANCE.createEAttribute();
8  directorAttribute.setName("director");
9  directorAttribute.setEType(EcorePackage.eINSTANCE.getEString());
10 //-- Adding the director Attribute to the Library EClass
11 lib.getEStructuralFeatures().add(directorAttribute);
12 //-- Creating the adress Attribute of the Library EClass
13 EAttribute adressAttribute = EcoreFactory.eINSTANCE.createEAttribute();
14 adressAttribute.setName("adress");
15 adressAttribute.setEType(EcorePackage.eINSTANCE.getEString());
16 //-- Adding the adress Attribute to the Library EClass
17 lib.getEStructuralFeatures().add(adressAttribute);
```

Bei der Erstellung der `EClass Medium` muss darauf geachtet werden, sie als `abstract` zu definieren. Ansonsten verläuft die Instanziierung wie bei den `EClasses LibraryDatabase` und `Library`.

```
1 //Creating the Medium EClass
2 //-- Creating an empty EClass
```

```
3  EClass medium = EcoreFactory.eINSTANCE.createEClass();
4  //—— Setting the name to "Medium"
5  medium.setName("Medium");
6  //—— Setting the EClass abstract
7  medium.setAbstract(true);
8  //—— Creating the bibID Attribute of the Medium EClass
9  EAttribute bibIdAttribute = EcoreFactory.eINSTANCE.createEAttribute();
10 bibIdAttribute.setName("bibID");
11 bibIdAttribute.setEType(EcorePackage.eINSTANCE.getEString());
12 //—— Adding the bibID Attribute to the Medium EClass
13 medium.getEStructuralFeatures().add(bibIdAttribute);
14 //—— Creating the title Attribute of the Medium EClass
15 EAttribute titleAttribute = EcoreFactory.eINSTANCE.createEAttribute();
16 titleAttribute.setName("title");
17 titleAttribute.setEType(EcorePackage.eINSTANCE.getEString());
18 //—— Adding the title Attribute to the Medium EClass
19 medium.getEStructuralFeatures().add(titleAttribute);
```

Die `EClass Book` ist eine Spezialisierung der `EClass Medium`. Aus diesem Grund muss das Objekt `medium` der Liste der Superklassen von `book` hinzugefügt werden.

```
1  //Creating the Book EClass that is a specialized Medium
2  //—— Creating an empty EClass
3  EClass book = EcoreFactory.eINSTANCE.createEClass();
4  //—— Setting the name to "Book"
5  book.setName("Book");
6  //—— Setting the EClass Medium as SuperClass
7  book.getESuperTypes().add(medium);
8  //—— Creating the author Attribute of the Book EClass
9  EAttribute authorAttribute = EcoreFactory.eINSTANCE.createEAttribute();
10 authorAttribute.setName("author");
11 authorAttribute.setEType(EcorePackage.eINSTANCE.getEString());
12 //—— Adding the author Attribute to the Book EClass
13 book.getEStructuralFeatures().add(authorAttribute);
```

Die `EClass Radioplay` hat eine Besonderheit, die sie von allen bisher instanziierten `EClasses` unterscheidet. Sie besitzt das `speakers`-Attribut vom Typ `EString`, welches für ein Objekt beliebig oft vorkommen darf. Aus diesem Grund müssen hier explizit die Multiplizitäten gesetzt werden.

```
1  //Creating the Radioplay EClass that is a specialized Medium
2  //—— Creating an empty EClass
3  EClass rapl = EcoreFactory.eINSTANCE.createEClass();
4  //—— Setting the name to "Radioplay"
5  rapl.setName("Radioplay");
6  //—— Setting the EClass Medium as SuperClass
7  rapl.getESuperTypes().add(medium);
8  //—— Creating the speakers Attribute of the Radioplay EClass
9  EAttribute speakersAttribute = EcoreFactory.eINSTANCE.createEAttribute();
10 speakersAttribute.setName("speakers");
11 speakersAttribute.setEType(EcorePackage.eINSTANCE.getEString());
12 speakersAttribute.setLowerBound(0);
13 speakersAttribute.setUpperBound(-1);
14 //—— Adding the speakers Attribute to the Radioplay EClass
```

```
15 rapl.getEStructuralFeatures().add(speakersAttribute);
```

Sind alle `EClasses` erstellt, können Referenzen zwischen ihnen erzeugt werden. Als erstes wird die `EReference` der `EClass LibraryDatabase` betrachtet. Analog zur Erstellung der `EClasses` wird auch hier zunächst eine leere `EReference` erzeugt.

```
1 //Creating the EReference saying LibraryDatabase has Library entries
2 EReference entry = EcoreFactory.eINSTANCE.createEReference();
```

Im Anschluss wird der Name und die EClass, welche referenziert werden soll, gesetzt. Im Falle des Beispiels wird als Name `"entry"` und als `eType` das `EClass`-Objekt, das die Bibliothek repräsentiert, übergeben.

```
1 entry.setName("entry");
2 entry.setEType(lib);
```

Als nächstes werden die Multiplizitäten der `EReference` festgelegt. Da die `LibraryDatabase` beliebig viele Einträge haben darf, wird die Anzahl hier nicht begrenzt.

```
1 entry.setLowerBound(0);
2 entry.setUpperBound(-1);
```

Da es sich um eine Teil-Ganzes Beziehung handelt, muss die boolsche Variable `containment` auf `true` gesetzt werden.

```
1 entry.setContainment(true);
```

Nun ist die `EReference` fertig erstellt und kann ihrem `EClass`-Objekt, in diesem Fall der `LibraryDatabase` als `EStructuralFeature` hinzugefügt werden.

```
1 libBase.getEStructuralFeatures().add(entry);
```

Die Erstellung der `EReference` von `Library`, die auf `Medium` referenziert wird dann analog erstellt.

```
1 //Creating the EReference saying Library has Medium items
2 EReference item = EcoreFactory.eINSTANCE.createEReference();
3 item.setName("item");
4 item.setEType(medium);
5 item.setLowerBound(0);
6 item.setUpperBound(-1);
7 item.setContainment(true);
8 lib.getEStructuralFeatures().add(item);
```

Im Beispiel-Bibliothekensystem ist es möglich, dass Bibliotheken Medien an andere Bibliotheken verleihen. Diese Beziehung wird durch zwei `EReferences` modelliert, da die verleihende Bibliothek den Ausleiher kennen muss und die ausleihende Bibliothek auch dem Verleiher bekannt sein sollte.

Es werden zunächst einmal beide `EReferences` erstellt.

```
1  //Creating the duplicated EReference saying Libraries lend to Libraries
2  //--- Create the EReference lenderOfItems
3  EReference lenderOfItems = EcoreFactory.eINSTANCE.createEReference();
4  lenderOfItems.setName("lenderOfItems");
5  lenderOfItems.setEType(lib);
6  lenderOfItems.setLowerBound(0);
7  lenderOfItems.setUpperBound(-1);
8  lenderOfItems.setContainment(false);
9  //--- Create the EReference borrowerOfItems
10 EReference borrowerOfItems = EcoreFactory.eINSTANCE.createEReference();
11 borrowerOfItems.setName("borrowerOfItems");
12 borrowerOfItems.setEType(lib);
13 borrowerOfItems.setLowerBound(0);
14 borrowerOfItems.setUpperBound(-1);
15 borrowerOfItems.setContainment(false);
```

Anschließend können sie zur jeweils anderen als `eOpposite` gesetzt werden.

```
1  //--- Set the created EReferences as Opposites
2  lenderOfItems.setEOpposite(borrowerOfItems);
3  borrowerOfItems.setEOpposite(lenderOfItems);
```

Zum Schluss werden beide erstellten `EReferences`, wie bisher, ihren jeweiligen Klassen als `EStructuralFeature` hinzugefügt.

```
1  //--- Add both References to the Library EClass
2  lib.getEStructuralFeatures().add(lenderOfItems);
3  lib.getEStructuralFeatures().add(borrowerOfItems);
```

Nun sind alle Klassen und Referenzen für das Beispiel-Bibliothekensystem erzeugt. Die Klassen können jetzt als `EClassifier` in das root-Paket eingeordnet werden.

```
1  //Adding all Classes to the rootPackage
2  rootPackage.getEClassifiers().add(libBase);
3  rootPackage.getEClassifiers().add(lib);
4  rootPackage.getEClassifiers().add(medium);
5  rootPackage.getEClassifiers().add(book);
6  rootPackage.getEClassifiers().add(rapl);
```

Damit ist das Metamodell komplett erzeugt.

3.5.3 Speichern eines Metamodells

Um das Metamodell im XMI-Format abzuspeichern, wird zuerst eine `XMIResource` zur gewünschten URI erstellt.

```
1  //Saving the Metamodel
2  //--- URI from the .ecore File
3  URI fileURI = URI.createFileURI("generatedLibrary.ecore");
4  //--- Generate a new Resource for the URI
5  XMIResource xmiResource = new XMIResourceImpl(fileURI);
```

Anschließend wird das `rootPackage` zum Inhalt der `Resource` gemacht.

```
1 //--- Add the root package to the Resource
2 xmiResource.getContents().add(rootPackage);
```

Ist dies geschehen, kann die `XMIResource` gespeichert werden.

```
1 //--- Save it with Default Options
2 xmiResource.save(xmiResource.getDefaultSaveOptions());
```

3.6 Ecore API für Modelle

Mit Hilfe des Metamodells können dann passende Modelle erstellt, geladen und gespeichert werden. Der folgende Abschnitt soll einen kurzen Überblick darüber geben, wie das geschieht.

3.6.1 Metamodell-spezifische Erstellung von Modellen

Das Eclipse Modeling Framework ermöglicht es, zu dem Metamodell Javaklassen zu generieren, die dann in eigene Projekte eingebunden und dazu verwendet werden können, Modelle zu erzeugen.

Die Instanziierung der Objekte erfolgt über eine Factory Instanz. Für das Beispiel-Modell zum Bibliothekensystem soll dies nun im Folgenden beschrieben werden.

Zunächst wird mit Hilfe des `LibrarysystemFactory` ein `LibraryDatabase`-Objekt erstellt.

```
1 LibraryDatabase libbase = LibrarysystemFactory.eINSTANCE.createLibraryDatabase();
```

Das `admin`-Attribut der Datenbank kann dann belegt werden.

```
1 libbase.setAdmin("Cam Puter");
```

Im Anschluss sollen Einträge für die Datenbank erstellt werden. Dazu müssen `Library`-Objekte erzeugt und initialisiert werden.

```
1 Library lib1 = LibrarysystemFactory.eINSTANCE.createLibrary();
2 lib1.setDirector("Hans Lovebook");
3 lib1.setAdress("Goodroute 4, Phantasia");
```

Ein `Library`-Objekt wird ebenfalls mit Hilfe des `LibrarysystemFactorys` instanziiert. Danach können die Attribute `director` und `adress` belegt werden. Für die anderen beiden Bibliotheken des Beispiel-Modells verläuft das analog.

```
1 Library lib2 = LibrarysystemFactory.eINSTANCE.createLibrary();
2 lib2.setDirector("Alice Wonderland");
3 lib2.setAdress("Readstreet 4, Fictivia");
4
5 Library lib3 = LibrarysystemFactory.eINSTANCE.createLibrary();
6 lib3.setDirector("James Mybooks");
7 lib3.setAdress("Totalroute 5, Darktown");
```

Anschließend werden die drei `Libraries` zu den Einträgen der Datenbank hinzugefügt.

```
libbase.getEntry().add(lib1);
libbase.getEntry().add(lib2);
libbase.getEntry().add(lib3);
```

Die einzelnen Bibliotheken besitzen Bücher und Hörspiele. Eine `Book`-Instanz wird ebenfalls mittels der `Factory` erzeugt und anschließend mit Attributen besetzt. Zum Schluss wird dieses Buch der Medienliste der Bibliothek mit dem Leiter `"James Lovebook"` zugewiesen.

```
Book lib1_it1 = LibrarysystemFactory.eINSTANCE.createBook();
lib1_it1.setBibID("B_J_Row_1");
lib1_it1.setAuthor("Joanne K. Rowling");
lib1_it1.setTitle("Harry Potter and the Philosopher's Stone");
lib1.getItem().add(lib1_it1);
```

Bei der Instanziierung eines `Radioplay`-Objektes fällt eine Besonderheit auf. Da es beliebig viele Sprecher geben kann, ist das Attribut `speakers` als Liste implementiert.

```
Radioplay lib1_it2 = LibrarysystemFactory.eINSTANCE.createRadioplay();
lib1_it2.setBibID("R_SL_Tom_1");
lib1_it2.setTitle("Crazy World");
lib1_it2.getSpeakers().add("Tom Tomsen");
lib1_it2.getSpeakers().add("Karl Karlsen");
lib1_it2.getSpeakers().add("Hugo Hugosen");
lib1.getItem().add(lib1_it2);
```

Die verbleibenden Medien des Beispiel-Modells werden ebenso erstellt.

```
Book lib2_it1 = LibrarysystemFactory.eINSTANCE.createBook();
lib2_it1.setBibID("B_SL_Tol_1");
lib2_it1.setTitle("Der Herr der Ringe");
lib2_it1.setAuthor("J. R. R. Tolkien");
lib2.getItem().add(lib2_it1);

Book lib2_it2 = LibrarysystemFactory.eINSTANCE.createBook();
lib2_it2.setBibID("B_SL_Tol_2");
lib2_it2.setTitle("Der kleine Hobbit");
lib2_it2.setAuthor("J. R. R. Tolkien");
lib2.getItem().add(lib2_it2);

Book lib3_it1 = LibrarysystemFactory.eINSTANCE.createBook();
lib3_it1.setBibID("B_J_Roe_1");
lib3_it1.setTitle("Robin Hood");
lib3_it1.setAuthor("Tilman Roehrig");
lib3.getItem().add(lib3_it1);

Book lib3_it2 = LibrarysystemFactory.eINSTANCE.createBook();
lib3_it2.setBibID("B_SL_Pra_1");
lib3_it2.setTitle("The Unseen Academicals");
lib3_it2.setAuthor("Terry Pratchett");
lib3.getItem().add(lib3_it2);
```

3.6.2 Generische Erstellung von Modellen

Stehen die generierten Javaklassen nicht zur Verfügung, zum Beispiel weil erst zur Laufzeit entschieden wird, welches Metamodell geladen und instanziiert wird, gibt es auch die Möglichkeit die Modelle dynamisch zu erzeugen. Dazu muss dann allerdings das Metamodell im Speicher vorliegen.

Als erstes wird wiederum ein `Factory` benötigt. Dazu wird das `rootPackage` des Metamodells verwendet.

```
1 //Creating a factory for the librarysystem
2 EFactory libFactory = rootPackage.getEFactoryInstance();
```

Anschließend wird eine `LibraryDatabase` erzeugt. Um die Datenbank zu instanziieren muss die `EClass`-Instanz, welche bei der Erzeugung des Metamodells in Abschnitt 3.5 erstellt wurde, übergeben werden.

```
1 //Creating a LibraryDatabase object - or something
2 EObject aLibBase = libFactory.create(libBase);
3 aLibBase.eSet(adminAttribute, "Cam Puter");
```

Das Ergebnis ist dann allerdings kein `LibraryDatabaseImpl`-Objekt, sondern eine Instanz von `EObjectImpl`, welche eine Default-Implementation anbietet. Um das `admin`-Attribut der Datenbank zu setzen, wird dann auch das entsprechende `EAttribute`-Objekt benötigt.

Eine Bibliothek wird dann analog erstellt.

```
1 //Creating a Library object - or something
2 EObject aLib = libFactory.create(lib);
3 aLib.eSet(directorAttribute, "Hans Lovebook");
4 aLib.eSet(adressAttribute, "Goodroute 4, Phantasia");
5
6 //Creating another Library object - or something
7 EObject aLib2 = libFactory.create(lib);
8 aLib2.eSet(directorAttribute, "Alice Wonderland");
9 aLib2.eSet(adressAttribute, "Readstreet 4, Fictivia");
```

Nachdem die `Libraries` instanziiert wurden, können sie der Datenbank als Einträge übergeben werden. Die `eSet`-Methode, die bisher zur Belegung der Attribute verwendet wurde, erhält als Parameter ein `EStructuralFeature` und ein `Object` mit dem es besetzt werden soll. Sie wird also ebenfalls verwendet, um `EReferences` zu realisieren.

```
1 //Adding the Libraries to the Database
2 EList <EObject> libs = new BasicEList<EObject>();
3 libs.add(aLib);
4 libs.add(aLib2);
5 aLibBase.eSet(entry, libs);
```

Da es beliebig viele Einträge geben kann, wird als zweiter Parameter eine Liste mit den Bibliotheken übergeben. Das Objekt `entry` ist die `EReference`-Instanz, die zuvor bei der Erstellung des Metamodells erzeugt wurde.

Im Anschluss daran sollen noch Medien erstellt werden, die in den Bibliotheken enthalten sein können. Die Erzeugung eines Buches verläuft analog zu den bereits besprochenen Instanziierungen.

```
1 //Creating a Book
2 EObject aBook = libFactory.create(book);
3 aBook.eSet(bibIdAttribute, "B_J_Row_1");
4 aBook.eSet(titleAttribute, "Harry Potter and the Philosopher's Stone");
5 aBook.eSet(authorAttribute, "Joanne K. Rowling");
```

Bei der Erstellung eines `Radioplay` muss darauf geachtet werden, dass `speakers` ein beliebig oft vorkommendes Attribut ist. Aus diesem Grund wird hier, wie bei Referenzen, eine Liste mit allen Sprechern übergeben.

```
1 //Creating a Radioplay
2 EObject aRapl = libFactory.create(rapl);
3 aRapl.eSet(bibIdAttribute, "R_SL_Tom_1");
4 aRapl.eSet(titleAttribute, "Crazy World");
5 EList<Object> speakerList = new BasicEList<Object>();
6 speakerList.add("Tom Tomsen");
7 speakerList.add("Karl Karlsen");
8 speakerList.add("Hugo Hugosen");
9 aRapl.eSet(speakersAttribute, speakerList);
```

Zum Schluss werden die beiden erstellten Medien zur ersten Bibliothek hinzugefügt.

```
1 //Adding the Book and the Radioplay to the first Library
2 EList<EObject> mediumList = new BasicEList<EObject>();
3 mediumList.add(aBook);
4 mediumList.add(aRapl);
5 aLib.eSet(item, mediumList);
```

3.6.3 Speichern eines Modells

Sowohl das statische als auch das dynamische Modell lassen sich wie folgt speichern.

Zunächst wird eine `ResourceSet` erstellt. In einer `ResourceSet` können mehrere `Resourcen` gesammelt werden. Das ist vor allem dann von Bedeutung, wenn ein Modell nicht in einer einzigen XMI-Datei gespeichert wird. `ResourceSets` ermöglichen es, Referenzen zwischen verschiedenen Modelldateien zu verwalten.

```
1 //Create a ResourceSet
2 ResourceSet resourceset = new ResourceSetImpl();
```

Im Gegensatz zum Metamodell ist die Dateiendung bei der Speicherung von Modellen nicht in der API festgelegt, da sie sich aus dem `ns-prefix` des Metamodells bildet. Aus diesem Grund muss `ResourceFactoryRegistry` mitgeteilt werden, dass sie Dateien mit der übergebenen Endung als XMI-Dateien betrachtet.

```
1 //Teach the registry the file ending
2 resourceset.getResourceFactoryRegistry().getExtensionToFactoryMap()
3             .put("librarysystem", new XMIResourceFactoryImpl());
```

Im Anschluss wird eine Resource zur gewünschten `URI` erzeugt.

```
1 //Create a Resource
2 URI uri = URI.createURI("Test.librarysystem");
3 Resource resource = resourceset.createResource(uri);
```

Dann kann das zukünftige Wurzelelement der XML-Datei, in unserem Fall die `LibraryDatabase`-Instanz, als Inhalt in die `Resource` eingefügt werden.

```
1 //Add the root Element to the contents
2 resource.getContents().add(aLibBase);
```

Zum Schluss muss nur noch `save` aufgerufen werden.

```
1 //Save it
2 resource.save(null);
```

3.6.4 Laden eines Modells

Im Folgenden wird beschrieben, wie ein Modell als Objektstruktur in einem Programm geladen werden kann.

```
1 //Load the model
2 //The URI to load
3 URI uri = URI.createURI("My.librarysystem");
```

Soll das Modell dynamisch geladen werden, wird zunächst das root-Paket des Metamodells benötigt, um den dort definierten Namespace zu registrieren und abzugleichen.

```
1 //Register the Metamodel to verify the namespace
2 EPackage.Registry.INSTANCE.put(pack.getNsURI(), pack);
```

Stehen die generierten Javaklassen zur Verfügung wird diese Zeile durch die Instanziierung eines `LibrarysystemPackage` ersetzt.

```
1 //Register the Metamodel to verify the namespace
2 LibrarysystemPackage p = LibrarysystemPackage.eINSTANCE;
```

Danach wird eine leere `ResourceSet`-Instanz erzeugt, welche die Organisation des weiteren Ladens übernimmt.

```
1 //Create a ResourceSet
2 ResourceSet resSet = new ResourceSetImpl();
```

Die Modelldatei besitzt als Endung, wie oben bereits erwähnt, das `ns-prefix` des Metamodells. Damit das Modell nun korrekt geladen werden kann, muss diese Endung mit einem passenden `ResourceFactory` assoziiert werden. Im Falle des Beispiel-Modells muss der `ResourceSet` gesagt werden, dass sie Dateien mit der Endung `librarysystem` als `XMIResource` betrachten soll.

```
1 //Register the extension and connect it to an XMI ResourceFactory
2 resSet.getResourceFactoryRegistry().getExtensionToFactoryMap().
3         put("librarysystem", new XMIResourceFactoryImpl());
```

Anschließend kann die `Resource` aus der `ResourceSet` geladen werden.

```
1 //Load the resource
2 Resource res = resSet.getResource(uri, true);
```

Um nun von der `Resource` an das Modell zu gelangen, muss noch die `getContents()` Methode aufgerufen werden, die ein `EList<EObject>`-Objekt zurück gibt. Besitzt die Datei ein künstliches `xmi:XMI` Wurzelelement, dann sind in dieser Liste alle Kindelemente davon enthalten und es kann darüber iteriert werden. Existiert für das Modell ein "richtiges" Wurzelelement, dann befindet sich dies alleine in der Liste. Im Falle des Beispiel-Modells gibt es eine `LibraryDatabase`, welche das Wurzelelement der XMI-Datei ist, es befindet sich daher an der Stelle 0.

```
1 //Get the root Element - dynamic
2 EObject eob = res.getContents().get(0);
3 //---OR---
4 //Get the root Element - static
5 LibraryDatabase eob = res.getContents().get(0);
```

Von dem Wurzelelement aus können alle anderen Elemente erreicht werden. Eine kurze Methode, welche für die dynamische Variante nur einige Informationen über die jeweiligen Elemente ausgibt, sieht so aus:

```
1 public static void printObjectStructure(EObject ob){
2         System.out.println(ob.eClass().getName() + "::" + ob.toString());
3         Iterator <EObject> it = ob.eContents().iterator();
4         while(it.hasNext()){
5                 printObjectStructure(it.next());
6         }
7 }
```

3.7 Zusammenfassung

Ecore ist der Kernbestandteil des Eclipse Modeling Framework und wird dort dazu verwendet Metamodelle zu repräsentieren. Da die Betonung bei EMF auf den Möglichkeiten zur Codegenerierung liegt, schränkt es UML-Klassendiagramme soweit ein, dass aus den verbleibenden Konstrukten problemlos Javaklassen erzeugt werden können. Ecore-Metamodelle sind also Klassendiagramme, während Ecore-Modelle Objektgeflechte darstellen. Sowohl Modelle als auch Metamodelle werden im XMI-Format gespeichert. Für die Metamodelle wird die Dateiendung `"*.ecore"` genutzt, während die Dateiendung der Modelle von dem zugehörigen Metamodell abhängt. Um Ecore-Metamodelle programmatisch zu verarbeiten und um passende Javaklassen zu generieren wird die Ecore-API verwendet. Auch für die Erstellung von und den Umgang mit Modellen bietet sie Möglichkeiten.

4 Vergleich von Ecore und grUML

Auf den ersten Blick scheinen Ecore- und grUML-Diagramme sehr ähnlich. Die Semantik ist jedoch unterschiedlich, grUML beschreibt Graphenschemas und Graphen, während in Ecore Klassen und ihre Referenzen untereinander modelliert werden. Dieser Abschnitt soll sich mit den Gemeinsamkeiten und Unterschieden von grUML und Ecore beschäftigen.

4.1 Ecore-Metamodell als Graphenschema

Im Folgenden wird das Ecore-Metamodell aus Abbildung 12 betrachtet.

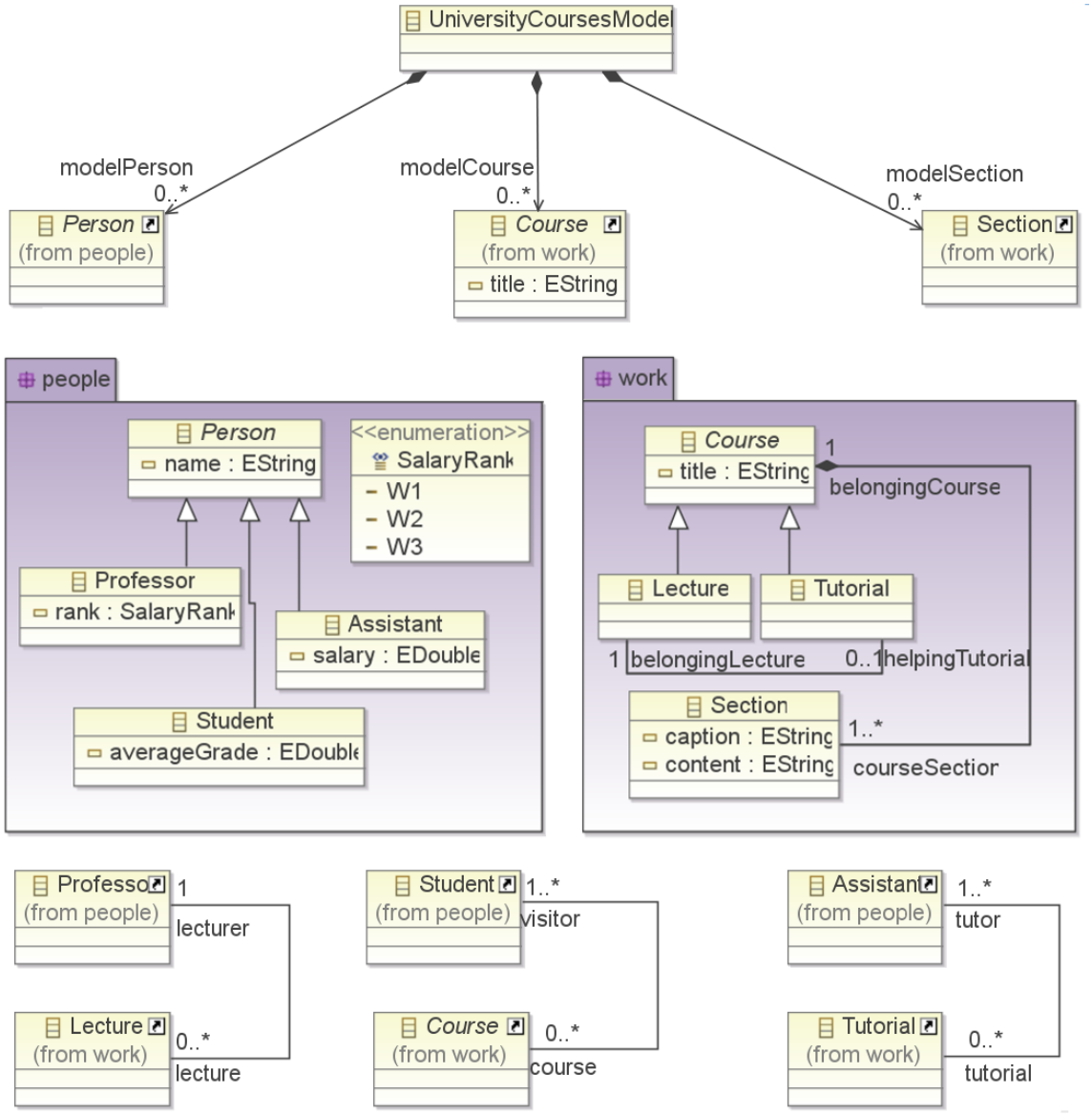

Abbildung 12: *Ecore-Metamodell Universities*

Es handelt sich um die *Modellierung eines Ausschnitts einer Universität*. Es gibt Personen, das können entweder Professoren, Assistenten oder Studenten sein. Personen haben einen Namen, Professoren haben zusätzlich einen Rang, Assistenten ein Einkommen und Studenten eine Durchschnittsnote. Außerdem gibt es Veranstaltungen, die von Studenten besucht werden. Diese haben einen Titel und können in unterschiedliche Kapitel mit Überschrift und Inhalt unterteilt werden. Veranstaltungen können Vorlesungen oder Tutorien sein. Ein Tutorium gehört immer zu einer Vorlesung, manche Vorlesungen haben aber auch keins. Eine Vorlesung wird von einem Professor gehalten, ein Tutorium von einem Assistenten.

Dieses Szenario ließe sich auch als Graphenschema mit grUML modellieren und könnte dort so aussehen, wie Abbildung 13 zeigt.

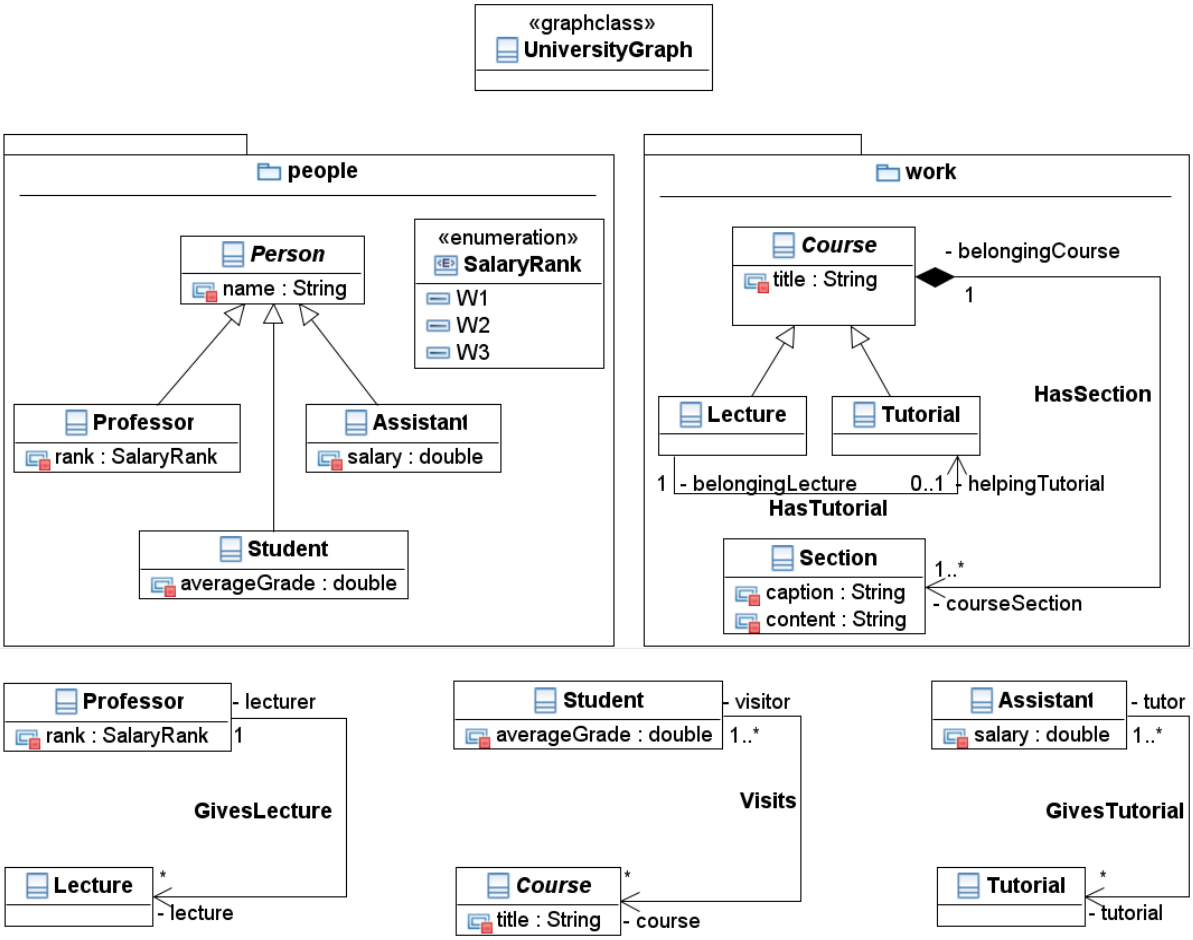

Abbildung 13: *grUML Graphenschema Universities*

Die beiden Diagramme sehen auf den ersten Blick fast identisch aus, haben aber eine andere Semantik. Personen, Professoren, Assistenten, Studenten, Veranstaltungen, Vorlesungen und Tutorien sind im Ecore-Metamodell Klassen, in grUML aber Knotenklassen. Dass ein Professor eine Vorlesung hält wird in Ecore dadurch ausgedrückt, dass die Klasse Professor eine Referenz auf die Klasse Vorlesung hat, während die Vorlesung ihrerseits auch den Professor referenziert. In grUML hat der Knoten Professor eine Kante mit dem Namen `GivesLecture` welche zu dem Knoten Vorlesung führt.

Werden also Klassen als Knoten und Referenzen als Kanten interpretiert, lassen sich auch mit Ecore Graphen modellieren.

4.2 Gegenüberstellung

Obwohl es möglich ist, Ecore-Metamodelle als Graphen zu interpretieren, sind beide Modellierungsarten nicht gleichmächtig. Im folgenden Abschnitt soll kurz auf die wichtigsten Unterschiede zwischen grUML und Ecore eingegangen werden.

4.2.1 Knotenklassen vs. Klassen

grUML-Knotenklassen und Ecore-Klassen sind sich sehr ähnlich, trotzdem gibt es einen klar erkennbaren Unterschied. Ecore-Klassen können *Operationen* besitzen, die dann später auch im generierten Programmcode vorkommen. Für grUML-Knotenklassen sind Methoden dagegen nicht vorgesehen.

4.2.2 Kantenklassen vs. Referenzen

Mit grUML modellierte Kanten haben einen eigenen *Typ*, eine Kantenklasse, und können *Attribute* besitzen. Das ist für Referenzen in Ecore nicht möglich. Als Typ wird dort nur der Typ der referenzierten Klasse bezeichnet.

Wie groß dieser Unterschied wirklich ist und was er bedeutet, lässt sich am Besten auf der Graphen-Ebene anhand der erstellten Javaklassen erklären.

Mit grUML modellierte Kanten werden zu *eigenständigen Klassen* mit einem Start- und Endpunkt sowie Attributen. Dadurch kann bei einer Traversierung des Graphen diese Kante in beide Richtungen durchlaufen werden. Ecore-Referenzen dagegen werden nur zu *Attributen der Klassen*, zu der sie gehören. Es ist also im Allgemeinen nur möglich von der Klasse, zu der die Referenz gehört, zur referenzierten Klasse zu gelangen, aber nicht umgekehrt. Soll es bidirektionale Verbindungen geben, muss das mit zwei Referenzen modelliert werden, die sich gegenseitig als `EOpposite` haben.

Außerdem besteht in Ecore keine Möglichkeit, einer `EReference` Attribute zu geben. Soll eine Beziehung zwischen zwei Klassen Attribute haben, muss diese als eigene Klasse oder über andere Tricks modelliert werden. In grUML dagegen können Kanten beliebige Attribute haben.

Desweiteren unterstützen grUML-Kantenklassen Generalisierung. Für Ecore-`EReferences` gilt das jedoch nicht. Sollte ein Szenario dies erfordern, muss die Generalisierung über Umwege, beispielsweise durch Modellierung der Beziehungen als Klassen statt Referenzen, realisiert werden.

4.2.3 Graphclass vs. ?

Ein weiterer wichtiger Unterschied ist das Existieren der *Graphklasse* in grUML. Im Diagramm drückt sich das nur durch eine weitere Klasse mit dem Stereotype `<<graphclass>>` aus. Auf Graphen-Ebene bedeutet das aber, dass dem `Graph` - Objekt alle Knoten und Kanten, welche im Graph vorkommen, bekannt sind. Ein solches Konstrukt gibt es für Ecore-Metamodelle nicht.

Stattdessen kann hier eine zusätzliche `EClass` definiert werden, welche alle anderen `EClasses` referenziert. Wird darauf verzichtet, gibt es keine einfache Möglichkeit, direkt zu allen Objekten im Geflecht zu navigieren.

4.2.4 Datentypen

In grUML gibt es andere Datentypen als in Ecore. Für das Ziel dieser Arbeit sollte es aber möglich sein, Datentypen aufeinander abzubilden. Die Tabelle aus Abbildung 14 zeigt eine Entsprechung für die einfachen Fälle sowie die Probleme bei den komplexeren.

Die mit grün gekennzeichneten Zeilen stellen kein Problem dar, bei allen anderen muss überlegt werden, ob eine Abbildung möglich ist. Kursiv geschrieben sind Vorschläge, wie mit einigen Unstimmigkeiten umgegangen werden kann.

Die Ecore Datentypen `EJavaObject` und `EJavaClass` kapseln die entsprechenden Java Klassen `java.lang.Object` und `java.lang.Class`. `EResource` und `EResourceSet` wurden in Kapitel 3.6.3 beschrieben, sie werden hauptsächlich zum Laden und Speichern von Ecore-Modellen benutzt. Es stellt sich die Frage, ob hier überhaupt eine Abbildung gefunden werden muss, da das Vorkommen dieser Datentypen als Attribute in Metamodellen sehr unwahrscheinlich ist. Gleiches gilt für die `EFeatureMap` mit ihren `EFeatureMapEntrys`. Dabei handelt es sich um ein Mapping von `EStructuralFeatures` auf `Objects`. Ein `ETreeIterator` wird wohl ebenso selten in einem Metamodell gefunden werden. Diese Aspekte sind vor allem deshalb modelliert, weil Ecore sein eigenes Metamodell ist und um Reflection zu unterstützen.

Ein weiteres Problem betrifft die Tatsache, dass mit Hilfe des `EDataTypes` *eigene Datentypen* definiert werden können, die nicht ausmodelliert sind, sondern nur als Javacode vorliegen. Hier stellt sich auch die Frage, wie mit so etwas bei der Abbildung von Ecore nach grUML umgegangen werden kann.

Wie die Datentypen nun konkret transformiert werden, wird in Kapitel 7 erläutert.

4.3 Beziehung zu den MOF Ebenen

Im Kontext von Metamodellen besitzen die *Ebenen der MOF* eine gewisse Bedeutung. Im folgenden Abschnitt wird kurz erläutert, wie diese Ebenen in Beziehung zu grUML und Ecore stehen.

Die OMG (Object Management Group) definiert *vier Metalevel* für die Modellierung. Auf dem obersten Level, M3 genannt, steht die MOF (Meta Object Facility), eine Modellierungssprache

Ecore	**grUML**
EBoolean	BooleanDomain
EByte	*IntegerDomain?*
EChar	*StringDomain?*
EDouble	DoubleDomain
EFloat	*DoubleDomain?*
EInt	IntegerDomain
ELong	LongDomain
EShort	*IntegerDomain?*
EString	StringDomain
EJavaObject	?
EJavaClass	?
EBooleanObject	*BooleanDomain?*
EByteObject	*IntegerDomain?*
ECharacterObject	*StringDomain?*
EDoubleObject	*DoubleDomain?*
EFloatObject	*DoubleDomain?*
EIntegerObject	*IntegerDomain?*
ELongObject	*LongDomain?*
EShortObject	*IntegerDomain?*
EDate	*Record mit 6 x IntegerDomain?*
EBigInteger	*LongDomain?*
EBigDecimal	*DoubleDomain?*
EResource	?
EResourceSet	?
EFeatureMapEntry	?
EFeatureMap	?
EEnumerator	EnumDomain
EEList	ListDomain
ETreeIterator	?
EEList?	SetDomain
java.util.Map als eigenen DataType kapseln?	MapDomain
EClass?	RecordDomain

Abbildung 14: Abbildung von Datentypen, Quellen: [BHR+10], [lit]

für Metamodelle. Alle Metamodelle, die mit MOF definiert werden können, gehören dann zum Level M2. Ein Beispiel ist hier das Metamodell von UML. Zu den Metamodellen passende Modelle stehen dann auf dem Level M1. Ein einfaches UML Klassendiagramm ist dafür ein Beispiel. Auf dem Level M0 finden sich dann die Laufzeit-Instanzen des Modells [SV06].

Ecore ist der MOF sehr ähnlich, verzichtet aber auf eine Reihe von Features um die Komplexität zu reduzieren sowie die Implementierung zu optimieren und anpassbar zu machen. EMOF (Essential Meta Object Facility) ist eine vereinfachte Form von MOF, die sich stark an der Entwicklung von Ecore orientiert hat und daher weitestgehend kompatibel ist [SBPM08].

Aus diesem Grund lassen sich die vier Ebenen der OMG auch auf Ecore übertragen. Daraus ergibt sich dann folgender Aufbau:

- **M3:** Ecore-Metametamodell
- **M2:** Ecore-Metamodell
- **M1:** Ecore-Modell
- **M0:** reale Welt

Wie in Abschnitt 4.1 erklärt, lassen sich Ecore-Metamodelle auch als Graphenschemas interpretieren. Auf grUML übertragen, ergibt sich dann folgender Ebenenaufbau:

- **M3:** grUML-Metaschema
- **M2:** grUML-Schema
- **M1:** grUML-Graph
- **M0:** reale Welt

4.4 Zusammenfassung

grUML und Ecore bieten beide Möglichkeiten zur Modellierung von Graphen. Sie sind sich in vielen Aspekten sehr ähnlich, trotzdem gibt es einige essentielle Unterschiede, die eine verlustfreie Abbildung von der einen in die andere Sprache schwierig machen.

Es lohnt sich also zu überlegen, wie Ecore-Metamodelle in grUML-Schemas umgewandelt werden können.

5 Untersuchungen zur Abbildung

Im folgenden Kapitel wird versucht, mögliche Abbildungen von Ecore-Metamodellen auf grUML-Graphenschemas und umgekehrt zu finden. Dabei wird untersucht, ob eine verlustfreie Hin- und Rücktransformation möglich ist. Die folgenden Überlegungen sollen helfen, eine Abbildung von Ecore-Metamodellen nach grUML-Graphenschemas zu finden, die so viele Informationen wie möglich erhält und die Semantik des Metamodells bewahrt.

5.1 Pakete

Ein `EPackage` kann auf ein grUML-`Package` abgebildet werden und umgekehrt. Dabei entspricht das ererbte Attribut `name` von `EPackage` dem ebenfalls ererbten Attribut `qualifiedName` von `Package`. Abbildung 15 zeigt die Entsprechungen graphisch.

Die beiden zusammengehörigen Referenzen `eSuperPackage` und `eSubpackages` entsprechen der `ContainsSubPackage` Kante. Beide Konstrukte dienen der Hierarchisierung von Paketen und haben die gleiche Semantik.

Die ebenfalls zusammengehörigen Referenzen `eClassifiers` und `ePackage` entsprechen semantisch entweder der `ContainsDomain` oder der `ContainsGraphElementClass` Kante. Beide Konstrukte sagen etwas darüber aus, welche Elemente in einem Paket enthalten sind. Bei Ecore sind ausschließlich `EDataTypes` und `EClasses` enthalten, während bei grUML `Domains`, `VertexClasses` und auch `EdgeClasses` zu einem Paket gehören.

Problem: Paket einer Kante, die aus einer Referenz erzeugt wurde

Bei der Abbildung von Ecore nach grUML muss daher darauf geachtet werden, dass die neu erzeugten Kanten, welche zum Teil aus den Referenzen erstellt wurden, ebenfalls zu einem Paket hinzugefügt werden müssen. Verbindet die Kante Knoten aus verschiedenen Paketen, muss eine Entscheidung darüber getroffen werden, zu welchem Paket die neue Kante nun gehört. Ist die Kante aus einer `EReference` ohne `eOpposite` entstanden, bietet es sich an, sie zu dem Paket der Klasse, die sie enthielt, hinzuzufügen. Andernfalls bleibt nur eine willkürliche Auswahl, wie zum Beispiel die Kantenklasse des im Alphabet als erstes kommenden Paketes zu wählen, oder die Möglichkeit dem Benutzer anzubieten, selbst über die Paketzuordnung zu entscheiden.

Problem: grUML besitzt keine `nsURI` und kein `nsPrefix`

`EPackage` besitzt die Attribute `nsURI` und `nsPrefix`. Etwas Vergleichbares dazu findet sich in grUML nicht. Soll nun ein grUML-Schema auf ein Ecore-Metamodell abgebildet werden, müssen diese beiden Attribute generiert werden. Hier besteht die Möglichkeit beides aus dem Namen der Graphklasse abzuleiten, beispielsweise auf folgende Weise: `nsPrefix = NAME` und `nsURI = http://NAME.com`. Vorher könnte der Name noch normalisiert werden indem alle Buchstaben klein geschrieben werden oder Ähnliches. Zusätzlich könnte dem Benutzer eine Meldung über die gewählten Namen gegeben werden, so dass er sie nachträglich noch ändern kann, falls er dies wünscht.

Eine andere Variante wäre, dem Benutzer die Möglichkeit zu geben, die `nsURI` und das `nsPrefix` direkt anzugeben. Das hätte den Nachteil, dass keine voll automatische Transformation von mehreren Metamodellen, die sich der Benutzer vorher nicht angeschaut hat, möglich wäre. Auf der anderen Seite könnte der Benutzer aber dafür sorgen, dass die beiden Attribute sinnvoll besetzt werden.

Soll ein Ecore-Metamodell in ein grUML-Schema umgewandelt werden, können die nsURI sowie das nsPrefix als Attribute der Graphklasse hinzugefügt werden und die Werte als Default-Werte gespeichert werden. Das hat den Vorteil, dass diese Informationen nicht verloren gehen und bei einer eventuellen Rücktransformation wieder verwendet werden können.

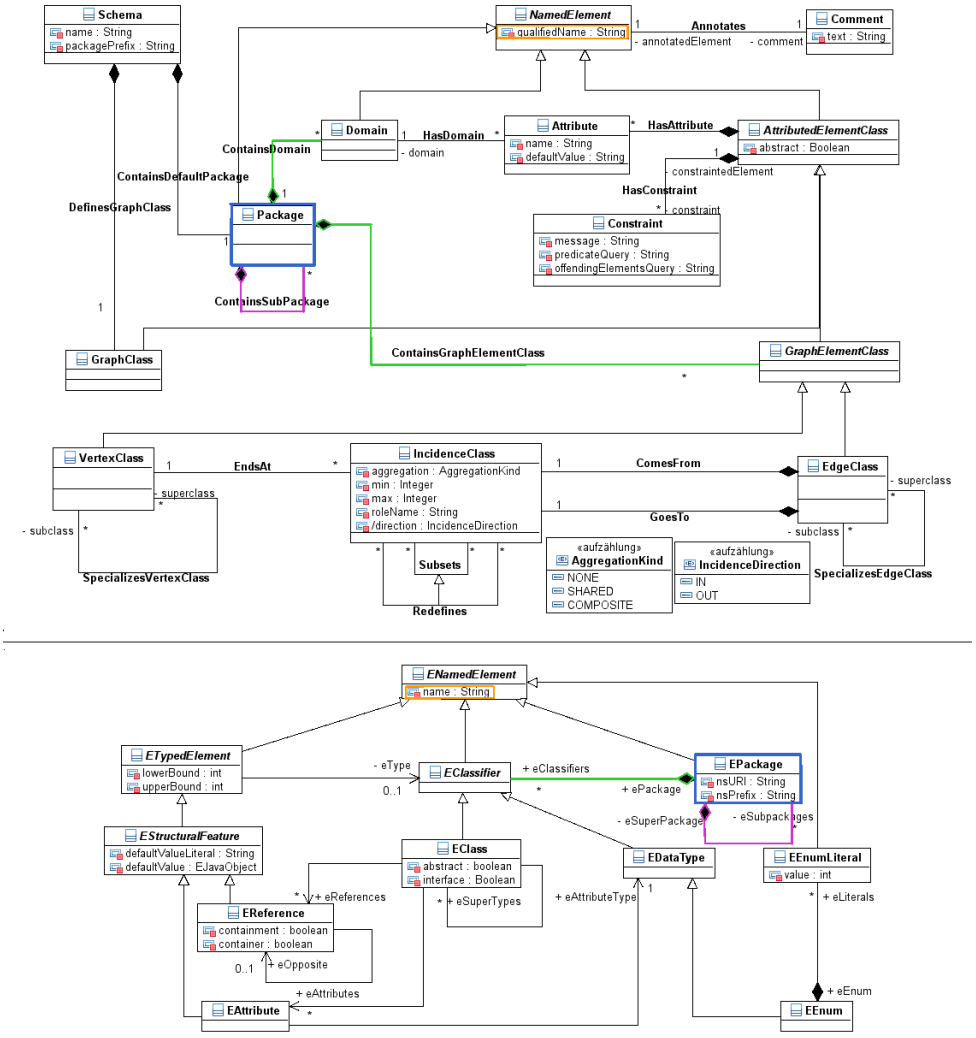

Abbildung 15: *Vorschlag Abbildung von Paketen*

5.2 Umgang mit der Graphklasse

Ecore-Metamodelle besitzen nichts Vergleichbares zur grUML-Graphklasse. Daher stellt sich die Frage, wie bei Abbildungen in beide Richtungen damit umgegangen werden kann.

Ecore -> grUML

Soll ein Ecore-Metamodell in ein grUML-Schema umgewandelt werden, muss die Graphklasse bei der Abbildung generiert werden.

Eine Möglichkeit besteht darin, die Graphklasse relativ unabhängig vom Rest des Metamodells zu erzeugen. In diesem Fall wird zur Erstellung nur ein Name benötigt. Dieser könnte beispielsweise vom Dateinamen des Metamodells abhängen. Eine Variante wäre an den Dateinamen ein “Graph” anzuhängen. Falls im Namen ein “Model” oder “Metamodel” vorkommt, könnte dieser Teil eventuell durch “Graph” ersetzt werden.

Ein Nachteil dieser Vorgehensweise fällt auf, wenn es im Metamodell eine Klasse gibt, die nur als Wurzelelement zur Navigation dient und keine semantische Bedeutung hat. Diese Klasse wird dann ebenfalls konvertiert, obwohl dies eigentlich nicht nötig ist, da die Graphklasse ihre Aufgabe übernimmt. (Beispiel: Universities aus Kapitel 4)

Eine andere Möglichkeit ist, die Graphklasse aus dem Wurzelelement, falls es eines gibt, zu generieren. In diesem Fall könnte der Name sowie gegebenenfalls Attribute übernommen werden. Dazu müsste aber zuvor untersucht werden, ob das gefundene Wurzelelement wirklich `containment`-Beziehungen zu allen `EClasses` hat, die nicht bereits in einer anderen `EClass` enthalten sind. Zusätzlich darf es darüber hinaus keine weiteren Referenzen besitzen, noch sollte es durch eine andere `EClass` referenziert werden. Ist das der Fall, stellt sich trotzdem die Frage, ob durch eine solche Vorgehensweise nicht Semantik verloren gehen kann, wenn diese Klasse vielleicht doch eine Aufgabe hat. (Beispiel: LibraryDatabaseSystem aus Kapitel 3)

grUML -> Ecore

Soll ein grUML-Schema in ein Ecore-Metamodell umgewandelt werden, muss entschieden werden, was aus der Graphklasse werden soll.

Eine Möglichkeit besteht darin, sie einfach wegzulassen. Falls die Graphklasse jedoch Attribute hat, gehen diese damit verloren. Außerdem würde es erheblich mehr Aufwand kosten, innerhalb des Metamodells zu navigieren.

Die andere Variante wäre, eine `EClass` zu erzeugen, die alle verbleibenden `EClasses` enthält, welche nicht bereits in anderen `EClasses` enthalten sind. In diesem Fall könnten die Attribute übernommen werden und die Navigation wäre zumindest insoweit gewährleistet, dass alle Klassen sofort erreicht werden können. Aus diesem Grund ist diese Variante zu bevorzugen.

5.3 Knotenklassen

grUML-Knotenklassen werden immer auf `EClasses` abgebildet. Die umgekehrte Richtung ist dagegen nicht eindeutig. Eine `EClass` kann zu einer `VertexClass` werden, aber auch zu

einer `EdgeClass`, wenn erkannt wird, dass es sich hierbei um eine konzeptuelle Kantenklasse handelt. Der zweite Fall wird in Abschnitt 5.4 näher beschrieben. Dieser Abschnitt beschäftigt sich nur mit `EClasses`, die eindeutig Knotenklassen werden müssen.

Sowohl `EClass` als auch `VertexClass` besitzen ein Namensattribut. `qualifiedName` von `VertexClass` kann daher sofort auf `name` von `EClass` abgebildet werden und umgekehrt.

Eine `EClass` kann als `abstract` oder als `interface` definiert werden. Für den Fall, dass sie `abstract` ist, gibt es eine direkte Entsprechung. `VertexClass` besitzt ebenfalls ein solches boolsches Attribut, welches sie von `AttributedElementClass` geerbt hat. Ist eine Klasse jedoch als `interface` definiert, gibt es keine eindeutige Abbildung. Das Attribut könnte einfach ignoriert und die `EClass` als normale `VertexClass` aufgefasst werden. Eine andere Variante wäre, wenn `interface` gesetzt ist, das `abstract` Attribut der `VertexClass` auf true zu setzen. Diese Variante erhält die Semantik von `interface` zumindest in soweit, dass von dieser Knotenklasse keine direkten Instanzen erstellt werden können. Um die Information nicht zu verlieren, kann auch ein Kommentar `<<interface>>` hinzugefügt werden.

Die Referenz `eSuperTypes` entspricht der Kante `SpecializesVertexClass`, was die Bedeutung betrifft und wird daher darauf abgebildet. Beide Konstrukte definieren die Vererbungshierarchie.

Die Referenz `eAttributes` wird auf die Kante `HasAttribute`, welche die `VertexClass` von `AttributedElementClass` geerbt hat, übertragen und umgekehrt. Sowohl Kante als auch Referenz dienen dazu, einer Knotenklasse beziehungsweise einer `EClass` Attribute zuzuordnen. Wie Attribute abgebildet werden, wird in Abschnitt 5.5 beschrieben.

5.4 Kantenklassen

Dieser Abschnitt beschäftigt sich mit der Transformation von und nach Kantenklassen. Eine Kantenklasse verbindet zwei Knotenklassen. Sie besitzt einen Namen, eine Richtung, eine Aggregationsart sowie eventuell Attribute. Für jedes Ende der Kante können Multiplizitäten und Rollennamen angegeben werden. Die Kantenklasse stellt den Typ der späteren Kante da. Kantenklassen unterstützen Generalisierung.

5.4.1 Einfache Kantenklassen

Als erstes muss die Frage geklärt werden, wie mit einfachen Kantenklassen, ohne Teil-Ganzes-Beziehung, ohne Attribute und ohne Vererbung umgegangen werden soll.

Abbildung 16 zeigt eine schlichte, einseitig navigierbare `EReference`, die in eine Kantenklasse transformiert werden soll.

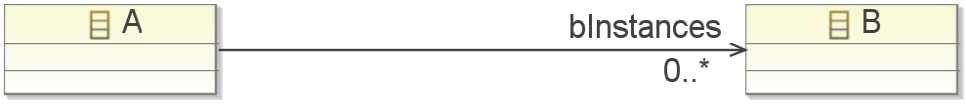

Abbildung 16: *Einseitig navigierbare EReference*

Die Referenz besitzt den Namen `bInstances` und die Multiplizität 0..*. Eine Instanz der Klasse `A` kann also beliebig viele Instanzen der Klasse `B` referenzieren.

Eine entsprechende, einfache grUML-Kantenklasse ist in Abbildung 17 dargestellt.

Abbildung 17: *Einfache grUML Kante*

Die Kantenklasse `EdgeAtoB` verläuft von der Knotenklasse `A` zur Knotenklasse `B`. Am Startknoten der Kante lautet der Rollenname `aInstances` und die Multiplizität *. Der Rollenname am Endknoten der Kante ist `bInstances` und die Multiplizität ebenfalls *.

Problem: Fehlende Rollen- und Kantennamen sowie Multiplizitäten bei Ecore -> grUML

Soll die `EReference` aus Abbildung 16 auf die Kantenklasse aus Abbildung 17 abgebildet werden, fällt auf, dass Informationen fehlen. Die Namen der Referenzen entsprechen semantisch dem *Rollennamen* auf der Ende-Seite der Kantenklasse. Einen Rollennamen für den Anfang der Kantenklasse gibt es nicht, genauso wenig wie einen Kantennamen.

Um den *Rollennamen* für den Anfang der Kantenklasse zu generieren, könnte der Name der `EClass`, zu der die Referenz gehört, in Kleinbuchstaben gewählt werden. Daran könnte dann ein “for” gefolgt von dem Namen der Referenz angehängt werden, um doppelt vorkommende Rollennamen zu vermeiden. Da ein so generierter Rollennamen jedoch keinerlei zusätzliche Semantik enthält, könnte es auch sinnvoll sein, den Rollennamen in diesem Fall einfach wegzulassen.

Die zweite Frage lautet, wie nun aus den Rollennamen ein *Name für die Kantenklasse* generiert werden kann. Eine Variante kann ein solcher String sein: “Is” + ROLLENNAME_ENDE + “Of” + ROLLENNAME_ANFANG Falls es einen generierten ROLLENNAME_ANFANG gibt, würde das aber zu unverständlichen langen Namen führen. Eine andere Möglichkeit ist, den zweiten Teil wegzulassen, also nur “Is” + ROLLENNAME_ENDE zu wählen. Alternativ wäre auch “Is” + ROLLENNAME_ENDE + “Of” + KLASSENNAME_ANFANG denkbar.

Außer dem Rollennamen vom Anfang der Kante und dem Kantennamen selbst, fehlt auch noch eine Angabe zur Multiplizität am Anfang der Kante. Da darüber jedoch nichts gesagt ist und es im Beispiel von Abbildung 16 ohne weiteres möglich ist, dass verschiedene Instanzen der Klasse `A` dieselben Instanzen der Klasse `B` referenzieren, sollte hier immer * gewählt werden.

Problem: Kantenrichtung bei Ecore -> grUML

Die Richtung der Kantenklasse ist im Fall von Abbildung 16 eindeutig übertragbar. `EClass A` referenziert `EClass B`, also gilt für die erzeugte Kante, dass sie vom Knoten `A` zum Knoten `B` verläuft. Es gibt jedoch auch `EReferences`, bei denen die Richtung nicht erkennbar ist, Abbildung 18 zeigt ein Beispiel.

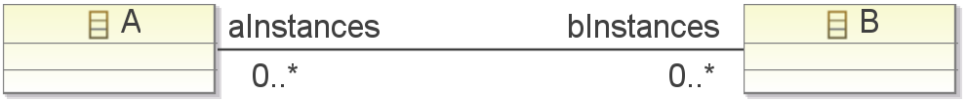

Abbildung 18: *Beidseitig navigierbare EReference*

Hier werden zwei `EReferences` durch die Nutzung der `eOpposite`-Attribute zu einer Aussage zusammengefasst. Klasse `A` referenziert `B` über die `EReference bInstances`, während `B A` durch `aInstances` referenziert. Die beiden `EReferences` haben die jeweils andere als `eOpposite` eingetragen.

Diese beiden Referenzen zusammen sollen in eine Kantenklasse transformiert werden. Beide Rollennamen und Multiplizitäten sind nun vorhanden und müssen nicht mehr generiert werden. Dafür fehlt jetzt die Kantenrichtung. Es muss daher eine Default-Einstellung gefunden werden. Beispielsweise kann die Knotenklasse, welche in alphabetischer Reihenfolge weiter vorne steht, als Startpunkt gewählt werden.

Problem: grUML -> Ecore, ein- oder zweiseitige Referenzen aus einer Kante?

Soll eine einfache grUML-Kantenklasse, wie in Abbildung 17 nach Ecore transformiert werden, kann daraus entweder eine `EReference` (Abbildung 16) oder zwei `EReferences` (Abbildung 18) werden. In beiden Fällen können nicht alle vorhandenen Informationen übertragen werden.

Wird die eine `EReference` ausgewählt, kann die Information der Richtung, sowie die Multiplizität und der Rollenname auf der Ende-Seite der Kante, beibehalten werden. Verloren gehen bei dieser Transformation der Rollennamen am Anfang der Kante, die Multiplizität dort und der Kantennamen. Lautet die Multiplizität am Anfang der Kante jedoch etwas anderes als *, wird hier bereits das Schema verfälscht. Auch wenn bei `EReferences` keine zweite Multiplizität notiert wird, ist sie implizit *. Bei der Rücktransformation würde die Multiplizität auf der Anfangsseite der Kante auf * gesetzt werden.

Wird stattdessen entschieden, dass eine grUML-Kantenklasse, wie in Abbildung 17 zu sehen, auf zwei zusammengehörige `EReferences` abgebildet werden soll, können beide Rollennamen und Multiplizitäten weiterhin gespeichert werden. Verloren gehen bei dieser Transformation die Richtung der Kante und der Kantenname.

5.4.2 Aggregationen und Kompositionen

`EReferences` besitzen ein `containment`-Attribut, welches definiert, ob es sich bei der Referenz um eine Komposition handelt. Abbildung 19 zeigt ein Beispiel.

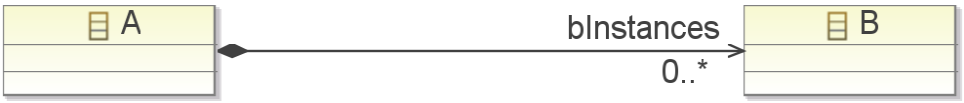

Abbildung 19: *Einseitig navigierbare Containment-EReference*

Die `EClass A` enthält beliebig viele Instanzen der `EClass B`. Der Name der `EReference` lautet `bInstances` und die Multiplizität ist auf * gesetzt.

Abbildung 20 zeigt ebenfalls eine Komposition in Ecore, nur handelt es sich hierbei um zwei zusammengehörige Referenzen.

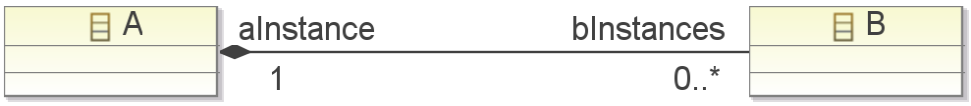

Abbildung 20: *Beidseitig navigierbare Containment-EReference*

In diesem Beispiel enthält die Klasse `A` wiederum beliebig viele Instanzen der Klasse `B`. Einziger Unterschied ist, dass den `B`-Instanzen bekannt ist, in welcher `A`-Instanz sie enthalten ist. Die Komposition ist exklusiv, eine Instanz kann also immer nur in einer anderen Instanz enthalten sein. In diesem Beispiel ist die Multiplizität auf der `Container`-Seite auf 1 gesetzt. Ansonsten ist, um die Exklusivität zu erhalten, nur noch 0..1 möglich, wenn mehrere Klassen Kompositionen zur betrachteten Klassen besitzen.

Die `bInstances` Referenz hat das `containment`-Attribut gesetzt, wie in Abbildung 16 auch. Zusätzlich hat die `aInstance` Referenz das `container`-Attribut gesetzt, um auszusagen, dass es sich bei der referenzierten Instanz der Klasse `A` um den Behälter der `B`-Instanz handelt.

Abbildung 21 zeigt eine entsprechende Komposition in grUML.

Abbildung 21: *grUML Kompositions-Kante*

Für die Kantenklasse `EdgeAtoB` ist in diesem Fall das `aggregation`-Attribut der `IncidenceClass` am Ende der Kante auf `COMPOSITE` gesetzt. In grUML wird die Komposition immer auf der Teil-Seite der Kante definiert.

Prinzipiell können Kompositionen genauso konvertiert werden, wie die einfachen Kantenklassen beziehungsweise Referenzen, die bereits in Unterabschnitt 5.4.1 beschrieben wurden. Allerdings sollten sie in Bezug auf Namen und Richtungen anders behandelt werden.

Bei der Transformation von Ecore nach grUML würde sich als Kantenname “Contains” + ROLLENNAME_TEIL anbieten. Existieren zwei Referenzen zu einer Kante, könnte auch alternativ ROLLENNAME_GANZES + “Contains” + ROLLENNAME_TEIL gewählt werden.

Bei zusammengehörigen `EReferences` wurde in Unterabschnitt 5.4.1 erklärt, dass eine deterministische Festlegung der Richtung für die zu erzeugende Kante kaum möglich ist. Für Kompositionen muss das nicht gelten. Es ist in grUML zwar durchaus möglich, eine Kompositions-Kante zu definieren, die von der Teil-Seite zur Ganzes-Seite verläuft. Trotzdem besteht die Möglichkeit, bei der Transformation von Ecore-Metamodellen in grUML-Schemas die Regel aufzustellen, dass die generierten Kanten immer von der Ganzes-Seite auf die Teil-Seite zeigen sollen. Somit wäre es zumindest möglich eine eindeutige Abbildungsvorschrift aufzustellen,

auch wenn die semantisch beste Richtung nicht bekannt ist.

Problem: Keine Aggregationen in Ecore

Bei der Transformation von grUML nach Ecore existiert das Problem, dass grUML auch Kantenklassen erlaubt, die eine einfache Aggregation modellieren und keine Komposition. Abbildung 22 zeigt ein Beispiel.

Abbildung 22: *grUML Aggregations-Kante*

In Ecore gibt es nichts Vergleichbares zu einer Aggregation. Es existieren also nur die beiden Möglichkeiten, die Kante in eine Komposition oder in eine normale Referenz umzuwandeln.

Wird die Kante in eine Komposition umgewandelt, kann es sein, dass damit die ursprüngliche Bedeutung verloren geht. Eine Aggregation kann auch auf der Seite mit der weißen Raute eine andere Multiplizität als 1 haben. Das ist für Kompositionen nicht möglich.

Wird die Kante in eine normale Referenz umgewandelt, geht die Information über die Aggregation verloren. Möglicherweise ließen sich die Referenznamen benutzen, um zumindest einen Hinweis zu erhalten. Beispielsweise könnten sie folgendermaßen lauten: “aggregatedROLLENNAME” und “aggregatingROLLENNAME”. Eine andere Variante wäre, eine Annotation einzuführen.

5.4.3 Kanten mit Attributen

grUML-Kantenklassen können Attribute besitzen. Abbildung 23 zeigt ein Beispiel.

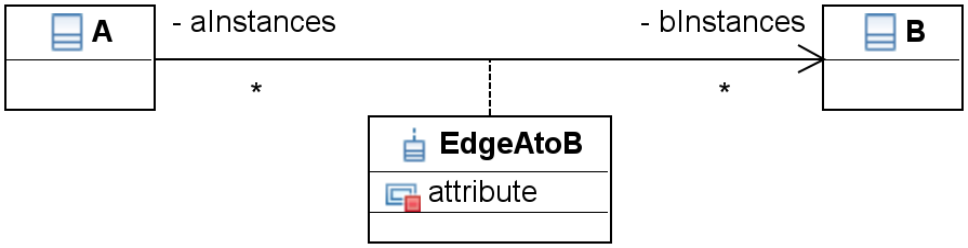

Abbildung 23: *grUML Kante mit Attribut*

Die Kantenklasse `EdgeAtoB` verbindet die Knotenklasse `A` mit der Knotenklasse `B`. Sie besitzt das Attribut `attribute`, welches eine Bedeutung bezüglich der Verbindung hat.

Das Gleiche gilt natürlich auch für Aggregationen und Kompositionen. Abbildung 24 zeigt ein Beispiel für eine Aggregation, Abbildung 25 für eine Komposition.

Nun stellt sich die Frage, wie eine solche Kantenklasse nach Ecore transformiert werden soll. Eine `EReference` kann keine Attribute besitzen, also bleibt nur die Möglichkeit, die Kantenklasse in eine `EClass` mit passenden Referenzen zu transformieren.

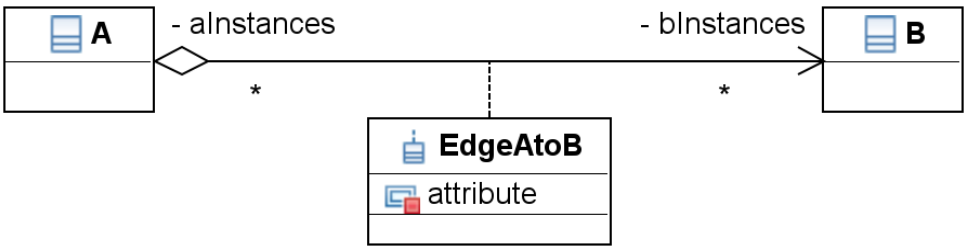

Abbildung 24: *grUML Aggregations-Kante mit Attribut*

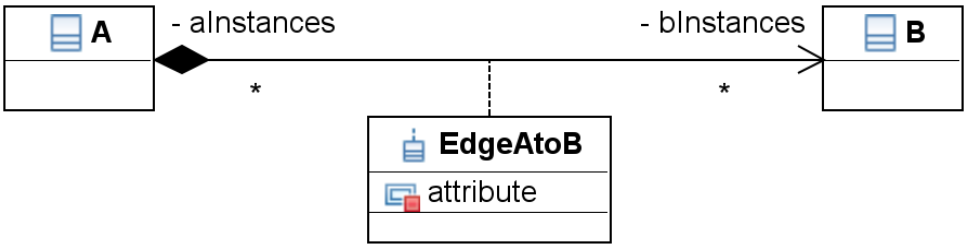

Abbildung 25: *grUML Kompositions-Kante mit Attribut*

Abbildung 26 zeigt einen ersten Versuch, der nur mit einzelnen Referenzen arbeitet.

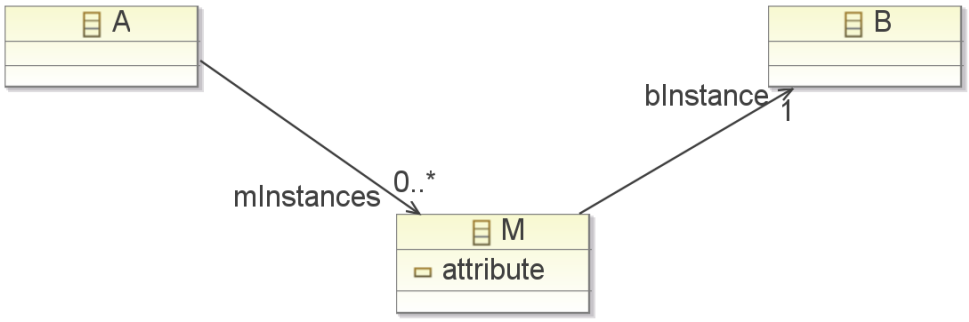

Abbildung 26: *Kante mit Attribut in Ecore - Einseitig navigierbar*

Das Attribut ist in diesem Lösungsvorschlag einfach transformierbar, allerdings stellt sich die Frage, ob die Semantik der Kante erhalten geblieben ist. Wird Abbildung 23 als Ausgangsschema betrachtet, kommen Zweifel auf. Die expliziten Multiplizitäten sind zwar korrekt übertragen worden, jedoch wurden die impliziten verletzt. Eine Kante verbindet immer genau 2 Instanzen, so dass sowohl bei `A` als auch bei `B` als Multiplizität eine 1 stehen müsste. Für Klasse `A` ist das aber nicht erfüllt, da bei einer einzelnen Referenz die implizite Multiplizität * ist.

Es sollten also zwei zusammengehörige Referenzen genutzt werden, um mit Hilfe einer `EClass` wirklich eine Kante repräsentieren zu können. Abbildung 27 zeigt den neuen Vorschlag.

Jetzt ist sichergestellt, dass eine Instanz von `M` wirklich immer nur eine Instanz von `A` mit einer Instanz von `B` verbinden kann.

Die Aggregation aus Abbildung 24 sollte genauso wie die normale attributierte Kante aus Abbildung 23 behandelt werden da Ecore keine Aggregationen kennt.

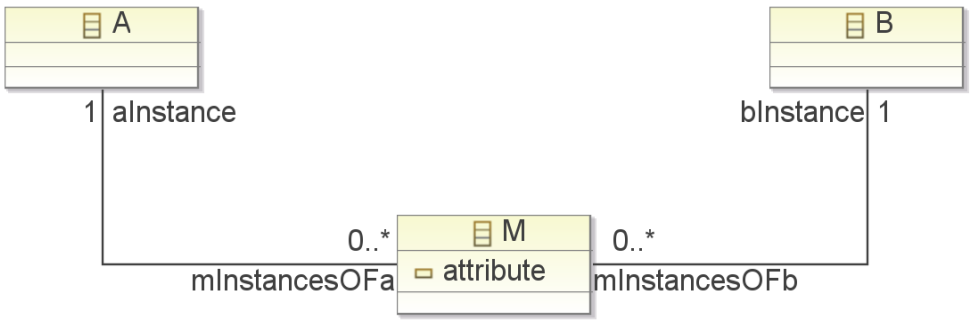

Abbildung 27: *Kante mit Attribut in Ecore - Beidseitig navigierbar*

Für die Komposition aus Abbildung 25 stellt sich die Frage, ob einzelne Referenzen ausreichen könnten, da die Multiplizitäten hier nicht verletzt werden. Abbildung 28 zeigt eine mögliche Transformation.

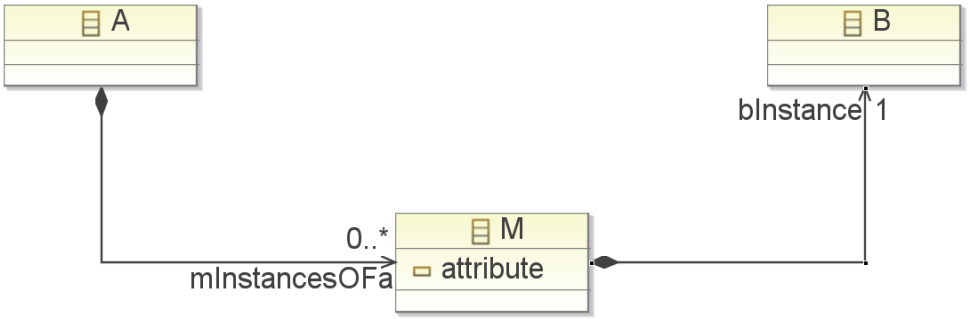

Abbildung 28: *Kompositions-Kante mit Attribut in Ecore - Einseitig navigierbar*

Problem: Erkennen einer Klasse als konzeptuelle Kantenklasse?

Ist die Abbildung einer attributierten Kante von grUML nach Ecore vollständig definiert, stellt sich die Frage nach der Rücktransformation. Ist es möglich ehemalige Kanten in einem Ecore-Metamodell zu erkennen und anschließend wieder auf eine grUML Kante abzubilden? Im Prinzip kann nach dem Charakteristikum einer Kante gesucht werden. Dazu gehört zum Beispiel, dass eine Kante immer genau zwei Knoten verbindet.

Eine andere Möglichkeit wäre, dem Benutzer die Möglichkeit zu geben, diejenigen `EClasses` zu benennen, die zu Kanten werden sollen. Dann muss jedoch überlegt werden, was mit der `EClass` geschehen soll, wenn das Charakteristikum nicht erfüllt ist. Abbildung 29 zeigt eine solche Problemklasse.

In diesem Fall könnte die Transformation mit einer Exception abgebrochen, oder die Benutzer-Eingabe nach einer Konsolenausgabe ignoriert werden. Eine Möglichkeit, diese Klasse in eine Kante zu transformieren und die Semantik der Multiplizitäten beizubehalten, gibt es nicht.

Wird eine grUML-Schema in ein Ecore-Metamodell transformiert, kann eine Möglichkeit geschaffen werden, sicher zu erkennen, welche Klassen eigentlich Kantenklassen sind. Dazu wird entweder ein Marker-Interface eingeführt, welches alle `EClasses`, die aus einer Kantenklasse entstanden sind, implementieren oder die Eigenschaft als Annotation gespeichert. Dies kann dann bei der Rücktransformation erkannt und entsprechend behandelt werden.

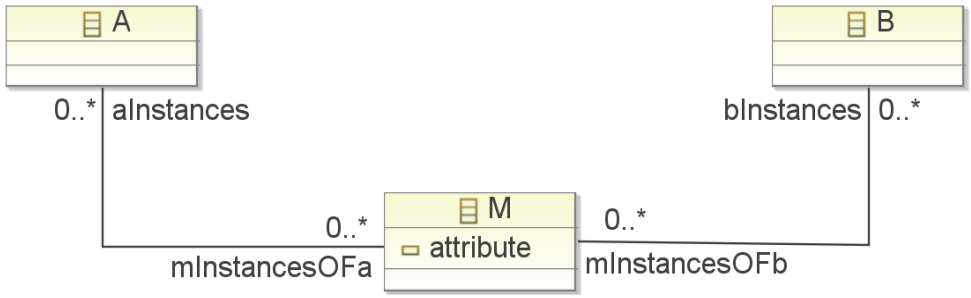

Abbildung 29: *Ausschnitt aus einem Ecore-Metamodell*

5.4.4 Spezialisierte Kanten

grUML-Kantenklassen unterstützen Generalisierung. Abbildung 30 zeigt ein Beispiel.

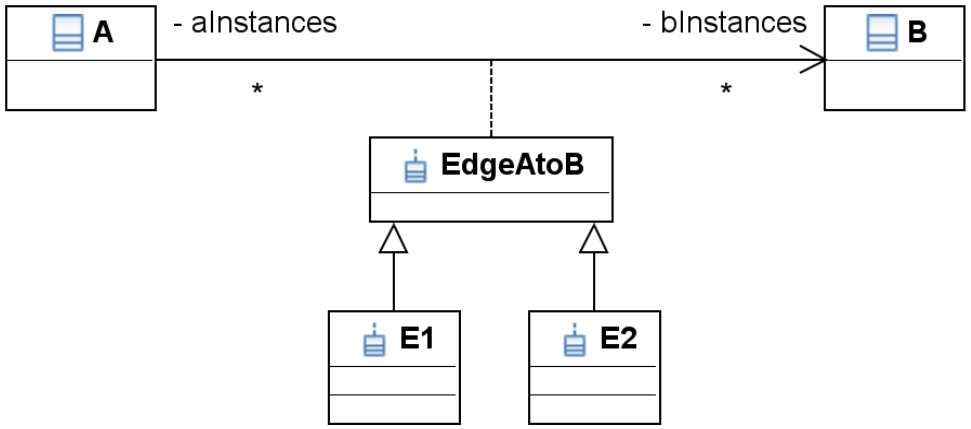

Abbildung 30: *grUML Kante mit Spezialisierungen*

Die Beispiel-Kante kann natürlich auch eine Aggregation oder Komposition sein, wie die Abbildungen 31 und 32 zeigen.

In allen Fällen hat die Kantenklasse `EdgeAtoB` zwei Spezialisierungen, `E1` und `E2`. `EReferences` unterstützen keine Vererbung, aus diesem Grund ist eine Abbildung wie in Abschnitt 5.4.1 beschrieben nicht möglich. Dafür könnte Abbildung 30 ähnlich wie in Abschnitt 5.4.3 erläutert konvertiert werden. Abbildung 33 zeigt eine mögliche Lösung.

Um die Kantenklasse zu modellieren, wird wiederum eine `EClass` benötigt. `EClasses` unterstützen Generalisierung, also können nun zwei Subklassen erzeugt werden. Zu beachten sind hier wieder alle Aspekte, die bereits bei der Transformation von Kanten mit Attributen in 5.4.3 erläutert wurden.

Da die Aggregation wiederum nicht modelliert werden kann, wird Abbildung 31 genauso transformiert, wie Abbildung 30.

Die Komposition wird genauso transformiert. Abbildung 34 zeigt ein mögliches Ergebnis.

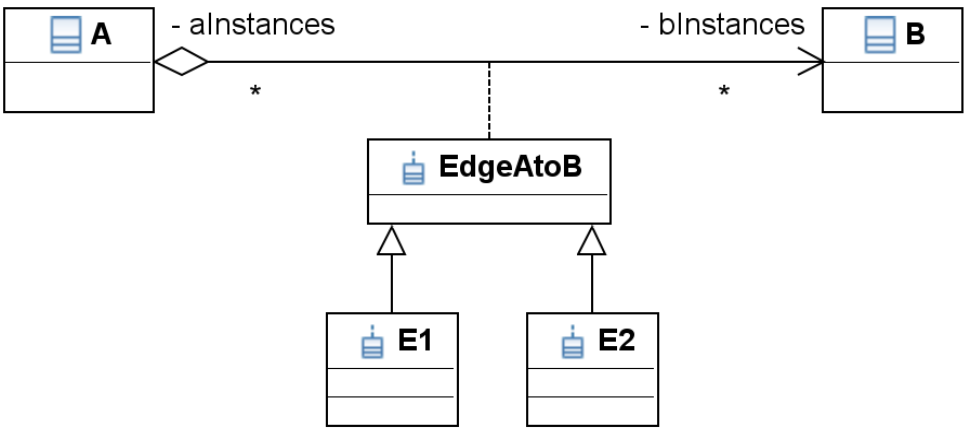

Abbildung 31: *grUML Aggregations-Kante mit Spezialisierungen*

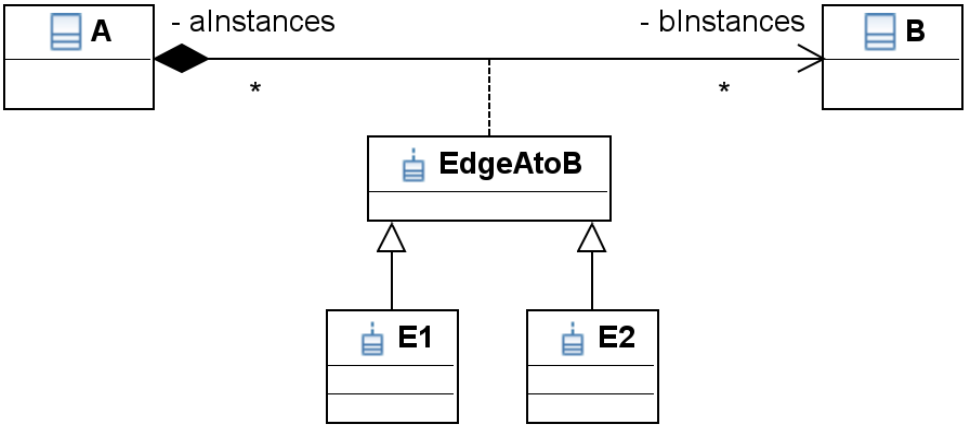

Abbildung 32: *grUML Kompositions-Kante mit Spezialisierungen*

Bei der Abbildung von Ecore nach grUML treten weitere Probleme auf. In einigen Fällen kann es vorkommen, dass eine `EClass`, welche eine konzeptuelle Kantenklasse darstellt, nur einige `EReferences` ihrer Superklasse überschreibt. Abbildung 35 zeigt einen solchen Fall.

Es ist nicht zu erkennen, ob die `EReference startChild` die `EReference start` oder `end` überschreibt. Die konzeptuelle Kantenklasse `EdgeChild` verbindet die beiden `EClasses A` und `ChildA`. Durch die Unklarheit der Richtung, ist nicht eindeutig erkennbar, wo die Kante beginnen soll.

Das gewünschte Ergebnis zeigt Abbildung 36.

Ohne zusätzliche Informationen, kann die Kantenklasse `EdgeChild` aber auch von `A` nach `ChildA` führen.

Um dieses Problem zu lösen, kann dem Benutzer die Möglichkeit gegeben werden, anzugeben, welche `EReferences` welche ererbten `EReferences` überschreiben.

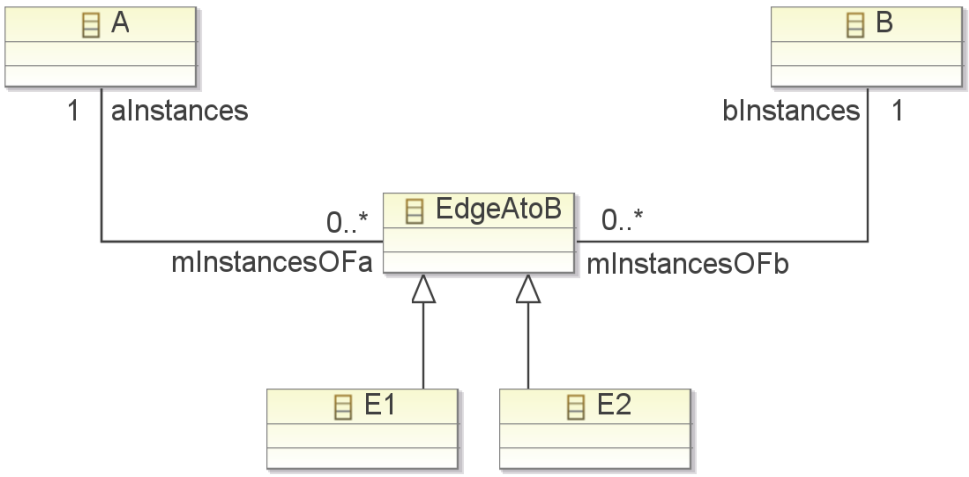

Abbildung 33: *Transformation einer Kante mit Spezialisierungen*

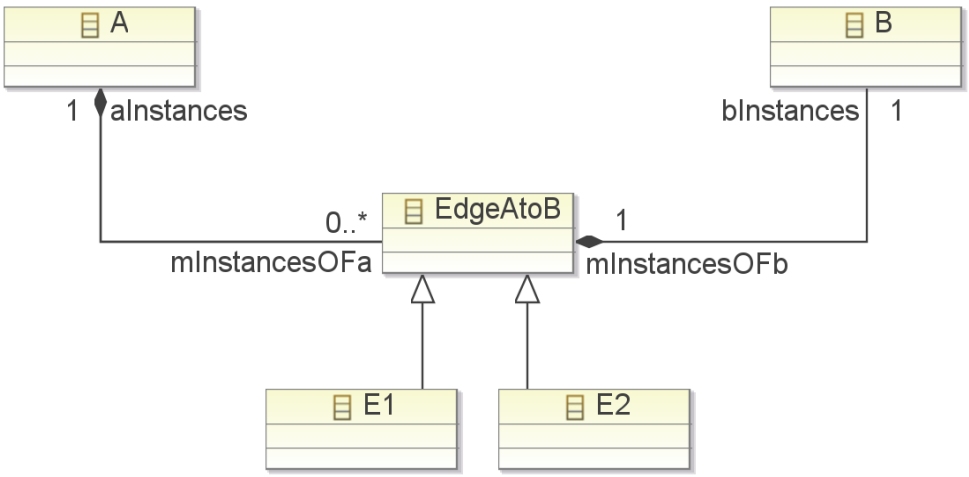

Abbildung 34: *Transformation einer Kompositions-Kante mit Spezialisierungen*

5.5 Attribute

Ein `EAttribute` in Ecore entspricht einem `Attribute` in grUML. Dabei kann die Eigenschaft `name`, welche `EAttribute` von `ENamedElement` geerbt hat, als `name` von `Attribute` genutzt werden und umgekehrt. Abbildung 37 zeigt graphisch, welche Elemente zugeordnet werden können.

`Attribute` besitzt ein `defaultValue`, welches durch einen `String` repräsentiert wird. Die Klasse `EAttribute` besitzt dementsprechend ein `defaultValueLiteral`, welches ebenfalls den Default-Wert eines Attributs als Text angibt. Beide Konstrukte sind daher semantisch gleich, wodurch die Abbildung eindeutig ist. `EAttribute` besitzt auch noch ein `defaultValue` vom Typ `EJavaObject`. Dies muss jedoch nicht zwangsläufig belegt werden und kann daher bei der Abbildung von grUML nach Ecore nicht besetzt werden. Bei der Abbildung von Ecore nach grUML kann dieses Attribut nicht transformiert werden.

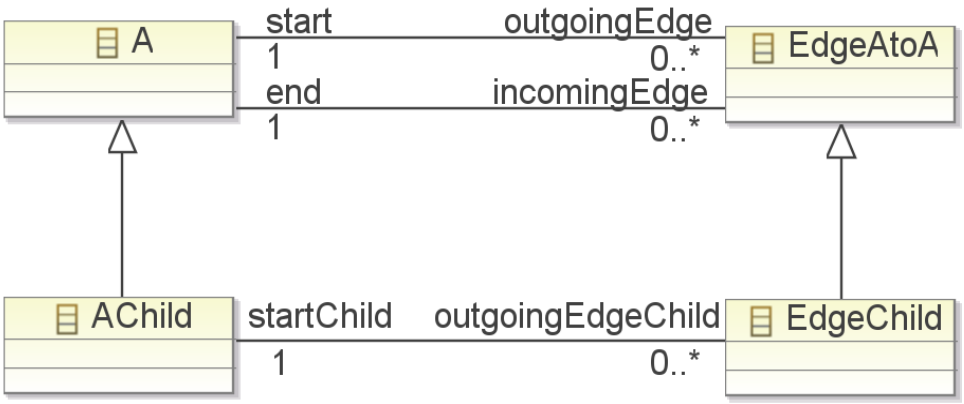

Abbildung 35: *Eine konzeptuelle Kantenklasse in Ecore mit uneindeutig überschreibenden Referenzen*

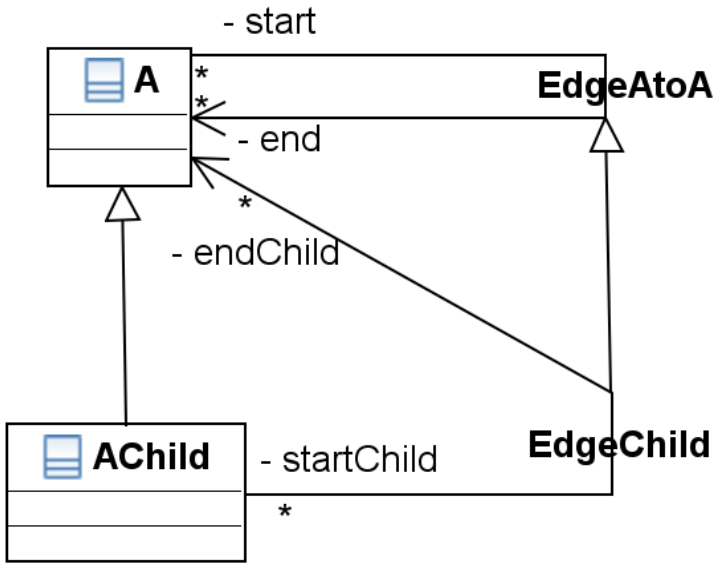

Abbildung 36: *Kantenklasse in grUML, welche aus Abbildung 35 entstanden ist*

Die Referenz `eAttributeType` entspricht semantisch der Kante `HasDomain`. Allerdings müssen hier einige Aspekte beachtet werden. Wie die `DataTypes` im Einzelnen auf die `Domains` abgebildet werden, wird in Abschnitt 5.7 näher erläutert. Interessant sind hier die Multiplizitäten von `EAttribute`, welche angeben, wie oft ein Attribut vorkommen kann. `Attributes` haben keine Multiplizitäten, eine direkte Abbildung ist also nicht möglich.

Bei dem Versuch, ein Ecore-Metamodell in ein grUML-Schema umzuwandeln, muss bei der Betrachtung der `EAttributes` die `upperBound` berücksichtigt werden. Ist diese größer als eins, dann muss das erzeugte `Attribute` als `Domain` eine Liste oder eine Menge erhalten, ansonsten genügt die direkte Typumwandlung.

Entsprechen `lowerBound` und `upperBound` nicht den Multiplizitäten 0..1, 1 oder 0..*, ist es nicht möglich dies korrekt in grUML nachzumodellieren. Möglicherweise könnten an dieser Stelle jedoch Constraints benutzt werden, um die Semantik zu erhalten.

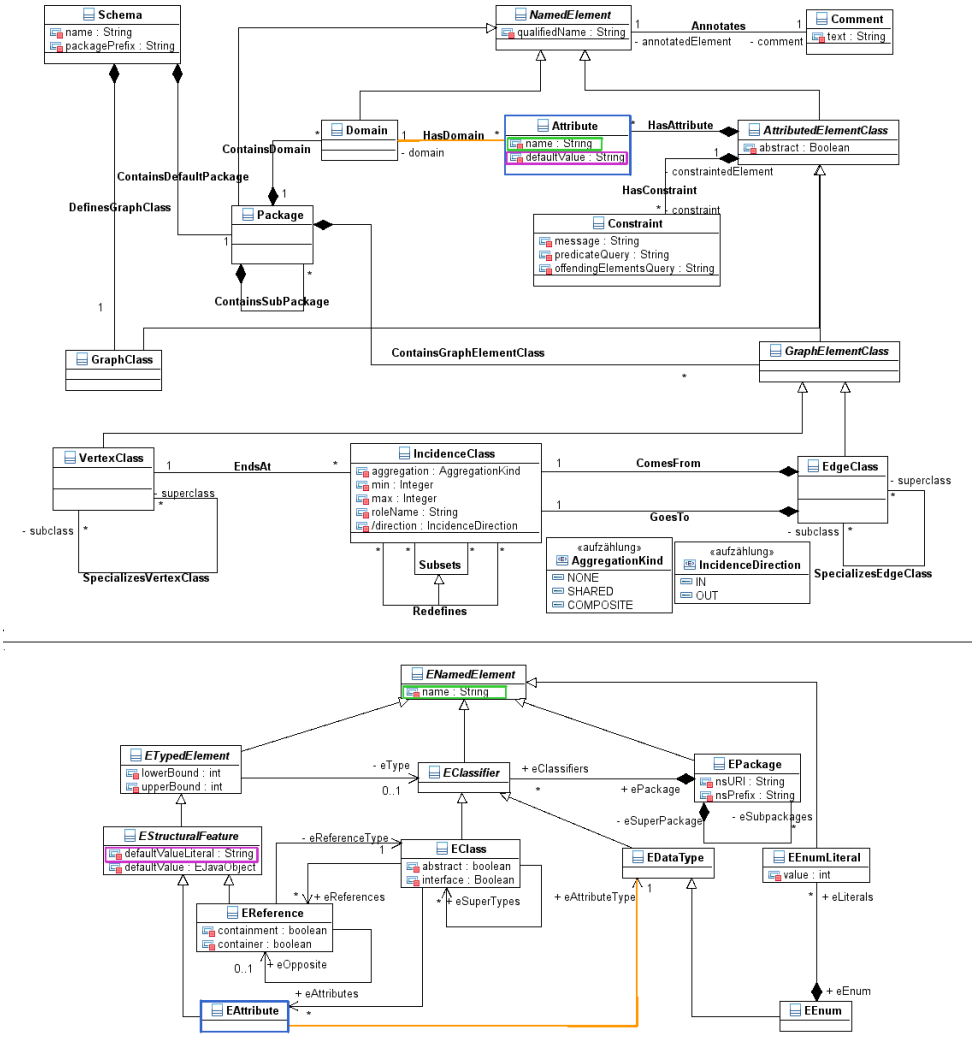

Abbildung 37: *Vorschlag Abbildung von Attributen*

Soll ein grUML-Schema auf ein Ecore-Metamodell abgebildet werden und ein `Attribute` besitzt den Datentyp `ListDomain`, muss das erzeugte `EAttribute` als `EDataType` die Typabbildung des in der Liste enthaltenen Datentyps erhalten und `upperBound` sollte auf -1, was unbeschränkt bedeutet, gesetzt werden. Gleiches gilt für den Datentyp `SetDomain`.

5.6 Annotationen, Kommentare, Constraints

grUML und Ecore bieten unterschiedliche Möglichkeiten, Zusatzinformationen zu Elementen zu speichern.

Ecore-Metamodellelemente besitzen `EAnnotations`, die als `source` einen Typ haben, der durch eine URI repräsentiert wird. Mögliche Typen sind beispielsweise Ecore, GenModel und Extended Metadata. Ecore `EAnnotations` speichern allgemeine Modellinformationen, die sowohl zur Laufzeit als auch zur Codegenerierung wichtig sein können, der Typ ermöglicht es,

Constraints zu spezifizieren. GenModel `EAnnotations` speichern Informationen, die nur für die Codegenerierung relevant sind. Extended Metadata `EAnnotations` sind nur in Metamodellen vorhanden, die aus XML-Schemas erstellt wurden. Sie speichern Informationen, die bei der Transformation in das Ecore-Metamodell nicht übertragen werden konnten. [SBPM08]

grUML-Graphelemente sowie die Graphklasse können Constraints oder Kommentare besitzen. Kommentare bestehen aus einem einfachen Text, der zusätzliche Informationen zum Element speichert. Constraints dagegen bestehen aus einer textuellen Beschreibung, einem entsprechenden GReQL Prädikat und einer GReQL Query. [BHR+10]

Nun stellt sich die Frage, ob es möglich ist, die Zusatzinformationen zu einem Modell zu erhalten. Da die Systeme, mit denen Constraints definiert werden sehr unterschiedlich sind, wird es nicht möglich sein, Ecore-Constraints in grUML-Constraints umzuwandeln. Allerdings könnte eventuell der Inhalt eines Ecore-Constraints in Textform als grUML-Kommentar abgespeichert werden.

Im Falle der anderen Richtung scheint es schwierig zu sein, die Ecore `EAnnotations` zu erhalten, da sie dazu gedacht sind, zur Laufzeit beziehungsweise zur Codegenerierung überprüft zu werden. Einfache Textannotation ist nicht möglich.

5.7 Datentypen

In der folgenden Tabelle werden zu den vordefinierten Ecore-Datentypen ihre Entsprechungen in grUML angegeben und umgekehrt.

Ecore	**grUML**
EBoolean	BooleanDomain
EBooleanObject	BooleanDomain
EDouble	DoubleDomain
EFloat	DoubleDomain
EDoubleObject	DoubleDomain
EFloatObject	DoubleDomain
EInt	IntegerDomain
EIntegerObject	IntegerDomain
EShort	IntegerDomain
EShortObject	IntegerDomain
EByte	IntegerDomain
EByteObject	IntegerDomain
ELong	LongDomain
ELongObject	LongDomain
EString	StringDomain
EChar	StringDomain
ECharacterObject	StringDomain
EDate	*RecordDomain mit 6 x IntegerDomain*
EEnumerator	EnumDomain
*t umgewandelt und Multiplizität auf **	ListDomain<Domain t>
EEList	*ListDomain?*
*t umgewandelt und Multiplizität auf **	SetDomain<Domain t>
java.util.Map als eigenen DataType kapseln?	MapDomain
EClass?	RecordDomain
EBigInteger	?
EBigDecimal	?
EJavaObject	?
EJavaClass	?
EResource	?
EResourceSet	?
EFeatureMapEntry	?
EFeatureMap	?
ETreeIterator	?

Quellen: [BHR+10], [lit]

Der erste Teil der Tabelle, bis `ECharacterObject`, ist relativ eindeutig. Einziges Problem ist, dass bei der Abbildung von grUML nach Ecore bei einigen Datentypen eine der Möglichkeiten ausgewählt werden muss. `IntegerDomain` könnte so zum Beispiel nach `EShort`, `EShortObject`, `EInt` oder `EIntegerObject` transformiert werden. Da `EInt` semantisch am

ähnlichsten ist, sollte das per default gewählt werden. Ebenso sollte `BooleanDomain` auf `EBoolean`, `DoubleDomain` auf `EDouble`, `LongDomain` auf `ELong` und `StringDomain` auf `EString` abgebildet werden.

Wird die Abbildung so definiert, tritt folgendes Problem auf: Wird ein Ecore-Metamodell mit einer Klasse, die ein Attribut vom Typ `EByte` besitzt, in ein grUML-Schema umgewandelt, so wird daraus ein Knoten mit einem Attribut vom Typ `IntegerDomain`. Soll dieses Schema jetzt wieder zurück in ein Ecore-Metamodell transformiert werden, dann erhält das Attribut den `DataType EInt`. Die Information, dass das Attribut vorher den Typ `EByte` besessen hat, ist verloren gegangen. Das gleiche gilt auch für die anderen nicht eindeutigen Datentypen.

Der Ecore-Datentyp `EDate` kann nicht direkt auf einen grUML-Datentyp abgebildet werden, da eine Datumsklasse dort nicht vorgesehen ist. Eine Möglichkeit wäre aber, eine `RecordDomain` zu erstellen, die sechs `IntegerDomains` für Jahr, Monat, Tag, Stunde, Minute und Sekunde enthält. Hier besteht nur auch wieder das Problem, dass bei der Rücktransformation erkannt werden müsste, dass diese `RecordDomain` dem Datentyp `EDate` entspricht.

Die unterschiedliche Modellierung von Listen in grUML und Ecore wurde bereits in Kapitel 5.5 kurz angesprochen. grUML bietet eine `ListDomain` an, die eine `BaseDomain` besitzt. Ecore dagegen besitzt keinen Datentyp, der genau dem entspricht. Es gibt zwar das Interface `EEList`, allerdings ist dies hauptsächlich deshalb vorhanden, weil Ecore seine eigene API modelliert. Stattdessen haben `EAttributes` in Ecore Multiplizitäten. Handelt es sich um eine Liste, sollte `upperBound` größer als 1 beziehungsweise gleich -1 sein.

Für die grUML `SetDomain` gibt es keine direkt Entsprechung. Soll daher ein grUML-Schema, welches Attribute vom Typ `SetDomain` enthält in ein Ecore-Metamodell transformiert werden, kann es behandelt werden, als sei es vom Typ `ListDomain`. Um die Semantik zu erhalten kann dann das Attribut `unique` auf true sowie `ordered` auf false gesetzt werden.

grUML besitzt eine `MapDomain` während Ecore nichts Vergleichbares anbietet. Eine Möglichkeit, diesen Datentyp trotzdem zu transformieren, wäre einen `EDataType` zu definieren, der `java.util.Map` kapselt und somit anbietet.

Desweiteren gibt es in grUML die `RecordDomain`, welche als Behälter für verschiedene Datentypen genutzt werden kann. In Ecore gibt es so etwas nicht. Eine Idee wäre, für eine `RecordDomain` eine `EClass` zu generieren, welche die enthaltenen Datentypen als Attribute besitzt. Dann müsste für diejenige `EClass`, welche aus der `VertexClass` generiert wurde, die ein Attribut vom Typ der speziellen `RecordDomain` besitzt, aber kein `EAttribute` erstellt werden, sondern eine `EReference`. Diese `EReference` kann dann eine Instanz der `EClass` referenzieren, die der `RecordDomain` entspricht. Um die Information nicht zu verlieren, sollte die neu erzeugte `EClass` entsprechend annotiert werden.

Die `DataTypes EBigInteger` und `EBigDecimal` kapseln die entsprechenden Datentypen aus der Java-Mathebibliothek. grUML besitzt keine vergleichbaren Konstrukte. Allerdings ist es wenig wahrscheinlich, dass `EBigInteger` und `EBigDecimal` in vielen Ecore-Metamodellen vorkommen. Aus diesem Grund mag es ausreichen, die beiden Datentypen in normale Integer beziehungsweise Double zu verwandeln oder die Transformation, sollte einer von beiden wirklich vorkommen, abzubrechen.

Die verbleibenden Datentypen, das sind `EJavaObject`, `EJavaClass`, `EResource`, `EResourceSet`, `EFeatureMap`, `EFeatureMapEntry` und `ETreeIterator`, werden mit hoher Wahrscheinlichkeit in keinem Metamodell vorkommen. Sie sind ausschließlich deshalb enthalten, weil Ecore sein eigenes Metamodell ist.

Neben den vordefinierten Datentypen ist es in Ecore auch noch möglich, selbst `EDataTypes` zu definieren, die vorhandene ausprogrammierte Javaklassen kapseln. Es stellt sich also die Frage, wie mit solchen Datentypen bei der Transformation nach grUML umgegangen werden kann. Da grUML keine vergleichbare Möglichkeit bietet, wird es wohl nicht möglich sein, diese `EDataTypes` zu konvertieren. Es bleibt also nur, sollte ein solcher Datentyp vorkommen, entweder die Transformation mit einer Fehlermeldung abzubrechen oder den Datentyp nicht zu konvertieren, das entsprechende Attribut wegzulassen und eine entsprechende Ausgabe zu generieren.

5.8 Zusammenfassung

Es ist klar zu erkennen, dass eine eindeutige verlustfreie Abbildung von grUML nach Ecore ohne das Hinzufügen von zusätzlichen Informationen in Kommentaren und Annotationen nicht möglich ist. Entweder geht die Kantenrichtung verloren, oder Semantik bezüglich der Multiplizitäten.

Die umgekehrte Abbildung von Ecore nach grUML kann ohne Zusatzinformationen genauso wenig eindeutig und verlustfrei sein. Eine Kante, die aus einer einzelnen `EReference` entstanden ist, sieht nicht anders aus als eine, die aus einem zusammengehörigen Paar generiert wurde. Eine Möglichkeit wäre hier noch keinen fehlenden Rollennamen zu generieren, dann könnte diese Information noch erhalten bleiben. Was aber auf jeden Fall verloren geht und auch nicht durch Kommentare erhalten werden kann, sind die selbst definierten `DataTypes`. Ohne Kommentare gehen auch die Quellinformationen verloren, wenn aus einem `EByte` eine `IntegerDomain` wird.

6 Anforderungen

In diesem Abschnitt werden Anforderungen an die Software, die Ecore-Metamodelle in grUML-Schemas und Ecore-Modelle in grUML-Graphen umwandeln soll, aufgestellt.

6.1 Allgemeines

A1 Die Software muss es ermöglichen, Ecore-Metamodelle in grUML-Schemas zu transformieren.

A2 Die Software muss es ermöglichen, Ecore-Modelle, zu denen ein passendes Metamodell vorliegt, in TGraphen zu transformieren.

A3 Die Software soll JGraLab benutzen.

A4 Die Software soll die Ecore API benutzen.

A5 Die Ecore-Metamodelle sollen im Ecore XMI-Format vorliegen.

A6 Ecore-Modelle müssen als Metamodell konforme XMI- oder XML-Datei vorliegen.

A7 Die Software muss als Ausgabe TG-Dateien mit den erstellten Schemas und Graphen liefern können.

A8 Die Software muss als Ausgabe das erzeugte Schema als de.uni_koblenz.jgralab.schema.SchemaGraph Objekt liefern können.

A9 Die Software muss als Ausgabe die erzeugten Graphen als de.uni_koblenz.jgralab.Graph Objekte liefern können.

6.2 Funktionalität

F1 Die Software soll eine automatische Transformation ohne Benutzerinteraktion ermöglichen.

F2 Die Software soll die Möglichkeit bieten, eine Konfigurationsdatei mitzugeben. In dieser soll der Nutzer alle Einstellungen, die in F4 - F13 genannt werden, vornehmen können.

F3 Der Nutzer soll die Einstellungen F4 - F13 direkt über die API konfigurieren können.

F4 Die Software soll es dem Benutzer ermöglichen, den zu erstellenden grUML-Kanten Namen zu geben.

F5 Die Software soll es dem Benutzer ermöglichen, zu entscheiden, in welches Paket eine Kantenklasse einzuordnen ist.

F6 Die Software soll es dem Benutzer ermöglichen zu entscheiden, welche Richtung eine Kantenklasse haben soll.

F7 Für Aggregationen und Kompositionen soll der Benutzer die Einstellungsmöglichkeit besitzen, alle Richtungen vom Ganzen zum Teil oder umgedreht zu definieren.

F8 Die Software soll es dem Benutzer ermöglichen, für `EClasses`, die konzeptuelle Kantenklassen sind und Superklassen besitzen, zu definieren, welche `EReferences` der Superklassen durch die eigenen überschrieben werden.

F9 Die Software soll die Option bieten, dass der Benutzer `EClasses` angeben kann, die im grUML-Schema zu Kantenklassen werden sollen.

F10 Die Software soll die Option bieten, automatisch nach `EClasses` zu suchen, die Kanten repräsentieren. Diese sollen dann auch in grUML-Kantenklassen transformiert werden.

F11 Die Software soll im Default-Fall eine zusätzliche Graphklasse für das grUML-Schema generieren. In diesem Fall soll die Software dem Benutzer die Möglichkeit geben, den Namen der Graphklasse zu definieren.

F12 Die Software soll die Option bieten, das Wurzelelement des Ecore-Metamodells in die Graphklasse des grUML-Schemas zu transformieren.

F13 Die Software soll es dem Benutzer ermöglichen zu entscheiden, ob bei einseitigen Ecore-Referenzen fehlende Rollennamen generiert werden sollen.

F14 `nsURI` und `nsPrefix` sollen Attribute der erzeugten Graphklasse werden. Ihre Werte im Ecore-Metamodell sollen die Default-Werte der grUML-Attribute ergeben.

6.3 Fehlerbehandlung

E1 Die Software soll mit einer aussagekräftigen Fehlermeldung abbrechen, wenn die übergebene XMI-Datei kein valides Ecore-Metamodell darstellt.

E2 Die Software soll die Transformation eines Modells mit einer Fehlermeldung abbrechen, wenn das mit übergebene Metamodell nicht zum Modell passt.

E3 Die Software soll im Falle von nicht transformierbaren Ecore `EDataTypes` das zugehörige Attribut sowie den `EDataType` selber bei der Transformation weglassen und den Benutzer mit einer Warnung darüber informieren.

E4 Gibt der Benutzer eine `EClass` ein, die zur Kante werden soll, aber nicht die Eigenschaften einer Kante erfüllt, dann soll die Transformation des Metamodells mit einer Fehlermeldung abgebrochen werden.

6.4 Transformationen

T1 `EPackages` sollen in grUML `Packages` transformiert werden.

T2 `EAttributes` sollen in `Attributes` transformiert werden.

T3 `EClasses` sollen in `VertexClasses` bzw. `EdgeClasses` transformiert werden.

T4 `EReferences` sollen in `EdgeClasses` transformiert werden. Dabei wird entweder aus einer `EReference` eine `EdgeClass`, aus zwei zusammengehörigen oder aus 4 `EReferences` bei denen 2 jeweils zusammengehörig sind plus einer `EClass`.

T5 Ein `EBoolean` wird zu `BooleanDomain`.

T6 Ein `EByte` wird zu `IntegerDomain`.

T7 Ein `EChar` wird zu `StringDomain`.

T8 Ein `EDouble` wird zu `DoubleDomain`.

T9 Ein `EFloat` wird zu `DoubleDomain`.

T10 Ein `EInt` wird zu `IntegerDomain`.

T11 Ein `ELong` wird zu `LongDomain`.

T12 Ein `EShort` wird zu `IntegerDomain`.

T13 Ein `EString` wird zu `StringDomain`.

T14 Für `EJavaObject` wird die Transformation nicht unterstützt.

T15 Für `EJavaClass` wird die Transformation nicht unterstützt.

T16 Ein `EBooleanObject` wird zu `BooleanDomain`.

T17 Ein `EByteObject` wird zu `IntegerDomain`.

T18 Ein `ECharacterObject` wird zu `StringDomain`.

T19 Ein `EDoubleObject` wird zu `DoubleDomain`.

T20 Ein `EFloatObject` wird zu `FloatDomain`.

T21 Ein `EIntegerObject` wird zu `IntegerDomain`.

T22 Ein `ELongObject` wird zu `LongDomain`.

T23 Ein `EShortObject` wird zu `IntegerDomain`.

T24 Ein `EDate` wird zu `RecordDomain` mit 6 `IntegerDomains`.

T25 Für `EBigInteger` wird die Transformation nicht unterstützt.

T26 Für `EBigDecimal` wird die Transformation nicht unterstützt.

T27 Für `EResource` wird die Transformation nicht unterstützt.

T28 Für `EResourceSet` wird die Transformation nicht unterstützt.

T29 Für `EFeatureMap` wird die Transformation nicht unterstützt.

T30 Für `EFeatureMapEntry` wird die Transformation nicht unterstützt.

T31 Für `ETreeIterator` wird die Transformation nicht unterstützt.

T32 Für `EEList` wird die Transformation nicht unterstützt.

T33 Ein `EEnumerator` wird zu `EnumDomain`.

7 Entwurfsentscheidungen

Zwischen Ecore-Metamodellen und grUML-Schemas gibt es einige Unterschiede. Im Folgenden wird beschrieben, wie in dieser Arbeit mit den Unterschieden umgegangen wird.

Abschnitt 7.1 erläutert kurz das prinzipielle Vorgehen und erklärt allgemeine Konzepte. Welche Möglichkeiten der Benutzer besitzt, Einfluss auf die Transformation zu nehmen, wird in Abschnitt 7.2 beschrieben. Abschnitt 7.3 beschreibt, welche Kriterien `EClasses` erfüllen müssen, die als konzeptuelle Kantenklassen betrachtet werden sollen. Die Regeln, nach denen die Transformation im Default-Fall abläuft, werden in Abschnitt 7.4 aufgeführt.

7.1 Allgemeines Vorgehen

Für *uneindeutige Situationen*, die während der Transformation auftreten können, müssen Entscheidungen getroffen werden. In Unterkapitel 7.4 wird aufgeführt, wie diese Entscheidungen getroffen werden, falls keine zusätzlichen Informationen vorliegen. Das hat den Vorteil, dass die *Transformation automatisch und deterministisch* ablaufen kann.

Bevor jedoch diese Default-Regeln verwendet werden, muss überprüft werden ob keine zusätzlichen *Informationen*, die nach definierten Regeln dem Modell hinzugefügt werden können, vorhanden sind. Zusätzliche Informationen können in Kommentaren, Annotationen oder zusätzlichen Attributen enthalten sein. Näheres dazu ist in Unterkapitel 7.4 beschrieben.

Die *höchste Entscheidungskompetenz besitzt der Benutzer*. Bei uneindeutigen Transformationen hat er die Möglichkeit, die Transformation nach seinen Wünschen zu beeinflussen, indem er zusätzliche Informationen angibt. Welche Optionen er besitzt, beschreibt Unterkapitel 7.4.

Gehen bei einer gewählten Transformation Informationen verloren, werden für die Richtung grUML-Schemas nach Ecore-Metamodellen *Annotationen* genutzt, um die Information zu bewahren. Für die andere Richtung erfüllen *Kommentare* die gleiche Aufgabe.

7.2 Optionen des Benutzers

Dieser Abschnitt beschreibt, welche *Möglichkeiten* dem Benutzer gegeben werden, um die *Transformation zu beeinflussen*. Es wird beschrieben, wie die Optionen direkt über die Programmierschnittstelle gesetzt werden. Alle im Folgenden erklärten Methoden sind in der Klasse `Ecore2Tg` enthalten. Die Festlegung dieser Optionen in einer Konfigurationsdatei oder bei Nutzung des Kommandozeilen-Werkzeugs werden in 8.4 und 8.5 erklärt.

7.2.1 Benennung des Wurzelpaketes

Im Default-Fall wird der Name des Schemas aus dem `nsPrefix` des Wurzelpaketes des Metamodells generiert. Da Ecore aber die Möglichkeit bietet, auch mehrere Wurzelpakete zuzulassen, ist diese Lösung nicht eindeutig. Wenn sichergestellt sein soll, dass ein bestimmtes Paket als Wurzelpaket betrachtet wird, kann dies durch die folgende Methode ausgedrückt werden.

```
void setRootPackageOfMetamodel(String qualifiedEPackageName)
```

Der übergebene Parameter stellt den Namen desjenigen Paketes da, welches zum Wurzelpaket ernannt werden soll. Für die Erstellung der Graphklasse werden auch dessen `nsPrefix` und `nsURI` genutzt.

7.2.2 Deklaration der Graphklasse

Soll eine vorhandene `EClass` in die `GraphClass` anstatt in eine `VertexClass` transformiert werden, gibt der Benutzer dies durch die folgende Methode an.

```
setAsGraphClass(String qualifiedEClassName)
```

Handelt es sich bei dem übergebenen Parameter um den qualifizierten Namen einer im Ecore-Metamodell vorhandenen `EClass`, dann wird diese zur `GraphClass` transformiert. Das bedeutet, dass die `EAttributes` der `EClass` in Attribute der Graphklasse konvertiert werden. Die `EReferences` werden nicht transformiert, da die `GraphClass` keine ein- oder ausgehenden Kantenklassen besitzen darf. Ist der String dagegen kein gültiger qualifizierter Name, wird er als Name der für das transformierte Schema zusätzlich erstellten Graphklasse genutzt.

7.2.3 Deklaration von Kantenklassen

Soll das Programm automatisch nach konzeptuellen Kantenklassen suchen und diese als solche deklarieren, muss dies vor dem Start der Transformation angegeben werden. Es existieren hier die folgenden drei Möglichkeiten:

- Das Programm sucht nicht nach konzeptuellen Kantenklassen.
- Das Programm sucht nach konzeptuellen Kantenklassen und gibt die Ergebnisse auf der Konsole aus. Bei der anschließenden Transformation werden für die gefundenen `EClasses` aber keine `EdgeClasses` erstellt, die Resultate der Suche werden nicht berücksichtigt.
- Das Programm sucht nach konzeptuellen Kantenklassen und transformiert die gefundenen `EClasses` anschließend in `EdgeClasses`.

Welche `EClasses` gefunden werden, wird durch Kriterien bestimmt, die in 7.3.1 erklärt sind. Angegeben wird diese Option über die folgende Methode.

```
setTransformationOption(TransformParams t)
```

Als Parameter werden Literale der Enumeration `Ecore2Tg.TransformParams` verwendet. Die Literale sind `JUST_LIKE_ECORE`, `PRINT_PROPOSALS` und `AUTOMATIC_TRANSFORMATION`.

Will der Nutzer eine bestimmte `EClass` in eine Kantenklasse transformieren lassen, fügt er deren qualifizierten Namen als String der entsprechenden `ArrayList` hinzu. Die Methode, welche die Liste bereitstellt lautet folgendermaßen.

```
ArrayList<String> getEdgeClassesList()
```

Wird ein Name eingefügt, zu der keine `EClass` im Metamodell existiert, wird eine Warnung ausgegeben. Bei spezialisierten Klassen muss nur die allgemeinste `EClass` angegeben werden, ihre Subklassen sind damit automatisch auch als Kantenklassen deklariert.

7.2.4 Bestimmung der Richtung einer Kantenklasse

Um die Richtung einer Kantenklasse zu bestimmen, gibt es zwei Möglichkeiten. Diese werden im Folgenden beschrieben.

Für Kompositionen kann angeben werden, dass die daraus resultierenden Kantenklasse vom Ganzen zum Teil oder umgekehrt verläuft. Dazu wird folgende Methode bereit gestellt.

```
setAggregationInfluenceOnDirection(int i)
```

Als Parameter erwartet sie eine der Konstanten `Ecore2Tg.NO_DIRECTION_FROM_AGGREGATION`, `Ecore2Tg.DIRECTION_WHOLE_TO_PART`, und `Ecore2Tg.DIRECTION_PART_TO_WHOLE`

Die Richtung einzelner Kantenklassen kann gesetzt werden, indem der qualifizierte Name von einer der `EReferences`, die zur Bildung der `EdgeClass` verwendet werden, angegeben wird, zusammen mit einer der Konstanten `Ecore2Tg.FROM` und `Ecore2Tg.TO`. Dadurch kann spezifiziert werden, ob die Richtung der EReference zum Start- oder Endpunkt der Kantenklasse führt. Dieses Paar aus Name und Konstante wird dann der Map hinzugefügt, die durch folgende Methode zugänglich ist.

```
HashMap<String, Integer> getDirectionMap()
```

Wird ein anderer Integerwert, als die beiden Konstanten, übergeben, wird eine Warnung ausgegeben. Die Transformation wird weiter durchgeführt, das betroffene Paar aber nicht berücksichtigt.

7.2.5 Deklaration des Namen einer Kantenklasse

Soll eine Kantenklasse einen bestimmten Namen erhalten, kann dies ausgedrückt werden, indem ein Paar aus qualifiziertem Namen einer `ERefererence` und gewünschtem Namen für die Kantenklasse, die aus dieser `EReference` entsteht, angegeben wird. Die beiden Strings werden wieder in einer Map gespeichert, die durch folgende Methode zugänglich ist.

```
HashMap<String, String> getNamesOfEdgeClassesMap()
```

Handelt es sich bei dem Wunschnamen um einen qualifizierten Namen, wird dadurch auch direkt das Paket der `EdgeClass` festgelegt. Anderenfalls kann es auch getrennt angegeben werden, wie in 7.2.7 beschrieben.

7.2.6 Generierung von Rollennamen

Kantenklassen, die aus unidirektionalen `EReferences` entstehen, besitzen auf einer Seite keinen Rollennamen. Wenn die fehlenden Namen generiert werden sollen, muss diese Option gesetzt sein. Die Methode dazu ist:

`setGenerateRoleNames(boolean b)`

Der fehlende Rollennamen entspricht dann dem vorhandenen Rollennamen um ein “Opposite” ergänzt. Diese Option ist im Default-Fall deaktiviert.

7.2.7 Deklaration des Paketes einer Kantenklasse

`EReferences` besitzen kein Paket, da sie zur enthaltenden `EClass` gehören. Aus diesem Grund besteht die Möglichkeit, direkt anzugeben, welchem Paket eine Kantenklasse zugeordnet werden kann. Dazu werden Paare von Strings einer Map hinzugefügt, die über folgende Methode zugänglich ist.

`HashMap<String, String> getDefinedPackagesOfEdgeClassesMap()`

Jedes Paar besteht aus dem qualifizierten Namen einer `EReference` angegeben, die zur Enstehung der `EdgeClass` beiträgt und dem qualifizierten Namen des Paketes, welches die Kantenklasse enthalten soll.

7.2.8 Benennung der überschreibenden Referenzen

Wie in 5.4.4 beschrieben, kann es für manche Metamodelle sinnvoll sein, anzugeben, welche `EReferences` welche ererbten überschreiben. Daher bietet diese Option die Möglichkeit, ein Paar aus dem qualifizierten Namen der `EReference`, die überschreiben soll zusammen mit dem derjenigen, die überschrieben wird, anzugeben. Dieses Paar wird in die Map eingefügt, die durch folgende Methode zugänglich ist.

`HashMap<String, String> getPairsOfOverwritingEReferences()`

Dabei genügt es, die direkten Überschreibungen anzugeben. Wenn `a b` überschreibt und `b c`, dann reicht die Angabe `(a,b)` und `(b,c)`. Dass `a` auch `c` überschreibt wird transitiv abgeleitet.

7.3 Konzeptuelle Kantenklassen

Bei der Transformation von Ecore zu grUML können `EClasses`, die konzeptuell Beziehungen darstellen, zu Kantenklassen konvertiert werden. Der folgende Abschnitt erläutert, welche `EClasses` *als konzeptuelle Kantenklassen akzeptiert* werden. Der erste Teil wird sich mit den strengeren Kriterien für Kantenklasse beschäftigen, die während der automatischen Suche angewendet werden. Im zweiten Teil wird beschrieben, welche zusätzlichen `EClasses` als konzeptuelle Kantenklassen benannt werden dürfen.

7.3.1 Automatische Suche von Kantenklassen

Das wichtigste Kriterium für eine konzeptuelle Kantenklasse ist, dass die `EClass` *genau zwei andere* `EClasses` *miteinander verbinden* muss. Sie muss also mindestens zwei bis maximal drei `EReferences` besitzen, die alle die Multiplizität eins besitzen.

Abbildung 38 zeigt welche maximale Menge von eigenen und referenzierenden `EReferences` eine konzeptuelle Kantenklasse haben darf.

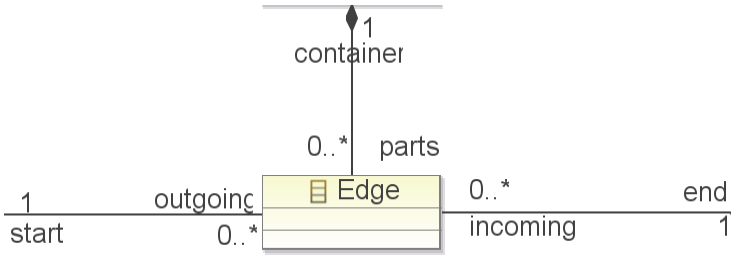

Abbildung 38: *Konzeptuelle Kantenklasse mit maximaler Anzahl an eigenen und referenzierenden Referenzen*

Die `EReferences start` und `end` sind nötig, um zu garantieren, dass genau zwei Klassen verbunden werden. Würde eine davon fehlen, wäre dies nicht sichergestellt. Die `EReferences outgoingEdge` und `incomingEdge` als Opposites von `start` und `end` spielen ebenfalls eine wichtige Rolle. Sie liefern die Inzidenzen für die resultierende Kantenklasse. Hier reicht es allerdings, wenn eine der `EReferences` vorhanden ist, entweder `outgoingEdge` oder `incomingEdge` kann weggelassen werden, ohne dass die `EClass` ihre Kanteneigenschaft verliert.

Die `EReferences start` und `incomingEdge` müssen entweder beide Kompositionen sein, oder keine davon. Das gleiche gilt für die `EReferences end` und `outgoingEdge`.

Die Rollen der beiden `ERefences container` und `parts` lassen sich mit Hilfe von Abbildung 39 erläutern.

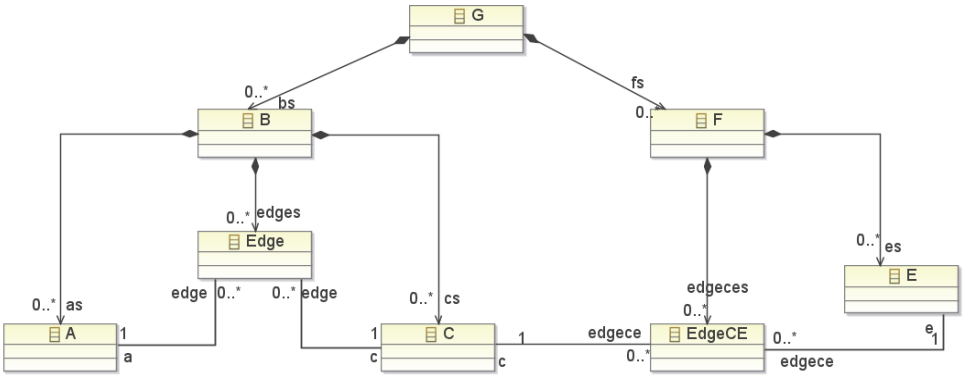

Abbildung 39: *Beispiel für konzeptuelle Kantenklassen mit zusätzlicher Containment-Referenz*

In diesem Beispiel sind `Edge` und `EdgeCE` konzeptuelle Kantenklassen. Um die *Navigierbarkeit im Metamodell* zu optimieren, wurden jedoch die `EClasses G, B` und `F` hinzugefügt. Die zusätzlichen Referenzen `edges` und `edgesces` sollen keinen Einfluss auf die Erkennung der Kantenklassen haben. Sie entsprechen der `parts-EReference` aus Abbildung 38. Falls `edges` oder `edgesces` Opposites hätten, würden diese die Rolle der `EReference container` aus Abbildung 38 übernehmen.

Ein weiteres Kriterium für eine Kantenklasse ist, dass *keine Sub- oder Superklasse die Kanteneigenschaft zerstört.* Bei Subklassen, die konzeptuelle Kantenklassen als Supertypen besitzen, darf auch `start` oder `end` aus Abbildung 38 fehlen, da die fehlende `EReference` dann

vom Supertyp übernommen werden kann. Dafür dürfen aber die `EReferences container` und `parts` nicht auftreten, da eine `Containment-EReference`, die der Navigierbarkeit dient, die oberste Klasse in der Vererbungshierarchie enthalten muss.

Zusätzlich dazu muss darauf geachtet werden, dass die *Start- und Endklassen von Super- und Subklassen kompatibel* sind. Abbildung 40 zeigt einen Fall, wo dieses Kriterium nicht erfüllt ist.

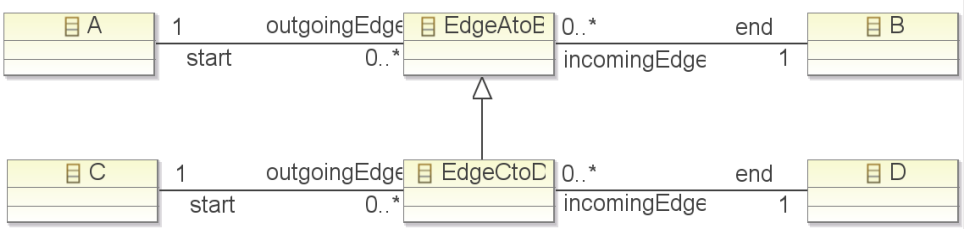

Abbildung 40: *Keine konzeptuellen Kantenklassen, trotz korrekter Referenzen*

Die `EClasses EdgeAtoB` und `EdgeCtoD` sind keine konzeptuellen Kantenklassen, obwohl die Menge der `EReferences` die Kriterien erfüllt. Subklassen von konzeptuellen Kantenklassen müssen grundsätzlich die gleichen Klassen verbinden, wie ihre Superklassen. Einzige Ausnahme ist, dass sie auch Spezialisierungen der Endpunkte ihrer Superklassen verbinden können.

Abschließend darf *keine Start- oder Endklasse* einer konzeptuellen Kantenklasse *selbst eine Kantenklasse* sein. In diesem Fall würde sonst die Transformation fehlschlagen, da eine `EdgeClass` nur `VertexClasses` verbinden kann und keine anderen `EdgeClasses`.

7.3.2 Benutzerdefinierte Kantenklassen

Der Benutzer kann zunächst alle konzeptuellen Kantenklassen als solche deklarieren, die auch von der automatischen Suche gefunden werden würden. Er kann jedoch auch einige *Kriterien aufheben*.

Wenn nur die `EReferences outgoingEdge` und `end` aus Abbildung 38 vorhanden sind, kann aus der `EClass` eine Kantenklasse generiert werden. In diesem Fall muss aber sichergestellt sein, dass in zugehörigen Modellen nicht mehrere `outgoingEdge`-Links auf ein `Edge`-Objekt zeigen, da sonst die Transformation fehlschlägt.

Außerdem kann der Benutzer `EClasses` als konzeptuelle Kantenklassen deklarieren, wenn `outgoingEdge` und `end` oder `incomingEdge` und `start` aus Abbildung 38 unterschiedlich in Bezug auf ihre `containment`-Eigenschaft sind.

7.4 Entscheidungstabelle

Der folgende Abschnitt erläutert, welche *Entscheidungen bei der automatischen Transformation* getroffen werden. Die Tabelle beschreibt, welche Regeln angewendet werden, falls keine zusätzlichen Informationen vorhanden sind.

Unterschied	**Behandlung**
In grUML gehören Kantenklassen eindeutig zu einem Paket, zusammengehörige Referenzen in Ecore haben keine solche Zuordnung.	Ecore -> grUML: Einzelne Referenzen gehören in das Paket der besitzenden Klasse, bei zusammengehörigen Referenzen wird die Kante in das Paket der alphabetisch ersten Klasse eingeordnet, sofern keine `EAnnotation` vorliegt, die etwas anderes definiert. grUML -> Ecore: Es wird eine `EAnnotation` genutzt, um zu vermerken in welchem Paket sich die Kantenklasse befunden hat, aus der nun beidseitige Referenzen geworden sind.
In grUML gibt es nichts Vergleichbares zu `nsPrefix` und `nsURI`.	Ecore -> grUML: Beide Informationen können als Attribute der Graphklasse gespeichert werden. Ihre Werte werden dann als Default-Werte eingetragen. grUML -> Ecore: Wenn die Graphklasse entsprechende Attribute besitzt, werden diese gewählt. Ansonsten muss beides wie folgt generiert werden: nsURI = http://GRAPHCLASSNAME.com; nsPrefix = GRAPHCLASSNAME. Zusätzlich erfolgt eine Meldung an den Benutzer, dass er dies ändern kann, wenn er das möchte.
Ecore besitzt nichts Vergleichbares zur grUML-Graphklasse.	Ecore -> grUML: Die Graphklasse wird zusätzlich erzeugt. Ist eine `EClass` als `GraphClass` annotiert wird diese entsprechend transformiert. grUML -> Ecore: Es wird ein zusätzliches Wurzelelement erzeugt, welches die Funktion der Graphklasse übernimmt. Dieses Wurzelelement wird mit einer Annotation versehen, dass es sich um eine ursprüngliche Graphklasse handelt.
Eine Ecore `EClass` kann als Interface definiert werden.	Ecore -> grUML: Die neu erzeugte grUML-Knotenklasse wird als `abstract` definiert. Zusätzlich wird ein grUML-Kommentar hinzugefügt, der `<<interface>>` lautet. grUML -> Ecore: Es wird geschaut, ob eine abstrakte grUML-Knotenklasse einen Kommentar `<<interface>>` besitzt. Ist das der Fall wird aus der Knotenklasse ein Interface in Ecore.
Einseitige Ecore-Referenzen haben keinen Rollennamen und keine Multiplizitäten am Anfang der Kante.	Ecore -> grUML: Die Multiplizität ist implizit *, bei Kompositionen 0..1. Der Rollenname wird weggelassen.

Unterschied	**Behandlung**
Zusammengehörige Ecore-Referenzen haben keine Kantenrichtung.	Ecore-> grUML: Die Kantenklasse beginnt bei der alphabetisch ersten Knotenklasse, sofern keine Annotation etwas anderes definiert. Bei Aggregationen und Kompositionen kann der Benutzer entscheiden, ob die Kante immer vom Ganzen zum Teil laufen soll oder umgedreht. grUML -> Ecore: Diejenige Referenz, die der Richtung entspricht, enthält eine entsprechende Annotation.
grUML-Kantenklassen können in einseitige oder beidseitige `EReferences` transformiert werden.	grUML -> Ecore: grUML-Kantenklassen werden in beidseitige `EReferences` transformiert.
Eine grUML-Kante braucht einen Namen, die Ecore-Referenz besitzt als Name nur den Rollennamen vom Ende der Kantenklasse.	Ecore -> grUML: Es muss ein Name generiert werden. JGraLab bietet dazu bereits Funktionalität.
In Ecore gibt es keine Aggregation.	grUML -> Ecore: Die Kante wird wie eine einfache Kante transformiert, zusätzlich wird eine `EAnnotation` eingeführt, die die Information der Aggregation bewahrt.
grUML-Kanten können Attribute besitzen.	grUML -> Ecore: Eine grUML-Kante mit Attributen wird in eine `EClass` mit den entsprechenden Attributen sowie passenden Referenzen transformiert. Um die Information, dass es sich um eine Kante handelt zu bewahren, wird ein Annotation eingeführt.
In grUML unterstützen Kantenklassen Generalisierung. Ecore-Referenzen kennen so etwas nicht.	grUML -> Ecore: Die Kanten werden in Klassen mit passenden Referenzen transformiert, so dass eine Vererbungshierarchie der Klassen möglich ist. Um die Information, dass es sich um eine Kante handelt zu bewahren, wird ein Annotation eingeführt.
In Ecore haben die Attribute der Klassen Multiplizitäten sowie unique und ordered Eigenschaften.	Ecore -> grUML: Die grUML-Attribute werden entsprechend als Liste (wenn ordered gesetzt ist) oder Menge realisiert. Die unique Eigenschaft wird nicht beachtet.
Ecore und grUML haben vollständig unterschiedliche Systeme für Annotationen, Constraints und Kommentare.	Ecore -> grUML: Ecore-Annotationen werden in grUML-Kommentare umgewandelt. grUML -> Ecore: grUML-Constraints und -Kommentare werden nicht transformiert.

7.5 Zusammenfassung

Die Transformation wird nach den in Abschnitt 7.4 erklärten Regeln durchgeführt. Dabei wird definiert, wie in uneindeutigen Situationen *Entscheidungen* getroffen werden. Durch die Optionen, die in Abschnitt 7.2 beschrieben wurden, kann *Einfluss auf die Transformation* genommen werden. Dies geschieht über entsprechende Getter- und Setter-Methoden. Darunter wird auch die Möglichkeit erläutert, spezielle `EClasses` als `EdgeClasses` zu definieren oder allgemein im Ecore-Metamodell nach *konzeptuellen Kantenklassen suchen* zu lassen. Abschnitt 7.3 definiert, welche Kriterien diejenigen `EClasses` erfüllen müssen, die als konzeptuelle Kantenklassen betrachtet werden sollen.

8 Implementierung

Im folgenden Kapitel werden die *wichtigsten Aspekte der Implementierung* kurz erläutert. In Abschnitt 8.1 wird grob beschrieben, wie das Programm arbeitet. Es wird ein *Überblick* gegeben und sowohl die *Metamodell- als auch die Modelltransformation erläutert*. Im Anschluss wird in Abschnitt 8.2 detailliert beschrieben, wie das Verfahren zur *automatischen Kantenklassensuche* abläuft. Abschnitt 8.3 erklärt, wie für eine `EClass`, die eine konzeptuelle Kantenklasse darstellt, diejenigen `EReferences` gefunden werden, die zur Erstellung der `EdgeClass` nötig sind. Das Programm bietet die Möglichkeit, die Optionen in einer *Konfigurationsdatei* zu speichern. Wie diese aufgebaut ist, wird in Abschnitt 8.4 dargestellt. Auf welche Weise die Optionen beim Start des Programms über die *Kommandozeile* gesetzt werden, beschreibt Abschnitt 8.5.

8.1 Ablauf

Dieser Abschnitt beschäftigt sich mit dem *Ablauf der Transformation*. Es wird erläutert, wie das Programm aufgebaut ist und wie die Transformation von Metamodellen in Schemas und von Modellen in Graphen durchgeführt wird.

8.1.1 Überblick

Grob betrachtet besteht das Programm aus 2 Teilen, der Metamodelltransformation und der Modelltransformation. Das Aktivitätsdiagramm aus Abbildung 41 zeigt den Ablauf.

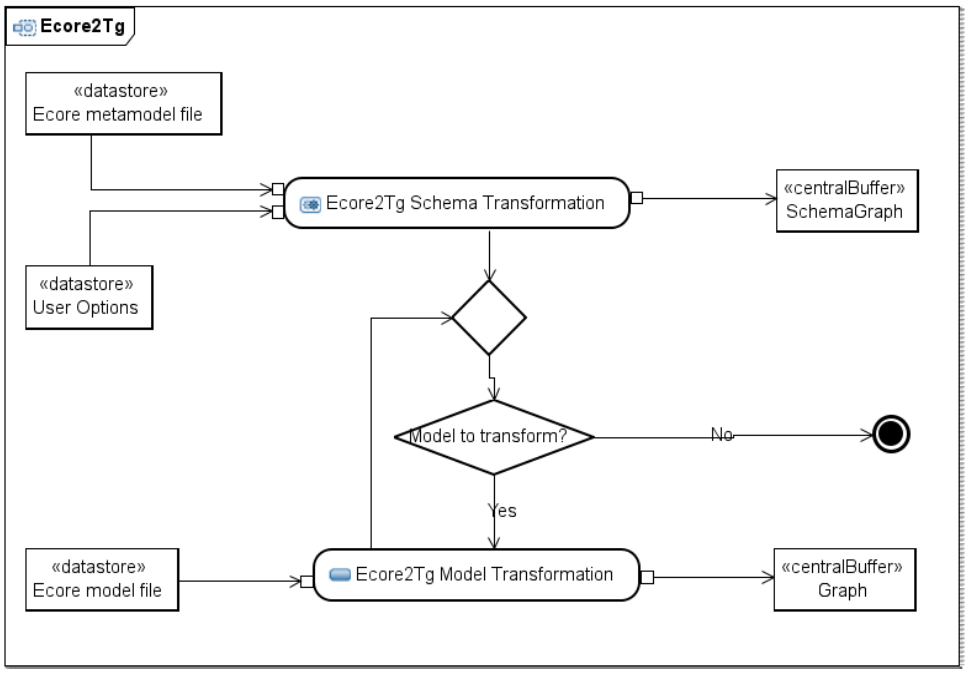

Abbildung 41: *Die Transformation als Programm*

Für den Start der Transformation wird die Ecore-Datei des Metamodells, sowie optional zusätzliche Angaben, benötigt. Die Optionen können entweder direkt über die Programmierschnittstelle gesetzt, in einer Konfigurationsdatei mitgegeben (Abschnitt 8.4) oder über die Kommandozeile eingegeben werden (Abschnitt 8.5).

Anschließend folgt die Transformation des Metamodells, die in 8.1.2 näher erläutert wird. Das Ergebnis ist dann der resultierende SchemaGraph.

Nachdem das Metamodell transformiert ist, können die Modelle folgen. Diese werden als XMI-Dateien der Transformation übergeben. Das Ergebnis ist jeweils ein Graph-Objekt. Zu einem Metamodell können beliebig viele Modelle transformiert werden.

8.1.2 Metamodell -> SchemaGraph

Abbildung 42 zeigt die Transformation eines Metamodells in einen SchemaGraphen.

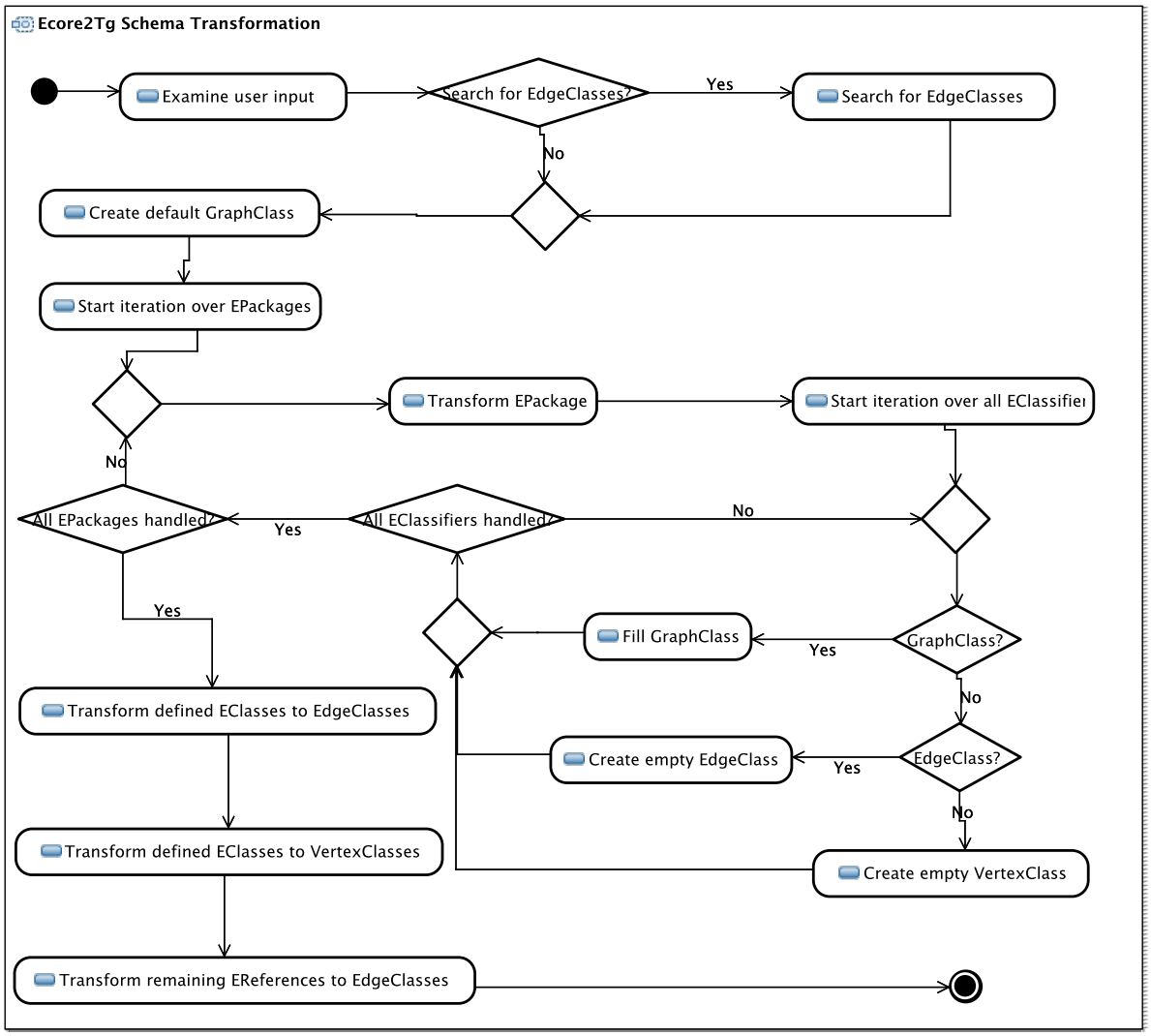

Abbildung 42: *Grober Ablauf der Transformation eines Ecore-Metamodells in ein grUML-Schema*

Zunächst werden die *Angaben des Benutzers untersucht*. Hier wird überprüft, ob die für die jeweiligen Optionen durch den qualifizierten Namen angegebenen `EPackages`, `EClasses` und `EReferences` existieren. Wenn ja, werden die entsprechenden Objekte im Sinne der angegebenen Option in einer Map oder Liste zur weiteren Verwendung während der Transformation gespeichert. Falls nicht, wird auf der Konsole eine entsprechende Fehlermeldung ausgegeben.

Falls der Benutzer das Programm automatisch nach `EdgeClasses` suchen lassen möchte, folgt dies als nächster Schritt. Wie diese Suche abläuft, wird in Unterkapitel 8.2 näher erklärt.

Anschließend beginnt die eigentliche Transformation. Zuerst wird eine `GraphClass` erstellt, die als Namen per default das `nsPrefix` des Metamodells um den String “Graph” erweitert erhält.

Ecore-Metamodelle sind in einer Paketstruktur organisiert, es muss mindestens ein “root package” geben. Aus diesem Grund wird zu Beginn der Transformation über alle Pakete iteriert. Für jedes `EPackage` wird ein grUML-`Package` erstellt. Die Namen werden übernommen. In jedem `EPackage` befinden sich `EClassifier`, über die ebenfalls iteriert wird.

Für jeden EClassifier wird untersucht, ob es sich um einen `EDataType` oder eine `EClass` handelt. `EDataTypes` werden nicht transformiert, `EClasses` müssen auf Benutzer-Definitionen überprüft werden. Ist eine `EClass` als `GraphClass` deklariert, wird der Name der per default erstellten `GraphClass` verändert. Zusätzlich werden mögliche Attribute der `EClass` zur Graphklasse hinzugefügt. Ist eine `EClass` als `EdgeClass` deklariert, wird eine leere `EdgeClass` mit dem Namen der `EClass` erstellt. Leer bedeutet in diesem Fall, dass die Kantenklasse noch keine Attribute, Superklassen und Inzidenzen besitzt. Das `EClass`-Objekt wird nun als Key, das `EdgeClass`-Objekt als Value in einer Map gespeichert. Ist nichts davon der Fall, wird eine leere `VertexClass` erstellt, die den Namen der `EClass` trägt. Hier wird ebenfalls das `EClass`-Objekt als Key und das `VertexClass`-Objekt als Value in eine Map eingefügt.

Nachdem über alle `EPackages` und `EClassifier` iteriert wurde, sind alle `VertexClasses` vorhanden und es kann mit der Transformation der Kantenklassen begonnen werden.

Als erstes werden alle als `EdgeClass` deklarierten `EClasses` betrachtet. Für die bisher leeren `EdgeClasses` werden Attribute und Supertyp-Beziehungen erstellt. Nachdem die für die Inzidenzen relevanten `EReferences` gefunden wurden, was in Unterkapitel 8.3 genauer erklärt wird, können die Inzidenzen erstellt, gefüllt und den entsprechenden Start- und End-`VertexClasses` zugeordnet werden.

Im Anschluss werden die `VertexClasses` mit Attributen gefüllt, sowie die Supertyp-Beziehungen entsprechend erstellt.

Zuletzt werden diejenigen `EReferences` betrachtet, welche noch nicht bei der Erstellung der `EdgeClasses` aus `EClasses` Verwendung gefunden haben. Sie werden dann ebenfalls in `EdgeClasses` transformiert. Es wird zwischen uni- und bidirektionalen `EReferences` unterschieden. Die Erstellung und Füllung der Inzidenzen ist hier einfacher, als bei den aus `EClasses` erstellten `EdgeClasses`, da Multiplizität und Rollenname einfach direkt aus den `EReferences` übernommen werden können.

Damit ist die Transformation des Metamodells abgeschlossen. Damit die Modelltransformationen folgen können, wird der SchemaGraph intern in ein Schema umgewandelt und dieses kompiliert. Die Methode, welche die Transformation durchführt lautet `Ecore2Tg.transform()`.

8.1.3 Modell -> Graph

Die Transformation der Modelle in Graphen erfolgt analog zu der des Metamodells in den SchemaGraph. Mit der Methode `Graph Ecore2Tg.transformModel(String[] paths)` wird eine Modelltransformation gestartet. Als Parameter werden Pfade der Dateien übergeben, die zu einem Modell gehören. Im Folgenden wird beschrieben, wie die Transformation abläuft.

Zuerst wird ein neuer Graph zum Schema erstellt. Dann wird über das gesamte Modell iteriert und für jedes `EObject` entweder ein `Vertex` beziehungsweise eine `Edge` erstellt, oder, falls das `EObject` das der als `GraphClass` deklarierten `EClass` ist, der Graph mit Attribut-Werten gefüllt.

Im Anschluss werden Start- und Endpunkte der erstellten Kanten gesucht und hinzugefügt. Da es sich hier um die `Edges` handelt, die Instanzen der `EdgeClasses` sind, welche aus `EClasses` entstanden sind, können sie auch Attribute haben. Ist dies der Fall werden sie mit den entsprechenden Attributwerten gefüllt.

Danach werden den Attributen der Knoten die entsprechenden Werte zugewiesen. Dabei muss darauf geachtet werden, dass Werte von Attributen, die bei der Schematransformation ausgeschlossen wurden, nun ebenfalls nicht berücksichtigt werden.

Zum Schluss werden für die noch nicht behandelten Links zwischen den `EObjects` ebenfalls entsprechende Kanten erstellt. Nicht berücksichtigt werden an dieser Stelle Links, die zu demjenigen `EObject` führen, welches zum Graph ernannt wurde.

8.2 Automatische Suche nach Kantenklassen

Der folgende Abschnitt beschreibt das Verfahren, mit dem das Programm automatisch nach Kantenklassen sucht.

Es wird zunächst nur nach `EClasses` gesucht, die keine Superklassen besitzen. In Abschnitt 8.2.1 wird beschrieben, welche Kriterien überprüft werden. Die Subklassen der resultierenden Kandidaten werden dann in einem weiteren Schritt betrachtet. Die Abschnitte 8.2.2 und 8.2.3 erläutern die Kriterien, mit denen untersucht wird, ob diese die Kanteneigenschaft verletzen. Abschnitt 8.2.4 erläutert im Anschluss die abschließenden Kontrollen, die nötig sind, um eine `EClass` endgültig als konzeptuelle Kantenklasse zu akzeptieren.

8.2.1 Überprüfung der EReferences der Supertypen

In diesem Schritt wird eine Kandidatenmenge mit `EClasses` erstellt, die potentielle `EdgeClasses` sind. Abbildung 43 zeigt, welche Konstrukte erlaubt sind.

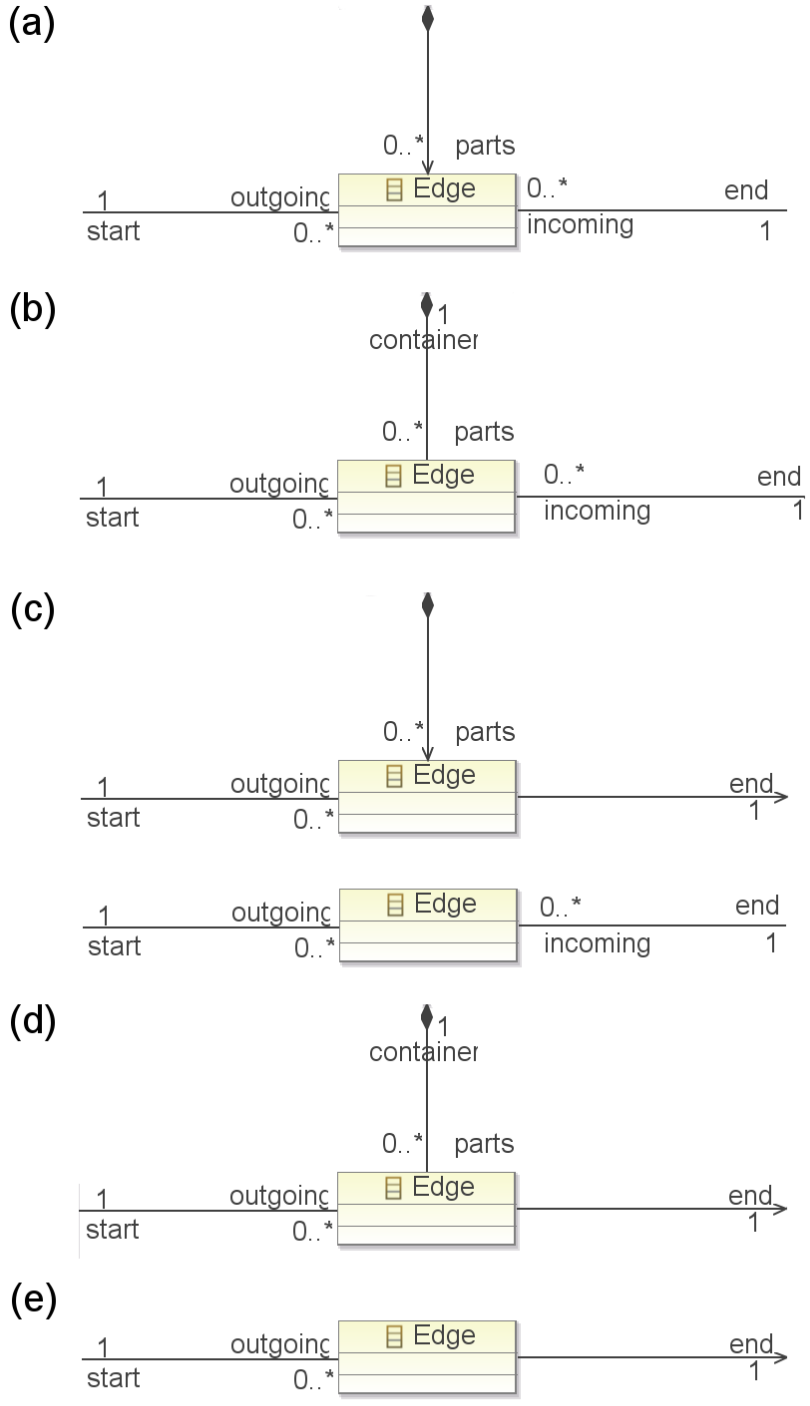

Abbildung 43: *Gültige auftretende Fälle bei der Überprüfung von ein- und ausgehenden EReferences der Kantenklassen-Kandidaten - Betrachtung der Superklassen*

Prinzipiell muss eine Kantenklasse mindestens zwei Verbindungen haben. Eine dritte kann existieren, wenn diese nur eine Enthalten-Sein-Beziehung ausdrückt. Abbildung 43 zeigt unter b), welche `EReferences` maximal vorhanden sein dürfen. Bei allen anderen Fällen fehlen `EReferences`, die verbleibenden genügen jedoch, um die `EClass` in der Kandidatenmenge zu behalten.

Aus der Menge aller EClasses wird eine Kandidatenmenge nach folgenden Kriterien herausgefiltert:

1. Die EClass darf keine Superklassen besitzen.
2. Die EClass darf nur 2 `EReferences` besitzen, die beide die Multiplizität von 1 haben müssen.
   Alternativ darf sie auch 3 `EReferences` besitzen, wenn eine davon eine `Container-EReference` ist und für die anderen beiden gilt, dass die Multiplizität 1 ist und das `containment`-Attribut nicht gesetzt ist.

Es werden diejenigen `EReferences` untersucht, welche die `EClasses` aus der Kandidatenmenge referenzieren. Alle Kandidaten, die nicht einen der folgenden Fälle erfüllen, werden entfernt:

1. Die `EClass` wird durch 3 `EReferences` referenziert und besitzt selbst 2 EReferences. Die beiden eigenen `EReferences` besitzen `EOpposites`, die 2 der 3 referenzierenden `EReferences` entsprechen. Die dritte `EReference` ist eine `Containment-EReference`. Für die beiden eigenen `EReferences` gilt, dass die `containment-` und `container`-Attribute nicht gesetzt sind. Dieser Fall entspricht (a) aus Abbildung 43.
2. Die `EClass` wird durch 3 `EReferences` referenziert und besitzt selbst 3 EReferences. Alle 3 eigenen `EReferences` besitzen `EOpposites`, die der Menge der referenzierenden `EReferences` entsprechen. Genau eine referenzierende `EReference` ist eine `Containment-EReference`. Dieser Fall entspricht (b) aus Abbildung 43.
3. Die `EClass` wird durch 2 `EReferences` referenziert und besitzt selbst 2 `EReferences`. Hier gibt es zwei Fälle.
   Im ersten Fall besitzen die beiden eigenen `EReferences` `EOpposites`, welche den referenzierenden `EReferences` entsprechen. Das `containment`-Attribut einer referenzierenden `EReference` muss gleich dem `Containment-` Attribut der `EOpposite` der anderen referenzierenden `EReference` sein.
   Im zweiten Fall gilt nur für ein Paar aus eigenen und referenzierenden `EReferences`, dass sie jeweils `EOpposites` sind. Die verbliebene referenzierende `EReference` hat das `containment`-Attribut gesetzt. Für die beiden eigenen `EReferences` gilt, dass keines der beiden Attribute `containment` und `container` gesetzt sein darf. Die Fälle entsprechen (c) aus Abbildung 43.
4. Die `EClass` wird durch 2 `EReferences` referenziert und besitzt selbst 3 `EReferences`. Die zwei referenzierenden `EReferences` müssen `EOpposites` besitzen, die den eigenen `EReferences` entsprechen. Dieser Fall entspricht (d) aus Abbildung 43.

5. Die `EClass` wird durch 1 `EReference` referenziert und besitzt selbst 2 `EReferences`. Die referenzierende `EReference` muss hier ein `EOpposite` besitzen, welches einer der beiden eigenen `EReferences` entspricht. Außerdem muss ihr `containment`-Attribut den gleichen Wert besitzen, wie das der eigenen `EReference`, die nicht ihr `EOpposite` ist. Dieser Fall entspricht (e) aus Abbildung 43.

Die verbleibenden Kandidaten erfüllen nun selbst die Kriterien für konzeptuelle Kantenklassen. Sie können jedoch nur in solche konvertiert werden, wenn keine ihrer Subklassen die Kanteneigenschaft zerstört. In den folgenden Unterabschnitten 8.2.2 und 8.2.3 wird erläutert, welche Kriterien speziell für Subklassen existieren und wie sie überprüft werden.

8.2.2 Überprüfung der EReferences der Subklassen

Um überprüfen zu können, ob Subklassen von potentiellen konzeptuellen Kantenklassen ebenfalls die Kriterien erfüllen, werden sie zunächst gesucht und in einer Menge der Kindelemente gespeichert. Erfüllt eine Subklasse ein Kriterium nicht, werden auch ihre Superklassen aus der Kandidatenmenge entfernt.

Dieser Unterabschnitt beschäftigt sich damit, alle Elemente auf ihre Form, in Bezug auf ein- und ausgehenden `EReferences`, zu überprüfen. Die möglichen auftretenden Fälle unterscheiden sich hier von denen der Superklassen, da die dritte Enthalten-Sein-Verbindung wegfällt, dafür aber nur eine der beiden anderen Verbindungen überschrieben werden kann. Abbildung 44 zeigt die Fälle, die nun akzeptiert werden.

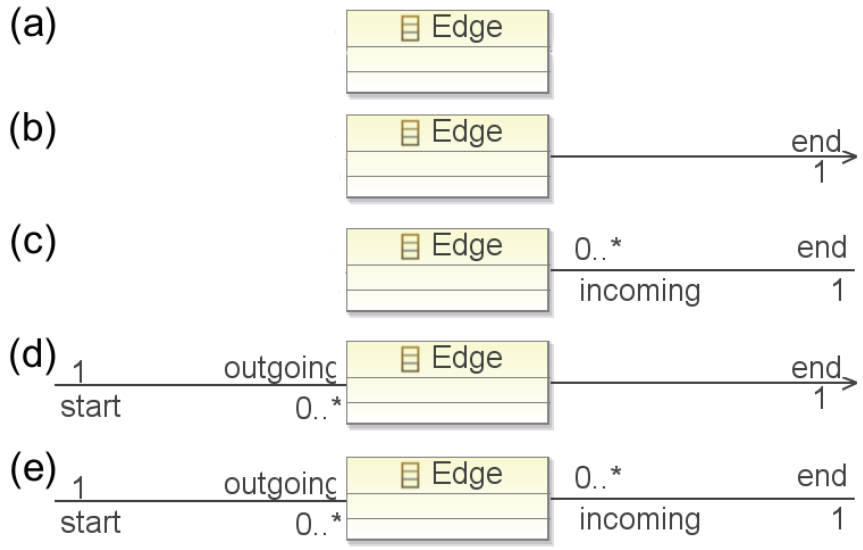

Abbildung 44: *Gültige auftretende Fälle bei der Überprüfung von ein- und ausgehenden EReferences der Kantenklassen-Kandidaten - Betrachtung der Subklassen*

Als erstes werden alle Elemente der Menge der Kindelemente bezüglich ihrer eigenen Referenzen untersucht. Gilt für ein Element, dass die Anzahl ihrer `EReferences` größer als 2 ist oder die Multiplizität einer `EReference` ungleich 1, dann wird sie mit allen Superklassen aus der

Menge der Kindelemente entfernt. Ihre höchsten Superklassen werden zusätzlich auch aus der Kandidatenmenge entfernt.

Anschließend werden alle Elemente der Menge der Kindelemente auf die `EReferences` hin untersucht, die sie referenzieren. Alle Kinder, die nicht einen der folgenden Fälle erfüllen, werden zusammen mit ihren Superklassen aus der Menge der Kindelemente entfernt. Ihre höchsten Superklassen werden zusätzlich aus der Kandidatenmenge entfernt.

1. Die `EClass` besitzt weder eigene noch referenzierende `EReferences`. Dieser Fall entspricht (a) aus Abbildung 44.
2. Die `EClass` besitzt eine eigene und keine referenzierende `EReference`. Dieser Fall entspricht (b) aus Abbildung 44.
3. Die `EClass` besitzt eine eigene und eine referenzierende `EReference`, die sich gegenseitig als `EOpposite` besitzen. Dieser Fall entspricht (c) aus Abbildung 44.
4. Die `EClass` besitzt zwei eigene und eine referenzierende `EReference`. Die referenzierende `EReference` hat ein `EOpposite`, welches einer der beiden eigenen `EReferences` entspricht. Dieser Fall entspricht (d) aus Abbildung 44.
5. Die `EClass` besitzt zwei eigene und zwei referenzierende `EReferences`. Beide referenzierenden `EReferences` besitzen `EOpposites`, welche den beiden eigenen `EReferences` entsprechen. Dieser Fall entspricht (e) aus Abbildung 44.

Diejenigen Subklassen, welche den Fällen entsprechen, bleiben vorerst in der Menge der Kindelemente enthalten. Sie werden nun auf weitere Kriterien überprüft.

8.2.3 Kompatibilität der Elternklassen

Nachdem die Subklassen der Kandidaten für konzeptuelle Kantenklassen nun auf ihre ein- und ausgehenden `EReferences` hin untersucht wurden, wird in diesem Unterabschnitt ein weiteres Kriterium untersucht. Alle verbliebenen Elemente der Menge der Kindelemente werden auf Kompatibilität und Eindeutigkeit der Endpunkte zu den Endpunkten der Superklassen überprüft.

Während der Prüfung wird über alle Superklassen iteriert und jeweils die beiden `EClasses`, welche die Endpunkte darstellen, gespeichert. Nun wird untersucht, ob ein Endpunkt des Elementes gleich einem Endpunkt der Superklasse, oder einer Subklasse des Eltern-Endpunktes ist. Wenn dies der Fall ist, wird noch festgestellt, ob die `containment`-Attribute der `EReferences`, die zu dem Element gehören (Damit sind die maximal 4 `EReferences` gemeint, aus denen die Inzidenzen der Kantenklasse abgeleitet werden würden), gleich den `containment`-Attributen der `EReferences`, die zu dem Supertyp gehören, sind. Sind diese Kriterien erfüllt, ist die Kompatibilität gewährleistet. Abbildung 45 zeigt unter (a) einen solchen Fall.

Falls jedoch nicht nur einer, sondern beide Endpunkte des zu untersuchenden Elements kompatibel zu beiden Endpunkten der Superklasse ist, kann nicht mit Sicherheit gesagt werden, welche der `EReferences` des Elements welche `EReferences` der Superklasse überschreibt.

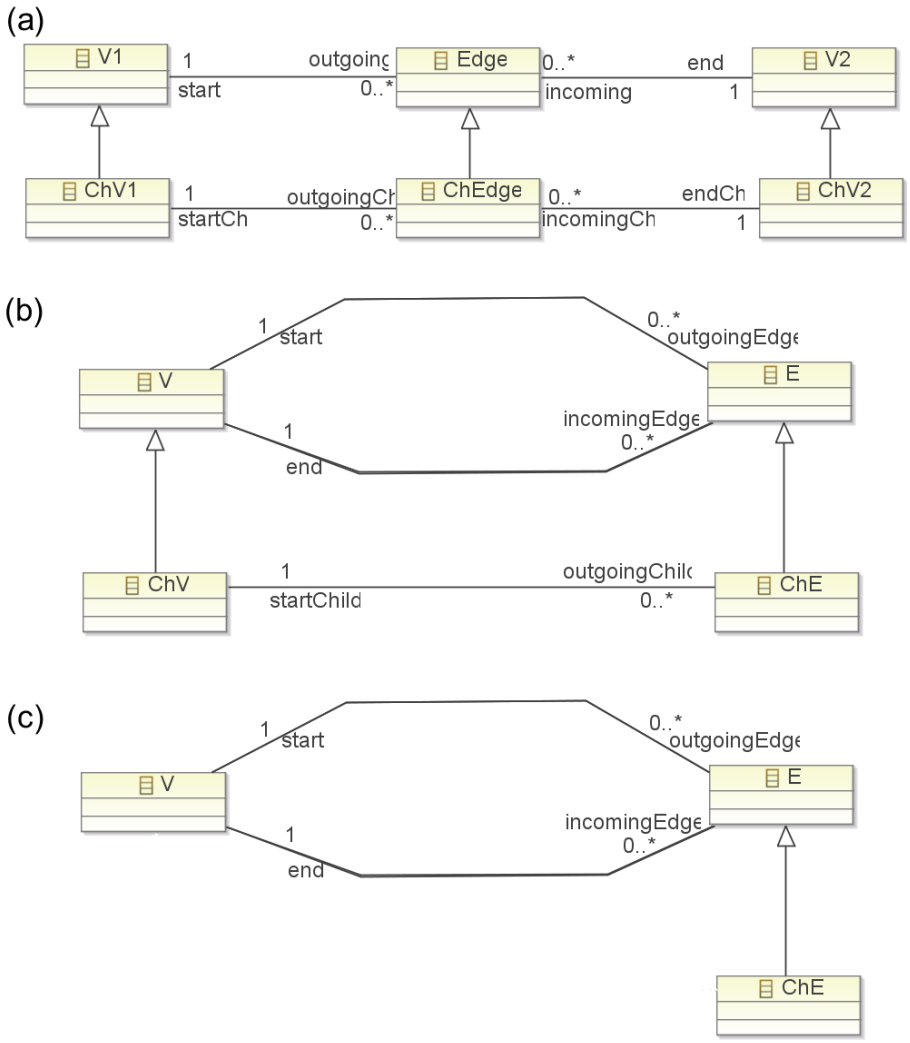

Abbildung 45: *Betrachtung der Kompatibilität von Endpunkten*

Aus diesem Grund werden beidseitig kompatible Elemente nur dann in der Menge der Kindelemente belassen, wenn die zugehörigen `EReferences` namensgleich mit denen der Eltern sind. Abbildung 45 zeigt unter (b) und (c) diese beiden Fälle. In Fall (c) sind die `EReferences` der Superklasse gleich denen der Subklasse. Im Fall (b) ist nicht klar, welche `EReferences` durch `startChild` und `outgoingChild` überschrieben werden. Wären in Fall (b) jedoch die Namen der `EReferences` `startChild` und `outgoingChild` gleich einem Paar der Eltern-Klassen, wäre klar welche `EReferences` überschrieben wurden und die Richtung könnte erkannt werden.

8.2.4 Abschließende Kontrolle

Nachdem die Subklassen der Kandidaten für konzeptuelle Kantenklassen untersucht wurden, ob sie ebenfalls die nötigen Kriterien erfüllen, wird jetzt die Menge der Subklassen zur Kandidatenmenge hinzugefügt. Im Anschluss daran werden noch einige abschließende Kriterien überprüft, die in diesem Unterabschnitt beschrieben werden.

Es wird über die komplette Kandidatenmenge iteriert und untersucht, ob für jedes Element noch alle Superklassen vorhanden sind. Unvollständige Superklassen-Hierarchien werden dann aus der Menge gelöscht. Es ist nicht nötig zu überprüfen, ob noch alle Subklassen da sind, da zuvor bei der Entfernung einer `EClass` aus der Kandidatenmenge immer alle Superklassen mit entfernt wurden.

Dann wird überprüft, ob die Endpunkte der Kandidaten ebenfalls in der Kandidatenmenge sind oder ob die Endpunkte von außen als Kantenklassen deklariert wurden. In diesem Fall wird der entsprechende Kandidat ebenfalls mit allen seinen Supertypen aus der Kandidatenmenge entfernt.

Zum Schluss wird noch einmal über die komplette Kandidatenmenge iteriert und überprüft, ob für jedes Element noch alle Superklassen vorhanden sind. Dies ist nötig, da durch den vorhergehenden Schritt möglicherweise eine Superklasse entfernt wurde. Unvollständige Superklassen-Hierarchien werden dann aus der Menge gelöscht.

Die verbleibenden Kandidaten bilden dann das Ergebnis der automatischen Suche nach Kantenklassen. Je nach Einstellung werden sie nun einfach ausgegeben oder zu `EdgeClasses` transformiert.

8.3 Ermittlung der EReferences zu einer Kantenklasse

Um aus einer `EClass` eine `EdgeClass` zu generieren, müssen zunächst die `EReferences` gefunden werden, welche Multiplizitäten, Aggregationstypen und Rollennamen für die zu erstellenden Inzidenzen liefern. Der Schwierigkeitsgrad ist hier abhängig davon, wie die Supertyp-Hierarchie der Kantenklasse aussieht. Das Verfahren unterscheidet daher zwischen `EClasses` mit und ohne Supertypen.

8.3.1 EClasses ohne Supertypen

Dieser Unterabschnitt erläutert, wie für `EClasses` ohne Supertypen die zur Erstellung einer `EdgeClass` nötigen `EReferences` ermittelt werden. Dabei werden verschiedene Fälle unterschieden. Es muss beachtet werden, dass es `Containment`-EReferences geben kann, die nicht zur Kantenklasse gehören sollen, da sie nur existieren um Navigierbarkeit zu gewährleisten. Abbildung 46 zeigt, welche Fälle bei `EClasses`, die Kantenklassen sind, auftreten können. Einige der Varianten gleichen denen, die im Unterkapitel 8.2 für Supertypen beschrieben wurden. Die zusätzlichen Fälle entstehen dadurch, dass der Nutzer auch `EClasses` als Kantenklassen definieren kann, die der automatische Such-Algorithmus nicht findet. In Abbildung 46 ist zum Beispiel 3(a) ein solcher Fall.

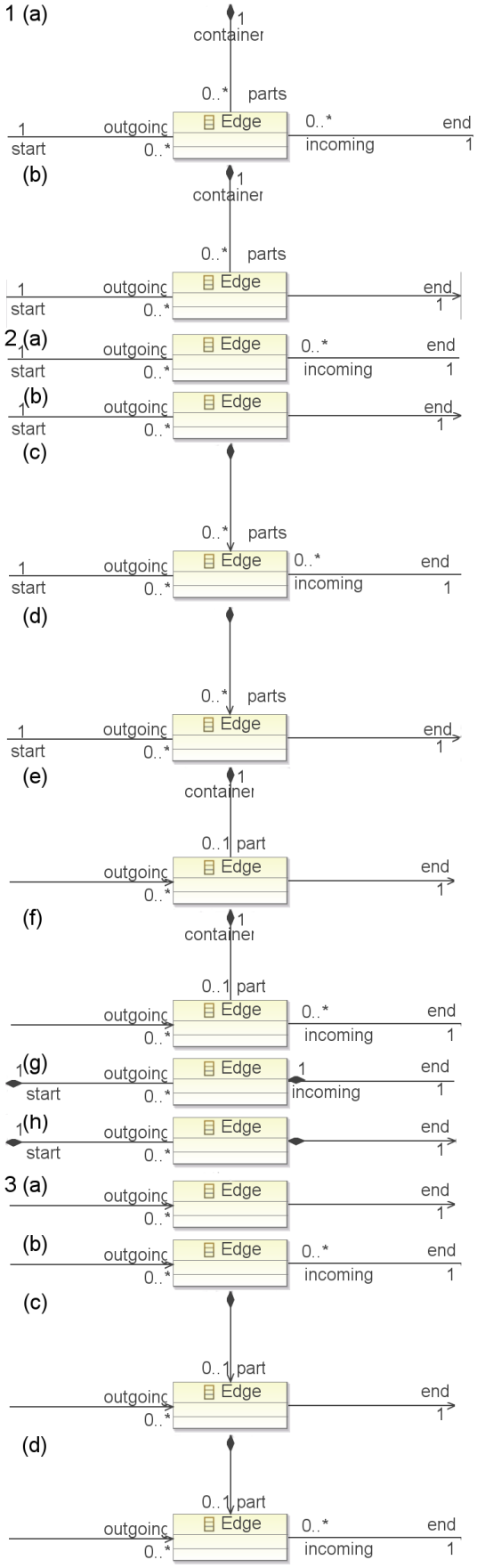

Abbildung 46: *Fälle von Kantenklassen ohne Superklassen*

1. Die `EClass` besitzt 3 eigene `EReferences`: In diesem Fall muss nur ermittelt werden, welche der `EReferences` das `container`-Attribut gesetzt hat. Die anderen beiden `EReferences` ergeben mit ihren `EOpposites`, falls sie existieren, die gesuchten `EReferences`. Abbildung 46 zeigt unter 1 die beiden Fällen, in denen 3 eigene Referenzen auftreten können. Sie unterscheiden sich darin, ob es drei (a) oder zwei (b) Referenzen gibt, welche die `EClass` referenzieren.

2. Die `EClass` besitzt 2 eigene `EReferences`:

   (a) Wenn beide `EReferences` kein `container`-Attribut gesetzt haben, ähnelt dieser Fall dem vorherigen. Die beiden eigenen `EReferences` mit ihren `EOpposites` können, falls vorhanden, als Ergebnis betrachtet werden. Abbildung 46 zeigt in Teil 2 (a), (b), (c) und (d) die Fälle, in denen das auftreten kann. Die Containment-`EReference part` aus (c) und (d) muss hier nicht beachtet werden, da mit Hilfe der eigenen beiden Referenzen `start` und `end` schon das gesamte Ergebnis bestimmt werden kann, ohne die Menge mit den 2 beziehungsweise 3 referenzierenden `EReferences` betrachten zu müssen.

   (b) Hat eine der beiden `EReferences` das `container`-Attribut gesetzt, wird überprüft, ob es noch eine referenzierende `EReference` gibt, welche kein `EOpposite` der eigenen ist. Wenn das so ist, dann wird die zusätzlich referenzierende `EReference` sowie die eigene `EReference`, die kein `Container` ist, mit ihrer `EOpposite` als Ergebnis zurück gegeben. Die Grafiken 2 (e) und (f) in Abbildung 46 sind die möglichen Fälle.

   Wenn das nicht so ist, sind beide eigenen `EReferences` mit ihren `EOpposites` das Ergebnis. Die Grafiken 2 (g) und (h) der Abbildung 46 zeigen dies.

3. Die `EClass` besitzt 1 eigene `EReference`:

   (a) Wenn die `EClass` 1 referenzierende `EReference` besitzt, bildet die eigene zusammen mit dieser referenzierenden `EReference` das Ergebnis. Abbildung 46 zeigt unter 3(a) einen solchen Fall.

   (b) Wenn die `EClass` 2 referenzierende `EReferences` besitzt, gibt es zwei Varianten. Falls diese `EReference` ein `EOpposite` besitzt, dann sind die 3 zu untersuchenden `EReferences` das Ergebnis. Abbildung 46 zeigt unter 3(b) diesen Fall. Falls dies nicht der Fall ist, werden beide referenzierenden `EReferences` bezüglich des `containment`-Attributs untersucht. Diejenige EReference, für die es gesetzt ist, wird aus der Menge herausgenommen, die verbleibenden beiden `EReferences` bilden das Ergebnis. Dieser Fall wird in Abbildung 46 Teil 3(c) gezeigt.

   (c) Wenn die `EClass` 3 referenzierende `EReferences` besitzt, muss eine davon eine `Containment - EReference` sein. Diese wird aus der Menge herausgenommen, die verbleibenden bilden das Ergebnis. Abbildung 46 zeigt in Teil 3(d) diesen Fall.

Die erkannten `EReferences`, welche für die entstehende `EdgeClass` nicht von Bedeutung sind, werden in einer Menge gespeichert, damit dies für die Subklassen und später für die Modelltransformation bekannt ist.

8.3.2 EClasses mit Supertypen

Dieser Unterabschnitt erläutert die Suche nach `EReferences`, die zur Erstellung einer Kantenklasse aus einer `EClass` nötig sind, für Klassen mit Supertypen. In diesem Fall muss nicht auf zusätzliche Kompositionen geachtet werden, die nicht zur Kantenklasse gehören und nur der Navigierbarkeit dienen. Dafür können ererbte `EReferences` genutzt werden.

Bei `EClasses` mit Supertypen muss darauf geachtet werden, dass die gesuchten `EReferences` möglicherweise von einer Superklasse ererbt werden beziehungsweise eine Superklasse referenzieren. Es besteht auch die Möglichkeit, dass eine Unterklasse nur eine der beiden Endpunktverbindungen der Superklasse überschreibt. Besonders beachtet werden muss, dass die Endpunkte einer potentiellen Kantenklasse immer die spezialisiertesten Endpunkte aller Superklassen werden müssen, damit keine Probleme bei der Vererbung der Kantenklassen auftreten. Für Abbildung 47 gilt also, dass die spezialisierte Kantenklasse `Edge` die Endpunkte `SubA` und `SubB` verbinden muss. Um eine korrekte Ermittlung der gesuchten `EReferences` zu ermöglichen, muss also zunächst festgestellt werden, welche Endpunkte eine Kantenklasse verbinden soll.

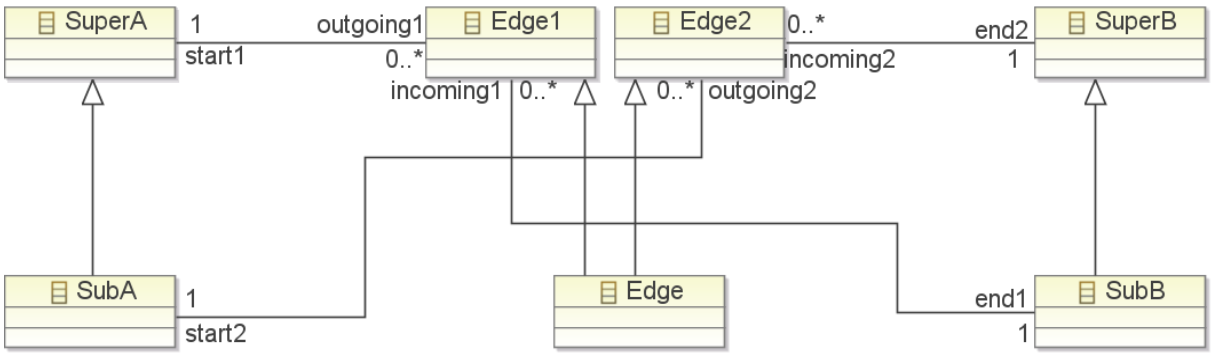

Abbildung 47: *Beispiel für eine Kantenklasse bei der Referenzen von zwei verschiedenen Elternklassen genutzt werden*

Das Verfahren zur Ermittlung der gesuchten `EReferences` besteht aus den im Folgenden erläuterten Schritten.

Zunächst werden potentielle Endpunkte für die Kantenklasse ermittelt. Dazu werden zwei Mengen von `EReferences` betrachtet: Zum einen die Menge aller eigenen inklusive der ererbten `EReferences`, sowie die Menge der `EReferences`, welche die `EClass` referenzieren. Aus diesen Mengen entfernt werden die `EReferences`, welche bei der Betrachtung der Superklassen als irrelevant gespeichert wurden.

Zu diesen Mengen werden nun die `EReferenceTypes` beziehungsweise `EContainingClasses` betrachtet. Diese werden dahingehend untersucht, ob sie zueinander über Superklassen-Beziehungen verwandt sind. Es werden zu allen unabhängigen Bäumen die spezialisierteste `EClass` sowie diejenige `EReference`, welche für ihr Vorkommen in der Menge verantwortlich war, gespeichert. Im Idealfall lassen sich genau 2 Bäume extrahieren, womit die beiden Endpunkte gefunden sind.

Sind 2 Klassen als Endpunkte gefunden worden, werden nun die gespeicherten `EReferences` sowie ihre `EOpposites`, falls vorhanden, als Ergebnis zurück geliefert.

Wurde nur 1 Endpunkt gefunden, gestaltet sich das weitere Vorgehen schwieriger. In diesem Fall werden zunächst alle `EReferences` zwischen dem gefunden Endpunkt und derjenigen `EClass` aus der Kantenklassen-Hierarchie, die damit in Verbindung steht, betrachtet. Bestehen zwei Verbindungen, können die `EReferences` dieser Verbindungen als Ergebnis zurück gegeben werden. Abbildung 48 zeigt ein Beispiel dazu.

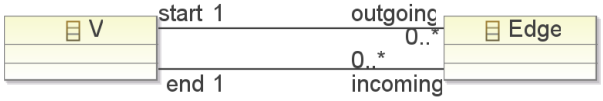

Abbildung 48: *Einfaches Beispiel für eine Kantenklasse in der eine Klasse die Rolle beider Endpunkte übernimmt*

Sind die beiden `EClasses` dagegen nur durch die eine `EReference`, mit oder ohne `EOpposite`, verbunden, dann muss es eine Superklasse des Endpunktes geben, welche die verbliebene Verbindung liefern kann. Abbildung 49 zeigt mit der `EClass Edge1` ein Beispiel für eine solche Situation. Um die andere Verbindung zu finden, wird die gefunden Verbindung aus den beiden Mengen gelöscht und anschließend noch einmal nach Endpunkten gesucht. Hat der Nutzer angegeben, welche `EReferences` von der gefundenen überschrieben werden, dann werden auch diese aus den Mengen gelöscht. Es sollte nun ein anderer Endpunkt gefunden werden, in diesem Beispiel `A`.

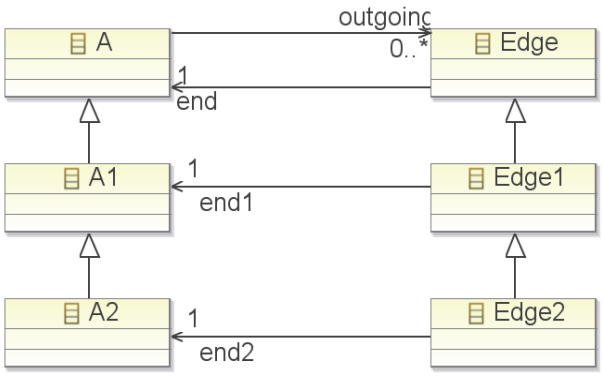

Abbildung 49: *Beispiel für eine Kantenklasse in der die Rollen beider Endpunkte von verwandten Klassen übernommen werden*

Besitzt dieser neue Endpunkt zwei Verbindungen, wird versucht, festzustellen, welche von der schon gefundenen Verbindung überschrieben wird. Das kann jedoch nur erkannt werden, wenn die Verbindungen unidirektional sind oder den gleichen Namen tragen. Für `E1` aus Abbildung 49 ist das der Fall, es ist klar zu erkennen, dass `end2` `end` überschreibt. Abbildung 50 zeigt dagegen einen Fall, wo diese Analyse kein Ergebnis bringt.

Besitzt der neue Endpunkt dagegen auch nur eine Verbindung muss versucht werden, festzustellen, ob diese als zweite Verbindung genutzt werden kann. Sind beide Verbindungen unidirektional und zeigen beide auf die Endpunkte oder von den Endpunkten weg, ist dies nicht der Fall. Abbildung 49 zeigt mit der `EClass Edge2` ein Beispiel. Hier muss die `EReference`

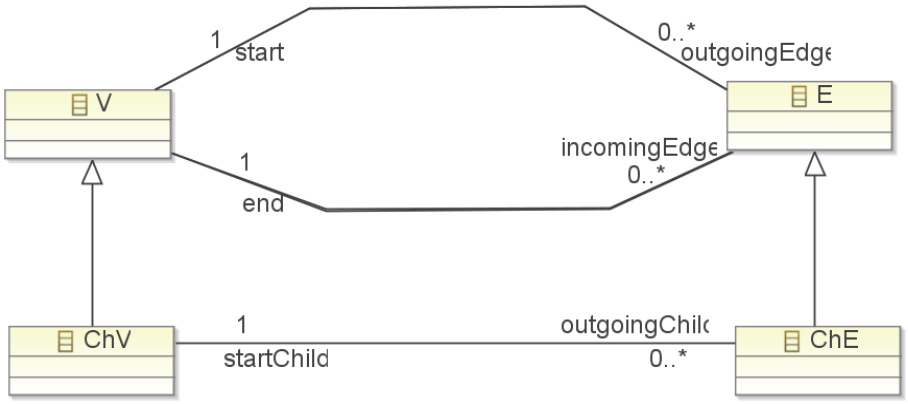

Abbildung 50: *Beispiel für eine Kantenklasse in der die Rollen beider Endpunkte von verwandten Klassen übernommen werden*

erneut aus der Menge gelöscht und noch einmal nach Endpunkten gesucht werden, solange bis eine ausreichende `EReference` gefunden wird. Ist eine der Verbindungen nicht unidirektional oder zeigen beide in unterschiedliche Richtungen (gemeint ist zum Endpunkt hin und vom Endpunkt weg) wird angenommen, dass dies die richtige Verbindung ist.

Die so gefundenen `EReferences` bilden das Ergebnis. Mit ihrer Hilfe kann die `EClass` korrekt in eine `EdgeClass` konvertiert werden.

8.4 Konfigurationsdatei

Um einem Schema individuelle Konfigurationen mitgeben zu können, werden “property lists” genutzt. Diese ermöglichen es, Daten in Arrays und Dictionaries zu organisieren. Unterstützte Datentypen sind `CFString`, `CFNumber`, `CFBoolean`, `CFDate`, `CFData`, `CFArray` und `CFDictionary`. Mit “property lists” können Daten sinnvoll strukturiert, transportiert, gespeichert und gelesen werden [App].

Eine einfache Struktur, die es ermöglicht Key-Value-Paare zu speichern, sieht so aus:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
3                 "http://www.apple.com/DTDs/PropertyList-1.0.dtd
4 <plist version="1.0">
5     <dict>
6           ...//Konfigurationen
7     </dict>
8 </plist>
```

Als Key wird nun die jeweilige Einstellungsmöglichkeit verwendet, während als Value die spezifischen Werte für das jeweilige Schema genutzt werden. Alle Einstellungsmöglichkeiten sind optional, müssen also nicht in der Konfigurationsdatei vertreten sein.

Im Folgenden wird erläutert, wie die bereits in Abschnitt 7.2 beschriebenen Optionen, in der Konfigurationsdatei syntaktisch angegeben werden. Dazu wird jeweils das Key-Value-Paar erklärt.

8.4.1 Benennung des Wurzelpaketes

Um ein `EPackage` als Wurzelpaket zu definieren, wird dies folgendermaßen in der Konfigurationsdatei angegeben:

```
<key>rootpackage</key>
<string>packagename</string>
```

Als Key wird der String “rootpackage” verwendet. Der Value-Part enthält den Namen des gewünschten Wurzelpaketes.

8.4.2 Deklaration der Graphklasse

Um eine `EClass` als Graphklasse zu definieren, werden in der Konfigurationsdatei folgende Zeilen genutzt:

```
<key>graphclass</key>
<string>packagename1.packagename2.eclassname1</string>
```

Als Key wird der String “graphclass” verwendet. Der Value-Part enthält den qualifizierten Namen der `EClass`.

8.4.3 Deklaration von Kantenklassen

Diese Option ermöglicht dem Nutzer die Auflistung der `EClasses` mit qualifiziertem Namen, die `EdgeClasses` werden sollen.

```
<key>edgeclasses</key>
<array>
    <string>packagename1.packagename2.eclassname1</string>
    <string>packagename1.packagename3.eclassname2</string>
</array>
```

Als Key wird der String “edgeclasses” verwendet. Der Value-Part besteht aus einem Array von String-Werten. Jeder String-Wert stellt einen qualifizierten Namen einer `EClass` da.

Die folgende Option ermöglicht es automatisch nach konzeptuellen Kantenklassen zu suchen.

```
 <key>search_for_edge_classes</key>
 <string>print_proposals</string>

 <key>search_for_edge_classes</key>
 <string>take_automatic</string>
```

Als Key wird der String “search_for_edge_classes” verwendet. Der Value-Part besteht aus einer der beiden Konstanten “print_proposals” und “take_automatic”. Enthält diese Option etwas anderes als Value, dann wird eine Fehlermeldung ausgegeben.

8.4.4 Bestimmung der Richtung einer Kantenklasse

Um die Richtung einer entstehenden Kantenklasse zu definieren, hat der Nutzer zwei Möglichkeiten.

Zum einen kann er für Kompositionen eine Richtung festlegen. Hier gibt es zwei Varianten, entweder die Richtung verläuft immer vom Ganzen zum Teil oder umgedreht.

```
<key>edgeclassdirection_aggregation</key>
<string>aggregation_whole_to_part</string>

<key>edgeclassdirection_aggregation</key>
<string>aggregation_part_to_wohle</string>
```

Als Key wird der String “edgeclassdirection_aggregation” verwendet. Der Value-Part besteht aus einer der beiden Konstanten “aggregation_whole_to_part” und “aggregation_part_to_wohle”. Enthält diese Option etwas anderes als Value, dann wird eine Fehlermeldung ausgegeben.

Die zweite Möglichkeit, eine Richtung festzulegen, besteht darin, sie direkt für eine Kantenklasse anzugeben.

```
<key>edgeclassdirection_reference_specific</key>
<array>
    <string>packagename1.eclassname1.ereferencename1, FROM</string>
    <string>packagename2.eclassname3.ereferencename2, TO</string>
</array>
```

Als Key wird der String “edgeclassdirection_reference_specific” verwendet. Der Value-Part besteht aus einem Array von Strings. Jeder String repräsentiert den qualifizierten Namen einer `EReference`, die zum Entstehen einer Kante beiträgt, mit Komma getrennt von “FROM” oder “TO”, je nachdem welche Richtung die benannte `EReference` haben soll.

8.4.5 Deklaration des Namen einer Kantenklasse

Diese Option ermöglicht es, den Namen einer entstehenden Kantenklasse festzulegen.

```
<key>reference_to_edgeclassname</key>
<array>
    <string>packagename1.eclassname1.ereferencename1, packagename1.userdefinedname1</string>
    <string>packagename2.eclassname3, packagename4.userdefinedname2</string>
</array>
```

Als Key wird der String “reference_to_edgeclassname” verwendet. Der Value-Part besteht aus einem Array von Strings. Jeder String repräsentiert den qualifizierten Namen einer `EReference`, die zum Entstehen einer Kante beiträgt, beziehungsweise den Namen einer `EClass` welche als `EdgeClass` definiert wurde, mit Komma getrennt vom gewünschten Namen für diese Kantenklasse.

8.4.6 Generierung von Rollennamen

Diese Option ermöglicht es zu entscheiden, ob Rollennamen generiert werden sollen.

```
<key>generate_role_names</key>
</true>

<key>generate_role_names</key>
</false>
```

Als Key wird der String “generate_role_names” verwendet. Der Value-Part besteht entweder aus true oder false.

8.4.7 Deklaration des Paketes einer Kantenklasse

`EReferences` in Ecore haben kein Paket. Um einer entstehenden Kantenklasse ein Paket zuzuordnen, gibt es diese Option.

```
<key>reference_to_packagename</key>
<array>
   <string>packagename1.eclassname1.ereferencename1, packagename1.packagename2</string>
   <string>packagename2.eclassname3, packagename4</string>
</array>
```

Als Key wird der String “reference_to_packagename” verwendet. Der Value-Part besteht aus einem Array von Strings. Jeder String repräsentiert den qualifizierten Namen einer `EReference`, die zum Entstehen einer Kante beiträgt, mit Komma getrennt vom Namen des gewünschten Paketes für diese Kantenklasse.

8.4.8 Benennung der überschreibenden Referenzen

Wenn Start- und Endpunkt einer Kantenklasse vom Typ der selben `EClass` sind, ist bei Vererbung nicht eindeutig, welche `EReferences` welche anderen überschreiben. Diese Option bietet dem Nutzer die Möglichkeit, dies in einem solchen Fall selbst zu definieren.

```
<key>reference_overwrites_reference</key>
<array>
   <string>packagename1.eclassname1.refname1, packagename1.eclassname2.refname2</string>
   <string>packagename2.eclassname4.refname4, packagename2.eclassname3.refname3</string>
</array>
```

Als Key wird hier der String “reference_overwrites_reference” verwendet. Der Value-Part besteht aus einem Array von Strings. Jeder String repräsentiert den qualifizierten Namen einer `EReference`, mit Komma getrennt von dem qualifizierten Namen der `EReference`, welche von ihr überschrieben wird. Zu Beachten ist hier, dass die Angaben auf Transitivität untersucht werden. Überschreibt die `EReference a` die `EReference b` und `b` überschreibt `c` dann genügt die Angabe der Paare `(a, b)` und `(b, c)`. Das Paar `(a, c)` wird dann automatisch ermittelt und hinzugefügt.

Neben der Möglichkeit, die Optionen in einer Konfigurationsdatei festzuhalten, können sie auch direkt über die Konsole eingegeben werden. Der folgende Abschnitt 8.5 erläutert die Vorgehensweise.

8.5 Kommandozeilen-Werkzeug

Dieser Abschnitt beschäftigt sich mit der Konsolenschnittstelle. Sie bietet die Möglichkeit, die Transformation direkt zu nutzen, ohne, dass ein zusätzliches Programm die entsprechenden Methoden aufrufen muss. Alle in Abschnitt 7.2 beschriebenen Optionen können auch auf diese Weise angegeben werden.

Im Folgenden wird beschrieben, welche Optionen es gibt und wie diese angegeben werden. Aufgerufen wird die Transformation auf diese Weise:

```
usage: java de.uni_koblenz.jgralab.utilities.ecore2tg.Ecore2Tg (-h ) (-v )
            -i <filename>
            (-o <filename>)
            (-c <filename>)
            (-m <filename>)
            (-g <qualifiedEClassName>)
            (-e <qualifiedEClassName>)
            (-s )
            (-t )
            (-a <1or2>)
            (-p <qualifiedReferenceName qualifiedPackageName>
                <qualifiedReferenceName qualifiedPackageName>)
            (-d <qualifiedReferenceName directionvalue>
                <qualifiedReferenceName directionvalue>)
            (-n <qualifiedReferenceName name>
                <qualifiedReferenceName name>)
            (-r )
            (-x <overwritingRefname overwrittenRefname>
                <overwritingRefname overwrittenRefname>)
            (-w <epackagename>)
```

Die Optionen `[-h ]` und `[-v ]` dienen nur der Ausgabe des Hilfe-Textes beziehungsweise der Version der Software. Die verbleibenden Parameter unterteilen sich in mehrere Gruppen.

8.5.1 Dateinamen

Die erste Gruppe bietet die Möglichkeit Dateinamen anzugeben. Dabei ist die Angabe des Pfads zum Ecore-Metamodell, durch die Option `-i` repräsentiert, für die Transformation notwendig. Die übrigen Parameter sind optional.

```
-i,--input <filename>
  (required): Ecore Metamodel file of the Schema.

-o,--output <filename>
  (optional): writes a TG-file of the Schema to the given filename. Free naming,
              but should look like this: '<filename>.rsa.tg.'

```

```
-c,--configuration <filename>
  (optional): loads configurations from the given filename.

-m,--modelfilename <filename>
  (optional): filename of a corresponding model file that beomes transformed
```

Die Option `-o` ermöglicht es, den Namen einer TG-Datei als Parameter anzugeben, unter der das resultierende Schema gespeichert wird. Wird kein Name angegeben, dann wird das Schema nicht gespeichert.

Die Option `-c` ermöglicht es, Konfigurationsdateien zu laden. Als Parameter erhält sie den Pfad zur entsprechenden Datei.

Die Option `-m` dient zur Angabe von Modelldateien. Sie kann beliebig oft vorkommen. Allerdings werden alle Dateien als zu einem Modell gehörig interpretiert. Es ist also mit dem Kommandozeilen-Tool nicht möglich mehrere verschiedene Modelle auf einmal zu transformieren. Gespeichert wird die TG-Datei des Modells automatisch unter dem Namen der ersten Modelldatei.

8.5.2 Grundlegende Einstellungen

Die zweite Gruppe von Optionen legt grundlegende Einstellungen für die Transformation fest. Ihre Elemente dürfen in einem Aufruf nur einmal benutzt werden.

```
-w,--rootEPackage <epackagename>
  (optional): declares the root EPackage of the metamodel

-g,--graphclassname <qualifiedEClassName>
  (optional): qualified name of EClass that should become the GraphClass

-s,--searchAutomaticAfterEdgeClasses
  (optional): if this flag is set, EdgeClasses are automatically searched

-t,--printFoundEdgeClasses
  (optional): if this flag is set, EdgeClasses proposals are printed

-r,--generateRoleNames
  (optional): defines that missing role names should become generated

-a,--influenceOfAggregation <1or2>
  (optional): 1 specifies direction whole to part, 2 specifies direction part to whole
```

Die Option `-w` gibt das Wurzelpaket des Metamodells an. Sie entspricht der Einstellung aus 7.2.1. Als Parameter wird der Name des Wurzelpaketes angegeben.

Die Option `-g` gibt den Namen der Graphklasse an. Der qualifizierte Name einer `EClass` bildet den Parameter. Diese Option entspricht der Einstellung aus 7.2.2.

Die Option `-s` gibt an, dass das Programm automatisch nach Kantenklassen suchen soll. Die Ergebnisse werden dann in `EdgeClasses` konvertiert. Die Option `-t` dagegen lässt das Programm nach Kantenklassen suchen, diese aber nur auf der Konsole ausgeben. Es darf nur

eine der beiden Parameter pro Programmaufruf genutzt werden. Diese Optionen entsprechen der Einstellung aus dem ersten Absatz von 7.2.3.

Die Option `-r` gibt an, dass das Programm automatisch fehlende Rollennamen generieren soll. Dies entspricht der Einstellung aus 7.2.6.

Die Option `-a` ermöglicht die Festlegung der Richtung von Kompositionen. Zur Auswahl stehen als Parameter die beiden Integerwerte 1 und 2. Wird 1 angegeben, werden die Richtungen vom Ganzen zum Teil definiert. Handelt es sich bei dem Parameter um die Zahl 2, werden die Richtungen vom Teil zum Ganzen festgelegt. Diese Option entspricht der Einstellung aus dem ersten Absatz von 7.2.4.

8.5.3 Einstellungen für Modellelemente

Die letzte Gruppe von Optionen gibt Informationen zur Transformation für einzelne Elemente des Metamodells an. Sie dürfen beliebig oft verwendet werden, da ein Metamodell beliebig viele Elemente enthalten kann.

```
1  -d,--directionOfEdgeClass <qualifiedReferenceName directionval>
2    (optional): defines the direction of the specified EReference, valid values for
3                directionval are "FROM" and "TO"
4
5  -e,--edgeclassname <qualifiedEClassName>
6    (optional): qualified name of EClass that should become an EdgeClass
7
8  -p,--definePackageOfEdgeClass <qualifiedReferenceName qualifiedPackageName>
9    (optional): defines a package for an EdgeClass
10
11 -n,--edgeclassname <qualifiedReferenceName name>
12   (optional): sets the EdgeClass, made frome the given EReference, the specified name
13
14 -x,--overwritingreferences <overwritingRefname overwrittenRefname>
15   (optional): defines which EReference overwrites which other EReference
```

Die Option `-d` gibt die Richtung für eine bestimmte `EdgeClass` an. Um welche Kantenklasse es sich handelt, wird durch den qualifizierten Namen einer `EReference` angegeben, die zur Entstehung der `EdgeClass` beiträgt. Diese Option entspricht dem zweiten Absatz aus 7.2.4.

Die Option `-e` gibt den Namen einer `EClass` an, die in eine Kantenklasse konvertiert werden soll. Sie entspricht der Einstellung aus 7.2.3 im zweiten Teil.

Die Option `-p` gibt das Paket für eine bestimmte `EdgeClass` an. Sie entspricht der Einstellung aus 7.2.7. Als Parameter wird hier entweder der qualifizierte Name einer `EClass` oder einer `EReference` angegeben.

Die Option `-n` gibt den Namen für eine bestimmte `EdgeClass` an. Um welche Kantenklasse es sich handelt, wird wieder durch den qualifizierten Namen einer `EReference` angegeben, die zur Entstehung der `EdgeClass` beiträgt. Diese Option entspricht der Einstellung aus 7.2.5.

Die Option `-x` gibt an, welche `EReference` welche andere überschreibt. Als Parameter werden die qualifizierten Namen der `EReferences` genutzt. Diese Option entspricht der Einstellung aus 7.2.8.

8.6 Zusammenfassung

Das Programm gliedert sich in eine *Metamodell- und eine Modelltransformation*. Erstere transformiert ein Ecore-Metamodell in ein grUML-Schema, letztere ein passendes Ecore-Modell in einen grUML-Graphen. In Abschnitt 8.1 werden die Einzelheiten genauer beschrieben.

Implementierungsdetails zur Metamodelltransformation werden in den Abschnitten 8.2 und 8.3 beschrieben. In ersterem wird erklärt, nach welchen Kriterien und auf welche Weise automatisch *nach Kantenklassen gesucht* wird. Der zweite Abschnitt beschreibt, wie für `EClasses`, die als konzeptuelle Kantenklassen erkannt wurden, diejenigen *Referenzen gefunden* werden, welche zur *Bestimmung der Inzidenzen* für die resultierende `EdgeClass` nötig sind.

Um das Programm zu nutzen gibt es zwei Möglichkeiten. Zum einen bietet die *Programmierschnittstelle* die Möglichkeit, alle Funktionen aus einem eigenen Programm heraus zu nutzen. Zum anderen kann die Transformation auch direkt von der *Kommandozeile* aus gestartet werden. Abschnitt 8.5 erläutert dies ausführlich.

Allgemein besteht die Möglichkeit die *Transformation durch eine Konfigurationsdatei zu beeinflussen*. In dieser können alle Optionen aus 7.2 angegeben werden. Unter Nutzung der Programmierschnittstelle werden alle enthaltenen Optionen geladen und den entsprechenden Maps und Listen zugefügt. Wie eine solche Datei syntaktisch aufgebaut ist, wird in Abschnitt 8.4 erklärt.

9 Ergebnisse

Dieses Kapitel zeigt Ergebnisse der vorliegenden Bachelorarbeit. Zunächst werden drei Ecore-Metamodelle betrachtet. Dazu wird jeweils das grUML-Schema dargestellt, welches durch die automatische Transformation generiert wurde. Im Abschnitt 9.2 wird die Verwendung der Optionen an einem Beispiel-Metamodell erläutert.

9.1 Automatische Transformation

Dieser Abschnitt zeigt einige Beispiele für Ergebnisse der Transformation. Die Beispiele wurden automatisch transformiert, die Suche nach Kantenklassen war aktiviert.

9.1.1 UML-Aktivitätsdiagramm

Abbildung 51 zeigt ein vereinfachtes Metamodell zu UML 1.4 Aktivitätsdiagrammen.

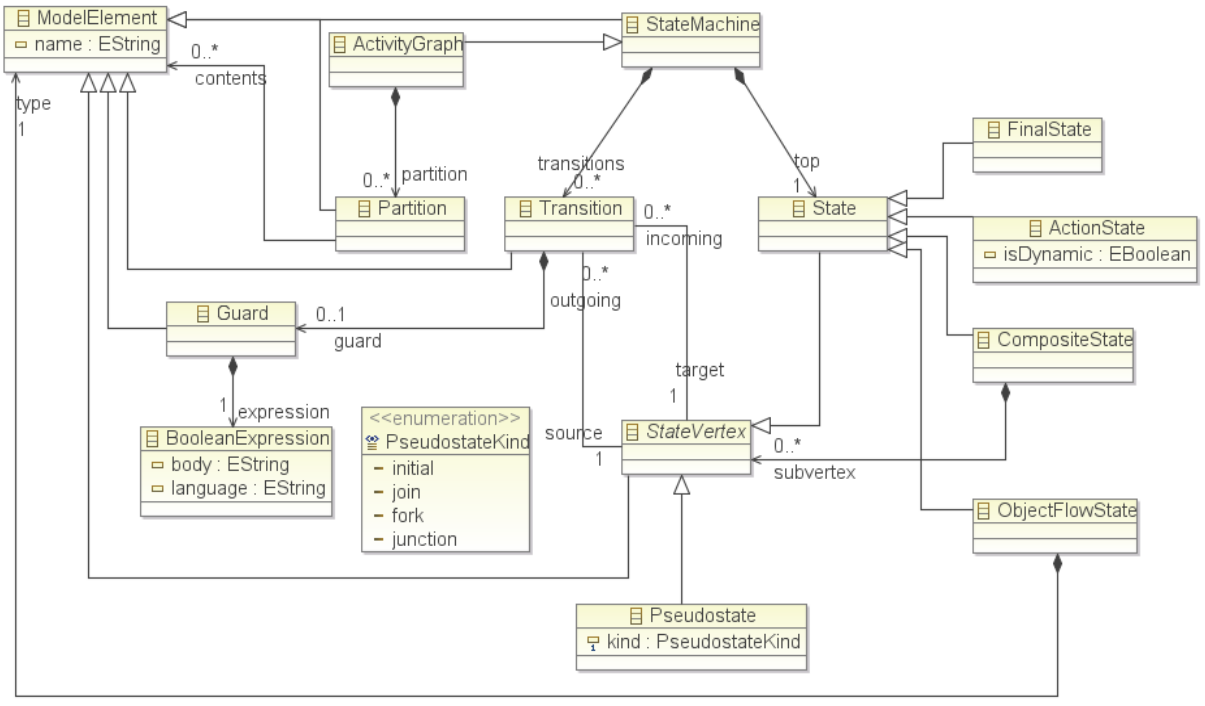

Abbildung 51: *UML-Aktivitätsdiagramm: Ecore-Metamodell*

In Abbildung 52 wird das resultierende grUML-Schema dargestellt.

Die sichtbaren `EClasses` sind in `VertexClasses` umgewandelt worden. Für die `EClass ModelElement` wurde beispielsweise die `VertexClass ModelElement` erstellt. Das zu `ModelElement` gehörenden `EAttribute name` wurde zum `Attribute name` der `VertexClass` konvertiert. Die `EReference contents`, welche zu `Partition` gehört und `ModelElement` referenziert, wurde in die Kantenklasse `PartitionLinksToContents` transformiert.

Bei der Transformation von `EReferences` zu `EdgeClasses` wurden sowohl Multiplizitäten als auch die Kompositions-Eigenschaft korrekt übertragen. So ist beispielsweise die Kantenklasse `ActivityGraphContainsPartition` aus der `EReference partition` der `EClass ActivityGraph` entstanden.

Für bidirektionale `EReferences` wurden Richtungen erzeugt. Die `EReferences outgoing` und `source` der `EClasses Transition` und `StateVertex` sind zu einer Kantenklasse mit Namen `SourceLinksToOutgoing` transformiert worden. Da keine explizite Richtung angegeben wurde, hat das Programm eine bestimmt. In diesem Fall führt sie von `StateVertex` nach `Transition`.

Die `EClass Transition` hat Ähnlichkeit mit einer Kantenklasse. Trotzdem wird sie bei der automatischen Transformation nicht als solche gefunden, da die `EReference guard` das Kriterium verletzt, dass es nur eine bestimmte Anzahl und Form von `EReferences` geben darf. Auch dem Benutzer ist es nicht erlaubt, diese `EClass` als `EdgeClass` zu definieren. `Transition` ähnelt einer Kantenklasse, erfüllt aber nicht alle Eigenschaften einer konzeptuellen `EdgeClass`.

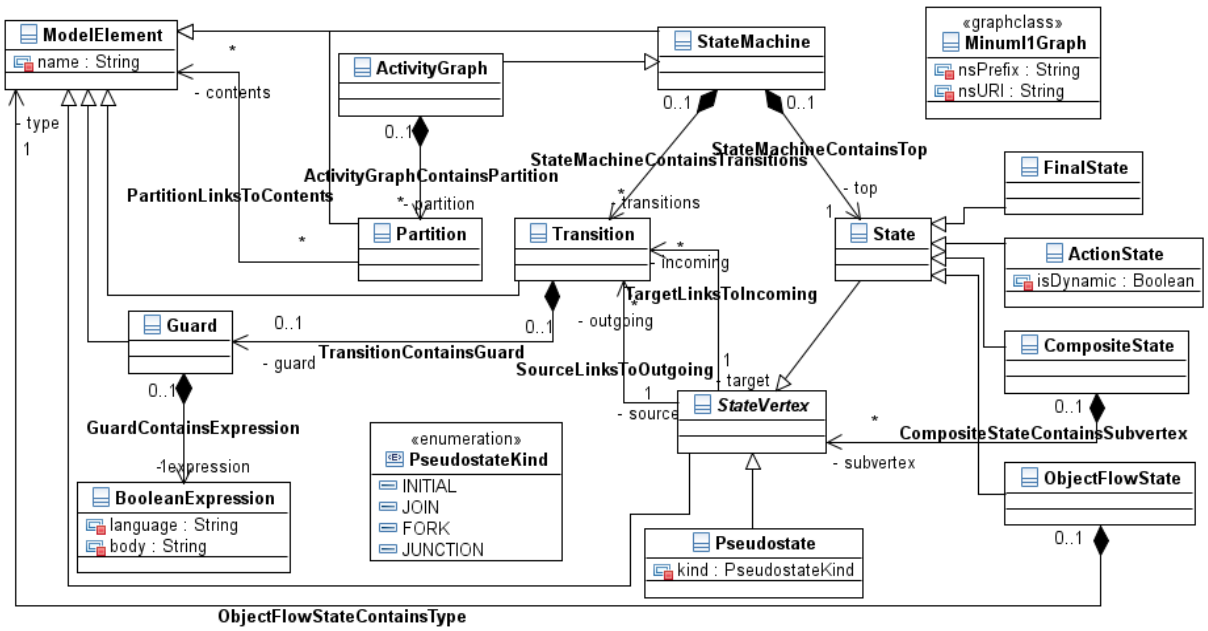

Abbildung 52: *UML-Aktivitätsdiagramm: grUML-Schema*

9.1.2 Grafcet

Grafcet ist ein Akronym aus **GRA**phe **F**onctionnel de **C**ommande **E**tapes/**T**ransitions. Es handelt sich um eine Spezifikationssprache für die Darstellung von Ablaufbeschreibungen [Sch08].

Abbildung 53 zeigt ein vereinfachtes Metamodell. Die Datei lautet `Grafcet.ecore` und ist unter folgender Adresse zu finden:

`http://www.eclipse.org/m2m/atl/atlTransformations/Grafcet2PetriNet/Grafcet2PetriNet.zip`.

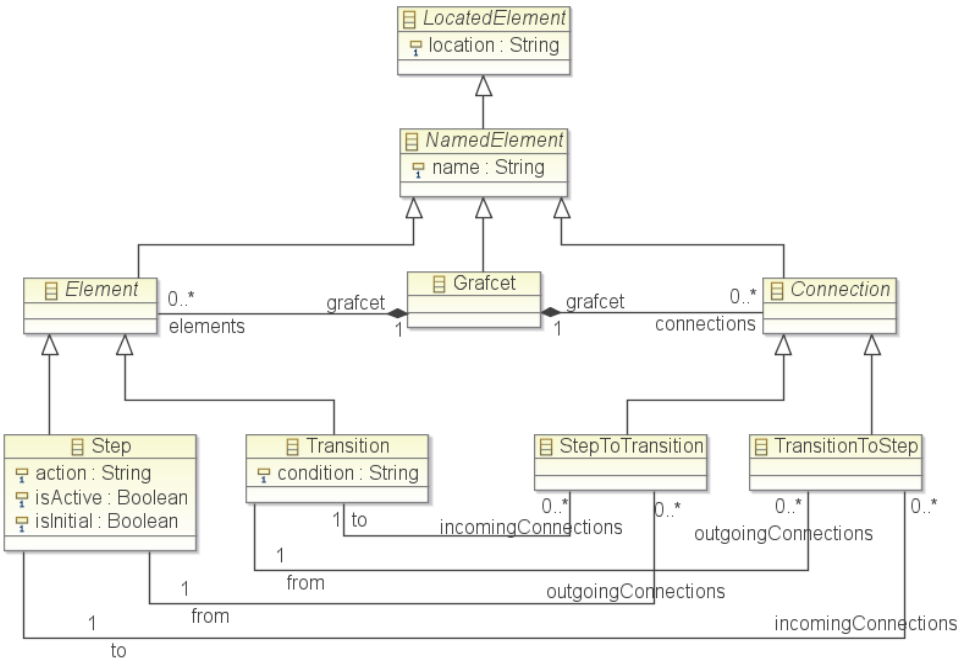

Abbildung 53: *Grafcet: Ecore-Metamodell*

Abbildung 54 zeigt das resultierende grUML-Schema.

Da sowohl zwischen den `EClasses` `Transition` und `TransitionToStep` sowie zwischen `Step` und `StepToTransition` die `EReferences` `from` und `outgoing` verwendet werden, wird dadurch der generierte Kantenklassenname identisch. Da dies zu einem nicht validen Schema führt wurde hier an einem Namen eine `2` ergänzt.

Abbildung 55 zeigt das Ergebnis der Transformation eines Modells zu dem Metamodell aus Abbildung 53.

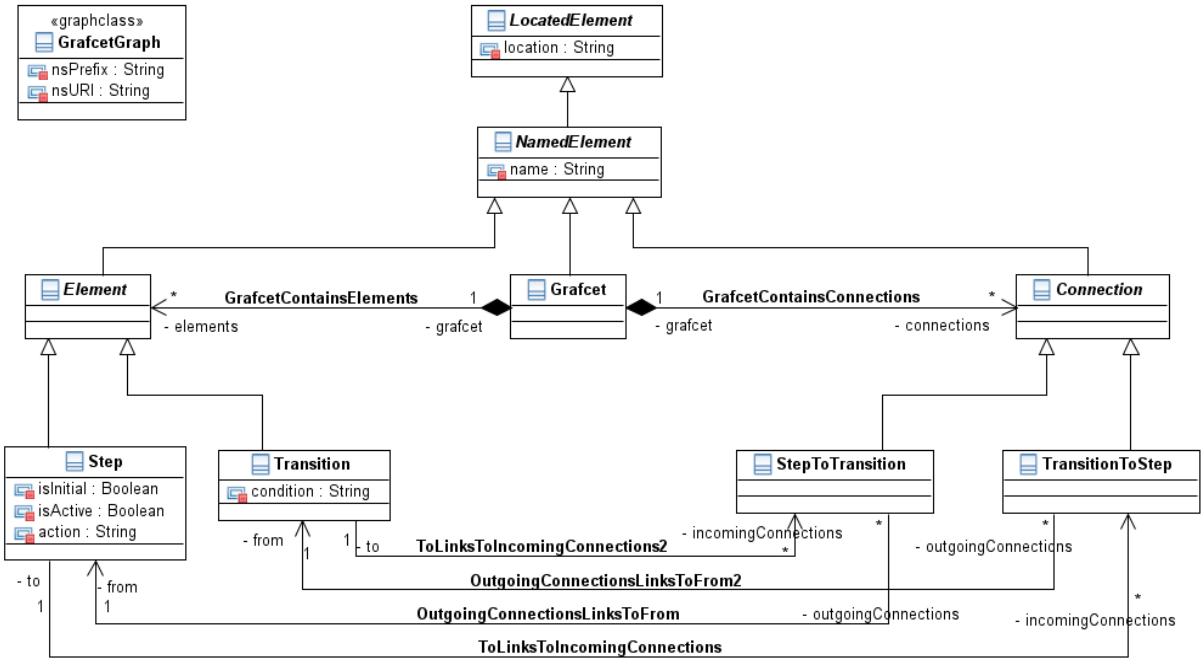

Abbildung 54: *Grafcet: grUML-Schema*

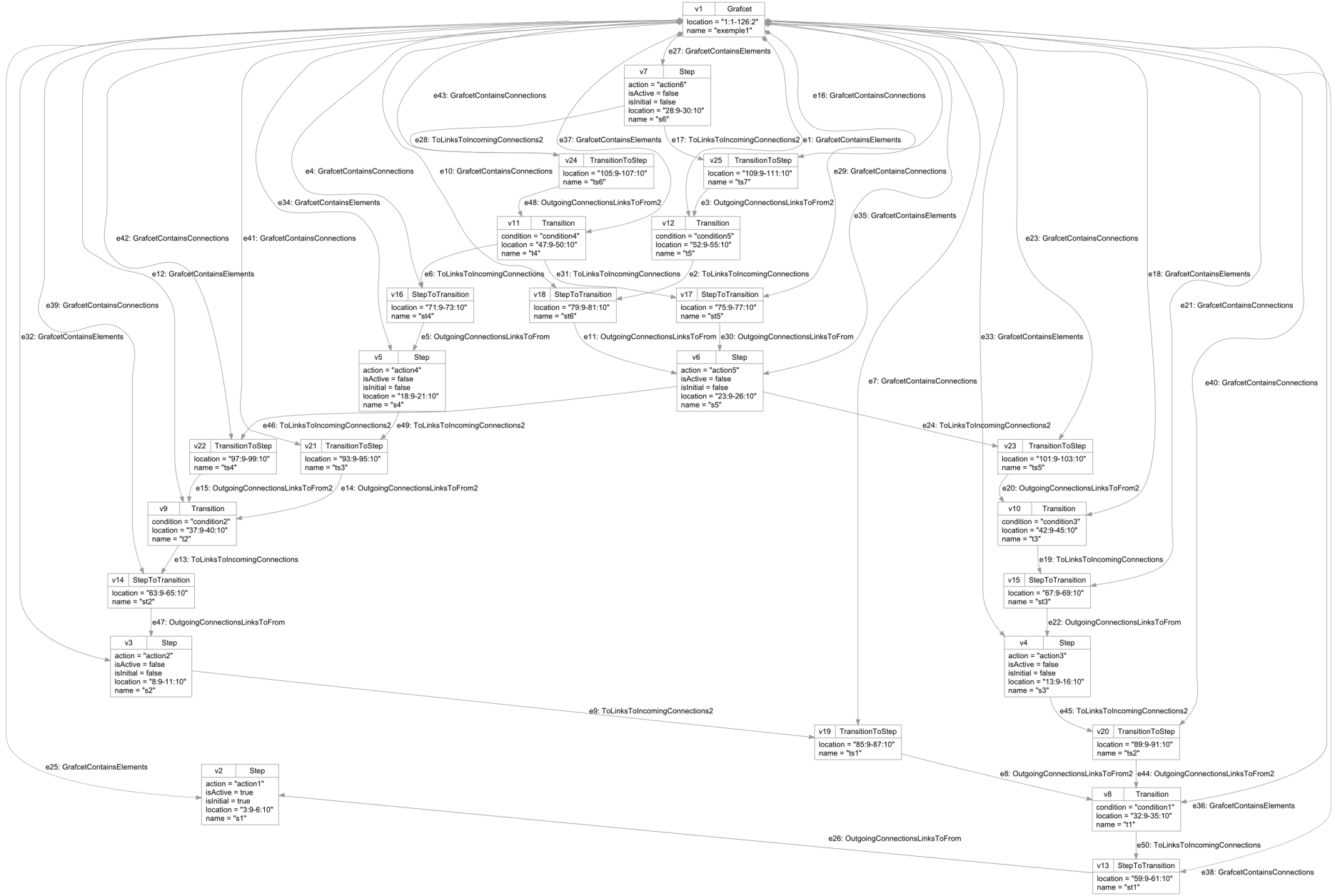

Abbildung 55: *Grafcet: grUML-Graph*

9.1.3 Metamodell mit erkannter Kantenklasse

Das Ausgangs Ecore-Metamodell zeigt Abbildung 56

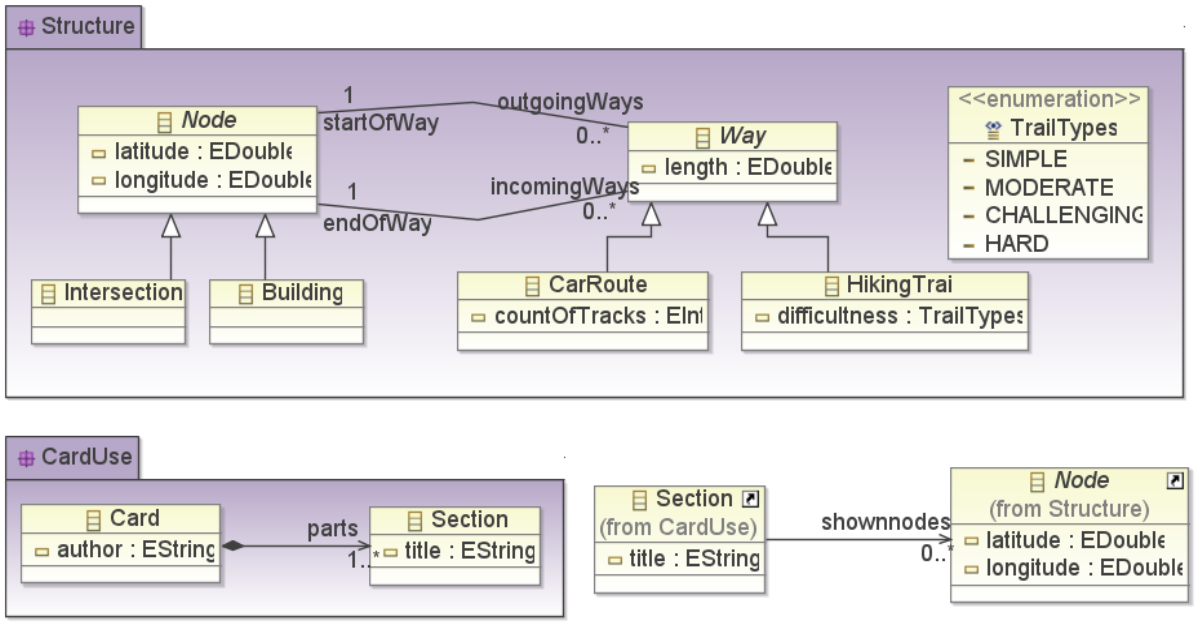

Abbildung 56: *Metamodell mit erkannter Kantenklasse: Ecore-Metamodell*

Abbildung 57 zeigt das Ergebnis der Transformation auf Schema-Ebene. Es wurde automatisch nach Kantenklassen gesucht und die konzeptuelle `EdgeClass Way` gefunden. Deren Subklassen `CarRoute` und `HikingTrail` verletzen ebenfalls keine Bedingung.

Außerdem sind die `EPackages structure` und `carduse` korrekt transformiert worden. Die `EClasses Card` und `Section` sind auch im grUML-Schema in `carduse` enthalten, während die verbliebenen `EClasses` zu `structure` gehören.

Abbildung 58 zeigt einen Beispiel-Graphen, der aus einem Modell, welches zum Metamodell in Abbildung 56 passt, entstanden ist.

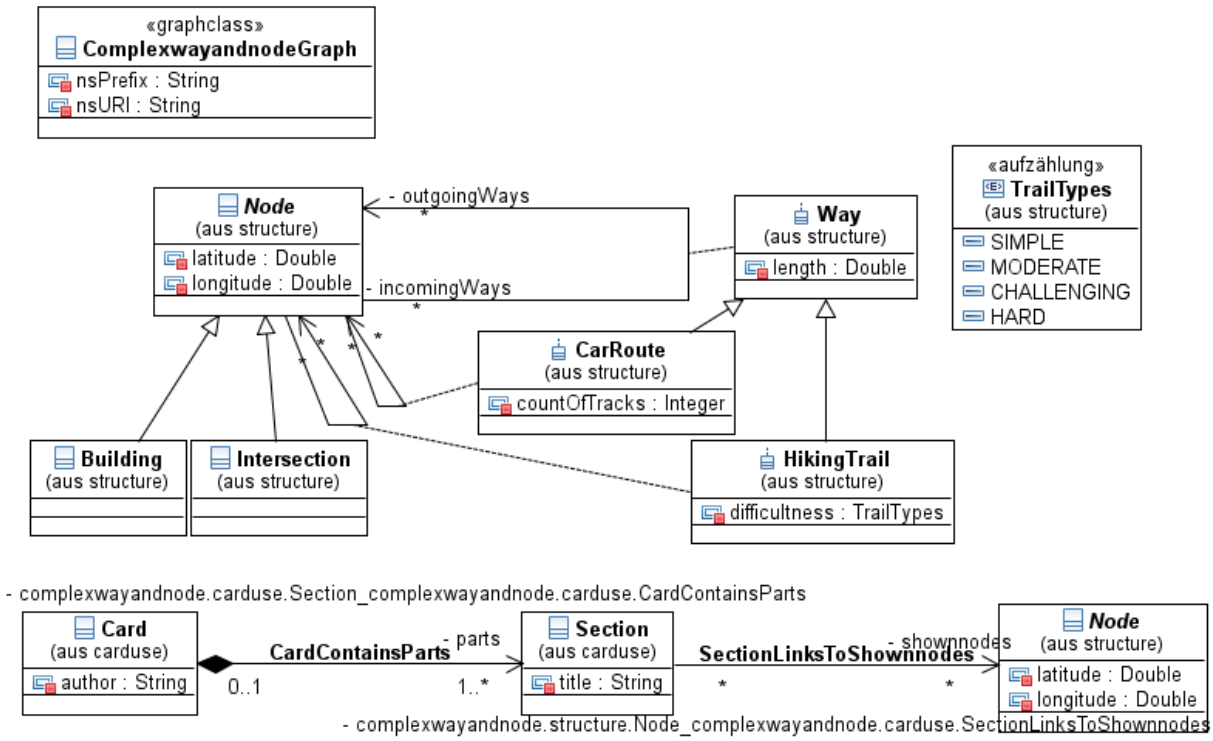

Abbildung 57: *Metamodell mit erkannter Kantenklasse: grUML-Schema*

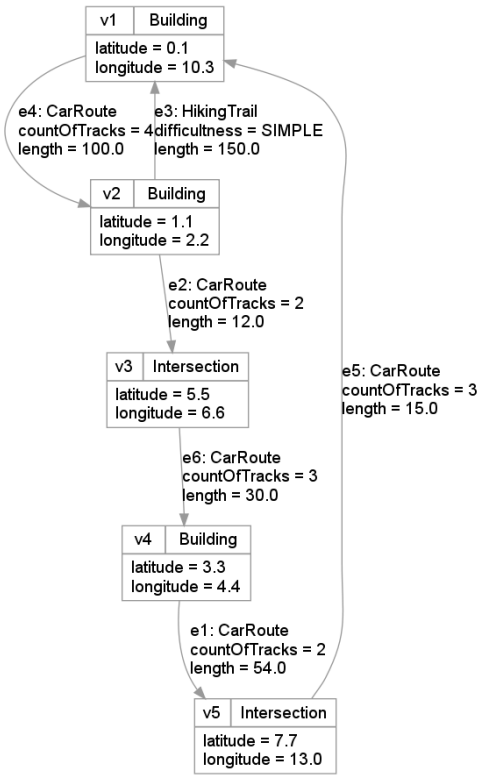

Abbildung 58: *Metamodell mit erkannter Kantenklasse: grUML-Graph*

9.2 Transformation mit Optionen

Neben der Möglichkeit, die Transformation automatisch ablaufen zu lassen, können auch die in 7.2 aufgeführten Optionen angeben werden. Dieser Abschnitt erläutert die verschiedenen Möglichkeiten Optionen anzugeben an einem Beispiel.

Abbildung 59 zeigt ein Ecore-Metamodell, welches einen kleinen Ausschnitt aus der Realität modelliert. Es handelt sich um die Modellierung einer Universität.

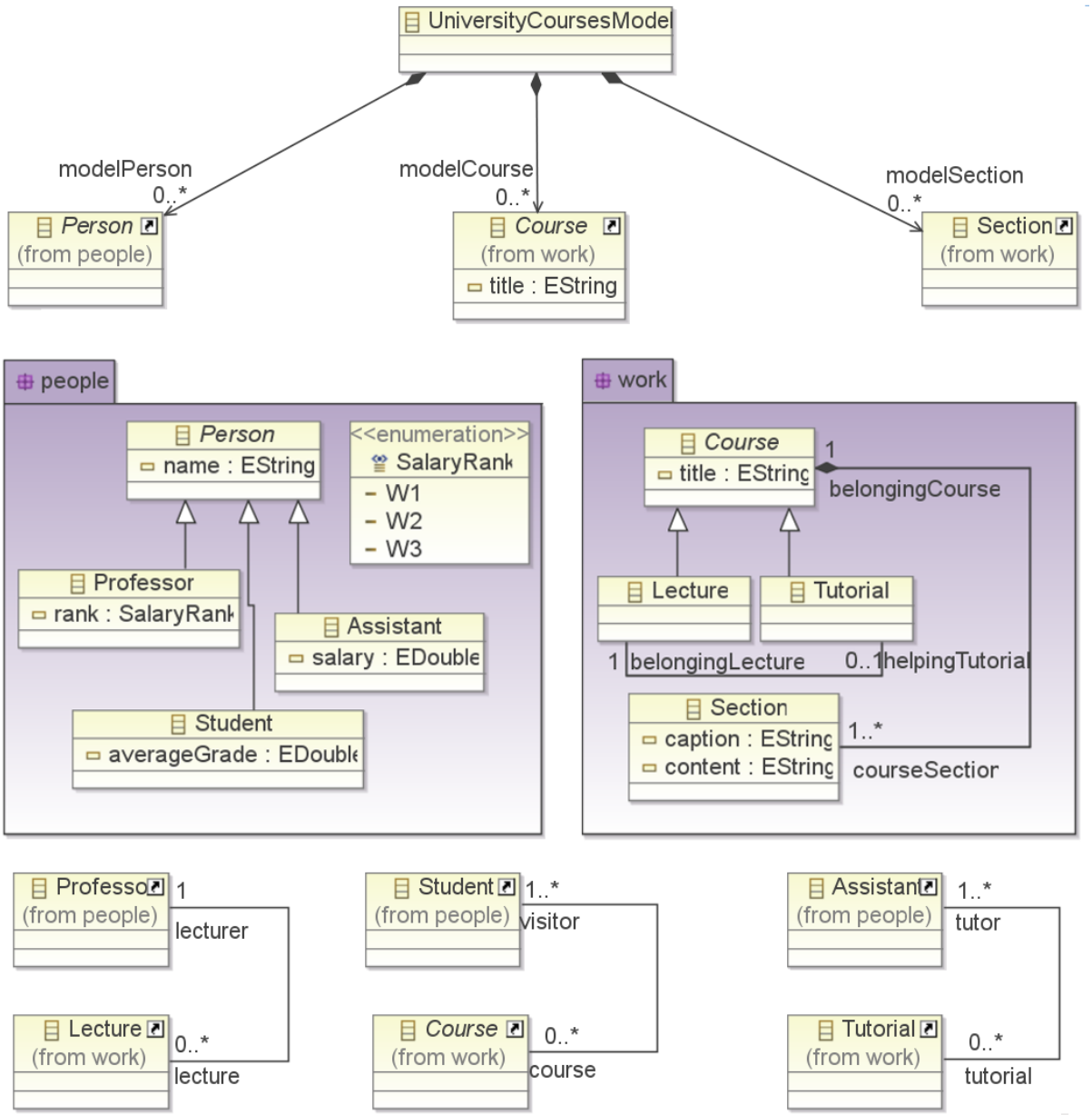

Abbildung 59: *UniversityModel: Ecore-Metamodell*

Abbildung 60 zeigt das Ergebnis der Transformation auf Schema-Ebene, wenn der Benutzer folgende Optionen gewählt hat:

- Die EdgeClass, welche aus der EReference “universitycourses.work.Tutorial.belongingLecture’ entsteht, soll den Namen “universitycourses.work.BelongsToLecture” erhalten.
- Die EdgeClass, welche aus der EReference “universitycourses.people.Assistant.tutorial” entsteht, soll den Namen “universitycourses.HoldsTutorium” erhalten.
- Die EdgeClass, welche aus der EReference “universitycourses.people.Professor.lecture” entsteht, soll dem Paket “universitycourses” zugeordnet werden.
- Die EdgeClass, welche aus der EReference “universitycourses.people.Professor.lecture” entsteht, soll die VertexClass “universitycourses.work.Lecture” als Startpunkt haben.
- Die EdgeClass, welche aus der EReference “universitycourses.people.Student.course” entsteht, soll die VertexClass “universitycourses.work.Course” als Startpunkt haben.
- Die EClass “universitycourses.UniversityCoursesModel” soll zur GraphClass werden.

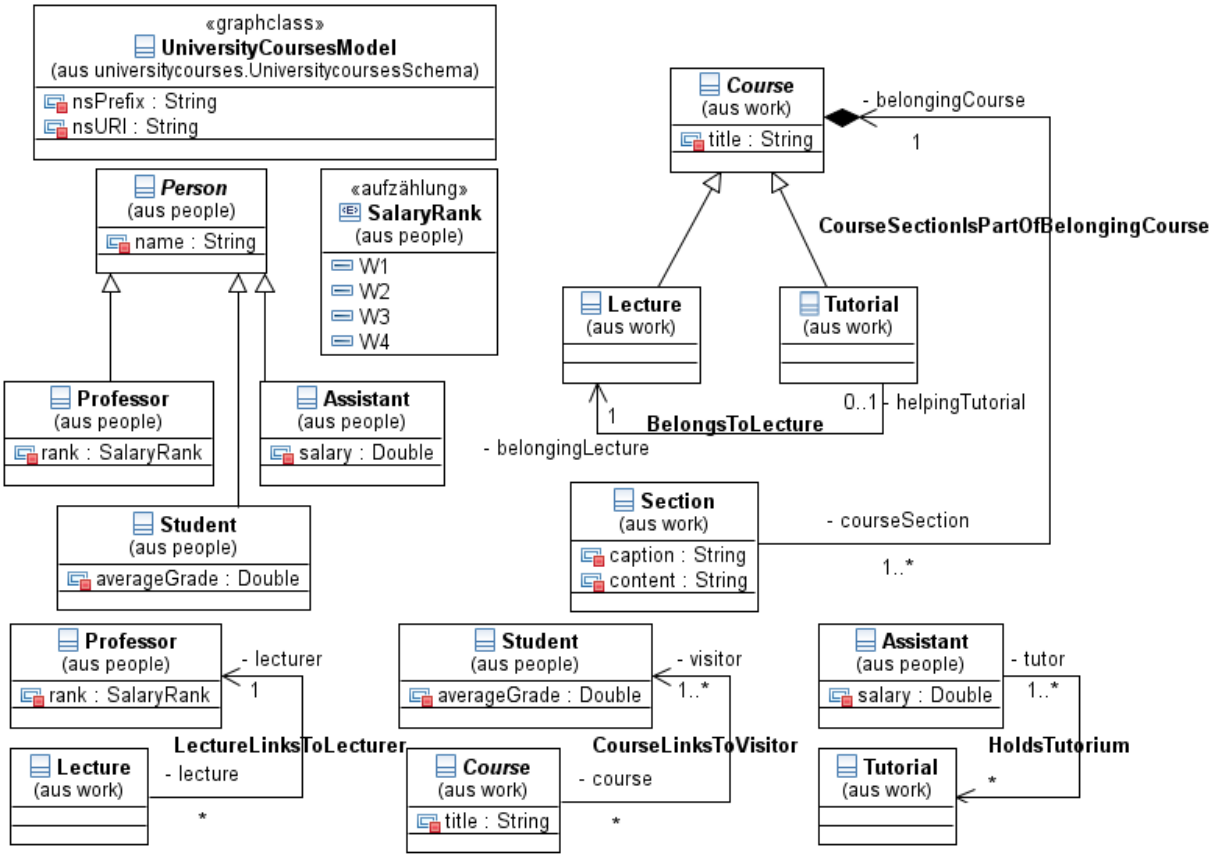

Abbildung 60: *UniversityModel: grUML-Schema*

Um diese Optionen zu setzen, hat der Nutzer die folgenden drei Möglichkeiten:

- Er gibt die Optionen direkt über die Programmierschnittstelle an.
- Er gibt der Transformation eine Konfigurationsdatei mit.
- Er verwendet die Möglichkeit, Optionen über die Kommandozeile anzugeben.

Diese werden im Folgenden im Detail mit Hilfe des Beispiels aus Abbildung 59 und der oben beschriebenen Optionen erläutert.

Verwendung der Programmierschnittstelle

Um die Api zu benutzen, werden die Optionen direkt den entsprechenden Maps, Sets oder Variablen hinzugefügt. Für die oben genannten Optionen ergibt sich folgender Code:

```
1  Ecore2Tg test = new Ecore2Tg(ecorefile);
2
3  //Choose names of EdgeClasses
4  test.getNamesOfEdgeClassesMap().put(
5                  "universitycourses.work.Tutorial.belongingLecture",
6                  "universitycourses.work.BelongsToLecture");
7  test.getNamesOfEdgeClassesMap().put(
8                  "universitycourses.people.Assistant.tutorial",
9                  "universitycourses.HoldsTutorium");
10
11 //Define Packages of EdgeClasses
12 test.getDefinedPackagesOfEdgeClassesMap().put(
13                 "universitycourses.people.Professor.lecture",
14                 "universitycourses");
15
16 //Define directions for EdgeClasses
17 test.getDirectionMap().put(
18                 "universitycourses.work.Lecture.lecturer",
19                 Ecore2Tg.TO);
20 test.getDirectionMap().put(
21                 "universitycourses.work.Course.visitor",
22                 Ecore2Tg.TO);
23
24 //Define GraphClass
25 test.defineAsGraphClass("universitycourses.UniversityCoursesModel");
```

Verwendung einer Konfigurationsdatei

Die passende Konfigurationsdatei für das University-Metamodell aus Abbildung 59, welche die bereits beschriebenen Optionen spezifiziert, hat folgendes Aussehen:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
3            "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
4  <plist version="1.0">
5     <dict>
6        <key>reference_to_edgeclassname</key>
7        <array>
8           <string>universitycourses.work.Tutorial.belongingLecture,
9                   universitycourses.work.BelongsToLecture</string>
10          <string>universitycourses.people.Assistant.tutorial,
11                  universitycourses.HoldsTutorium</string>
12       </array>
13       <key>reference_to_packagename</key>
```

```
14        <array>
15            <string>universitycourses.people.Professor.lecture,
16                    universitycourses</string>
17        </array>
18        <key>edgeclassdirection_reference_specific</key>
19            <array>
20                <string>universitycourses.work.Lecture.lecturer,TO</string>
21                <string>universitycourses.work.Course.visitor,TO</string>
22            </array>
23        <key>graphclass</key>
24        <string>universitycourses.UniversityCoursesModel</string>
25    </dict>
26 </plist>
```

Verwendung der Konsole

Soll das Konsolen-Werkzeug genutzt werden, geht dies zunächst auch direkt im Code.

```
1  String[] args = {"-i","metamodel.ecore",
2                   "-o", "schema.tg",
3                   "-n", "universitycourses.work.Tutorial.belongingLecture",
4                         "universitycourses.work.BelongsToLecture",
5                   "-n", "universitycourses.people.Assistant.tutorial",
6                         "universitycourses.HoldsTutorium",
7                   "-p", "universitycourses.people.Professor.lecture",
8                         "universitycourses",
9                   "-d", "universitycourses.work.Lecture.lecturer", "TO",
10                  "-d", "universitycourses.work.Course.visitor", "TO",
11                  "-g", "universitycourses.UniversityCoursesModel"};
12 Ecore2Tg.main(args);
```

Direkt in der Konsole wird die Transformation durch folgende Zeilen gestartet:

```
1  java Ecore2Tg -i metamodel.ecore
2                -o schema.tg
3                -n universitycourses.work.Tutorial.belongingLecture
4                   universitycourses.work.BelongsToLecture
5                -n universitycourses.people.Assistant.tutorial
6                   universitycourses.HoldsTutorium
7                -p universitycourses.people.Professor.lecture
8                   universitycourses
9                -d universitycourses.work.Lecture.lecturer TO
10               -d universitycourses.work.Course.visitor TO
11               -g universitycourses.UniversityCoursesModel
```

9.3 Zusammenfassung

Die aufgeführten Beispiele zeigen, dass die Transformation von Ecore nach grUML im Rahmen der Entscheidungen aus Kapitel 7 korrekt arbeitet.

Eine voll automatische Transformation bringt bereits gute Ergebnisse. Durch die Möglichkeit, Optionen anzugeben, erhält der Benutzer die Fähigkeit, Einfluss zu nehmen. Das Beispiel aus 9.2 zeigt, dass auch dies funktioniert.

10 Fazit und Ausblick

In diesem Kapitel werden die *Ergebnisse der Arbeit zusammengefasst*. Desweiteren wird ausgeführt, welche *weiterführenden Problemstellungen* sich ergeben und in welcher Weise mit den Ergebnissen weiter gearbeitet werden kann.

10.1 Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit der *Transformation von Ecore nach grUML*. Ziele der Arbeit waren die Entwicklung einer Abbildung von Ecore-Metamodellen nach grUML-Schemas und die Implementierung des Ergebnisses. Außerdem sollten zu den Metamodellen passende Modelle in Graphen konvertiert werden.

Hierzu wurden zunächst als Grundlage die Modellierungssprachen *Ecore und grUML* getrennt voneinander betrachtet. Sie wurden *anhand ihrer Metamodelle untersucht* und die einzelnen Elemente beschrieben. Dabei wurden auch die *Programmierschnittstellen* der einzelnen Sprachen vorgestellt und erläutert.

Weiterhin wurden Ecore und grUML auf *Unterschiede und Gemeinsamkeiten* hin untersucht. Für die einzelnen Elemente der Metamodelle wurde aufgeführt, wo eine direkte Abbildung problemlos möglich ist und wo Schwierigkeiten existieren und Entscheidungen getroffen werden müssen. Die Ergebnisse dieser Untersuchungen bildeten die Grundlage für die zu treffenden Entwurfsentscheidungen.

Der nächste Schritt beinhaltete das Festlegen der Vorgehensweise bei der *Behandlung der uneindeutigen Aspekte* der Abbildung. Die einzelnen Probleme wurden benannt und es wurde entschieden, wie damit umgegangen wird. Grundsätzlich wurde immer ein Default-Verhalten festgelegt, darüber hinaus kann die Transformation durch Angabe zusätzlicher Informationen in definierter Weise beeinflusst werden.

Im Anschluss wurden *Anforderungen* an die zu entwickelnde Software *formuliert*. Dies geschah auf Basis der zuvor getroffenen Entwurfsentscheidungen. Die Anforderungen wurden in vier Gruppen unterteilt. Darunter fanden sich allgemeine Anforderungen an die Schnittstellen der Software, Anforderungen an die Funktionalität und Beeinflussungsmöglichkeiten, Anforderungen in Bezug auf Fehlermeldungen und schließlich Anforderungen, welche direkt die Transformation betreffen.

Danach wurde mit der *Implementierung* begonnen. Hier war es nötig zunächst die Programmierschnittstellen festzulegen, mit denen die Software später aus anderen Programmen heraus genutzt werden kann. Anschließend wurde die Transformation von Ecore-Metamodellen nach grUML-Schemas realisiert. Es wurde sowohl die Verwendung einer Konfigurationsdatei als auch der Kommandozeile ermöglicht. Zum Schluss wurde die Implementierung der Transformation von passenden Ecore-Modellen nach Graphen fertiggestellt.

Während der Implementierung fiel auf, dass die Anforderung F12 nicht ausreichend durchdacht war. Sie lautete: “Die Software soll die Option bieten, das Wurzelelement des Ecore-Metamodells in die Graphklasse des grUML-Schemas zu transformieren.” Die Anforderung

wurde nun so realisiert, dass eine `EClass` angegeben werden kann, die dann zur Graphklasse konvertiert wird. Eine automatische Bestimmung des Wurzelelements wurde nicht implementiert. Ein Grund dafür war, dass bei der Transformation einer `EClass` in eine `GraphClass` alle `EReferences` dazwischen verloren gehen. Da nicht automatisch erkennbar ist, ob die `EReferences` Semantik besitzen, die über die reine Navigierbarkeit hinaus geht, kann dies unerwünscht sein. Den zweiten Grund lieferten konzeptuelle Kantenklassen. Ein Metamodell könnte in einer Form modelliert sein, in der eine `EClass` alle anderen Klassen über transitive `Containment`-Beziehungen enthält, die konzeptuellen Kantenklassen jedoch nicht.

Zuletzt wurden einige Beispiel-Metamodelle einschließlich passender Modellen mit Hilfe der Software transformiert und das Ergebnis untersucht. Dabei wurden sowohl selbst erstellte Beispiele als auch im Internet gefundene genutzt. Die Beispiele zeigten, dass die Transformation im Rahmen der Entwurfsentscheidungen korrekt abläuft.

10.2 Ausblick

Im Folgenden wird beschrieben, wie diese Arbeit in Zukunft fortgeführt werden könnte und welche Aspekte noch betrachtet werden sollten.

Die Software, die zu dieser Arbeit realisiert wurde, ermöglicht die automatische Transformation eines Ecore-Metamodells in ein grUML-Schema, sowie die Transformation eines passenden Ecore-Modells in einen Graphen. Um diese Implementierung zum Austausch zwischen den beiden Modellierungssprachen sinnvoll nutzen zu können, ist jedoch noch die Realisierung des umgedrehten Schrittes notwendig. Die Grundlage dafür wurde bereits gelegt. In Kapitel 5 ist bereits untersucht worden, welche Elemente zu Schwierigkeiten führen können und Abschnitt 7.4 zeigt, wie damit grundsätzlich umgegangen werden kann. Eine Fortführung dieser Arbeit würde also die Transformation von grUML-Schemas in Ecore-Metamodelle, sowie passende Ecore-Modelle in Graphen, realisieren.

Um eine möglichst verlustfreie Hin- und Rücktransformation zu ermöglichen, sollte dann auch die Auswertung von `EAnnotations` und Kommentaren der beiden Programme aufeinander abgestimmt werden. Im Anschluss sollten die Ergebnisse mehrmaliger Hin- und Rücktransformationen untersucht und in Bezug auf Informationsverlust bewertet werden.

Literatur

[App] Apple. The plist manual page. URL: http://developer.apple.com/mac/library/documentation/Darwin/Reference/ManPages/man5/plist.5.html. Zuletzt besucht: 22.09.2010.

[BHR+10] Daniel Bildhauer, Tassilo Horn, Volker Riediger, Hannes Schwarz, and Sascha Strauß. *grUML - A UML based modeling language for TGraphs*, 2010. unpublished.

[EB10] Jürgen Ebert and Daniel Bildhauer. Reverse Engineering Using Graph Queries. In Andy Schürr, Claus Lewerentz, Gregor Engels, Wilhelm Schäfer, and Bernhard Westfechtel, editors, *Graph Transformations and Model Driven Engineering*, LNCS 5765. Springer, 2010. to appear.

[HE10] Tassilo Horn and Jürgen Ebert. The GReTL Transformation Language. Technical report, University Koblenz-Landau, Institute for Software Technology, 2010. unpublished, draft at http://www.uni-koblenz.de/~horn/gretl.pdf.

[lit] Ecore api. URL: http://download.eclipse.org/modeling/emf/emf/javadoc/2.5.0/org/eclipse/emf/ecore/package-summary.html. Zuletzt besucht: 22.09.2010.

[SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional, 2 edition, 12 2008.

[Sch08] Gerhard Schmidt. *GRAFCET - Grundsätzliches zum Verständnis*. Festo Didactic GmbH & Co KG, 73770 Denkendorf, 2008.

[SV06] Thomas Stahl and Markus Voelter. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 1 edition, 5 2006.