universität wien

DIPLOMARBEIT

Titel der Diplomarbeit

Evaluierung von Clio zur Transformation von Metamodellen

Verfasser

Oliver Motschiunigg

angestrebter akademischer Grad

Magister der Sozial- und Wirtschaftswissenschaften (Mag. rer. soc. oec.)

Wien, 2008

Studienkennzahl laut Studienblatt:	A 175
Studienrichtung laut Studienblatt:	Wirtschaftsinformatik
Betreuerin:	o. Univ.-Prof. Mag. Dipl.-Ing. Dr. Gerti Kappel
Mitbetreuer:	Dr. Horst Kargl

Danksagung

Ich bedanke mich bei meiner Familie, meinen Freundinnen und meinen Freunden.

Kurzfassung

Clio ist ein Tool zur teilautomatischen Erzeugung von Schema Mappings und der anschließenden Transformation der Instanz eines Quellschemas in die Instanz eines Zielschemas.

Ein Metamodell ist das Modell eines Modells und dient zur Beschreibung seiner Elemente und ihrer Beziehungen zueinander. *Ecore* ist eine Implementierung der *Meta Object Facility*, der standardisierten Sprache der *Object Management Group* (OMG) zur Beschreibung von Metamodellen.

Diese Arbeit untersucht Clio in Anwendung auf Ecore-basierte Metamodelle. Es soll festgestellt werden, ob ein Einsatz von Clio zur Transformation dieser Metamodelle möglich und sinnvoll ist. Dabei wird die Bedienung Clios mit besonderem Augenmerk auf den notwendigen Input untersucht. Anschließend wird eine Methode entwickelt, um Metamodelle entsprechend umzuformen. Schließlich werden diese umgeformten Metamodelle verwendet, um sie mit Clio zu transformieren.

Abstract

Clio is a tool for the semi-automatic generation of schema mappings and the following transformation of an instance of a source schema into the instance of a target schema.

A metamodel is the model of a model. It is used to describe model elements and their relationships to each other. *Ecore* is an implementation of the *Meta Object Facility* – the standardized language of the *Object Management Group (*OMG) for the description of metamodels.

This thesis evaluates the application of Clio to Ecore-based metamodels. The goal is an evaluation of the pros and cons of using Clio as a tool for the transformation of Ecore-based metamodels. Therefore it is necessary to examine how to use Clio focusing on the required input. Subsequently, a method to translate metamodels is developed. Finally, Clio is used to transform these metamodels.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Einleitung

Datenintegration und -transformation sind zeitaufwendige und fehleranfällige Prozesse, die hauptsächlich manuell von ExpertInnen durchgeführt werden müssen. In den letzten Jahrzehnten wurden zahlreiche Methoden zur automatischen bzw. semiautomatischen Integration von Daten entwickelt. Einer dieser Ansätze ist die Erzeugung der Abbildung eines Quellschemas auf ein Zielschema durch Wertkorrespondenzen, genannt *Schema Mapping*.

Clio ist ein Tool zur teilautomatischen Erzeugung solcher Schema Mappings. Es untersucht die Struktur von Quell- und Zielschemata und verwendet von der/dem BenutzerIn erzeugte Wertkorrespondenzen, die beschreiben, wie der Wert eines Zielattributs aus einer Menge von Werten von Quellattributen abgebildet werden kann. Dabei erstellt Clio in Echtzeit Transformationsabfragen, mit denen Instanzen des Quellschemas in entsprechende Instanzen des Zielschemas transformiert werden können.

Die vorliegende Arbeit untersucht Clio in Anwendung auf Metamodelle, die auf *Ecore* basieren. Dabei soll evaluiert werden, ob sich Clio zur Transformation von Ecore-Metamodellen eignet.

1.1 Ziel

In dieser Arbeit soll festgestellt werden, ob und in welchem Ausmaß es möglich ist, Clio zur Transformation von auf Ecore basierenden Metamodellen zu verwenden. Das Ergebnis der Arbeit ist eine Evaluierung des Einsatzes von Clio für die Trans-

formation von Metamodellen. Dabei wird im Besonderen untersucht, wie sich auf Ecore basierende Metamodelle verlustfrei in Schemata überführen lassen, die von Clio bearbeitet werden können, bzw. inwieweit dabei Kompromisse eingegangen werden müssen.

1.2 Vorgangsweise

Nach Erarbeitung der notwendigen theoretischen Grundlagen wird die Bedienung von Clio anhand eines einfachen Testbeispiels erörtert. Dabei soll insbesondere festgestellt werden, in welcher Form die Inputdaten zur Verfügung stehen müssen, damit sie von Clio bearbeitet werden können. Im Anschluss soll Clios Funktionalität detailliert auf Basis diverser Testmodelle geprüft werden. Schließlich wird untersucht, ob und wie sich Ecore-Metamodelle in von Clio lesbare Daten transformieren lassen. Dabei soll ein Algorithmus entwickelt werden, der Metamodelle möglichst automatisch in die notwendigen Schemata umwandelt. Diese werden abschließend verwendet, um die Transformation von Metamodellen mit Hilfe von Clio zu evaluieren.

Kapitel 2

Theoretische Grundlagen

Dieses Kapitel beschreibt die für die vorliegende Arbeit notwendigen theoretischen Grundlagen. Konkret wird auf Schema Mapping und Ecore-Metamodelle eingegangen. Außerdem sollen die notwendigen XML-Grundlagen zusammengefasst werden.

2.1 Datentransformation mittels Schema Mapping

Laut [NL07] bezeichnet der Begriff Schema Mapping (oder Schema-Abbildung) zwei Dinge: Zum einen ist ein Schema Mapping eine Menge von Korrespondenzen zwischen Attributen unterschiedlicher Schemata, die als Wertkorrespondenzen bezeichnet werden. Zum anderen bezeichnet Schema Mapping einen komplexen Prozess, der ausgehend von Wertkorrespondenzen komplexere Schemakorrespondenzen und schließlich Datentransformationsvorschriften ableitet.

Die Transformation von Daten von einem Schema in ein anderes ist ein Vorgang, mit dem man bei der Wartung und Migration von Informationssystemen ständig konfrontiert wird. In der Praxis wird die Datentransformation von einem Schema in ein anderes hauptsächlich manuell von ExpertInnen durchgeführt; sie schreiben Anfragen oder Programme, mit denen die Daten der Quelle extrahiert, transformiert und in das Zielschema eingeführt werden. Wegen der Komplexität der zugrundeliegenden Schemata ist dieser Prozess sehr aufwändig und fehleranfällig. Schema Mapping ist eine wesentliche Unterstützung bei der Erstellung dieser Transformationen.

Im Bereich der Informationsintegration kann im Allgemeinen zwischen drei Anwendungsgebieten von Schema Mappings unterschieden werden: Datentransformation,

Schemaintegration und Schemaevolution. In der vorliegenden Arbeit wird lediglich die Datentransformation mit Hilfe von Schema Mappings betrachtet, die das Mapping verwendet, um ein Quellschema mittels einer aus ihm generierten Transformationsabfrage in Daten des Zielschemas umzuwandeln [Leg05].

2.2 Ecore-Metamodelle

Ein Metamodell ist das Modell eines Modells – es beschreibt die Elemente eines Modells und deren Beziehungen zueinander. Ein Modell stellt somit eine Instanz eines Metamodells dar.

Die *Meta Object Facility* (MOF) ist die standardisierte Sprache der *Object Managment Group* (OMG) [1] für die Beschreibung von Metamodellen [MOF04]. *Ecore* ist eine Implementierung von MOF, die vom *Eclipse Modeling Framework* (EMF)[2] zur Verfügung gestellt wird [WSK+06]. Die in der vorliegenden Arbeit verwendeten Metamodelle sind in Ecore modelliert. Auf die notwendigen Ecore-Elemente wird in Kapitel 5 eingegangen.

XML Metadata Interchange (XMI) ist ein Standard der OMG für den Austausch von Metadaten im XML-Format.

2.3 Die XML-Familie

Dieses Kapitel beschreibt die für die vorliegende Arbeit notwendigen XML-Standards und die Begriffe *XML*, *XML Schema*, *XPath*, *XQuery* und *XSLT*.

2.3.1 XML

Die *Extensible Markup Language* (XML) ist ein vom *World Wide Web Consortium* (W3C)[3] spezifiziertes, standardisiertes Textformat zur Darstellung und für den Austausch hierarchisch strukturierter Daten [BPS+06]. XML stellt eine Teilmenge der *Standard Generalized Markup Language* (SGML) dar.

---

[1] http://www.omg.org/

[2] http://www.eclipse.org/emf

[3] http://www.w3.org/

Prinzipiell handelt es sich bei XML um eine Formatbeschreibungssprache für Textdateien. Informationen in diesen Dateien werden mit sogenannten Tags umschlossen, wodurch eine Strukturierung der Daten erfolgt [NL07].

2.3.2 XML Schema

XML Schema ist ein W3C-Standard zur Spezifizierung der Struktur von XML-Dateien [siehe BM01, FW04 und TMB+04]. Mit Hilfe von XML Schema werden die in einem XML-Dokument erlaubten Elemente und Attribute, die Verschachtelungsstruktur und Häufigkeit von Elementen, Datentypen für Werte sowie Schlüssel und Fremdschlüssel definiert [NL07].

2.3.3 XPath

Die *XML Path Language* (XPath) ist eine vom W3C entwickelte Abfragesprache, um Teile von XML-Dokumenten zu selektieren [CD99 und RSC+07]. Ausgehend von der Betrachtung eines XML-Dokuments in Baumstruktur erlaubt XPath die Navigation in diesem Baum entlang seiner Knoten.

2.3.4 XQuery

Die *XML Query Language* (XQuery) ist eine vom W3C spezifizierte Abfragesprache für XML-Dokumente [RSB+07]. Eine zentrale Rolle in XQuery stellen die *FLWOR-Ausdrücke* dar, die aus einer FOR-Klausel, einer WHERE-Klausel, einer ORDER-BY-Klausel und einer RETURN-Klausel bestehen. FLWOR-Ausdrücke werden zur Abfrage von XML-Dokumenten sowie zur Strukturierung von Abfrageergebnissen verwendet.

2.3.5 XSLT

Die *Extensible Stylesheet Language for Transformations* (XSLT) ist Teil der *Extensible Stylesheet Language (XSL)* und stellt eine Sprache zur Umformung von XML-Dokumenten dar [Cla99 und Kay07]. XSLT-Programme bestehen aus Transformationsregeln, die auf die Elemente von XML-Dokumenten angewendet werden. Zur Selektion der Elemente werden XPath-Ausdrücke verwendet. XSLT-Programme sind selbst wiederum XML-Dokumente [NL07].

Kapitel 3

Clio und seine Handhabung

Clio ist ein Tool zur teilautomatischen Erstellung von Schema Mappings. Das Clio-Projekt wurde 1999 gemeinsam von der *University of Toronto* und dem *IBM Almaden Research Center* ins Leben gerufen. Mittlerweile hat die Clio-Technologie über den Status eines Entwicklungsprototypen hinaus Einzug in IBMs Unternehmenssoftware zur Datenmodellierung und Datenintegration *Rational Data Architect* gefunden [Mil07].

Clio war das erste System, das sich sowohl dem Problem der teilautomatischen Generierung von Schema-Mappings als auch dem nachfolgenden Problem der Benutzung dieser Mappings zur Generierung von Abfragen widmete. Ursprünglich wurde Clio nur für relationale Schemata entwickelt, die in XML Schema oder über eine relationale Datenbank zur Verfügung gestellt werden mussten. Mittlerweile handelt es sich um ein System für die Generierung von Mappings und Transformationen sowohl von relationalen als auch von hierarchisch modellierten, verschachtelten XML Schemata [HHH+05].

Die Spezifizierung der Schema-Mappings erfolgt durch die Erstellung von Wertkorrespondenzen (*engl. value correspondences*) durch den/die BenutzerIn. Laut [MHH00] ist eine Wertkorrespondenz ein Paar bestehend aus 1) einer Funktion, die definiert, wie ein Wert (oder eine Kombination von Werten) aus dem Quellschema benutzt werden kann, um einen Wert im Zielschema abzubilden, und 2) einem Filter, der angibt, welche Quellwerte dazu verwendet werden sollen. Wertkorrespondenzen spezifizieren also, wie ein Zielattribut aus einem oder mehreren Quellattributen generiert werden kann. Anhand dieser Wertkorrespondenzen erzeugt Clio eine Repräsen-

tation des Schema-Mappings als Abfrage. Abhängig von der Art der verwendeten Schemata generiert Clio Abfragen in XQuery, XSLT, SQL und SQL/XML [HHH+05 und PHV+02].

Dieses Kapitel soll einen Überblick über die grundsätzliche Funktionsweise von Clio geben. Im Gegensatz zu den existierenden Unterlagen zu und Beschreibungen von Clio (siehe [HHH+05, MHH00, MHH+01, Mill07, PHV+02]) wird aufgrund der Anforderungen für die nachfolgenden Kapitel insbesondere die Bedienung von Clio untersucht. In Bezug auf das in Abschnitt 3.1 beschriebene Testset erfolgt in Abschnitt 3.2 die Untersuchung von Clios Komponente zum Laden von Schemata (*engl. schema loader*). Abschnitt 3.3 befasst sich mit der Erstellung und Bearbeitung von Wertkorrespondenzen und den resultierenden Schema-Mappings mittels der *Mapping-Engine* bevor in Abschnitt 3.4 Clios *Query-Engine* untersucht wird, die für die Erzeugung der Abfragen verantwortlich ist. In Abschnitt 3.5 folgen einige Anmerkungen zur Handhabung von Clio.

Sämtliche Untersuchungen für diese Arbeit wurden mit Clio in der Version 02-05-2008 durchgeführt, die JRE[4] Version 1.4.2 oder höher benötigt.

3.1 Aufbau des Testsets und Beschreibung verwendeter Werkzeuge

Um Clios Handhabung zu überprüfen, wurde ein Testset erstellt, das aus je einem Quell- und Zielschema sowie einer Instanz des Quellschemas, dessen Daten in die Form des Zielschemas transformiert werden sollen, besteht. Sowohl das Quell- als auch das Zielschema bilden eine einfache Struktur einer Firma mit ihren Abteilungen und MitarbeiterInnen ab.

[4] Java Runtime Environment, www.java.com

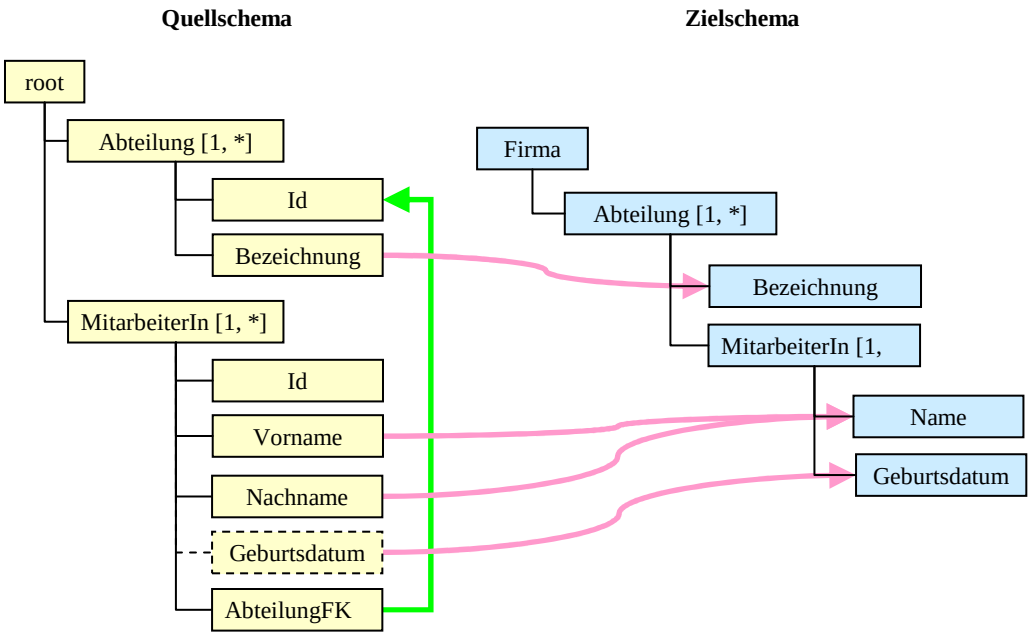

Abbildung 3.1: Quellschema, Zielschema und Wertkorrespondenzen

3.1.1 Das Quellschema – relationale Struktur

Im Quellschema werden die MitarbeiterInnen ihren Abteilungen über eine Fremdschlüsselbeziehung zugeordnet. Um diese abzubilden, erhält sowohl das Element *Abteilung* als auch das Element *MitarbeiterIn* ein Attribut mit der Bezeichnung *Id*. Listing 3.1 zeigt die Definition der Fremdschlüsselbeziehung für das Quellschema als XML Schema.

```
<xs:key name="AbteilungID">
    <xs:selector xpath="./Abteilung"/>
    <xs:field xpath="@Id"/>
</xs:key>
<xs:keyref name="AbteilungFK" refer="AbteilungID">
    <xs:selector xpath="./MitarbeiterIn"/>
    <xs:field xpath="AbteilungFK"/>
</xs:keyref>
```

Listing 3.1: Definition der Fremdschlüsselbeziehung

3.1.2 Das Zielschema – eingebettete Struktur

Für das Zielschema wurde eine eingebettete Struktur gewählt. Die Elemente *MitarbeiterIn* sind Kindesknoten ihres jeweiligen Elternknotens *Abteilung*, wodurch eine hierarchische Abbildung geschaffen wird.

3.1.3 Instanz des Quellschemas

Listing 3.2 zeigt eine Instanz des Quellschemas aus Abbildung 3.1. Diese enthält zwei Abteilungen und drei MitarbeiterInnen. Letztere sind ihren Abteilungen gemäß der Festlegung im Quellschema per Fremdschlüssel zugeordnet. Diese Beispielinstanz soll mit Clio in die Struktur des Zielschemas transformiert werden.

```
<?xml version="1.0" encoding="UTF-8"?>
<root xsi:noNamespaceSchemaLocation="abteilung_relational.xsd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <Abteilung Id="1">
       <Bezeichnung>Verkauf</Bezeichnung>
   </Abteilung>
   <Abteilung Id="2">
       <Bezeichnung>Logistik</Bezeichnung>
   </Abteilung>
   <MitarbeiterIn Id="1">
       <Vorname>Manfred</Vorname>
       <Nachname>Moser</Nachname>
       <Geburtsdatum>1984-06-13</Geburtsdatum>
       <AbteilungFK>1</AbteilungFK>
   </MitarbeiterIn>
   <MitarbeiterIn Id="2">
       <Vorname>Margit</Vorname>
       <Nachname>Müller</Nachname>
       <Geburtsdatum>1967-01-17</Geburtsdatum>
       <AbteilungFK>2</AbteilungFK>
   </MitarbeiterIn>
   <MitarbeiterIn Id="3">
       <Vorname>Simone</Vorname>
       <Nachname>Egger</Nachname>
       <Geburtsdatum>1975-10-20</Geburtsdatum>
       <AbteilungFK>1</AbteilungFK>
   </MitarbeiterIn>
</root>
```

Listing 3.2: Beispielinstanz des Quellschemas

3.1.4 Mapping des Quellschemas auf das Zielschema

Abbildung 3.1 zeigt die Wertkorrespondenzen zwischen Quell- und Zielschema. Damit die Quelldaten dem Zielschema entsprechen, müssen Elemente vom Typ *MitarbeiterIn* nach *Abteilung* gruppiert und *Bezeichnung*, *Geburtsdatum*, *Vorname* und *Nachname* übertragen werden, wobei die beiden letzteren mit Hilfe einer Funktion zu *Name* im Zielschema zusammengefügt werden sollen.

3.1.5 Hilfswerkzeuge

Zur Ausführung der XQuery- und XSLT-Scripts wurde das Tool Saxon[5] verwendet. Saxon ist ein XQuery- und XSLT-Prozessor, der in Java geschrieben wurde und über die Kommandozeile ausgeführt wird. Zur Benutzung von Saxon ist JRE 1.4 notwendig, wobei JRE 1.5 empfohlen wird. Für diese Arbeit wurde zur Ausführung der XQuery-Scripts die neueste Saxon-Version 9.1 verwendet. Für die XSLT-Scripts kam Version 6.5.5 zum Einsatz, da Version 9.1 nur mehr XSLT-2.0-Abfragen transformieren kann, von Clio aber XSLT-1.0-Scripts erzeugt werden.

Für eine zusätzliche Verifizierung der mit Saxon erzielten Ergebnisse wurden die Tools XMLSpy[6] für die XQuery- und Xalan[7] für die XSLT-Transformationen verwendet.

3.1.6 Ausführung der Clio-Abfragen

Sowohl für die Transformationsabfragen in XQuery als auch in XSLT gilt, dass sie nicht direkt in Clio ausgeführt werden können, obwohl dessen Interface diese Funktionalität vorsieht. Aus diesem Grund wurden die Transformationen mit Saxon durchgeführt. Für die Bedienung von Saxon wurden Scripts erstellt, die in den nachfolgenden Abschnitten beschrieben werden. Zunächst folgt eine allgemeine Namenskonvention für die Variablen bzw. Dateinamen, die in den Scripts verwendet werden:

- Die Variable *%TESTSETPATH%* enthält den Pfad, in dem sich die Inputdateien befinden.
- Die Variable *%TESTSET%* enthält den Namen des Transformationsbeispiels.
- Der Dateiname für die Beispielinstanz lautet *%TESTSET%.xml*.
- Der Dateiname für das XQuery Script ist *%TESTSET%.xquery*.
- Die Dateinamen für die XSLT Scripts lauten *%TESTSET%_script1.xslt* und *%TESTSET%_script2.xslt*.
- Das Transformationsergebnis für die XQuery-Abfrage wird in *%TESTSET%_ergebnis_xquery.xml* erzeugt.

---

5 Saxon v6.5.5 bzw. v9.1.0.1, saxon.sourceforge.net

6 Altova XMLSpy v2008 rel. 2, www.altova.com

7 Xalan Java v2.7.1, xml.apache.org/xalan-j/

- Das Transformationsergebnis für die XSLT-Abfrage wird in *%TESTSET%_ergebnis_xslt.xml* erzeugt.

3.1.6.1 Ausführung der XQuery-Abfragen

Will man die XQuery-Abfragen über Clios *Execute-Query*-Button ausführen, versucht Clio die Daten an das externe Tool Quip zu übergeben, das in C:\quip gesucht wird. Quip ist ein Prototyp eines XQuery-Prozessors, der von der Firma SoftwareAG[8] entwickelt wurde. Mittlerweile steht Quip nicht mehr als freier Download zur Verfügung, weshalb für die Tests der vorliegenden Arbeit lediglich die veraltete Version 1.4.1 verwendet werden konnte, die Clios XQuery-Abfragen nicht interpretieren kann.[9]

Angelehnt an oben festgelegte Namenskonvention werden die XQuery-Transformationen mit Saxon 9.1 und folgendem Script durchgeführt:

```
@set TESTSETPATH= ...
@set TESTSET= ...

java -cp saxon9.jar net.sf.saxon.Query
 -q:%TESTSETPATH%%TESTSET%.xquery -s:%TESTSETPATH%%TESTSET%.xml
 -o:%TESTSETPATH%%TESTSET%_ergebnis_xquery.xml
```

Listing 3.3: Script zur Ausführung der XQuery-Transformation mit Saxon 9.1

3.1.6.2 Ausführung der XSLT Abfragen

```
@set JAVACLASSPATH=xalan.jar;serializer.jar;
 xml-apis.jar;xercesImpl.jar
@set TESTSETPATH= ...
@set TESTSET= ...

java -jar saxon.jar -o %TESTSETPATH%%TESTSET%_schritt1.xml
 %TESTSETPATH%%TESTSET%.xml
 %TESTSETPATH%%TESTSET%_script1.xslt

java -jar saxon.jar -o
 %TESTSETPATH%%TESTSET%_ergebnis_xslt.xml
 %TESTSETPATH%%TESTSET%_schritt1.xml
 %TESTSETPATH%%TESTSET%_script2.xslt
```

Listing 3.4: Script zur Ausführung der XSLT Transformation mit Saxon 6.5.5

Auch für die XSLT-Scripts gilt, dass eine direkte Durchführung der Abfrage über Clio nicht möglich ist. Diese Funktionalität scheint überhaupt nicht implementiert zu

[8] www.softwareag.com

[9] Laut Clio Hilfe wurden diese mit *Quip* in der Version 2.2.1 getestet, das diese korrekt verarbeitete.

sein, weshalb die XSLT-Transformationen mit Saxon 6.5.5 und dem Script aus Listing 3.4 durchgeführt werden:

3.2 Input – Laden von Quell- und Zielschemata

Eine typische Clio-Session beginnt damit, Quell- und Zielschema zu laden. Dabei muss mindestens je ein Quell- und Zielschema angegeben werden, wobei jede Kombination von Schemata aus relationalen Datenbanken und XML Schema unterstützt wird[10]. Letztere wiederum können sowohl dem relationalen Schema entsprechen als auch eingebettete Strukturen (*engl. nested schemas*) beinhalten. Die geladenen Schemata werden von Clios Schema-Engine in eine interne Darstellungsweise transformiert und in der *Schema View* abgebildet, von wo aus Wertkorrespondenzen erstellt und manipuliert werden können [Cli08].

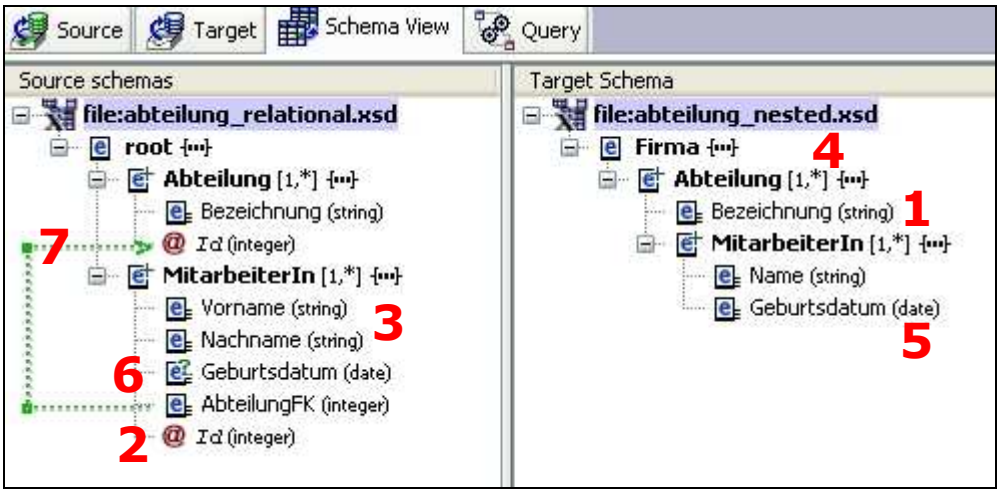

Abbildung 3.2: Darstellung des geladenen Quell- und Zielschemas in Clio

Abbildung 3.2 zeigt Clios Zustand, nachdem Quell- und Zielschema geladen wurden. Die Fremdschlüsselrelation *AbteilungFK* auf *Id* in der Abteilungstabelle des Quellschemas wurde erkannt und entsprechend abgebildet. Auf Seite des Zielschemas wiederum wird die eingebettete Datenstruktur hierarchisch dargestellt. Beide Strukturen modellieren die selbe Situation, nämlich eine Umgebung, in der ein/eine MitarbeiterIn Mitglied von einer oder mehreren Abteilungen ist.

[10] Diese Arbeit beschäftigt sich ausschließlich mit Daten aus XML Schemata, da die zu untersuchenden Ecore-Metamodelle in solche transformiert werden.

In Bezugnahme auf Abbildung 3.2 folgt eine Aufzählung der strukturellen Komponenten, die von Clios Schema-Engine erkannt werden. Für eine bessere Übersicht werden zusätzlich Codefragmente aus den zugehörigen XML Schemata angegeben.

- **1 – Elemente** werden wie das Element *Bezeichnung* im Zielschema dargestellt.

```
<xs:element name="Bezeichnung">
```

- **2 - Attribute** werden wie das Attribut *Id* im Quellschema dargestellt.

```
<xs:attribute name="Id">
```

- **3 - Komplexe Datentypen:** Elemente und Attribute innerhalb von komplexen Datentypen nach dem *xsd:sequence* oder dem *xsd:all* Modell werden hierarchisch unter deren Elternknoten aufgelistet.

```
<xs:element name="MitarbeiterIn" maxOccurs="unbounded">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Vorname"/>
            <xs:element name="Nachname"/>
            ...
        </xs:sequence>
    ...
</xs:element>
```

- **4 - Kardinalitäten** werden erkannt und in eckigen Klammern angegeben.

```
<xs:element name="Abteilung" minOccurs="1"
 maxOccurs="unbounded">
```

- **5 - Datentypen** werden in Klammern angegeben. Falls für ein Element kein Datentyp angegeben wurde, wird ihm automatisch der Typ *String* zugeordnet.

```
<xs:element name="Geburtsdatum" type="xs:date"/>
```

- **6 - Optionale Elemente bzw. Attribute** werden mit einem Fragezeichen markiert; siehe Element *Geburtsdatum* im relationalen Modell.

```
<xs:element ... minOccurs="0" maxOccurs="1"/>
<xs:attribute ... use="optional"/>
```

- **7 - Referenzen über Schlüssel:** Fremdschlüsselreferenzen werden erkannt und mit einem grünen Graphen markiert. Schlüssel, die in keiner Referenz enthalten sind, werden nicht besonders hervorgehoben.

```
<xs:key name="AbteilungID">
    <xs:selector xpath="./Abteilung"/>
    <xs:field xpath="@Id"/>
</xs:key>
<xs:keyref name="AbteilungFK" refer="AbteilungID">
    <xs:selector xpath="./MitarbeiterIn"/>
    <xs:field xpath="AbteilungFK"/>
</xs:keyref>
```

Darüber hinaus erkennt Clio noch folgende strukturelle Komponenten, die in Abbildung 3.3 ersichtlich sind und lediglich der Vollständigkeit halber erwähnt werden:

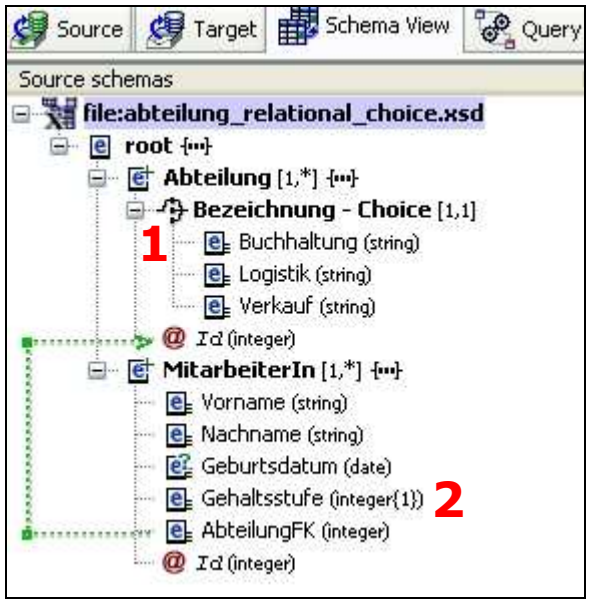

Abbildung 3.3: Choice Constraint und Standardwert

- **1 - Choice Constraints:** Komplexe Datentypen nach dem *xsd:choice*-Modell werden als solche erkannt und als spezielle Form von Sequenz angezeigt.

```
<xs:element name="Bezeichnung">
    <xs:complexType>
        <xs:choice>
            <xs:element name="Buchhaltung"/>
            <xs:element name="Logistik"/>
            <xs:element name="Verkauf"/>
        </xs:choice>
    </xs:complexType>
</xs:element>
```

- **2 - Standardwerte** werden nach dem jeweiligen Datentyp in geschwungenen Klammern angezeigt.

```
<xs:element name="Gehaltsstufe" ... default="1"/>
```

3.3 Schema-Mapping – Clios Mapping-Komponente

Abbildung 3.4 zeigt Clio nach Erstellung der Wertkorrespondenzen, indem Werte aus dem Quellschema Werten aus dem Zielschema zugewiesen werden. Die erste Wertkorrespondenz fordert, dass für jede Abteilungsbezeichnung im Quellschema eine Abteilung mit gleicher Bezeichnung im Zielschema existieren muss. Die zweite Wertkorrespondenz bildet Vor- und Nachnamen der MitarbeiterInnen aus dem Quellschema auf deren Namen im Zielschema ab. Aufgrund der Abhängigkeit der MitarbeiterInnen von ihren Abteilungen sowohl im Quellschema über die Fremdschlüsselbeziehung als auch im Zielschema über die eingebettete Struktur sind diese Wertkorrespondenzen ohne entsprechende Gruppierung nicht aussagekräftig. Diese Gruppierung wird von Clio erkannt und automatisch erstellt.

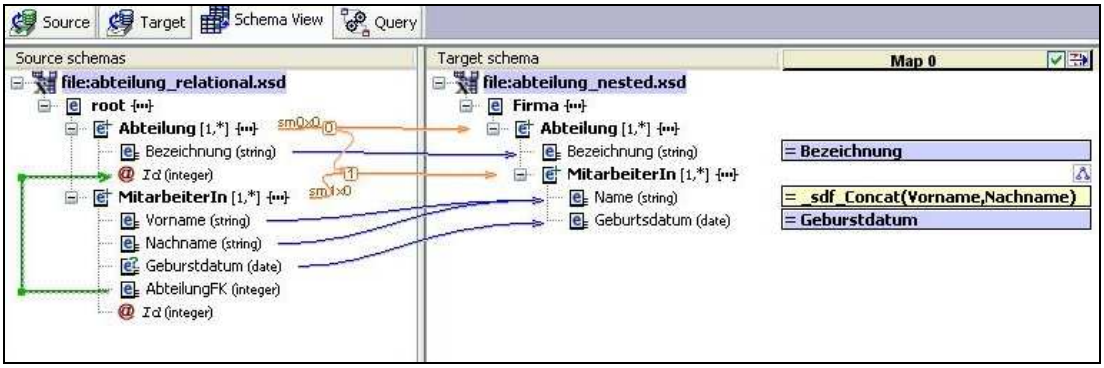

Abbildung 3.4: Wertkorrespondenzen in Clio

Falls Clios Mapping Engine erkennt, dass die Abbildungsfunktion einer Wertkorrespondenz keine Eins-zu-eins-Funktion ist, versucht es, automatisch eine komplexe Transformationsfunktion einzufügen. Die Funktion *_sdf_Concat()*[11], die von Clio nach Erstellung der entsprechenden Wertkorrespondenz selbständig eingefügt wurde,

[11] Hinweis: Die Funktion *_sdf_Concat()* kann weder von den getesteten XQuery- noch von den XSLT-Prozessoren interpretiert werden. Eventuell handelt es sich dabei um die Funktion einer Sprache, die für Clio entwickelt wurde und von entsprechenden IBM-Produkten weiterverarbeitet werden kann. In dieser Arbeit wird davon ausgegangen, dass sie das korrekte Ergebnis in Form einer Verknüpfung der übergebenen Parameter liefert.

erzeugt eine Verknüpfung (*engl. concatenation*) für die Abbildung *Vorname* und *Nachname* aus dem Quellschema auf *Name* im Zielschema. Weiters erkennt Clio Wertkorrespondenzen für Attribute mit unterschiedlichen Datentypen und fügt die Funktion *convert()* automatisch ein. Zusätzlich können komplexe Transformationsfunktionen manuell mit dem Expression Editor erstellt und bearbeitet werden.

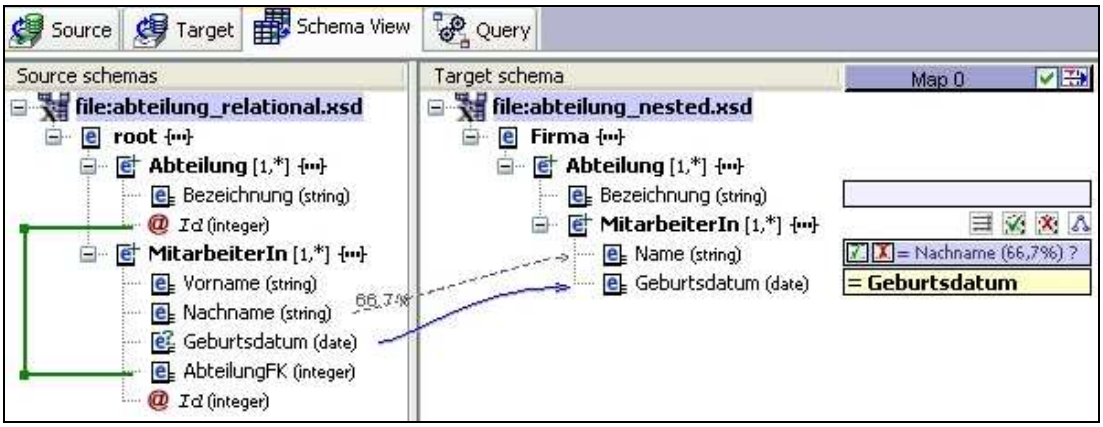

Abbildung 3.5: Clios Attribut-Matcher macht einen Vorschlag

Schlussendlich ist Clio mit einem Attribut-Matcher ausgestattet. In Abbildung 3.5 ist dargestellt, wie dieser eine Wertkorrespondenz von *Nachname* aus dem Quellschema auf *Name* im Zielschema vorschlägt. Solche Vorschläge können von der/dem BenutzerIn akzeptiert oder abgelehnt werden.

3.4 Output – Clios Query-Engine

Wie bereits eingangs erwähnt wurde, erzeugt Clio Transformationsscripts in den Formaten XQuery, XSLT, SQL und XML/SQL. Für diese Arbeit kommen lediglich Schemata im XML Schema-Format zum Einsatz, weshalb sich meine Untersuchungen auf Clios XQuery- und XSLT-Abfragen beschränken.

Anhand der Beispielinstanz aus Listing 3.2 wird in diesem Abschnitt Clios Output untersucht.

3.4.1 XQuery-Transformation

Clio erzeugt XQuery-Transformationsabfragen, die weder syntaktisch noch semantisch korrekt sind. Grund dafür ist Clios Verwendung einer veralteten XQuery-Spezifikation. Laut [Leg05] ist sich das Clio-Entwicklungsteam dieses Problems bewusst und kündigt an, dass zukünftige Clio-Versionen XQuery-Code erzeugen, der

aktuellen Standards entspricht. Bis zur vorliegenden Version wurde diese Ankündigung allerdings nicht eingehalten, weshalb es notwendig ist, die XQuery-Abfragen manuell anzupassen.

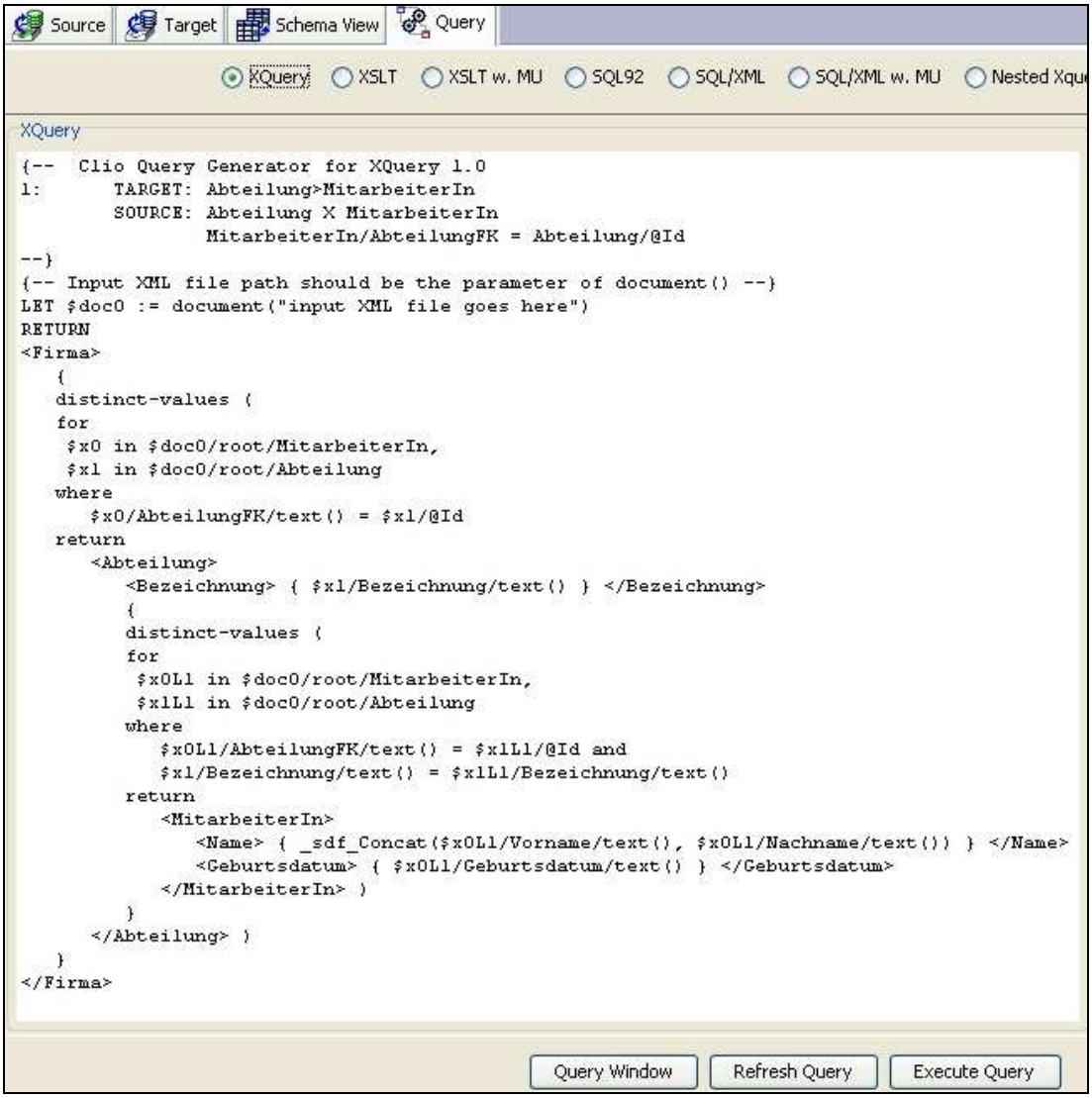

Abbildung 3.6: Schema-Mapping als XQuery-Transformationsabfrage

Abbildung 3.6 zeigt den Transformationscode für das Testset aus Abschnitt 3.1 in Clios XQuery-Viewer. Dieser Code enthält syntaktische und semantische Fehler, die in der Folge beschrieben und teilweise korrigiert werden.

3.4.1.1 Syntaktische Fehler[12]

- Die Kommentare sind syntaktisch falsch. Statt Verwendung der Schlüsselwörter *{--* und *--}* müssen die Kommentare mit *(:* eingeleitet und mit *:)* beendet werden.

- Die Schlüsselwörter *LET* und *RETURN* müssen klein geschrieben werden.

- Das zu transformierende XML-Dokument muss in der Funktion *doc()* anstatt der verwendeten Funktion *document()* angegeben werden.

- Die Funktion *_sdf_Concat()* muss in die XPath-Funktion *concat()* umbenannt werden.

```
let $doc0 := doc("abteilung.xml")
return
<Firma>
    {
    distinct-values (
    for
        $x0 in $doc0/root/MitarbeiterIn,
        $x1 in $doc0/root/Abteilung
    where
        $x0/AbteilungFK/text() = $x1/@Id
    return
        <Abteilung>
            <Bezeichnung>
                { $x1/Bezeichnung/text() }
            </Bezeichnung>
            {
            distinct-values (
            for
                $x0L1 in $doc0/root/MitarbeiterIn, $x1L1 in
                 $doc0/root/Abteilung
            where
                $x0L1/AbteilungFK/text() = $x1L1/@Id and
                 $x1/Bezeichnung/text() = $x1L1/Bezeichnung/text()
            return
                <MitarbeiterIn>
                    <Name>
                        { concat($x0L1/Vorname/text(),
                         $x0L1/Nachname/text()) }
                    </Name>
                    <Geburtsdatum>
                        { $x0L1/Geburtsdatum/text() }
                    </Geburtsdatum>
                </MitarbeiterIn> )
            }
        </Abteilung> )
    }
</Firma>
```

Listing 3.5: XQuery-Script zu Abbildung 3.6 mit syntaktischen Korrekturen

[12] [Leg05] enthält ein Perl-Script, das syntaktische Fehler in Clios XQuery-Abfragen korrigiert.

Listing 3.5 zeigt das XQuery-Script aus Abbildung 3.6 nach den syntaktischen Korrekturen[13]. Die Transformation dieser XQuery-Abfrage mit Saxon angewendet auf die Beispielinstanz aus Listing 3.2 bringt folgendes Ergebnis:

```
<Firma>VerkaufManfredMoser1984-06-13SimoneEgger1975-10-
 20LogistikMargitMüller1967-01-17</Firma>
```

Listing 3.6: Output der XQuery-Transformation

Obwohl die strukturellen Komponenten verloren gegangen sind, lässt sich feststellen, dass die Integrität der Daten gewährleistet ist. Sowohl die Gruppierung von *MitarbeiterIn* nach *Abteilung* als auch die Verknüpfung von *Vor- und Nachname* zu *Name* wurden durchgeführt, wodurch die Daten der Struktur des Zielschemas entsprechen.

3.4.1.2 Semantische Fehler

Wie man in Abbildung 3.6 erkennen kann, versucht Clio, Duplikate mit der Funktion *distinct-values()* zu entfernen. Diese Duplikateliminierung wird von Clio standardmäßig durchgeführt und lässt sich nicht deaktivieren. Die Spezifikation der Funktion *distinct-values()* definiert, dass als Eingabe eine Sequenz von atomaren Werten erwartet wird, die in eine Sequenz duplikatfreier atomarer Werte umgewandelt wird. Wenn die Funktion wie in diesem Fall keine Sequenz von atomaren Werten enthält, soll der Eingabewert atomisiert werden *(engl. atomize*), wodurch im Fall von Clio die gesamten strukturellen Informationen des Ergebnisses der FLWOR-Anfrage verloren gehen [Leg05]. Um das gewünschte Ergebnis zu erhalten, müsste die Anfrage aus Abbildung 3.6 so umgeschrieben werden, dass die Funktion *distinct-values()* auf die einzelnen Attribute angewendet wird, anstatt die gesamte FLWOR-Anfrage einzuschließen. [Leg05] beschreibt anhand eines Beispiels, wie dabei vorgegangen werden kann.

3.4.2 XSLT-Transformation

Clio generiert zwei XSLT-1.0-Scripts, die nacheinander ausgeführt werden müssen, um das gewünschte Ergebnis zu erhalten. Dabei sammelt das erste Script die Zieldaten, die im zweiten Script von Duplikaten bereinigt in die Struktur des Zielschemas überführt werden.

---

[13] Zur besseren Übersicht wurden die Kommentare weggelassen.

```
...
<xsl:for-each select="/root/MitarbeiterIn">
    <xsl:variable name="x0" select="."/>
<xsl:for-each select="/root/Abteilung">
    <xsl:variable name="x1" select="."/>

    <xsl:if test="$x0/AbteilungFK=$x1/@Id">
        <xsl:element name="Abteilung">
...
```

Listing 3.7: Ausschnitt aus dem ersten XSLT-Script

Angewendet auf die Instanz aus Listing 3.2 bringt die Transformation der XSLT-Scripts mit Saxon folgendes Ergebnis:

```
<Firma>
    <Abteilung>
        <Bezeichnung>Verkauf</Bezeichnung>
        <MitarbeiterIn>
            <Name>_sdf_Concat(Manfred, Moser)</Name>
            <Geburtsdatum>1984-06-13</Geburtsdatum>
        </MitarbeiterIn>
        <MitarbeiterIn>
            <Name>_sdf_Concat(Simone, Egger)</Name>
            <Geburtsdatum>1975-10-20</Geburtsdatum>
        </MitarbeiterIn>
    </Abteilung>
    <Abteilung>
        <Bezeichnung>Logistik</Bezeichnung>
        <MitarbeiterIn>
            <Name>_sdf_Concat(Margit, Müller)</Name>
            <Geburtsdatum>1967-01-17</Geburtsdatum>
        </MitarbeiterIn>
    </Abteilung>
</Firma>
```

Listing 3.8: Ergebnis der XSLT-Abfrage

Obwohl Saxon die Funktion *_sdf_Concat()* nicht interpretieren konnte, geht aus Listing 3.8 hervor, dass die Datentransformation korrekt durchgeführt wurde und die Datenintegrität gewährleistet ist. Wie bereits bei der XQuery-Transformation in Abschnitt 3.4.1 erfolgte die Gruppierung der entsprechenden MitarbeiterInnen nach deren zugehörigen Abteilungen.

3.4.3 XSML

XSML ist eine Erweiterung von XML zur Definition einer Schema-Mapping-Sprache. Clio kann Schema-Mappings im XSML-Format speichern, wodurch es möglich wird, diese Mappings in Clio selbst bzw. in anderen Applikationen wieder zu verwenden.

Listing 3.9 zeigt zwei Mappings des Testsets im XSML-Format. Das erste Mapping bildet *Vorname* und *Nachname* des Quellschemas auf *Name* im Zielschema mittels der Funktion *_sdf_Concat()* ab. Das zweite Mapping steht für die entsprechende Abbildung des Attributs *Bezeichnung*.

```
<xsml:valueMapping>
    <xsml:source>/root/MitarbeiterIn/Vorname</xsml:source>
    <xsml:source>/root/MitarbeiterIn/Nachname</xsml:source>
     <xsml:target>/Firma/Abteilung/MitarbeiterIn/Name
      </xsml:target>
    <xsml:function>_sdf_Concat(${0},${1})</xsml:function>
</xsml:valueMapping>
<xsml:valueMapping>
    <xsml:source>/root/Abteilung/Bezeichnung</xsml:source>
    <xsml:target>/Firma/Abteilung/Bezeichnung</xsml:target>
    <xsml:function>${0}</xsml:function>
</xsml:valueMapping>
```

Listing 3.9: Ausschnitt eines Schema-Mappings im XSML-Format

3.4.4 XQuery und XSLT Derivate

Der Vollständigkeit halber soll erwähnt werden, dass Clio Abfragecode in diversen XSLT- und XQuery-Derivaten erzeugen kann. Besonders interessant sind die *Nested-XQuery-(2Phased)*-Scripts, da diese als einzige direkt in Clio ausgeführt werden können. Zusätzlich sind sie syntaktisch korrekt, da sie der aktuellen XQuery-Spezifikation entsprechen. Aus diesen Tatsachen lässt sich schließen, dass die Nested-XQuery-Scripts von Clios EntwicklerInnen am besten gepflegt wurden, weshalb sie in den nachfolgenden Kapiteln verwendet werden sollen.

Angewendet auf das Testset aus Abbildung 3.1 schlägt die Auswertung der Nested-XQuery-Scripts jedoch fehl. Clio erzeugt nach Erstellung der Wertkorrespondenz *Vorname* und *Nachname* auf *Name* die in Abbildung 3.7 dargestellte Fehlermeldung, was dazu führt, dass das Nested XQuery Script nicht erzeugt wird. Wie im nächsten Abschnitt beschrieben, war es mir nicht möglich, nachzuvollziehen, wodurch dieser Fehler ausgelöst wird.

3.5 Erfahrungen mit der Benutzung von Clio

Clios Verhalten lässt sich oft nicht nachvollziehen, seine Benutzung ist teilweise sehr undurchsichtig. Folgende Erfahrungen erschweren dies:

- Es existieren keine Beschreibungen bzw. Anleitungen zur Bedienung von Clio. Lediglich technische Beschreibungen sind verfügbar.

- Die technischen Anleitungen beschreiben Komponenten von Clio, die nicht implementiert sind, beispielsweise den *Data Viewer*.
- Clios interne Hilfe ist veraltet – sie bezieht sich nicht auf die aktuelle Version und deren Funktionalität.

Es folgt eine Aufzählung diverser Situationen bei der Verwendung von Clio, die laut meinen subjektiven Erfahrungen zu Problemen geführt haben:

- Clios Fehlermeldungen sind zum Großteil nicht aussagekräftig. Clio erzeugt Fehlermeldungen mit dem Inhalt *1*, *-1* bzw. leere Fehlermeldungen. Ein Beispiel dafür ist in Abbildung 3.7 dargestellt.
- Nach Bestätigung diverser inhaltsloser Fehlermeldungen funktioniert der Vorgang, der diese erzeugt hat, scheinbar problemlos.
- Nach diversen Fehlermeldungen kann ein Mapping nicht mehr weiterbearbeitet werden. Jeder Versuch der Erzeugung von Wertkorrespondenzen ist vergeblich und erzeugt wieder eine Fehlermeldung. In diesem Fall muss der Vorgang neu gestartet werden, wodurch das bereits erstellte Mapping verloren geht.
- Die Wertkorrespondenzen müssen teilweise in einer Reihenfolge erzeugt werden, die nicht nachvollziehbar ist. Wird diese Reihenfolge nicht eingehalten, erzeugt Clio eine Fehlermeldung ohne Aussage.
- Der produzierte XQuery-Code entspricht nicht dem aktuellen Standard (siehe Abschnitt 3.4.1).
- Die Ausführung der erzeugten Abfragen funktioniert nicht, obwohl Clios Interface diese Funktionalität vorsieht.
- Der Zweck diverser Funktionen ist nicht erkennbar. Das Ausführen dieser Funktionen bringt weder eine Zustandsveränderung noch eine Fehlermeldung.

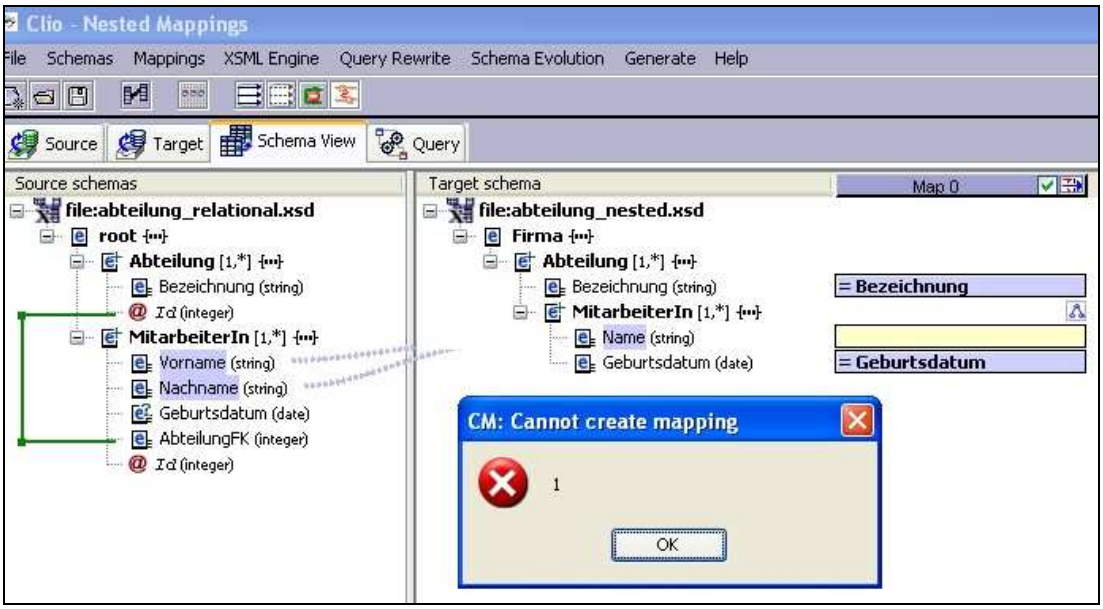

Abbildung 3.7: Fehlermeldung in Clio

Abbildung 3.7 zeigt eine Fehlermeldung in Clio, die bei der Erstellung der Wertkorrespondenz *Vorname* und *Nachname* auf *Name* entstanden ist. Nach Bestätigung der Fehlermeldung wird die entsprechende Wertkorrespondenz dennoch korrekt erzeugt. Wie in Abschnitt 3.4.4 beschrieben, führt der Fehler allerdings dazu, dass das Nested-XQuery-Script nicht generiert wird.

Kapitel 4

Evaluierung von Clio anhand diverser Mapping-Situationen

Als Grundlage für die in Kapitel 6 durchzuführende Evaluierung mit Metamodellen soll Clio in diesem Kapitel in Anwendung auf simple Mapping Situationen getestet werden. Legler führt in [Leg05] bzw. [LN07] eine Klassifikation von Mapping-Situationen ein und erstellt dazu einfache Schemata, um die Fähigkeiten diverser Mapping-Tools – darunter auch Clio – zu eruieren.

Als Vorbereitung auf die vorliegende Arbeit wurden die Tests aus [Leg05] mit Clio nachgestellt, wobei sich die von mir erzielten Ergebnisse teilweise von Leglers Resultaten unterschieden. Ein Grund für diese Abweichungen dürfte die Tatsache sein, dass für die vorliegende Arbeit eine neuere Version von Clio zum Einsatz kommt. Als weiteren Grund vermutete ich, dass die Modellierung meiner Testschemata nicht korrekt war, wodurch sich diese von denen aus [Leg05] unterscheiden. Diese Annahme kann ich mittlerweile ausschließen, da mir Frank Legler freundlicherweise seine Testschemata zur Verfügung gestellt hat.

In Abschnitt 4.1 werden Fälle untersucht, bei denen Attribute im Quell- oder Zielschema fehlen. Abschnitt 4.2 behandelt Mapping-Situationen, die sich aus der Betrachtung einzelner Korrespondenzen ergeben, während in Abschnitt 4.3 Mapping-Situationen untersucht werden, die mehrere Korrespondenzen beinhalten, wodurch sich strukturelle Abhängigkeiten im Quell- oder Zielschema ergeben.

Die Tests der einzelnen Mapping-Situationen werden nach folgendem Schema durchgeführt:

1. Visuelle Darstellung der Mapping-Situation
2. Beschreibung der Mapping-Situation und des erwarteten Sollergebnisses
3. Darstellung und Interpretation des Mappings aufgrund der erzeugten Wertkorrespondenzen inkl. Vergleich zum Ergebnis aus [Leg05][14]
4. Auswertung der erzeugten Scripts mittels Beispielinstanz:
   - XQuery: Freie Interpretation[15] der Abfrage
   - XSLT: Ausführung der Abfragen mit Saxon
   - Nested XQuery (2Phased): Direkte Ausführung in Clio

4.1 Mapping-Situationen infolge fehlender Korrespondenzen

Im Quell- oder Zielschema befinden sich Attribute, für die aufgrund ihres fehlenden Gegenübers keine Wertkorrespondenz erzeugt werden kann. Clio soll Fälle erkennen, in denen die Datenintegrität verletzt wird, und entsprechend reagieren, indem es selbständig Werte erzeugt, den/die BenutzerIn zur Eingabe auffordert oder das Mapping ablehnt.

4.1.1 Attribut des Quellschemas ist nicht Teil des Mappings

Beschreibung der Situation: Ein Attribut des Quellschemas ist nicht im Zielschema vorhanden, wodurch es im Mapping weggelassen werden kann. Obwohl dabei Quelldaten verloren gehen, können keine Komplikationen entstehen - das erzeugte Mapping ist korrekt. Durch den entstandenen Informationsverlust ist es allerdings nicht mehr möglich, die Quelldaten aus den Zieldaten wiederherzustellen.

[14] Der Vergleich zu [Leg05] wird nur bei abweichenden Ergebnissen gezogen.

[15] Aufgrund der fehlerhaften XQuery-Scripts werden diese nicht ausgeführt. Siehe Kapitel 3.4.1.

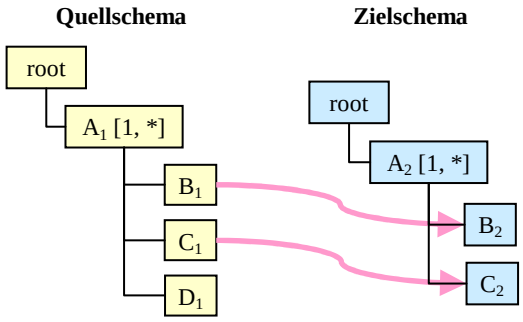

Abbildung 4.1: Quellattribut nicht Teil des Mappings

Erzeugung der Wertkorrespondenz: Nach Erzeugung der Wertkorrespondenzen wird das Attribut D_1 aus Abbildung 4.1 ignoriert, wodurch die Anforderungen des Zielschemas erfüllt sind.

Auswertung der erzeugten Scripts mittels Beispielinstanz:

- XQuery: Ergebnis korrekt
- XSLT: Ergebnis korrekt
- Nested XQuery: Ergebnis korrekt

4.1.2 Attribut des Zielschemas ist nicht Teil des Mappings

Dieser Abschnitt widmet sich Situationen, in denen ein oder mehrere Zielattribute existieren, die nicht im Quellschema vorhanden sind.

4.1.2.1 Zielattribute mit diversen Spezifikationen fehlen im Quellschema

Beschreibung der Situation: Wie Abbildung 4.2 zeigt, können folgende Fälle unterschieden werden, in denen Attribute des Zielschemas fehlen:

- K_2 ist ein Schlüsselattribut und darf weder leer sein noch weggelassen werden. Clio sollte entweder einen eindeutigen Wert für K_2 generieren oder den/die BenutzerIn zur Konfliktlösung auffordern. Gegebenenfalls sollte Clio das Mapping zurückweisen.

- U_2 ist ein Unikat. Ein entsprechender Wert sollte von Clio erzeugt bzw. der/die BenutzerIn zur Eingabe aufgefordert werden. Die Erzeugung eines Wertes durch eine Skolemfunktion ist nicht besonders sinnvoll, bewahrt al-

lerdings die Datenintegrität. Da die Unique Constraint nicht vorgibt, dass ein Wert vorhanden sein muss, ist auch ein NULL-Wert zulässig.

- Für C_2 ist ein Standardwert angegeben, der beim Fehlen einer Korrespondenz verwendet werden soll.
- D_2 muss im Zielschema vorhanden sein. Da es wenig Sinn macht, wenn Clio für D_2 einen Zufallswert errechnet, sollte der/die BenutzerIn zur Interaktion aufgefordert oder das Mapping zurückgewiesen werden.
- E_2 ist optional und kann weggelassen werden.

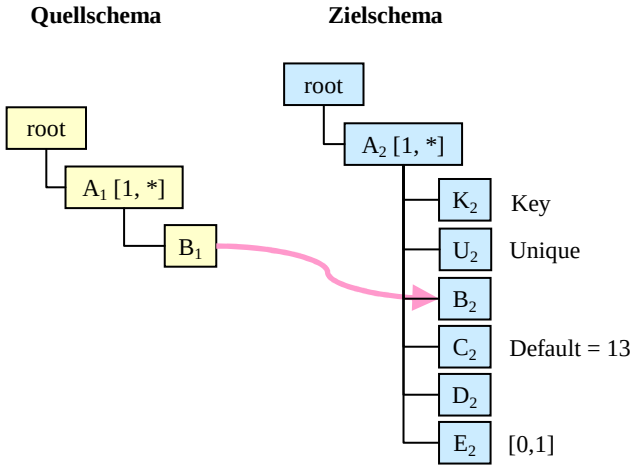

Abbildung 4.2: Zielattribute nicht Teil des Mappings

Erzeugung der Wertkorrespondenz: Wie Tabelle 4.1 zeigt, ergeben sich bei den Tests für die vorliegende Arbeit große Abweichungen zu [Leg05]. Die einzige Übereinstimmung gilt für das Attribut E_2, das sowohl in dieser als auch in Leglers Arbeit weggelassen wurde. Während in [Leg05] für K_2, U_2 und D_2 Skolemwerte berechnet wurden, werden in den vorliegenden Tests leere Attribute erzeugt. Da diese Werte zwingend im Zielschema vorhanden sein müssen, ist diese Lösung nicht zufriedenstellend. Lediglich für das Attribut C_2 lässt sich eine Verbesserung feststellen, da Clio den vorgegebenen Default-Wert einsetzt.

	K_2 **Key**	U_2 **Unique**	C_2 **Default**	D_2 **Notwendig**	E_2 **Optional**
[Leg05]	Skolemwert	Skolemwert	Skolemwert	Skolemwert	weggelassen
Clio v08	leer	leer	13	leer	weggelassen

Tabelle 4.1: Fehlende Zielattribute im Vergleich mit [Leg05]

Auswertung der erzeugten Scripts mittels Beispielinstanz:

- XQuery: Ergebnis korrekt
- XSLT: Ergebnis korrekt
- Nested XQuery: Ergebnis falsch - Standardwert für C_2 wird nicht übernommen

4.1.2.2 Fehlende Attribute sind Teil einer Fremdschlüsselrelation

Beschreibung der Situation: Im Zielschema sind Elemente über eine Fremdschlüsselbeziehung verknüpft, die im Quellschema nicht existiert. Clio muss diese Verknüpfung erhalten und notfalls selbständig Schlüsselwerte erzeugen.

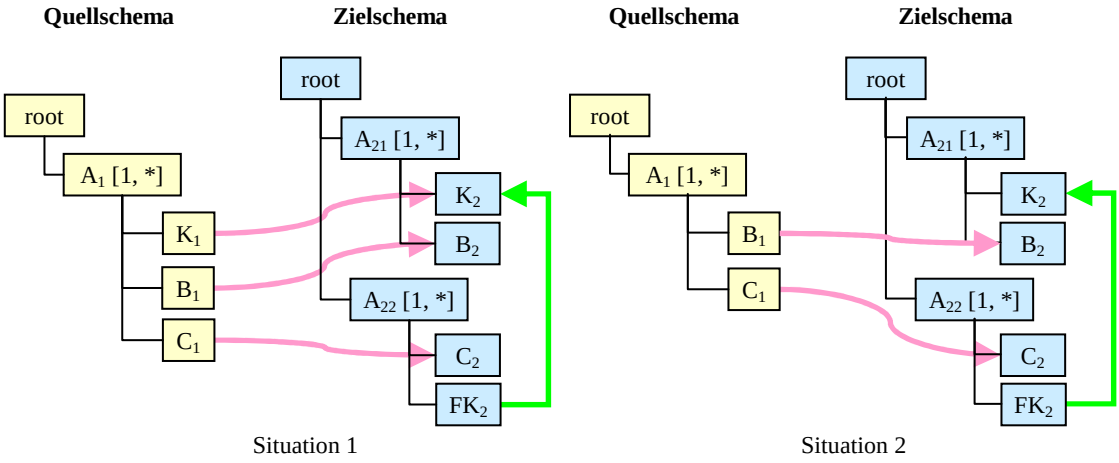

Abbildung 4.3: Fremdschlüssel bzw. Schlüssel fehlen

Erzeugung der Wertkorrespondenz:

Situation 1:

Mapping wird korrekt erzeugt. Für FK_2 wird der Wert von K_2 eingesetzt.

Situation 2:

Mapping wird korrekt erzeugt. Sowohl für K_2 als auch für FK_2 werden Skolemwerte erzeugt und die Fremdschlüsselbeziehung hergestellt.

Auswertung der erzeugten Scripts mittels Beispielinstanz:

Situation 1:

- XQuery: Ergebnis korrekt
- XSLT: Ergebnis korrekt
- Nested XQuery: Ergebnis falsch - Wert für FK_2 wird nicht übernommen

Situation 2:

- XQuery: Ergebnis korrekt
- XSLT: Ergebnis korrekt
- Nested XQuery: Ergebnis korrekt

4.2 Mapping-Situationen, die einzelne Korrespondenzen betreffen

Dieser Abschnitt betrachtet Mapping Situationen, die sich durch die Untersuchung einzelner Korrespondenzen ausmachen lassen. Abschnitt 4.2.1 behandelt 1:1-Korrespondenzen, während sich Abschnitt 4.2.2 mit 1:n, n:1 und n:m-Korrespondenzen beschäftigt.

4.2.1 1:1-Korrespondenzen

Als 1:1-Korrespondenzen bezeichnen wir Abbildungen von jeweils einem Quellattribut auf genau ein Zielattribut. Unterschieden werden kann zwischen Korrespondenzen in Bezug auf

- Typen der beteiligten Schema-Elemente
- Kardinalität der beteiligten Elemente
- Datentyp der beteiligten Elemente

4.2.1.1 Typ der beteiligten Schema-Elemente

Laut [Leg05] lassen sich Korrespondenzen nach deren beteiligten Elementen in nachstehende Mapping Situationen unterteilen. Mit *Attribut* wird dabei ein Blattknoten bezeichnet, während mit *Element* ein Innerer Knoten gemeint ist.

- Element – Element
- Element – Attribut
- Attribut – Element
- Attribut – Attribut

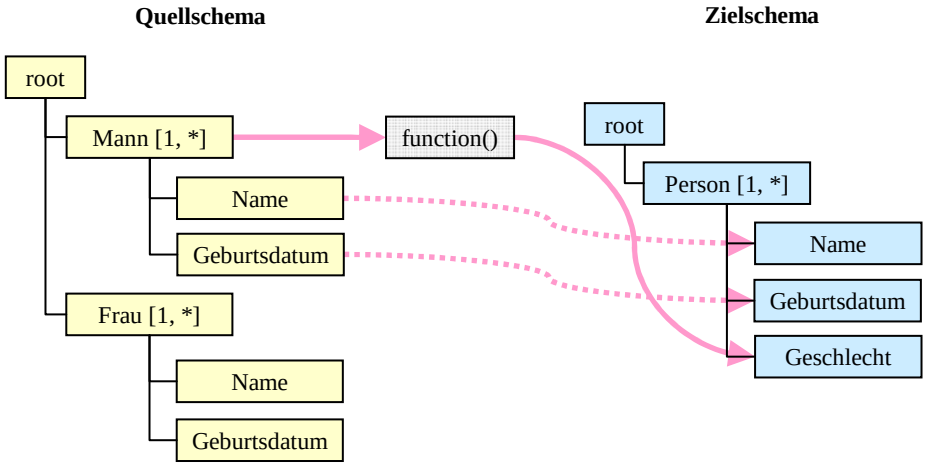

Abbildung 4.4: Abbildung eines Inneren Knotens auf ein Blattelement

In Clio können innere Knoten nicht Teil eines Mappings sein, weshalb für diese Arbeit nur Attribut – Attribut-Korrespondenzen relevant sind. Dadurch ist es beispielsweise nicht möglich, das Mapping aus Abbildung 4.4 zu erzeugen, da die Wertkorrespondenz von *Mann* aus dem Quellschema über die Funktion *function()* auf *Geschlecht* im Zielschema nicht abgebildet werden kann.

4.2.1.2 Kardinalität der beteiligten Elemente

Beschreibung der Situation: Wie bereits in Abschnitt 3.2 festgestellt wurde, erkennt Clio die Kardinalität von Attributen und Elementen. Bei abweichender Kardinalität der Attribute einer Wertkorrespondenz ergeben sich verschiedene Mapping Situationen, die laut [Leg05] in sechs verschiedene Kategorien eingeordnet werden können (siehe auch Tabelle 4.2):

1. Wert fehlt im Zielschema
2. Wert aus dem Quellschema muss verworfen werden
3. Wert fehlt eventuell im Zielschema und Wert aus Quellschema muss eventuell verworfen werden
4. Wert fehlt eventuell im Zielschema
5. Wert aus der Quellschema muss eventuell verworfen werden
6. kein Problem

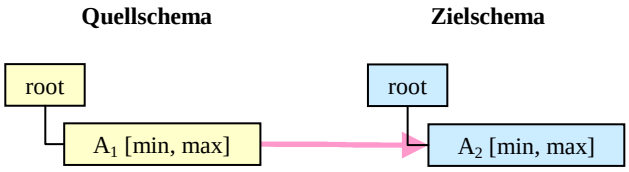

Abbildung 4.5: Mapping zum Testen von Kardinalitäten

Während die Fälle 1 und 2 aufgrund der Schemainformationen analysiert werden können, fallen die Testergebnisse für Situation 3, 4 und 5 abhängig von der verwendeten Instanz aus. Die Tests für diesen Abschnitt werden anhand der Kardinalitäten aus Tabelle 4.2 durchgeführt und interpretiert.

Erzeugung der Wertkorrespondenz:

Situation	A1	A2	[Leg05]	Ergebnis
1	[1, 1]	[2, 2]	nicht erkannt	Fehlermeldung1
2a	[2, 2]	[1, 1]	teilweise erkannt	Fehlermeldung2
2b	[3, 3]	[2, 2]		falsch
3a	[0, 2]	[1, 1]	teilweise erkannt	Fehlermeldung2
3b	[1, 3]	[2, 2]		abhängig von Beispielinstanz
4	[0, 1]	[1, 1]	nicht erkannt	Fehlermeldung3
5a	[1, 2]	[1, 1]	teilweise erkannt	Fehlermeldung2
5b	[2, 3]	[2, 2]		abhängig von Beispielinstanz
6	[1, 2]	[1, 3]	ok	korrekt

Tabelle 4.2: Kardinalitäten der Testattribute und Mapping Ergebnisse

Für die meisten Testsituationen lassen sich keine Mappings, nur Situation 6 ist korrekt. Situation 2b generiert drei *A2*-Elemente, wodurch das Ergebnis unbrauchbar ist.

3a und 5b produzieren ein richtiges Ergebnis, falls die Anzahl der *A1*-Elemente in der Beispielinstanz gleich der jeweiligen Kardinalität von *A2* im Quellschema sind. Für die Situationen 1, 2a, 3a, 4 und 5a werden hingegen nachstehende Fehlermeldungen erzeugt:

- Fehlermeldung1:
  Cannot create Mapping: Index:0, Size:0

- Fehlermeldung2:
  Cannot create Mapping: Cannot map a repeatable or choice element into a top-level, non-repeatable and non-choice element

- Fehlermeldung3:
  Cannot create Mapping – leere Fehlermeldung

4.2.1.3 Datentypen der beteiligten Elemente

Beschreibung der Situation: Dass Clio Datentypen der Quell- und Zielattribute erkennt, wurde bereits in Abschnitt 3.2 festgestellt. In diesem Abschnitt wird getestet, ob und wie Clio Attribute unterschiedlicher Datentypen aufeinander abbildet. Der ideale Zustand wäre, wenn Clio Datentypen, die zueinander kompatibel sind, erkennen und automatisch konvertieren würde (*engl. to cast*). Anderenfalls sollte das Programm ein Mapping zurückweisen. Dazu werden wie in [Leg05] drei Situationen getestet:

1. **Datentypen sind vollständig zueinander kompatibel**
   Es existiert eine eindeutige Regel, wie ein Datentyp auf einen anderen konvertiert werden kann.

2. **Datentypen sind fallweise kompatibel**
   Abhängig von der jeweiligen Instanz kann der Quelldatentyp auf den Zieldatentyp konvertiert werden. Beispielsweise kann ein Quellattribut vom Typ *String* nur auf ein Zielattribut vom Typ *Boolean* konvertiert werden, wenn die Quellinstanz der Anforderung des Boolean-Datentyps entspricht (*true, false, 0* oder *1*).

3. **Datentypen sind nicht kompatibel**
   Datentypen sind nicht zueinander kompatibel, falls für sie keine Konvertierungsregel existiert.

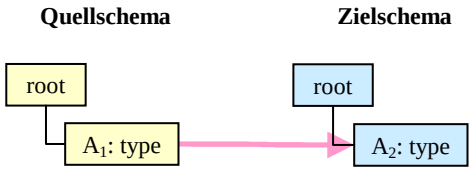

Abbildung 4.6: Mapping-Situationen mit unterschiedlichen Datentypen

Erzeugung der Wertkorrespondenz: Auch bei Mappings unterschiedlicher Datentypen gilt, dass die vorliegenden Ergebnisse von denen Leglers abweichen. Wie in Tabelle 4.3 ersichtlich ist, werden vollständig zueinander kompatible Datentypen erkannt und konvertiert, während Mapping-Situationen mit teilweise kompatiblen oder nicht kompatiblen Datentypen ignoriert werden. Für die Situationen 2 und 3 setzt Clio die Funktion *convert()*, die von den getesteten Script-Prozessoren nicht interpretiert werden kann. Zumindest für Situation 3 ist die Verwendung einer Konvertierungsfunktion auf jeden Fall falsch – das Mapping hätte zurückgewiesen werden müssen.

Situation	A1	A2	[Leg05]	Ergebnis:
1a	xs:integer	xs:integer	ok	ok
1b	xs:integer	xs:string	falsch, zurückgewiesen	ok, Funktion _sdf_Int2String
2	xs:string	xs:boolean	ok, zurückgewiesen	falsch, Funktion convert()
3	xs:integer	xs:date	ok, zurückgewiesen	falsch, Funktion convert()

Tabelle 4.3: Datentypen der Quell- und Zielattribute und Mapping-Ergebnis

Auswertung der erzeugten Scripts:

- XQuery: Ergebnis korrekt. Die erzeugten Funktionen können nicht interpretiert werden.
- XSLT: Ergebnis korrekt. Die erzeugten Funktionen können nicht interpretiert werden.
- Nested XQuery: Ergebnis falsch. Die eingesetzten Funktionen werden ignoriert, wodurch die Quellwerte 1:1 auf das Ziel übertragen werden.

4.2.2 n:1-, 1:n- und n:m-Korrespondenzen

[Leg05] unterscheidet für Korrespondenzen, in denen mehrere Quell- bzw. Zielattribute enthalten sind, folgende drei Fälle:

- n:1 – Mehrere Quellattribute werden auf ein Zielattribut abgebildet.
- 1:n – Ein Quellattribut wird auf mehrere Zielattribute abgebildet.
- n:m – Mehrere Quellattribute werden auf mehrere Zielattribute abgebildet.

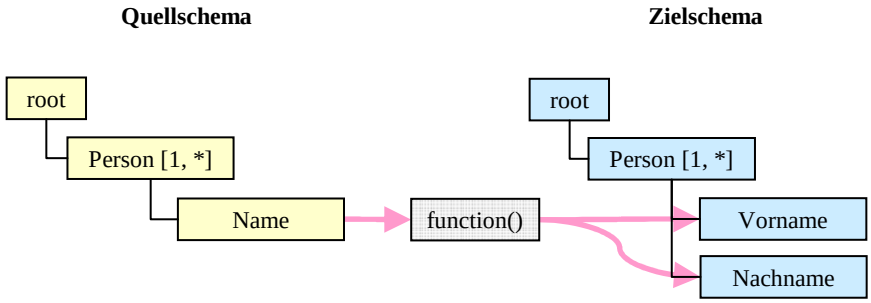

Abbildung 4.7: 1:n-Korrespondenz

In Clio können Wertkorrespondenzen auf mehrere Zielattribute nicht modelliert werden. Ein Beispiel für eine solche Situation ist in Abbildung 4.7 dargestellt. Auf Seite des Zielschemas darf in jeder Wertkorrespondenz nur ein Attribut teilnehmen, weshalb sich meine Tests auf den n:1-Fall beschränken. Für n:1-Korrespondenzen lassen sich folgende vier Fälle ausmachen:

- n:1-Korrespondenz mit dem selben Elternknoten
- n:1-Korrespondenz mit dem selben Vorfahren
- n:1-Korrespondenz mit einer Fremdschlüsselbeziehung
- n:1-Korrespondenz ohne strukturellen Zusammenhang

4.2.2.1 n:1-Korrespondenz mit dem selben Elternknoten

Beschreibung der Situation: Die beiden Attribute *Vorname* und *Nachname* mit dem selben Elternknoten *Person* aus dem Quellschema sollen auf das Attribut *Name* im Zielschema abgebildet werden. Die beiden Quellattribute sollten zum Zielattribut *Name* verknüpft werden.

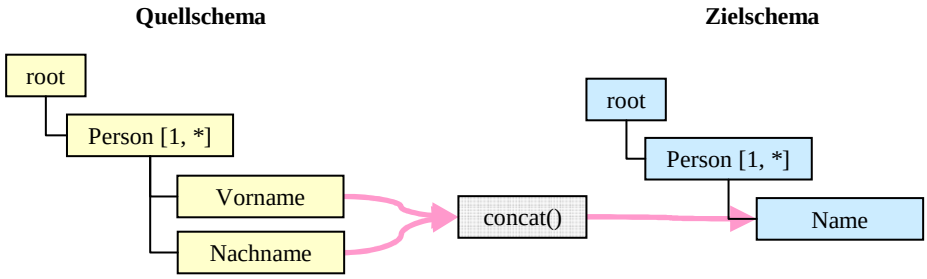

Abbildung 4.8: Gleicher Elternknoten

Erzeugung der Wertkorrespondenz: Clio erkennt den Zusammenhang der beiden Quellattribute durch ihren Elternknoten und fügt in der Wertkorrespondenz automatisch die Funktion *_sdf_Concat()* ein, um *Vorname* und *Nachname* zu verknüpfen.

Auswertung der erzeugten Scripts:

- XQuery: Ergebnis korrekt.
- XSLT: Ergebnis korrekt.
- Nested XQuery: Ergebnis falsch. Die Funktion *_sdf_Concat()* wird ignoriert. Nur *Vorname* wird auf das Zielschema übertragen.

4.2.2.2 n:1 Korrespondenz mit dem selben Vorfahren im Quellschema

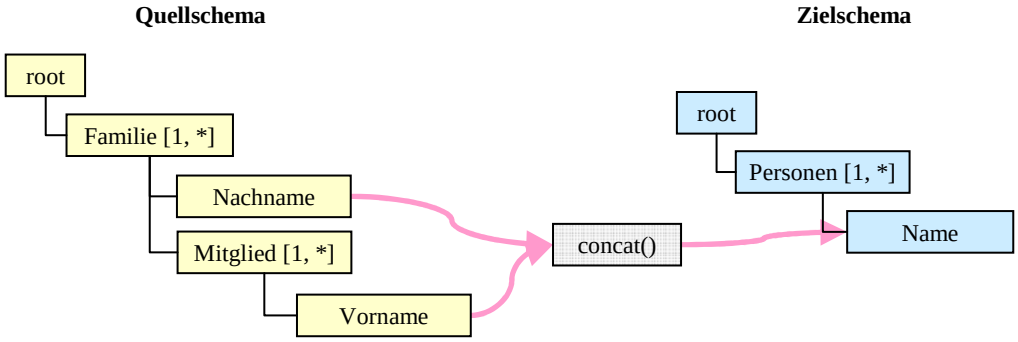

Abbildung 4.9: Gleicher Vorfahre

Beschreibung der Situation: Die beiden Attribute *Vorname* und *Nachname* mit dem selben Vorfahren *Familie* des Quellschemas sollen auf das Attribut *Name* im Zielschema abgebildet werden. Clio muss den Zusammenhang im Quellschema erkennen und eine entsprechende Verknüpfung der Quellattribute erzeugen.

Erzeugung der Wertkorrespondenz: Clio erkennt den Zusammenhang der Quellattribute über den selben Vorfahren und fügt im Mapping die Funktion *_sdf_Concat()* ein.

Auswertung der erzeugten Scripts

- XQuery: Ergebnis korrekt.
- XSLT: Ergebnis korrekt.
- Nested XQuery: Ergebnis falsch. Funktion *_sdf_Concat()* wird ignoriert. Lediglich *Nachname* wird auf das Attribut *Name* im Zielschema übertragen.

4.2.2.3 n:1 Korrespondenz verbunden über Fremdschlüssel

Beschreibung der Situation: Die beiden Attribute *Vorname* und *Nachname* aus dem Quellschema, die über eine Fremdschlüsselbeziehung verbunden sind, sollen auf das Attribut *Name* im Zielschema übertragen werden. Clio soll die Quellattribute miteinander verknüpfen, wobei deren Zusammenhang über die Fremdschlüsselbeziehung berücksichtigt werden muss.

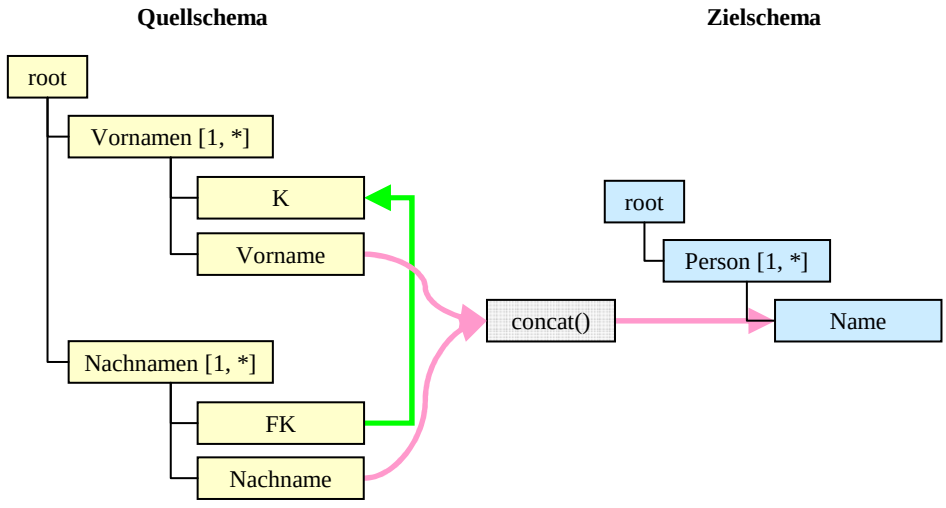

Abbildung 4.10: n:1-Korrespondenz über Fremdschlüssel

Erzeugung der Wertkorrespondenz: Die Quellattribute werden über die Funktion *_sdf_Concat()* verbunden, wobei die Fremdschlüsselbeziehung beachtet wird.

Auswertung der erzeugten Scripts:

- XQuery: Ergebnis korrekt. Fremdschlüsselkorrespondenz wird aufgelöst.
- XSLT: Ergebnis korrekt. Fremdschlüsselkorrespondenz wird aufgelöst.
- Nested XQuery: Ergebnis falsch. Lediglich *Vorname* wird auf das Zielschema übertragen.

4.2.2.4 n:1 Korrespondenz ohne Zusammenhang im Quellschema

Beschreibung der Situation: Die beiden Attribute *Vorname* und *Nachname* aus dem Quellschema, die in keinem Zusammenhang zueinander stehen, sollen auf *Name* im Zielschema abgebildet werden.

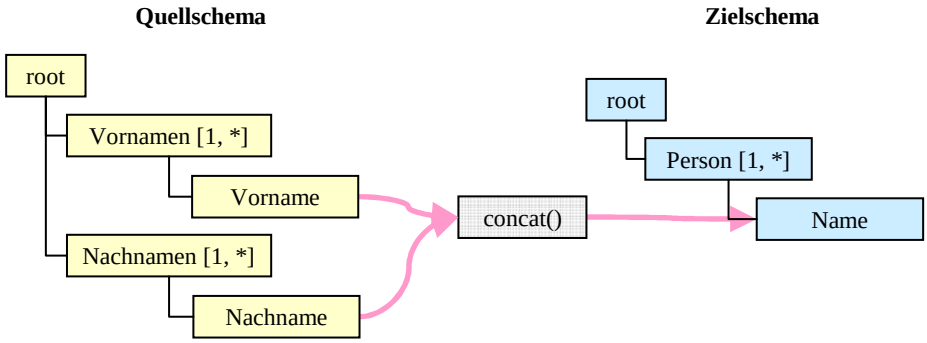

Abbildung 4.11: Kein Zusammenhang

Erzeugung der Wertkorrespondenz: Es ist nicht möglich, die Wertkorrespondenz zu erzeugen. Clio bringt die Fehlermeldung *value correspondence doesn't exist*.

4.3 Mapping-Situationen, die mehrere Korrespondenzen betreffen

Dieser Abschnitt untersucht Mapping-Situationen, die mehrere Korrespondenzen beinhalten. In [Leg05] werden solche Situationen auch als Mapping-Cluster bezeichnet. Abschnitt 4.3.1 befasst sich mit Mapping-Situationen, die sich durch Zusammenhänge im Quellschema zusammenfassen lassen, während sich Abschnitt 4.3.2 Zusammenhängen im Zielschema widmet.

4.3.1 Verbindung im Quellschema

Für Mapping Cluster mit Verbindungen im Quellschema können laut [Leg05] folgende Situationen unterschieden werden:

- Situationen mit strukturellem Zusammenhang im Quellschema
- Situationen mit einer Fremdschlüsselbeziehung im Quellschema
- Situationen ohne Verbindung im Quellschema

4.3.1.1 Struktureller Zusammenhang im Quellschema

Beschreibung der Situation: Die Attribute *Vorname* und *Nachname* mit dem selben Vorfahren *Familie* aus dem Quellschema sollen auf die Attribute *Vorname* und *Nachname* im Zielschema abgebildet werden. Clio sollte für jedes Familienmitglied aus dem Quellschema einen *Namen* bestehend aus *Vorname* und *Nachname* erzeugen.

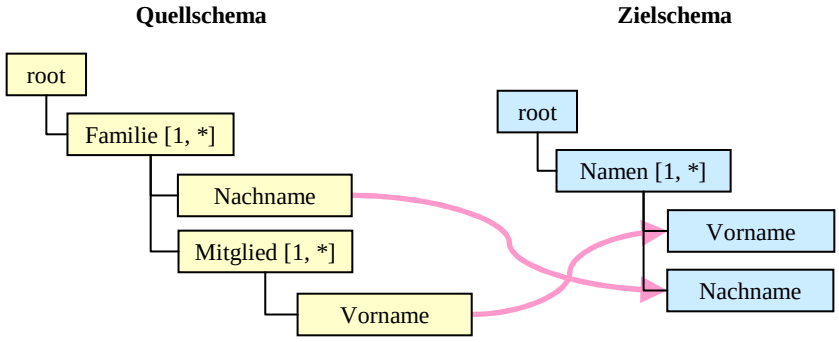

Abbildung 4.12: Struktureller Zusammenhang im Quellschema

Erzeugung der Wertkorrespondenz: Nach Erstellung der Wertkorrespondenz schlägt Clio zwei Mappings vor. Das erste Mapping bildet nur das Attribut *Nachname* des Quellschemas auf *Nachname* des Zielschemas ab. Das zweite Mapping enthält sowohl die Nachnamen als auch die Vornamen der beiden Inputschemata. Grundsätzlich sind beide Mappings korrekt, wobei das zweite eine bessere Lösung für das Testset aus Abbildung 4.12 ist. Beim Vorhandensein mehrere Mappings bietet Clio die Möglichkeit, nicht benötigte Mappings zu deaktivieren, was sich direkt auf die erzeugten Abfragescripts auswirkt.

Auswertung der erzeugten Scripts:

- XQuery: Ergebnis korrekt
- XSLT: Ergebnis korrekt
- Nested XQuery: Ergebnis korrekt

4.3.1.2 Verbindung über Fremdschlüssel im Quellschema

Beschreibung der Situation: *Vorname* und *Nachname* sind im Quellschema über eine Fremdschlüsselrelation verbunden. Diese sollen auf die Attribute *Vorname* und *Nachname* im Zielschema abgebildet werden. Clio sollte ein Mapping erzeugen, das die Fremdschlüsselverbindung der Quellattribute aufrecht erhält.

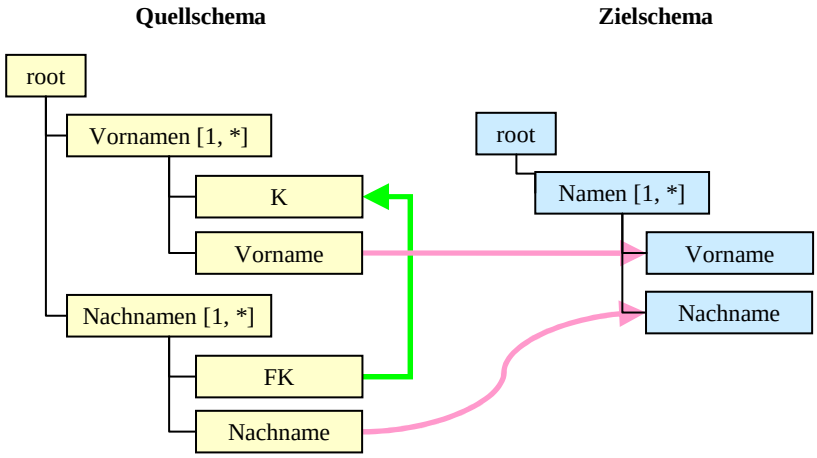

Abbildung 4.13: Verbindung über Fremdschlüssel im Quellschema

Erzeugung der Wertkorrespondenz: Nach Erzeugung der Wertkorrespondenzen schlägt Clio zwei Mappings vor. Das erste Mapping bildet lediglich die Vornamen aufeinander ab, während das zweite auch die Nachnamen berücksichtigt und die Fremdschlüsselverbindung erhält.

Auswertung der erzeugten Scripts mittels Beispielinstanz:

- XQuery: Ergebnis korrekt[16]
- XSLT: Ergebnis korrekt
- Nested XQuery: Ergebnis korrekt

4.3.1.3 Keine Verbindung im Quellschema

Beschreibung der Situation: Im Quellschema gibt es keinen Zusammenhang zwischen den Attributen *Vorname* und *Nachname*. Diese sollen auf die entsprechenden Pendants im Zielschema abgebildet werden. Laut [Leg05] können Mappingtools im Fall von zusammenhanglosen Quellschemata die Eingabewerte nach mehreren Möglichkeiten kombinieren. Beispielsweise kann ein Kreuzprodukt der Quelldaten erzeugt werden, indem jedes Attribut *Vorname* mit dem jedem Attribut *Nachname* kombiniert wird. Eine andere Interpretation wäre ein Outer-Union, der jedes Attribut mit einem NULL-Wert kombiniert.

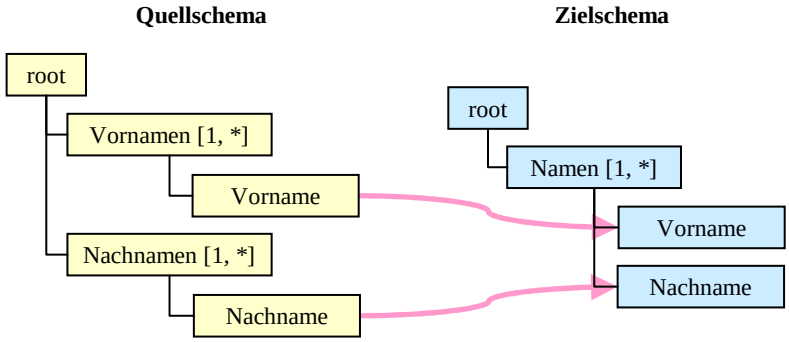

Abbildung 4.14: Keine Verbindung im Quellschema

Erzeugung der Wertkorrespondenz: Clio erzeugt zwei Mappings. Das erste Mappings bildet die Vornamen ab, das zweite die Nachnamen, was einem Outer-Union entspricht. Angesichts der Tatsache, dass die Quelldaten in keinem Zusammenhang zueinander stehen, ist Clios Verhalten korrekt.

---

[16] Je nachdem in welcher Reihenfolge die Wertkorrespondenzen erzeugt werden, kann das XQuery-Script nach Deaktivierung eines Alternativmappings falsch sein. Clio blendet im XQuery-Code eventuell das falsche Mapping aus. Diese Tatsache soll nur der Vollständigkeit halber erwähnt und bei den weiteren Untersuchungen vernachlässigt werden.

Auswertung der erzeugten Scripts mittels Beispielinstanz:

- XQuery: Ergebnis korrekt
- XSLT: Ergebnis korrekt
- Nested XQuery: Ergebnis korrekt

4.3.2 Verbindung im Zielschema

Für Mapping Cluster mit Verbindungen im Zielschema können laut [Leg05] folgende Situationen unterschieden werden:

- Situationen mit strukturellem Zusammenhang im Zielschema
- Situationen mit einer Fremdschlüsselbeziehung im Zielschema
- Situationen ohne Zusammenhang im Zielschema
- Spezialfall: Notwendigkeit von Gruppierung und Aggregation der Quelldaten, um strukturellem Zusammenhang im Zielschema zu entsprechen

4.3.2.1 Strukturelle Verbindung im Zielschema

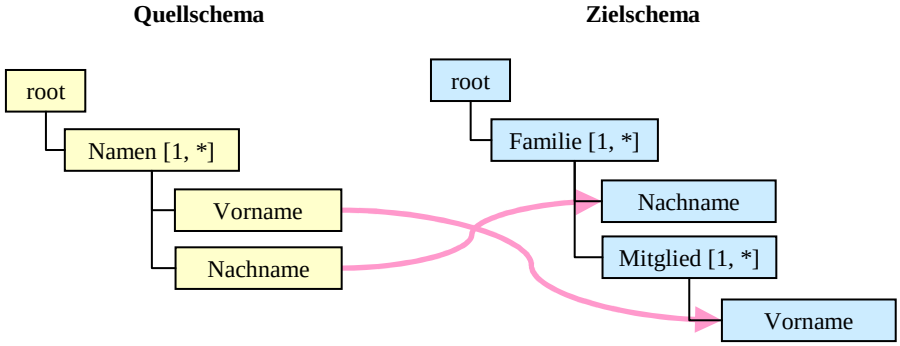

Abbildung 4.15: Strukturelle Verbindung im Zielschema

Beschreibung der Situation: Im Zielschema befinden sich zwei Attribute, die durch den selben Vorfahren strukturell verbunden sind. Clio sollte den Zusammenhang erkennen und die Quelldaten entsprechend dem Zielschema gruppieren oder dem/der BenutzerIn eine manuelle Gruppierung ermöglichen.

Erzeugung der Wertkorrespondenz: Nach Erstellung der Wertkorrespondenzen erfolgt automatisch eine Gruppierung nach *Nachname*, wodurch die Quelldaten korrekt auf das Zielschema abgebildet werden.

Auswertung der erzeugten Scripts mittels Beispielinstanz:

- XQuery: Ergebnis korrekt
- XSLT: Ergebnis korrekt
- Nested XQuery: Ergebnis falsch - Gruppierung wird nicht durchgeführt

4.3.2.2 Zusammenhang im Zielschema über Fremdschlüssel

Beschreibung der Situation: *Vorname* und *Nachname* sind im Zielschema über eine Fremdschlüsselbeziehung verbunden. Clio muss die Daten aus dem Quellschema verbinden, indem es selbständig eine Fremdschlüsselrelation erzeugt.

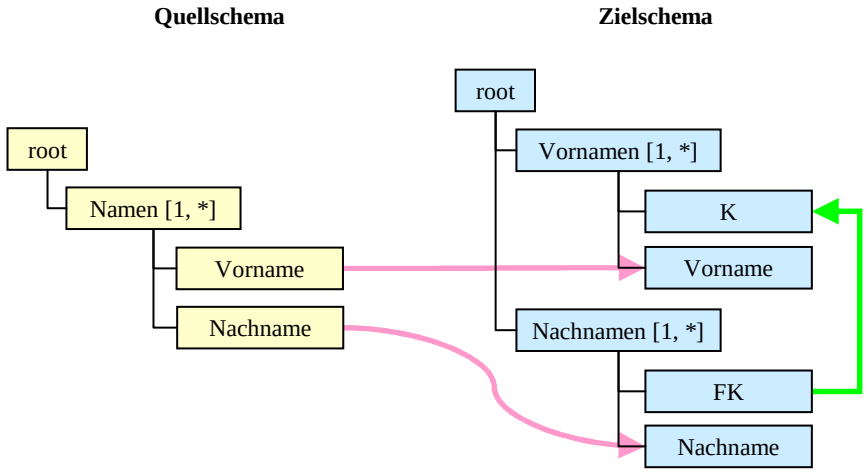

Abbildung 4.16: Zusammenhang im Zielschema über Fremdschlüssel

Erzeugung der Wertkorrespondenz: Nach Erzeugung der Wertkorrespondenzen werden Schlüssel und Fremdschlüssel durch eine Skolemfunktion erzeugt, wodurch die Datenintegrität gewährleistet wird.

Auswertung der erzeugten Scripts mittels Beispielinstanz:

- XQuery: Ergebnis korrekt
- XSLT: Ergebnis korrekt
- Nested XQuery: Ergebnis korrekt

4.3.2.3 Kein Zusammenhang im Zielschema

Beschreibung der Situation: Im Zielschema befinden sich Attribute, die nicht zusammenhängen. Clio kann die Daten beliebig in das Zielschema übertragen, um die Integrität zu gewährleisten.

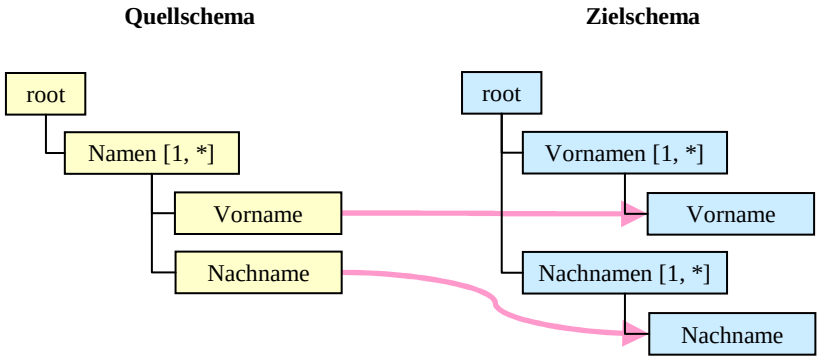

Abbildung 4.17: Kein Zusammenhang im Zielschema

Erzeugung der Wertkorrespondenz: Clio erzeugt zwei Mappings. Das erste Mapping bildet die Vornamen aufeinander ab, das zweite die Nachnamen. Dieses Ergebnis entspricht einem Outer-Union und stellt eine korrekte Lösung dar.

Auswertung der erzeugten Scripts mittels Beispielinstanz:

- XQuery: Ergebnis korrekt
- XSLT: Ergebnis korrekt
- Nested XQuery: Ergebnis korrekt

4.3.2.4 Gruppierung und Aggregation

Beschreibung der Situation: Die Daten aus dem Quellschema müssen nach *Nachname* gruppiert auf *Familie* im Zielschema übertragen werden. Um das Familieneinkommen zu ermitteln, muss zusätzlich eine Aggregation nach *Einkommen* durchgeführt werden.

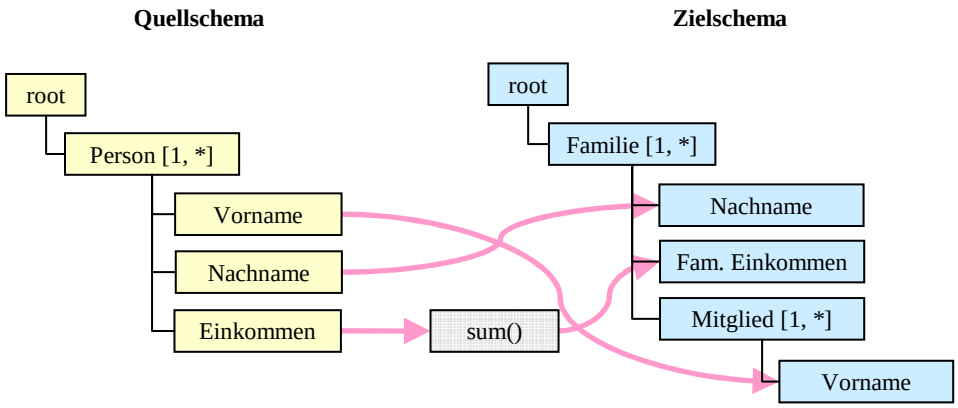

Abbildung 4.18: Gruppierung und Aggregation

Erzeugung der Wertkorrespondenz: Die Aggregation nach Einkommen lässt sich in Clio nicht modellieren. Es ist zwar möglich, eine beliebige Funktion einzufügen, die in die Abfragescripts übernommen wird. Da diese Funktionen jedoch nicht ausgeführt werden können, sondern als Konstante interpretiert werden, ist die Abbildung dieser Mapping Situation nicht möglich.

Kapitel 5

Flache Metamodelle – Von Ecore zu XML Schema

Um Ecore-Metamodelle mit Clio mappen zu können, müssen diese als XML Schemata vorliegen, die auf dem relationalen Modell basieren. Eclipse bietet die Möglichkeit, Metamodelle als XML Schema zu serialisieren. Da diese jedoch nicht dem relationalen Modell entsprechen, muss die Umformung von Ecore zu XML Schema händisch durchgeführt werden.

In diesem Kapitel wird anhand eines Testmodells gezeigt, wie die Umformung von Ecore-Metamodellen in XML Schema effizient durchgeführt werden kann. Es soll eruiert werden, welche Daten der Ecore-Modelle umgewandelt werden müssen bzw. welche vernachlässigt werden können. Dabei soll die Datenintegrität bewahrt bleiben. Außerdem sollen eventuell auftretende Datenverluste aufgedeckt werden. In Abschnitt 5.1 wird ein Testset definiert, das als Grundlage für die Arbeiten dieses Kapitels verwendet wird. Abschnitt 5.2 beschäftigt sich damit, wie einzelne Ecore-Objekte in XML Schema ausgedrückt werden können, bevor in Abschnitt 5.3 ein Algorithmus für eine Umformung von Ecore-Metamodellen in XML Schema entwickelt wird. In Abschnitt 5.4 wird das Testmodell mit diesem Algorithmus umgewandelt und das Resultat in Bezug auf dessen Verarbeitung durch Clio analysiert. Abschließend folgen in Abschnitt 5.5 Anmerkungen zur Datenintegrität.

5.1 Aufbau des Testsets

Zur Veranschaulichung der Konvertierung von Ecore-Objekten in XML Schema wurde ein Testmodell verwendet, das in Abbildung 5.1 als UML-Klassediagramm bzw. in Abbildung 5.2 modelliert in Eclipse dargestellt wird. Dieses Ecore-Testset soll in den folgenden Abschnitten in XML Schema konvertiert werden, das in der Folge als *Clio-Import-Schema* bezeichnet wird. Das Testmodell enthält folgende Ecore-Objekte:

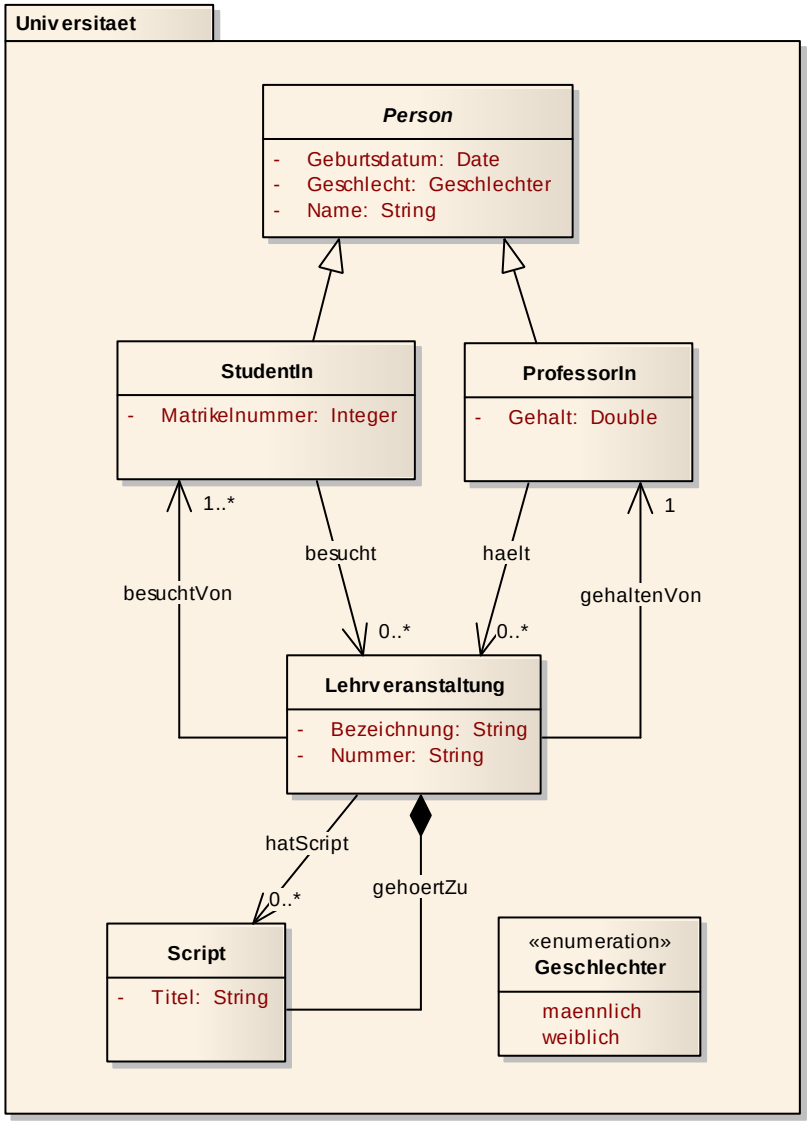

Abbildung 5.1: UML-Diagramm des Testmodells

- **EPackages** dienen zur Gruppierung der Ecore-Objekte zu einem Paket. Das gesamte Testset ist im Paket *Universitaet* verpackt.

- Das **EClass**-Objekt beschreibt die Klassen des Modells. Klassen werden durch einen Namen benannt und können Attribute und Referenzen beinhalten. Das Testset besteht aus den Klassen *Person, StudentIn, ProfessorIn, Lehrveranstaltung* und *Script*. Das *EClass*-Objekt kann unter anderem die Eigenschaft *abstract* sowie ein oder mehrere *ESuper Types* enthalten. Die Klasse *Person* ist eine abstrakte Klasse. Sie ist als diese in den Klassen *StudentIn* und *ProfessorIn* eingetragen, wodurch sie ihre Eigenschaften erben.

- Das Objekt **EAttribute** beschreibt die Attribute einer Klasse. Der Datentyp eines Attributs ist entweder ein simpler Datentyp oder vom Typ *EEnum* [WKS+06]. Die wesentlichen Eigenschaften von *EAttribute* sind *Name, EType, Lower Bound* und *Upper Bound*. Die Klasse *Person* besitzt die Attribute *Name, Geburtsdatum* und *Geschlecht* mit den in Tabelle 5.1 dargestellten Eigenschaften.

Name	EType	Lower Bound	Upper Bound	Default Value	Unique
Name	EString	1	1		true
Geburtsdatum	EDate	0	1		false
Geschlecht	Geschlechter	0	1	maennlich	false

Tabelle 5.1: Attribute der Klasse *Person* und ihre Eigenschaften

- **EEnum** dient zur Definition von Aufzählungstypen (*engl. enumeration type*). Das Testmodell enthält das *EEnum*-Konstrukt *Geschlechter* mit den Literalen *maennlich* und *weiblich* **.**

- Das **EReference**-Element dient zur Abbildung von Beziehungen zwischen Klassen. Die wesentlichen Eigenschaften von *EReference* sind *Name, Lower Bound, Upper Bound, EType* und *Containment*. Durch die Eigenschaft *Containment=true* lassen sich stärkere, existenzielle Abhängigkeiten modellieren. Das Testmodell beinhaltet sechs Referenzen. Die Referenz *haelt* der Klasse *ProfessorIn* bildet ab, dass jedEr ProfessorIn 0 oder mehrere Lehrveranstaltungen hält. Die Referenz *gehoertZu* ist eine existenzabhängige Beziehung zwischen *Script* und *Lehrveranstaltung* – ein Script existiert nur innerhalb der zugeordneten Lehrveranstaltung.

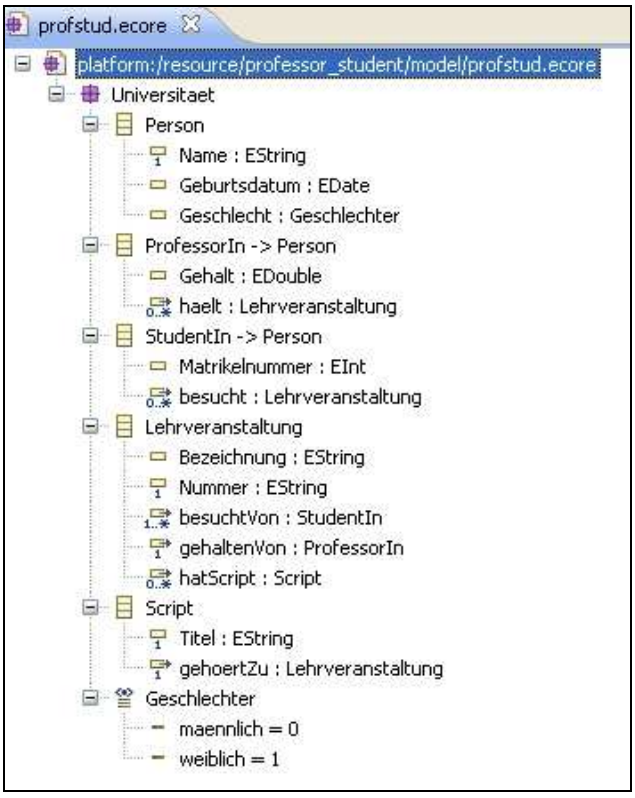

Abbildung 5.2: Modellierung des Testmodells in Eclipse

5.2 Ecore-Elemente und deren Abbildung in XML Schema

Dieser Abschnitt verwendet eine Serialisierung des Ecore-Testmodells im XMI-Format als Grundlage für die durchzuführende Umformung in XML Schema. Dabei soll für jedes Ecore-Konstrukt aus Abschnitt 5.1 ein passendes Gegenüber in XML Schema identifiziert werden.

5.2.1 Das Ecore-Paket

Das Paket *Universitaet* des Testmodells wird in Ecore folgendermaßen dargestellt:

Ecore

```
<ecore:EPackage ... name="Universitaet">
```

Dieses Paket wird als Wurzelelement im Clio-Import-Schema verwendet. Dabei wird die Eigenschaft *name* des Quellpakets in die gleichnamige Eigenschaft des Wurzelelements im Zieldokument übertragen. Für das Testmodell wird ein XML-Schema-Element mit *name=Universitaet* erzeugt.

XML Schema

```
<xs:element name="Universitaet">
    <xs:complexType>
        <xs:sequence>

        ...

        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Innerhalb des XML-Schema-Elements *xs:sequence* entstehen in der Folge Klassen, Referenzen und Attribute des Testmodells.

5.2.2 Umformung der Klassen

Für Ecore-Klassen werden entsprechende XML-Schema-Elemente erstellt, die den Namen der Ecore-Klasse erhalten. Handelt es sich um eine abstrakte Klasse, wird das jeweilige XML-Schema-Element zusätzlich um die Eigenschaft *abstract=true* erweitert. Die Klassen des Testmodells werden in Ecore folgendermaßen ausgedrückt:

Ecore

```
<eClassifiers xsi:type="ecore:EClass" name="Person"
 abstract="true">

<eClassifiers xsi:type="ecore:EClass" name="ProfessorIn"
 eSuperTypes="#//Person">

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

Nachstehend wird das Ergebnis der Umformung der Klasse *Lehrveranstaltung* in XML Schema dargestellt. Wie bereits beim Wurzelelement werden auch Ecore-Klassen auf das XML-Schema-Element *xs:element* übertragen, das deren Inhalt als komplexen Datentypen innerhalb einer Sequenz speichert [Pro02]. Auf die Behandlung der *EClass*-Eigenschaft *eSuperTypes* wird später im Rahmen der Umformung der Vererbung eingegangen.

XML Schema

```
<xs:element name="Universitaet">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Lehrveranstaltung">
                <xs:complexType>
                    <xs:sequence>

                    ...

                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            ...
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

5.2.3 Attribute einer Klasse

In XML Schema ist es gleichgültig, ob Informationen als *xs:attribute* oder *xs:element* abgespeichert werden. Beispielsweise bilden die beiden nachstehenden Codefragmente dieselbe Struktur ab:

XML Schema

```
<xs:element name="Person">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Name" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>


<xs:element name="Person">
    <xs:complexType>
          <xs:attribute name="Name" type="xs:string"
           use="required"/>
    </xs:complexType>
</xs:element>
```

Die Entscheidung, welches der beiden Elemente zur Umformung von *EAttribute* verwendet wird, liegt bei der/dem DesignerIn [Huc00]. Für eine bessere Übersicht der Schemata in Clio werden für die vorliegende Arbeit Klassenattribute vom Typ *EAttribute* als *xs:element* dargestellt. Diese Entscheidung hat einen weiteren Vorteil: In Ecore können Attribute mit multipler Kardinalität erzeugt werden, während das XML-Schema-Element *xs:attribute* lediglich die Multiplizitäten [0, 1] oder [1, 1] abbilden kann. Durch Verwendung einer Sequenz von *xs:element*-Elementen ist die Umformung von Ecore-Klassenattributen mit multipler Kardinalität möglich.

Zur Umformung der Ecore-Klassenattribute wird jeweils ein Element *xs:element* erzeugt, das folgende Bestandteile übernimmt:

ecore:EAttribute	**xs:element**
name	name
ecore:EDataType[17]	type
lowerBound	minOccurs
upperBound	maxOccurs

Tabelle 5.2: Attribute von EAttribute und deren Pendants in XML Schema

Es folgen zwei Beispiele für die Umformung von Attributen und deren Kardinalität:

Kardinalität in Ecore ist [0, 1]

Für ein Ecore-Attribut gilt, dass die Kardinalität standardmäßig als [0, 1] angenommen wird, wodurch folgender Code erzeugt wird:

Ecore

```
<eStructuralFeatures xsi:type="ecore:EAttribute" name="Gehalt"
 eType="ecore:EDataType
 http://www.eclipse.org/emf/2002/Ecore#//EDouble"/>
```

Ausgedrückt in XML Schema ergibt sich dieses Codefragement:

XML Schema

```
<xs:element name="Gehalt" type="xs:double" minOccurs="0"
 maxOccurs="1"/>
```

Kardinalität in Ecore ist [1, 1]

Ecore

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

XML Schema

```
<xs:element name="Name" type="xs:string" minOccurs="1"
 maxOccurs="1"/>
```

[17] Anmerkungen zur Übertragung der Datentypen folgen in Abschnitt 5.2.5.

5.2.4 Enumeration Type

Im Testmodell existiert ein EEnum-Objekt *Geschlechter*, für das die Werte *maennlich* und *weiblich* zugelassen sind.

Ecore

```
<eClassifiers xsi:type="ecore:EEnum" name="Geschlechter">
    <eLiterals name="maennlich"/>
    <eLiterals name="weiblich" value="1"/>
</eClassifiers>
```

Die Definition eines Datentypen mit Auswahlwerten erfolgt in XML Schema über das Element *xs:simpleType* und eine entsprechende Einschränkung [Pro02]. Umgeformt in XML Schema schaut der Aufzählungstyp *Geschlechter* folgendermaßen aus:

XML Schema

```
<xs:simpleType name="Geschlechter" final="restriction">
    <xs:restriction base="xs:string">
        <xs:enumeration value="maennlich"/>
        <xs:enumeration value="weiblich"/>
    </xs:restriction>
</xs:simpleType>
```

Wie dieses Beispiel zeigt, benötigt das zur Einschränkung verwendete XML Schema-Element *xs:restriction* zwingend einen Datentypen, der als Basis für die Einschränkung gilt. Da eine solche Information in den Quelldaten nicht vorhanden ist, wird für diese Arbeit als *xs:restriction base* grundsätzlich der Datentyp *xs:string* verwendet.

5.2.5 Umformung von Datentypen

Der Transfer von Ecore-Datentypen in entsprechende XML-Schema-Datentypen kann über eine Castingtabelle erfolgen. Tabelle 5.3 zeigt sowohl eine Lösung zur Umformung der Ecore Datentypen *EDataType* als auch von selbst erstellten Aufzählungen vom Typ *EEnum*. Je nach Bedarf muss die Castingtabelle auch auf andere Ecore-Datentypen erweitert werden.

eType	XML Schema type
ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString	xs:string
ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EDate	xs:date
ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt	xs:integer
ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EDouble	xs:double
ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBoolean	xs:boolean
#//Geschlechter	Geschlechter

Tabelle 5.3: Castingtabelle zur Umformung von Datentypen

5.2.6 Referenzen zwischen Klassen

Mit dem Ecore-Element *EReference* können Assoziationen zwischen Klassen modelliert werden. Zusätzlich ist es möglich, mit Hilfe des *EReference*-Attributs *containment=true* die semantisch strengere Komposition abzubilden.

Sowohl für die Assoziation als auch für die Komposition wird zunächst ein Element *xs:element* mit dem Namen der Referenz erzeugt.

Ecore

```
<eStructuralFeatures xsi:type="ecore:EReference"
 name="hatScript" upperBound="-1" eType="#//Script"
 containment="true"/>
```

XML Schema

```
<xs:element name="hatScript" minOccurs="0"
 maxOccurs="unbounded"/>
```

Auf dieser Grundlage können Assoziation und Komposition modelliert werden.

5.2.6.1 Die Assoziation – Referenz über Fremdschlüssel

Beziehungen zwischen Klassen können über eine Fremdschlüsselreferenz abgebildet werden [Pro02]. Folgender Ecore-Code stellt die Beziehung *besuchtVon* der Klasse *Lehrveranstaltung* dar, wobei *StudentIn* die Zielklasse der Beziehung ist.

Ecore

```
<eStructuralFeatures xsi:type="ecore:EReference"
 name="besuchtVon" lowerBound="1" upperBound="-1"
 eType="#//StudentIn"/>
```

Mittels der Elemente *xs:key* und *xs:keyref* ist es möglich, in XML Schema eine Fremdschlüsselbeziehung abzubilden. Die Pfade der betroffenen Elemente können über XPath-Ausdrücke definiert werden.

Der erste Schritt zur Umformung einer Assoziation ist die Definition eines Schlüsselelements der Klasse am anderen Ende der Beziehung.

XML Schema

```
<xs:key name="StudentIn_Key">
    <xs:selector xpath="./StudentIn"/>
    <xs:field xpath="Name"/>
</xs:key>
```

Im zweiten Schritt wird die eigentliche Referenz erstellt, die auf den zuvor definierten Schlüssel verweist.

XML Schema

```
<xs:keyref name="Lehrveranstaltung_besuchtVon"
 refer="StudentIn_Key">
    <xs:selector xpath="./Lehrveranstaltung"/>
    <xs:field xpath="besuchtVon"/>
</xs:keyref>
```

5.2.6.2 Die Komposition – geschachtelte Struktur

Die Komposition kann durch eine Verschachtelung (*engl. nesting*) von Elementen umgesetzt werden [Pro02]. Nachstehende Beziehung *hatScript* der Klasse *Lehrveranstaltung* verweist auf die Klasse *Script*, die nur existieren soll, wenn auch *Lehrveranstaltung* existiert.

Ecore

```
<eStructuralFeatures xsi:type="ecore:EReference"
 name="hatScript" upperBound="-1" eType="#//Script"
 containment="true"/>
```

Obiges Ecore-Fragment wird in XML Schema umgeformt, indem das zuvor erstellte Element *hatScript* eine Sequenz *xs:sequence* erhält. Innerhalb dieser Sequenz wird das Element *Script* erzeugt, wodurch eine existenzabhängige Beziehung modelliert wird.

XML Schema

```
<xs:element name="hatScript" ... >
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Script">

                ...

            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

5.2.7 Behandlung der Vererbung

Durch Vererbung ist es möglich ein Klassenobjekt zu definieren, das als Erweiterung oder Ableitung einer anderen Klasse dient [Huc00]. Folgender Ecore-Code weist der Klasse *ProfessorIn* die Klasse *Person* als Superklasse zu, wodurch *ProfessorIn* sämtliche Eigenschaften von *Person* erbt.

Ecore

```
<eClassifiers xsi:type="ecore:EClass" name="ProfessorIn"
 eSuperTypes="#//Person">
```

In XML Schema kann Vererbung durch den *xs:extension-base*-Mechanismus implementiert werden, welcher einem XML-Element ein anderes Element als Basis zuweist. Nachstehende XML-Schema-Ausschnitte zeigen die Modellierung der Erweiterung der Klasse *ProfessorIn* um die abstrakte Klasse *Person*. Die Umsetzung umfasst zwei Schritte:

1. Definition der Klasse *Person* als komplexer Datentyp.

XML Schema

```
<xs:complexType name="Person" abstract="true">
    <xs:sequence>
        <xs:element name="Name" ... />
        <xs:element name="Geburtsdatum" ... />
        <xs:element name="Geschlecht" ... />
    </xs:sequence>
</xs:complexType>
```

2. Verwendung des komplexen Datentypen zur Erweiterung von *ProfessorIn*.

XML Schema

```
<xs:element name="ProfessorIn" ... >
    <xs:complexType>
        <xs:complexContent>
            <xs:extension base="ProfessorIn">
                <xs:sequence>
                    <xs:element name="Gehalt" ... />
                    <xs:element name="haelt" ... />
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
</xs:element>
```

5.3 *EcoreToXsd* – Algorithmus zur Umformung von Metamodellen in XML Schema

Im vorigen Abschnitt wurde aufgezeigt, wie einzelne Ecore-Objekte in XML-Schema-Elemente umgeformt werden können. Ziel dieses Abschnittes ist es, einen Algorithmus zu entwickeln, der diesen Transfer nach Möglichkeit automatisch durchführen kann. Falls Interaktionen durch den/die BenutzerIn notwendig sind, sollen diese erkannt und erläutert werden. Der Umformungsalgorithmus, der in der Folge auch als *EcoreToXsd* bezeichnet wird, besteht aus sechs Schritten:

Schritt 1 - Behandlung der Objekte *EEnum*

Im ersten Schritt wird für jedes Objekt *EEnum* ein XML-Schema-Element vom Typ *xs:simpleType* wie in Abschnitt 5.2.4 beschrieben erzeugt.

Schritt 2 - Umformung der Ecore-Klassen *EClass*

Sämtliche Klassen werden zunächst als komplexe Datentypen definiert, damit diese als Basis zur Erweiterung anderer Klassen verwendet werden können.

1. Erzeugung eines XML Schema-Elements *xs:complexType* für jedes Objekt vom Typ *EClass* wie in Abschnitt 5.2.2 beschrieben. Zusätzlich zum Namen der Klasse wird dabei die Information übernommen, ob es sich um eine abstrakte Klasse handelt oder nicht. Für die Klassen *Person* bzw. *ProfessorIn* wird dabei folgender Code erstellt:

```
<xs:complexType name="Person" abstract="true">
</xs:complexType>

<xs:complexType name="ProfessorIn">
</xs:complexType>
```

2. Für jede Klasse, die von einer Superklasse abgeleitet ist, bekommt der zuvor erstellte komplexe Datentyp diese als Erweiterung zugewiesen.

```
<xs:complexType name="ProfessorIn">
    <xs:complexContent>
        <xs:extension base="Person">
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

3. Erzeugung einer Sequenz für jede Klasse, die ihre Attribute und Referenzen als Elemente vom Typ *xs:element* beinhaltet.

```
<xs:complexType name="ProfessorIn">
    <xs:complexContent>
        <xs:extension base="Person">
            <xs:sequence>
                <xs:element name="Gehalt">
                <xs:element name="haelt"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

4. Zuweisung der Datentypen für alle Klassenattribute gemäß Tabelle 5.3.

```
...

<xs:element name="Gehalt" type="xs:double"/>
<xs:element name="haelt"/>

...
```

5. Vergabe der Kardinalität für sämtliche Klassenattribute und Referenzen.

```
...

<xs:element name="Gehalt" type="xs:double" minOccurs="0"/>
<xs:element name="haelt" minOccurs="0" maxOccurs="unbounded"/>

...
```

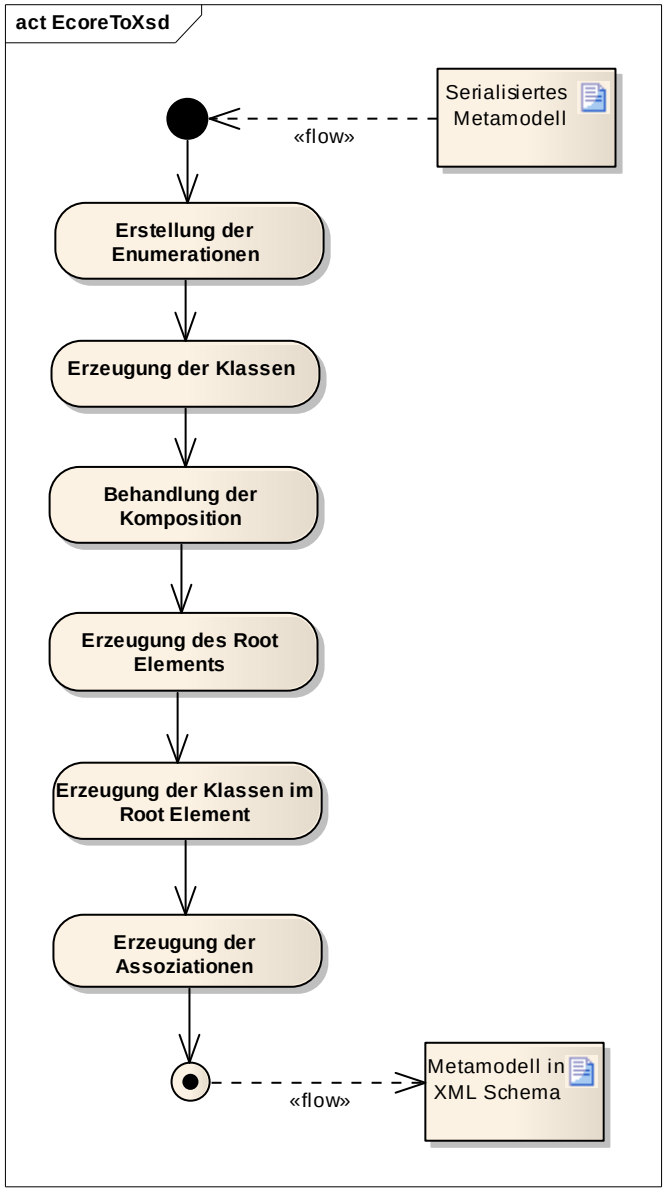

Abbildung 5.3. *EcoreToXsd* dargestellt als UML-Aktivitätsdiagramm

Schritt 3 - Behandlung der Komposition

Wie in Abschnitt 5.2.6.2 festgestellt, wird für die Komposition eine geschachtelte Struktur verwendet. Dazu wird die existenzabhängige Klasse ihrem Kompositum hierarchisch unterstellt. Für das konkrete Beispiel *hatScript* der Klasse *Lehrveranstaltung* wird in einer *xs:sequence* ein neues Element mit dem Namen *Script* erstellt, das wiederum den Inhalt der Klasse *Script* erhält. Da in Schritt 2 bereits sämtliche Klassen mittels komplexer Datentypen erzeugt wurden, können diese jetzt verwendet werden, indem sie über das XML-Schema-Element *xs:extension* als Basis für eine Erweiterung angegeben werden. Untenstehender Code weist dem Element *Script* der

Komposition *hatScript* eine Erweiterung um sämtliche Eigenschaften der in Schritt 2 erzeugten Klasse *Script* zu.

```
<xs:element name="hatScript" minOccurs="0"
 maxOccurs="unbounded">
   <xs:complexType>
       <xs:sequence>
           <xs:element name="Script" minOccurs="0"
            maxOccurs="unbounded">
              <xs:complexType>
                  <xs:complexContent>
                      <xs:extension base="Script"/>
                  </xs:complexContent>
             </xs:complexType>
           </xs:element>
       </xs:sequence>
   </xs:complexType>
</xs:element>
```

Zusätzlich erhält das neu erstellte Element die Kardinalität seines Kompositums. Im obigen Beispiel wurde *Script* dieselbe Multiplizität wie *hatScript* zugewiesen.

Schritt 4 - Erstellung des Root-Elements

Für das Ecore-Paket *EPackage* wird das Root-Element des XML-Schema-Dokuments erzeugt. Siehe dazu Abschnitt 5.2.1.

```
<xs:element name="Universitaet">
   <xs:complexType>
       <xs:sequence>

           ...
```

Schritt 5 - Erzeugung der Klassen im Wurzelelement

Das Root-Element beinhaltet eine Sequenz von XML-Schema-Elementen, die die Klassen des Ecore-Modells darstellen. Die Übersetzung der Ecore-Klassen auf XML Schema erfolgt durch Verwendung des Elements *xs:element*. Zusätzlich bekommen diese Elemente jeweils ihren in Schritt 2 erzeugten komplexen Datentypen als *xs:extension* zugewiesen. Dabei gelten folgende Besonderheiten:

- Abstrakte Klassen werden nicht erstellt, da sie lediglich als Erweiterungsbasis ihrer Subklassen dienen. Die Realisierung dieser Erweiterungen wurde bereits in Schritt 2 bei der Erstellung der Klassen umgesetzt.

- Im Root-Element werden nur jene Klassen erzeugt, die nicht als Komponente in einer Kompositionsbeziehung verwendet werden. Klassen, die als Komponenten dienen, wurden bereits in Schritt 3 verschachtelt.

- Sämtliche Kinder des Wurzelelements bekommen die Kardinalität [0, *].

Für das Testmodell werden Elemente für die Klassen *ProfessorIn*, *StudentIn* und *Lehrveranstaltung* erstellt. Für die Klassen *Person* und *Script* werden hingegen keine Elemente erzeugt, da es sich bei *Person* um eine abstrakte Klasse handelt und *Script* nur als Komponente der Beziehung *hatScript* verwendet wird.

```
<xs:element name="ProfessorIn" minOccurs="0"
 maxOccurs="unbounded">
   <xs:complexType>
       <xs:complexContent>
           <xs:extension base="ProfessorIn"/>
       </xs:complexContent>
   </xs:complexType>
</xs:element>
<xs:element name="StudentIn" minOccurs="0"
 maxOccurs="unbounded">
   <xs:complexType>
       <xs:complexContent>
           <xs:extension base="StudentIn"/>
       </xs:complexContent>
   </xs:complexType>
</xs:element>
<xs:element name="Lehrveranstaltung" minOccurs="0"
 maxOccurs="unbounded">
   <xs:complexType>
       <xs:complexContent>
           <xs:extension base="Lehrveranstaltung"/>
       </xs:complexContent>
   </xs:complexType>
</xs:element>
```

Schritt 6 - Erzeugung der Assoziationen mittels Fremdschlüsselreferenzen

Im letzten Schritt werden die Fremdschlüsselreferenzen erstellt. Zunächst muss für jede Klasse, auf die eine Referenz erstellt werden soll, ein Attribut als eindeutiger Schlüssel ausgewählt werden. Da in den zugrundeliegenden Metamodellen keine Schlüssel existieren, muss diese Auswahl manuell von der/dem BenutzerIn durchgeführt werden. Durch die Notwendigkeit dieser Interaktion kann das ursprüngliche Ziel einer automatisierten Abarbeitung nicht erreicht werden.

Bei der manuellen Schlüsselauswahl kommen sämtliche Attribute der Klasse sowie ihrer eventuell vorhandenen Superklasse in Frage. Dabei sollte beachtet werden, dass die Eindeutigkeit des entsprechenden Attributs nur gewährleistet ist, falls es die Eigenschaft *unique=true* hat. Die ausgewählten Schlüsselattribute werden mit dem XML-Schema-Element *xs:key* definiert. Zur Navigation innerhalb des Dokumentes kommt XPath zum Einsatz. Tabelle 5.4 zeigt die Klassen des Beispielmodells mit den verwendeten Schlüsselattributen. Die Definition des Schlüssels für die Klasse *Lehrveranstaltung* erfolgt durch folgendes Codefragment:

```
<xs:key name="Lehrveranstaltung_Key">
    <xs:selector xpath="./Lehrveranstaltung"/>
    <xs:field xpath="Nummer"/>
</xs:key>
```

Klasse	Schlüsselattribut
ProfessorIn	Name aus der Superklasse Person
StudentIn	Name aus der Superklasse Person
Lehrveranstaltung	Nummer

Tabelle 5.4: Schlüsselattribute der Klassen des Beispielmodells

Nachdem die benötigten Schlüssel erzeugt wurden, können die Fremdschlüsselbeziehungen hergestellt werden. Dazu wird das XML Schema-Element *xs:keyref* verwendet. Ziel der Referenz ist der bereits erstellte Schlüssel. Die Navigation im XML-Schema-Dokument erfolgt wieder mittels XPath. Folgender Code bildet die Assoziation *haelt* der Klasse *ProfessorIn* unter Verwendung des erstellten Schlüssels ab:

```
<xs:keyref name="ProfessorIn_haelt"
 refer="Lehrveranstaltung_Key">
    <xs:selector xpath="./ProfessorIn"/>
    <xs:field xpath="haelt"/>
</xs:keyref>
```

Alternativer Ansatz durch Vergabe eindeutiger Klassen-IDs

Wie in Schritt 6 festgestellt wurde, führt das Nichtvorhandensein von IDs in den Klassen der Metamodelle zu zwei Problemen:

1. Da die Auswahl von Schlüsselattributen manuell durchgeführt werden muss, kann das ursprünglich definierte Ziel der Entwicklung eines Algorithmus, der automatisch abgearbeitet werden kann, nicht erreicht werden.

2. Die Eindeutigkeit des Schlüsselattributs ist nicht gewährleistet, sofern die jeweilige Klasse kein Attribut mit der Eigenschaft *unique=true* besitzt.

Aus diesem Grund wurde für die vorliegende Arbeit ursprünglich ein Ansatz gewählt, der für jede Klasse zusätzlich ein Attribut erzeugt, das als ID verwendet wird. Außerdem soll dessen Eindeutigkeit durch eine *unique constraint* garantiert werden.

```
<xs:element name="Person">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ID" type="xs:integer"/>
            <xs:element name="Name" type="xs:string"/>

            ...

        </xs:sequence>
    </xs:complexType>
</xs:element>

<xs:unique name="Person_ID">
    <xs:selector xpath="./Person"/>
    <xs:field xpath="ID"/>
</xs:unique>
```

Obwohl durch diese Idee die oben beschriebene Problematik umgangen werden kann, ist sie nicht praktikabel, da die in der Folge erzeugten Schema-Mappings durch die zusätzlich generierten Schlüsseldaten verfälscht werden. Zur Umformung der Metamodelle wird in dieser Arbeit deshalb der ursprüngliche Ansatz verwendet, bei dem die Schlüsselattribute manuell ausgewählt werden. Zusätzlich wird davon ausgegangen, dass deren Inhalt eindeutig ist.

5.4 Anwendung von *EcoreToXsd* auf das Testmodell

Abbildung 5.4 zeigt das Testmodell nach der Umformung durch *EcoreToXsd* dargestellt in Clios *Schema View*[18]. Es folgen einige Anmerkungen zur Interpretation des Schemas in Clio.

5.4.1 Auflösung der Vererbung

Obwohl das Konzept der Vererbung im XML Schema modelliert wurde, wird diese von Clio aufgelöst, indem es Klassen mit Superklassen um deren Attribute erweitert. Dieser Umstand ist ein logischer Schritt der Umwandlung einer objektorientierten Struktur in ein relationales Schema.

Für den konkreten Fall - *ProfessorIn* hat *Person* als Superklasse -, hat diese Auflösung die Konsequenz, dass *ProfessorIn* in Clio dieselbe Struktur hat, wie wenn sie durch nachstehenden Code modelliert worden wäre.

[18] Das serialisierte Ecore-Metamodell und das resultierende XML Schema nach Anwendung von *EcoreToXsd* befinden sich im Anhang der vorliegenden Arbeit.

```
<xs:element name="ProfessorIn">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Name" ... />
            <xs:element name="Geburtsdatum" ... />
            <xs:element name="Geschlecht" ... />
            <xs:element name="Gehalt" ... />
            <xs:element name="haelt" ... />
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Die Elemente *Name*, *Geburtsdatum* und *Geschlecht* stammen dabei ursprünglich aus der Klasse *Person*, während *Gehalt* und *haelt* in der Klasse *ProfessorIn* definiert sind.

5.4.2 Darstellung der Referenzen

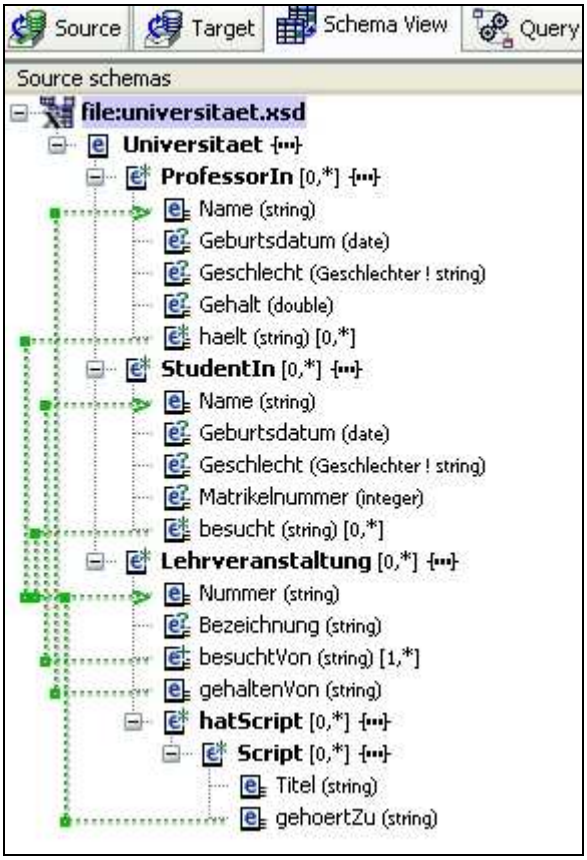

Abbildung 5.4: Darstellung des Testmodells in Clios *Schema View*

Wie in Abschnitt 3.2 festgestellt, werden Referenzen in Clio durch Pfeile abgebildet, die auf das jeweilige Attribut zeigen, das als Schlüssel definiert wurde.

5.4.3 Darstellung der Komposition

Die Komposition wird durch eine hierarchische Struktur dargestellt. Abbildung 5.4 zeigt die Einbettung der Klasse *Script* in die abhängigkeitsbedingte Referenz *hatScript*. Man beachte auch die Auflösung der Klasse *Script*, die gemäß Schritt 3 von *EcoreToXsd* durch Erweiterung um den komplexen Datentypen *Script* modelliert wurde.

5.5 Anmerkungen zur Datenintegrität bei der Umformung der Metamodelle

Auch wenn die in diesem Kapitel dargestellte Methodik zur Umwandlung von Ecore-Metamodellen in XML-Schema-Dokumente ausreichend ist, um die Erstellung von Schema Mappings mit Clio zu testen, ist die Integrität der Daten nicht gewährleistet. Dazu muss erwähnt werden, dass eine Übertragung von objektorientierten Strukturen in eine flachere, relationale Abbildung auf jeden Fall einen Informationsverlust bedeutet. Zusammenfassend sollen in diesem Abschnitt noch einmal die wesentlichen Einschränkungen bei der Schemaumformung hervorgehoben werden.

5.5.1 Problematik der Umformung von Assoziationen

Zur Darstellung von Assoziationen zwischen Klassen objektorientierter Strukturen können im relationalen Modell Fremdschlüsselbeziehungen verwendet werden. Da in den Klassen der vorliegenden Ecore-Metamodelle keine Schlüsselattribute definiert sind, müssen diese hinzugefügt oder aus den bestehenden Attributen ausgewählt werden. Wie in Abschnitt 0 festgestellt wurde, ist das zusätzliche Hinzufügen von Schlüsselattributen nicht praktikabel, da es die Mapping-Ergebnisse verfälscht. Aufgrund dieser Tatsache wurden in der vorliegenden Arbeit die Schlüsselattribute manuell ausgewählt, wodurch es nicht möglich ist, eine Implementierung von *EcoreToXsd* zu erstellen, die keiner BenutzerInneninteraktion bedarf.

Weitere Probleme können entstehen, falls in den Quellklassen keine eindeutigen Attribute vorhanden sind bzw. falls eine Referenz auf eine Klasse zeigt, in der kein Attribut existiert. Solche Situationen bedürfen einer individuellen Untersuchung und müssen entsprechend manuell aufgelöst werden. Eine Betrachtung sämtlicher Eventualitäten bei der Umformung der Ecore-Referenzen nach XML Schema würde den Rahmen der vorliegenden Arbeit sprengen und ist für die Evaluierung in Kapitel 6 nicht notwendig.

5.5.2 Probleme bei der Übertragung der Vererbung

Vererbung wird beim Einlesen der Schemata von Clio aufgelöst, wodurch ein Teil der Struktur der Ecore-Metamodelle verloren geht. Wie bereits erwähnt, lässt sich dieser Informationsverlust nicht vermeiden, da eine objektorientierte Struktur in eine relationale überführt wird, was nur durch eine entsprechende Verflachung der Quelldaten möglich ist.

Weiters ist es nicht möglich, Vererbung abzubilden, falls eine Klasse mehrere Superklassen hat. Für das XML-Schema-Element *xs:extension base*, das zur Modellierung von Vererbung verwendet wird, kann lediglich ein komplexer Datentyp als Basis angegeben werden [KST02]. Die Lösung des Problems könnte durch Verwendung von Dummyklassen verwirklicht werden. Andererseits besteht die Möglichkeit, komplett auf die Abbildung von Vererbung zu verzichten, da diese Information von Clio ohnehin verworfen wird. Stattdessen könnten die Eigenschaften der jeweiligen Superklassen direkt in den Klassen modelliert werden.

5.5.3 Sonstiges

Attribute und Referenzen von Ecore-Metamodellen können Eigenschaften aufweisen, die in diesem Kapitel nicht behandelt wurden. Teilweise besteht die Möglichkeit, zusätzliche Informationen der Ecore-Metamodelle in XML Schema abzubilden. Das ist für die vorliegende Arbeit nicht notwendig, da die XML Schemata, die durch den in diesem Kapitel entwickelten Algorithmus erzeugt werden, ausreichend genau sind, um die Evaluierung im folgenden Kapitel durchführen zu können.

Kapitel 6

Schema Mapping mit Clio angewendet auf Metamodelle

In diesem Kapitel soll evaluiert werden, inwieweit sich Clio zum Mappen von Metamodellen eignet. Dazu wird in Abschnitt 6.1 ein Testset erzeugt, in dem zwei Ecore-basierte Metamodelle mit Hilfe des in Abschnitt 5.3 entwickelten Algorithmus *EcoreToXsd* in XML Schema umgewandelt werden. Zusätzlich sollen die überlappenden Teile der beiden Testmodelle ausgemacht werden. In Abschnitt 6.2 wird die Integration der Testmodelle mit Clio anhand von Beispielinstanzen evaluiert. Schließlich werden in 6.3 die Testergebnisse zusammengefasst.

6.1 Das Testset

Für die Evaluierung von Clio zur Integration von Metamodellen wurde mit verschiedenen Inputschemata experimentiert. Aufgrund des hohen praktischen Stellenwerts von UML erfolgten die Tests der vorliegenden Arbeit mit den Metamodellen von UML 1.4 und UML 2.0. Weitere Experimente wurden mit den Metamodellen von *Entity-Relationship*-(ER)-Diagrammen und Teilen der *Web Modeling Language* (WebML) durchgeführt.

Abbildung 6.1und Abbildung 6.2 zeigen die Klassendiagramme der beiden UML-Metamodelle.

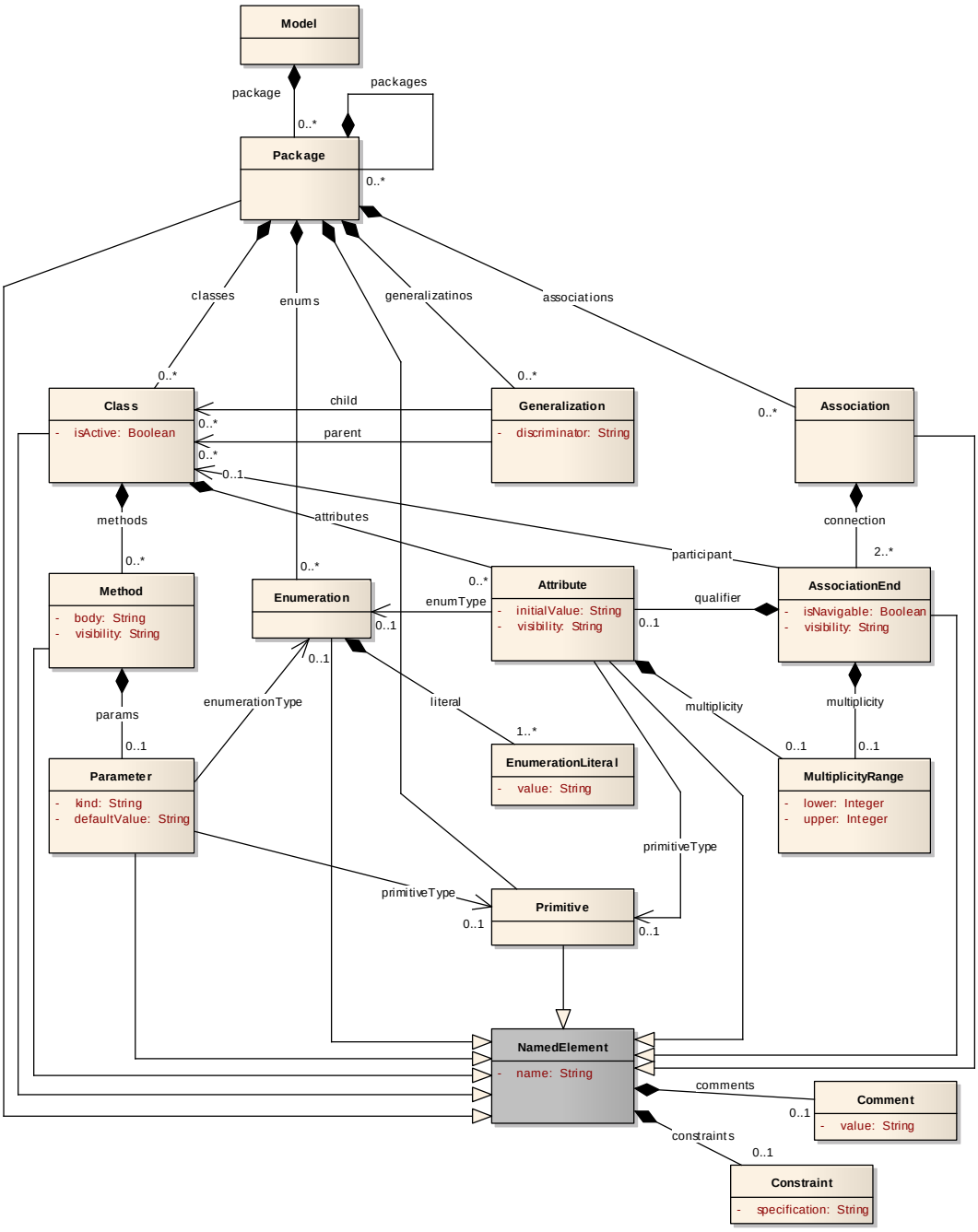

Abbildung 6.1: Klassendiagramm des UML-1.4-Metamodells aus [Kar08]

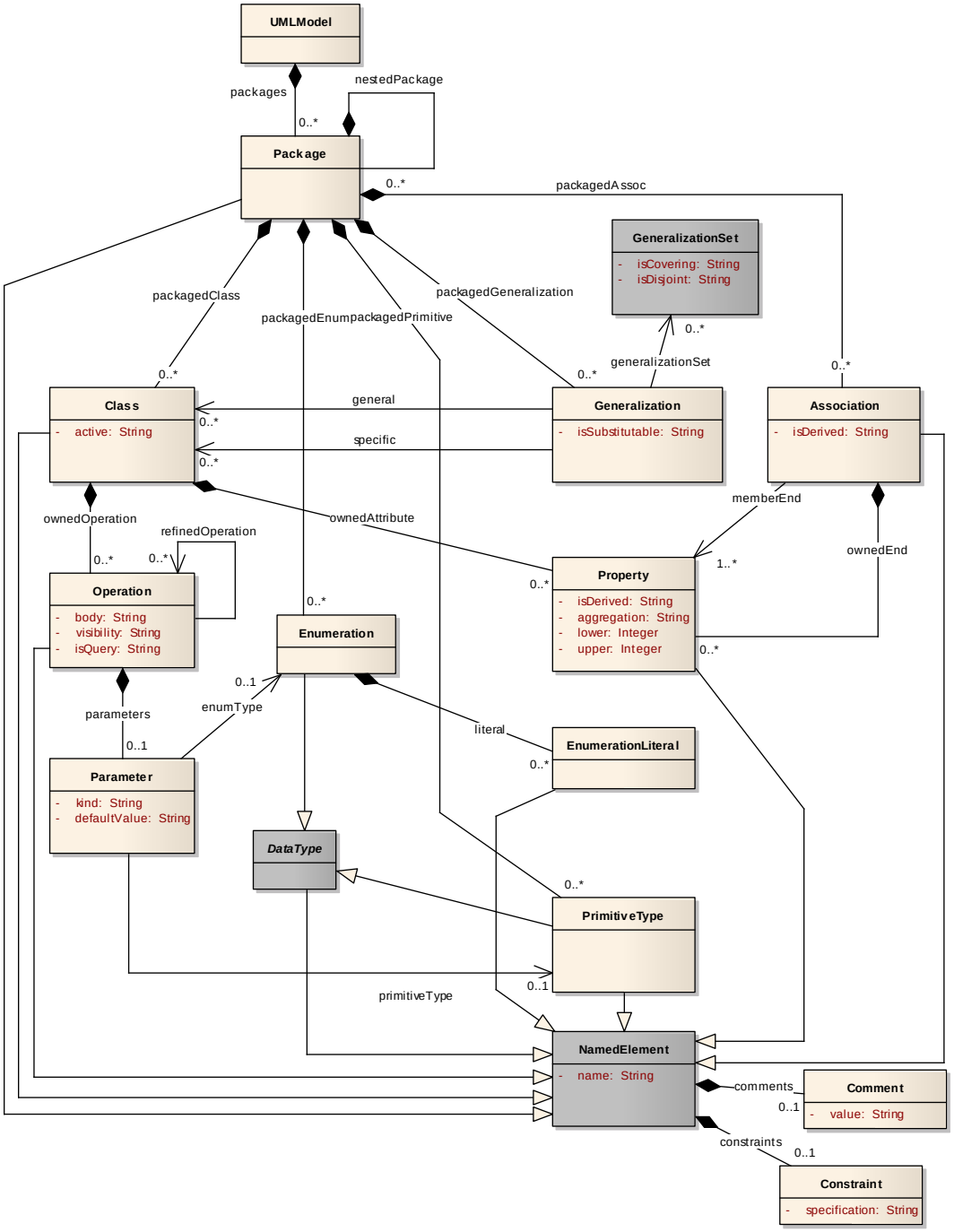

Abbildung 6.2: Klassendiagramm des UML-2.0-Metamodells aus [Kar08]

6.1.1 Erzeugung der Testschemata: *EcoreToXsd* in der Praxis

Die Ecore-basierten Metamodelle von UML 1.4 und UML 2.0 wurden mit *EcoreToXsd* in XML Schemata umgeformt. Zur besseren Übersicht wurde der Inhalt der

Klasse *NamedElement* in beiden Modellen auf das Attribut *name* reduziert[19]. Bei der Umformung ergaben sich die folgenden Besonderheiten.

6.1.1.1 Rekursive Komposition

Sowohl das Metamodell von UML 1.4 als auch jenes von UML 2.0 enthalten eine rekursive Kompositionsbeziehung. Die Klassen *Package* aus Abbildung 6.1 und Abbildung 6.2 enthalten mit *packages* bzw. *nestedPackages* Kompositionen auf sich selbst. Gemäß Abschnitt 5.2.6.2 wird das Konzept der Komposition in XML Schema durch eine geschachtelte Struktur dargestellt. Dabei wird eine Erweiterung über den entsprechenden komplexen Datentypen durch Verwendung von *xs:extension base* erzeugt. Listing 6.1 zeigt den dafür generierten XML-Schema-Code der Klasse *Package* aus UML 1.4.

```
<xs:complexType name="Package">

    ...

    <xs:element name="packages" ... >
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Package" ... >
                    <xs:complexType>
                        <xs:complexContent>
                            <xs:extension base="Package"/>
                        </xs:complexContent>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    ...

</xs:complexType>
```

Listing 6.1: Rekursive Komposition in UML 1.4

Diese Modellierung bildet eine endlose Verschachtelung ab und erzeugt eine Fehlermeldung beim Einlesen des Schemas in Clio mit der Folge, dass das Schema nicht geöffnet werden kann. Das Problem kann umgangen werden, indem die Klasse *Pa-*

---

[19] Aufgrund der Tatsache, dass die meisten Klassen der UML-Modelle von *NamedElement* abgeleitet sind, ist der Evaluierungsteil der vorliegenden Arbeit übersichtlicher, wenn *NamedElement* so einfach wie möglich gehalten wird. Aus diesem Grund wurde in *NamedElement* auf die Kompositionsbeziehungen *comments* und *constraints* auf die Klassen *Comment* bzw. *Constraint* verzichtet – lediglich das Attribut *name* wurde modelliert.

ckage in XML Schema über das *xs:element* Attribut *type=Package* modelliert wird wie in Listing 6.2 dargestellt.

```
...

<xs:element name="packages" ... >
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Package" type="Package" ... />
        </xs:sequence>
    </xs:complexType>
</xs:element>

...
```

Listing 6.2: Modellierung der rekursiven Komposition unter Verwendung von *xs:element type*

Diese Struktur kann von Clio gelesen werden, wodurch eine endlose Verschachtelung erreicht wird. Wird in diesem Fall allerdings ein Mapping erzeugt, bringt es Clio zum Absturz, weshalb auf die Modellierung von rekursiven Kompositionen verzichtet werden muss.

6.1.1.2 Assoziationen auf Klassen ohne Schlüsselattribut

Wie bereits in Abschnitt 5.5.1 angedeutet, kommt es zu Problemen bei der Erstellung von Referenzen auf Klassen, die kein Attribut haben, das als Schlüsselattribut verwendet werden kann. Ein Beispiel dafür ist die Referenz *generalizationSet* der Klasse *Generalization* aus dem UML-2.0-Metamodell. Diese Assoziation zeigt auf die Klasse *GeneralizationSet*, in der kein geeignetes Schlüsselattribut vorhanden ist. Obwohl es möglich wäre, solche Assoziationen über ein zusätzlich eingefügtes Attribut abzubilden, das als ID verwendet werden kann, wird in der vorliegenden Arbeit auf eine derartige Modellierung verzichtet, um die resultierenden Mappings nicht zu verfälschen.

6.1.2 Korrespondenzen der Testmodelle

Als Grundlage für die Erzeugung der Wertkorrespondenzen in Clio steht ein Mapping sämtlicher sich überlappender Elemente der beiden Testmodelle zur Verfügung. [WKS+07] enthält eine Klassifizierung möglicher Kombinationen, die sich beim Mappen von Elementen von Metamodellen ergeben können. Diese sind in Tabelle 6.1 dargestellt.

	Class	Attribute	Relationship
Class	C2C	C2A	C2R
Attribute	A2C	A2A	A2R
Relationship	R2C	R2A	R2R

Tabelle 6.1: Mögliche Kombinationen von Mappings

Wie bereits in Abschnitt 4.2.1.1 festgestellt, können in Clio nur Wertkorrespondenzen zwischen Blattelementen erstellt werden. Umgekehrt ist es nicht möglich, Wertzuweisungen für Innere Knoten zu erstellen. In Bezug auf die verwendeten XML-Schema-Metamodelle bedeutet das, dass lediglich zwei Kombinationen aus Tabelle 6.1 verwendet werden können, deren Bestandteile als Blattelemente modelliert wurden:

- A2A – Attribute To Attribute
- R2R – Relationship To Relationship

Für R2R Kombinationen gilt außerdem die Einschränkung, dass nur Wertkorrespondenzen für Assoziationen erstellt werden können, nicht aber für Kompositionen. Letztere werden in XML Schema durch die Verwendung der Verschachtelung als Innere Knoten modelliert.

Die überlappenden Teile der Klassendiagramme von UML 1.4 und UML 2.0 sind in Tabelle 6.2 dargestellt [Kar08]. Die Elemente von UML 1.4 werden in der zweiten Spalte gezeigt. In der vierten Spalte finden sich die dazugehörenden Äquivalenzen aus UML 2.0. Die für die vorliegende Arbeit relevanten Wertkorrespondenzen sind farblich hervorgehoben.

Nr.	UML 1.4	Typ	UML 2.0
1	Class	C2C	Class
2	Class.name	A2A	Class.name
3	Class.isActive	A2A	Class.active
4	Class_methods_Method	R2R	Class_ownedOperation_Operation
5	Class_comments_Comment	R2R	Class_comments_Comment
6	Class_constraints_Constraint	R2R	Class_constraints_Constraint
7	Method	C2C	Operation
8	Method.name	A2A	Operation.name
9	Method.visibility	A2A	Operation.visibility
10	Method.body	A2A	Operation.body
11	Method_comments_Comment	R2R	Operation_comments_Comment
12	Method_constraints_Constraint	R2R	Operation_constraints_Constraint
13	Method_params_Parameter	R2R	Operation_parameters_Parameter
14	Attribute	C2C	Property
15	MultiplicityRange.lower,	A2A	Property.lower
16	Multiplicity.Range.upper,	A2A	Property.upper
17	AssociationEnd	cond. C2C	Property
18	Class_attributes_Attribute	R2R	Class_ownedAttribute_Property
19	Attribute.name	A2A	Property.name
20	Generalization	C2C	Generalization
21	Generalization_child_Class	R2R	Generalization_specific_Class
22	Generalization_parent_Class	R2R	Generalization_general_Class
23	Package	C2C	Package
24	Package.name	A2A	Package.name
25	Package_generalizations_Generalization	R2R	Package_packagedGeneralizations_Generalization
26	Package_packages_Package	R2R	Package_nestedPackage_Package
27	Package_classes_Class	R2R	Package_packagedClass_Class
28	Package_comments_Comment	R2R	Package_comments_Comment
29	Package_enums_Enumeration	R2R	Package_packagedEnum_Enumeration
30	Package_associations_Association	R2R	Package_packagedAssoc_Association
31	Comment	C2C	Comment
32	Comment.body	A2A	Comment.value
33	Package_constraints_Constraint	R2R	Package_constraints_Constraint
34	Package_primitiveTypes_Primitive	R2R	Package_packagedPrmitive_PrimitveType
35	Association	C2C	Association
36	Association.name	A2A	Association.name
37	Association_comments_Comment	R2R	Association_comments_Comment
38	Association_constraints_Constraint	R2R	Association_constraints_Constraint
39	Model	C2C	UMLModel
40	Model_package_Package	R2R	UMLModel_packages_Package
41	Primitive	C2C	PrimitiveType
42	Primitive.name	A2A	PrimitiveType.name
43	Primitive_comments_Comment	R2R	PrimitiveType_comments_Comment
44	Primitive_constraints_Constraint	R2R	PrimitiveType_constraints_Constraint
45	Enumeration	C2C	Enumeration
46	Enumeration.name	A2A	Enumeration.name
47	Enumeration_comments_Comment	R2R	Enumeration_comments_Comment
48	Enumeration_constraints_Constraint	R2R	Enumeration_constraints_Constraint
49	EnumerationLiteral.value	A2A	EnumerationLiteral.name
50	EnumerationLiteral	C2C	EnumerationLiteral
51	Enumeration_literal_EnumerationLiteral	R2R	Enumeration_ownedLiteral_EnumerationLiteral
52	Constraint	C2C	Constraint
53	Constraint.body	A2A	Constraint.specification
54	Parameter	C2C	Parameter
55	Parameter.name	A2A	Parameter.name
56	Parameter.kind	A2A	Parameter.kind
57	Parameter.defaultValue	A2A	Parameter.defaultValue
58	Parameter_comments_Comment	R2R	Parameter_comments_Comment
59	Parameter_constraints_Constraint	R2R	Parameter_constraints_Constraint
60	Parameter_enumType_Enumeration	R2R	Parameter_enumType_Enumeration
61	Parameter_primitiveType_Primitive	R2R	Parameter_primitiveType_Primitive

Tabelle 6.2: Überlappende Teile von UML 1.4 und UML 2.0 gemäß [Kar08]

6.2 Erstellung der Mappings und Untersuchung des erzeugten Outputs

Aufgrund der Größe des Testmodells aus Abschnitt 6.1 und der damit verbundenen Komplexität der Testsituation werden in diesem Abschnitt zunächst zwei vereinfachte Darstellungen des UML-Klassendiagramms – *UML-light* und *UML-medium* – untersucht, bevor das Gesamtmodells *UML-heavy* zur Evaluierung herangezogen wird. Dabei wird jeweils der Output anhand der erzeugten XSLT- und Nested-Xquery-(2Phased)-Scripts betrachtet.

6.2.1 Testmodell 1: *UML-light*

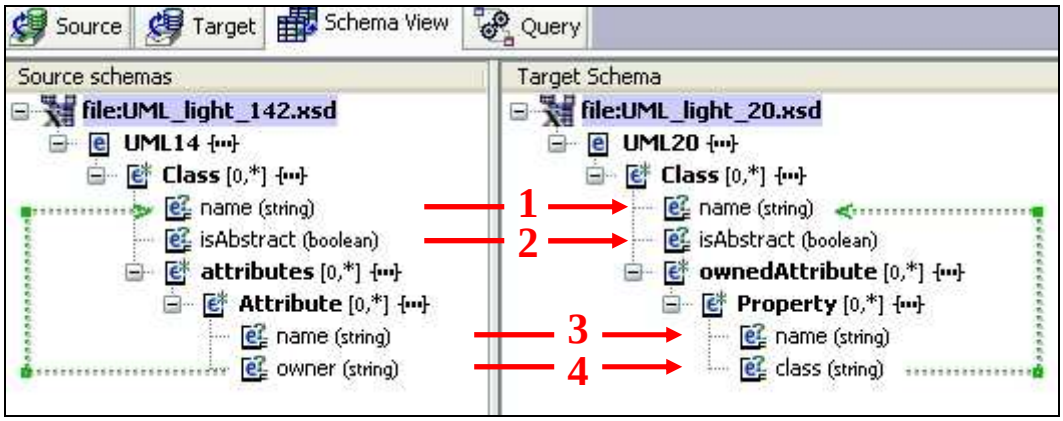

Abbildung 6.3: UML-light in Clios *Schema View*

Abbildung 6.3 zeigt die beiden UML-light-Schemata in Clios *Schema View*. UML-light besteht lediglich aus den Klassen *Class* und *Attribute* bzw. *Property* mit ihren jeweiligen Attributen. In der Folge sollen die vier Wertkorrespondenzen aus Tabelle 6.3 erstellt und soll untersucht werden, wie Clio die UML-1.4-light-Beispielinstanz aus Listing 6.3 auf das UML-2.0-light-Schema überträgt.

Nr.	UML 1.4 light	Typ	UML 2.0 light
1	Class.name	A2A	Class.name
2	Class.isAbstract	A2A	Class.isAbstract
3	Attribute.name	A2A	Property.name
4	Attribute.owner	R2R	Property.class

Tabelle 6.3: Semantische Übereinstimmungen der beiden UML-light-Schemata

Die UML-1.4-light-Beispielinstanz enthält die Klasse *Person* mit den beiden Attributen *svnr* und *name*.

```
<?xml version="1.0" encoding="UTF-8"?>
<UML14 xsi:noNamespaceSchemaLocation="UML_light_142.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Class>
        <name>Person</name>
        <attributes>
            <Attribute>
                <name>svnr</name>
            </Attribute>
            <Attribute>
                <name>name</name>
            </Attribute>
        </attributes>
    </Class>
</UML14>
```

Listing 6.3: Instanz des UML 1.4-light-Klassendiagramms

6.2.1.1 Erste Stufe: Wertkorrespondenz 1

Zunächst wird nur die erste Wertkorrespondenz Class.name aus UML-1.4-light auf Class.name im UML-2.0-light-Modell erzeugt. Während das XSLT-Script das erwartete Ergebnis liefert, generiert die Nested-XQuery-(2Phased)-Abfrage bereits ein leeres *<isAbstract-Tag>*, das nicht Teil des Mappings sein sollte.

XSLT

```
<?xml version="1.0" encoding="UTF-8"?>
<UML20>
    <Class>
        <name>Person</name>
    </Class>
</UML20>
```

Nested-XQuery-(2Phased)

```
<?xml version="1.0" encoding="UTF-8"?>
<UML20>
    <Class>
        <name>Person</name>
        <isAbstract/>
    </Class>
</UML20>
```

6.2.1.2 Zweite Stufe: Wertkorrespondenzen 1 und 2

Nachdem in der zweiten Teststufe zusätzlich die Wertkorrespondenz für das Attribut *isAbstract* erzeugt wird, liefert Clio für beide getesteten Scripts ein brauchbares Ergebnis. Fraglich ist, ob Clio im Falle eines Nichtvorhandensein eines Wertes für *isAbstract* dieses XML-Tag überhaupt erzeugen soll. Meiner Meinung nach ist diese Vorgangsweise falsch, da das entsprechende Element im Zielschema als optional definiert wurde und somit ignoriert werden kann.

XSLT

```
<?xml version="1.0" encoding="UTF-8"?>
<UML20>
    <Class>
        <name>Person</name>
        <isAbstract/>
    </Class>
</UML20>
```

Nested-XQuery-(2Phased)

```
<?xml version="1.0" encoding="UTF-8"?>
<UML20>
    <Class>
        <name>Person</name>
        <isAbstract/>
    </Class>
</UML20>
```

6.2.1.3 Dritte Stufe: Wertkorrespondenzen 1, 2, 3 und 4

Nach Erzeugung der Wertkorrespondenzen 3 und 4, die die Eigenschaften von *Attribute* auf *Property* abbilden, sind die Ergebnisse unbrauchbar. Das XSLT-Script liefert das leere Wurzelelement, das Nested-XQuery-Script das selbe Schema wie die Teststufen 1 und 2.

XSLT

```
<?xml version="1.0" encoding="UTF-8"?>
<UML20/>
```

Nested-XQuery-(2Phased)

```
<?xml version="1.0" encoding="UTF-8"?>
<UML20>
    <Class>
        <name>Person</name>
        <isAbstract/>
    </Class>
</UML20>
```

6.2.1.4 Vierte Stufe: Wertkorrespondenzen 3 und 4

Nachdem Stufe drei Probleme mit der Erzeugung der Attribute-/Property-Korrespondenzen hatte, wird in der vierten Stufe auf die Wertkorrespondenzen *Class.name* und *Class.isAbstract* verzichtet und nur die Abbildung *Attribute.name* aus dem UML-1.4-light-Modell auf *Property.name* im UML-2.0-light-Schema modelliert. Die XSLT-Query generiert ein Schema, das zwar ein Tag <*ownedAttribute*> enthält, dieses aber leer lässt. Die eigentlichen Tags für die Properties werden nicht

erzeugt. Noch schlechter schneidet das Nested-XQuery-Script ab, das nur ein leeres Wurzelelement erzeugt.

XSLT

```
<?xml version="1.0" encoding="UTF-8"?>
<UML20>
    <Class>
        <ownedAttribute/>
    </Class>
</UML20>
```

Nested-XQuery-(2Phased)

```
<?xml version="1.0" encoding="UTF-8"?>
<UML20/>
```

6.2.1.5 Fünfte Teststufe: Wertkorrespondenzen 1 und 3 ohne Assoziationen im Zielschema

Aufgrund der schlechten Ergebnisse, die in den bisherigen Teststufen erzielt wurden, entsteht der Verdacht, dass Clio ein Problem mit Assoziationen hat, falls deren Abbildung nicht dem relationalen Modell entspricht. Sowohl das UML-1.4-light- als auch das UML-2.0-light-Schema können über *Attribute.owner* bzw. *Property.class* zu ihrem Elternknoten navigieren – eine Modellierung, die aus dem objektorientierten Paradigma hervorgeht.

Testweise wird für Stufe fünf die Fremdschlüsselbeziehung im Zielschema entfernt, was keine Verbesserung von Clios Output bewirkt. Wieder entstehen leere bzw. unvollständige UML-light-2.0-Instanzen.

XSLT

```
<?xml version="1.0" encoding="UTF-8"?>
<UML20/>
```

Nested-XQuery-(2Phased)

```
<?xml version="1.0" encoding="UTF-8"?>
<UML20>
    <Class>
        <name>Person</name>
        <isAbstract/>
    </Class>
</UML20>
```

6.2.1.6 Sechste Teststufe: Wertkorrespondenzen 1 und 3 ohne Assoziationen in Quell- und Zielschema

In der sechsten Teststufe wird auch die Assoziation im Quellschema entfernt, wodurch die besten Ergebnisse erzielt werden. Das XSLT-Script generiert ein UML-2.0-Schema, das der Beispielinstanz entspricht. Nested-XQuery-(2Phased) erzeugt wieder leere Tags für *Class.ownedAttribute* bzw. *Property.class*, obwohl die entsprechenden Wertkorrespondenzen nicht existieren.

XSLT

```
<?xml version="1.0" encoding="UTF-8"?>
<UML20 xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <Class>
        <name>Person</name>
        <ownedAttribute>
            <Property>
                <name>svnr</name>
            </Property>
            <Property>
                <name>name</name>
            </Property>
        </ownedAttribute>
    </Class>
</UML20>
```

Nested-XQuery-(2Phased)

```
<?xml version="1.0" encoding="UTF-8"?>
<UML20>
    <Class>
        <name>Person</name>
        <isAbstract/>
        <ownedAttribute>
            <Property>
                <name>svnr</name>
                <class/>
            </Property>
        </ownedAttribute>
        <ownedAttribute>
            <Property>
                <name>name</name>
                <class/>
            </Property>
        </ownedAttribute>
    </Class>
</UML20>
```

6.2.1.7 Fazit der Tests mit UML-light

Die besten Testergebnisse mit dem UML-light-Modell werden erzielt, wenn Assoziationen aus Quell- und Zielschema entfernt werden. Der Grund dafür dürfte sein, dass Clio auf Schemata ausgelegt ist, die dem relationalen Datenmodell entsprechen. Eine

Modellierung wie die Assoziation *Attribute.owner* auf *Class* aus dem UML-1.4-light-Modell, die durch die Komposition *Class.attributes* auf *Attribute* in die entgegengesetzte Richtung abgebildet ist, entspricht dem relationalen Modell nicht, wodurch Clio sie nicht verarbeiten kann.

Daher stellt sich die Frage, ob bidirektionale Beziehungen überhaupt modelliert werden sollen. Generell sollte die Überlegung angestellt werden, auf die Abbildung von Assoziationen zu verzichten, da die zu übersetzenden Metamodellinstanzen auch ohne diese gemapt werden können. In den folgenden Evaluierungen wird auf Assoziationsbeziehungen verzichtet, falls dadurch die Mappingergebnisse verbessert werden können.

6.2.2 Testmodell 2: *UML-medium*

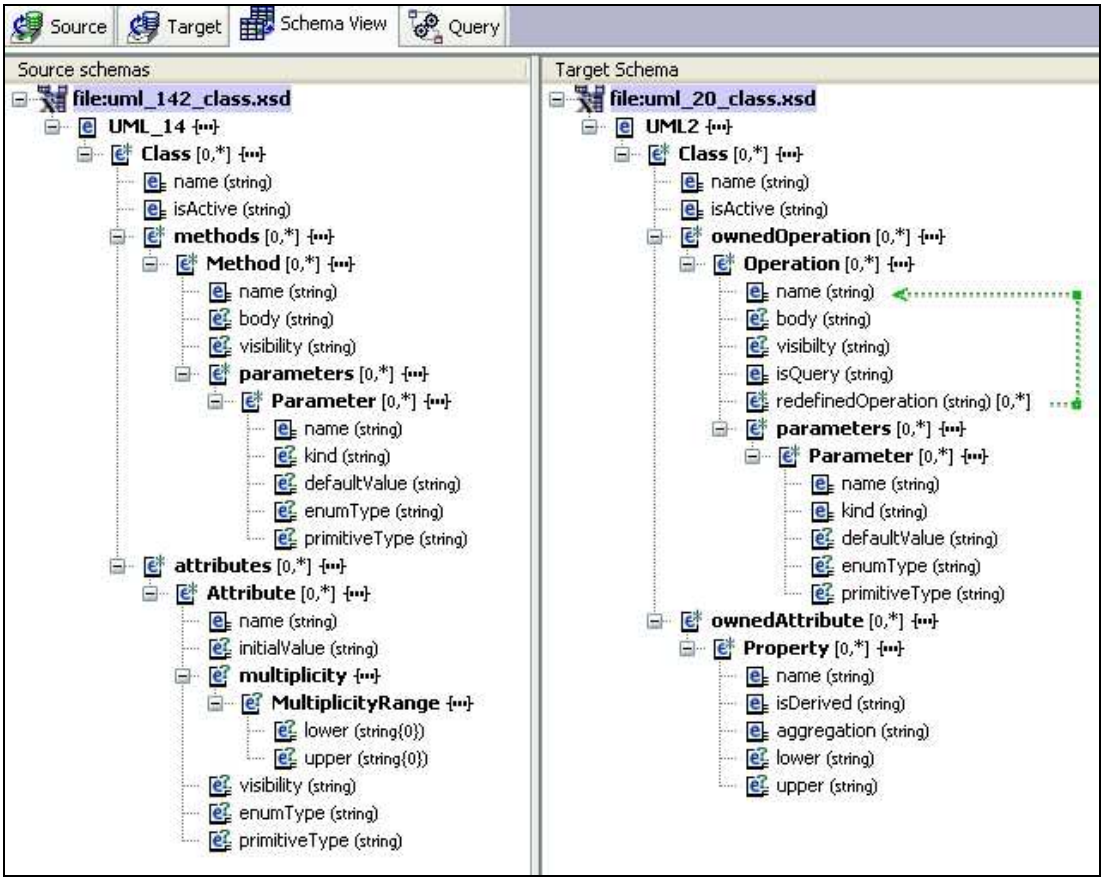

Abbildung 6.4: Die UML-medium-Schemata in Clios *Schema View*

Das UML-1.4- und das UML-2.0-medium-Schema sind auf das *Class*-Objekt aus dem Testmodell aus Abschnitt 6.1 beschränkt und werden in Abbildung 6.4 dargestellt. Die Klasse *Class* enthält jeweils eine oder mehrere Klassen *Method* bzw. *Operation* sowie eine oder mehrere Klassen *Attribute* bzw. *Property*.

Listing 6.4 zeigt die **UML-1.4-medium-Instanz**, die auf das UML-2.0-Schema übertragen werden soll. Sie enthält

- eine Klasse mit dem Namen *Person*,
- eine Methode *getSvnr* mit dem Parameter *name* und
- zwei Attribute *name* und *svnr* mit ihrer jeweiligen Multiplizität.

```
<?xml version="1.0" encoding="UTF-8"?>
<UML_14 xsi:noNamespaceSchemaLocation="uml_142_class.xsd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <Class>
       <name>Person</name>
       <isActive>true</isActive>
       <methods>
           <Method>
               <name>getSvnr</name>
               <body>return Person.svnr</body>
               <visibility>public</visibility>
               <parameters>
                   <Parameter>
                       <name>name</name>
                       <kind>String</kind>
                   </Parameter>
               </parameters>
           </Method>
       </methods>
       <attributes>
           <Attribute>
               <name>name</name>
               <multiplicity>
                   <MultiplicityRange>
                       <lower>1</lower>
                       <upper>1</upper>
                   </MultiplicityRange>
               </multiplicity>
           </Attribute>
           <Attribute>
               <name>svnr</name>
               <multiplicity>
                   <MultiplicityRange>
                       <lower>0</lower>
                       <upper>1</upper>
                   </MultiplicityRange>
               </multiplicity>
           </Attribute>
       </attributes>
   </Class>
</UML_14>
```

Listing 6.4: Beispielinstanz für das UML-medium-Testset

Nach der Erstellung der Wertkorrespondenzen und der Ausführung der von Clio erzeugten Queries soll eine **UML-2.0-medium-Instanz** erzeugt werden, die folgende Elemente enthält:

- eine Klasse mit dem Namen *Person*
- eine Operation *getSvnr* und dem Parameter *name* inklusive ihrer jeweiligen Attribute
- zwei Properties *name* und *svnr* mit ihren jeweiligen Attributen und ihrer Multiplizität

Besonderes Augenmerk liegt auf der Übertragung der Kardinalität von *Attribute* auf *Property*. Während im UML-1.4-Modell die Multiplizität über die Kompositionsbeziehung *multiplicity* auf die Attribute *MultiplicityRange.lower* und *MultiplicityRange.upper* realisiert ist, wird diese im UML-2.0-Metamodell direkt in der Klasse *Property* als deren Attribut *lower* bzw. *upper* modelliert.

Die Wertkorrespondenzen *2, 3, 8, 9, 10, 15, 16, 19, 55, 56, 57, 60* und *61* aus Tabelle 6.2 lassen sich in Clio problemlos erstellen. Dabei werden sowohl das XSLT- als auch das Nested-XQuery-(2Phased)-Script erzeugt. Die beiden UML-2.0-medium-Instanzen nach Anwendung dieser Scripts werden in Listing 6.5 und Listing 6.6 dargestellt.

XSLT

```
<?xml version="1.0" encoding="UTF-8"?>
<UML20>
    <Class>
        <name>Person</name>
        <isActive>true</isActive>
        <ownedOperation>
            ...
        </ownedOperation>
    </Class>
    <Class>
        <name>Person</name>
        <isActive>true</isActive>
        <ownedAttribute>
            ...
        </ownedAttribute>
    </Class>
</UML20>
```

Listing 6.5: Ausschnitt aus der UML 2.0-medium-Instanz nach Ausführung des XSLT-Scripts

Der Output des XSLT-Scripts ist unbrauchbar. Die Klasse *Person* wurde zwei Mal erstellt, wobei sie einmal nur aus der Operation *getSvnr* besteht und ein anderes Mal lediglich die beiden Properties *name* und *svnr* enthält. Dieses Ergebnis geht nicht mit der UML-1.4-medium-Instanz konform und ist falsch.

Nested-XQuery-(2Phased)

```
<?xml version="1.0" encoding="UTF-8"?>
<UML20>
    <Class>
        <name>Person</name>
        <isActive>true</isActive>
        <ownedOperation>
            <Operation>
                <name>getSvnr</name>
                <body>return Person.svnr</body>
                <visibilty>public</visibilty>
                <isQuery/>
                <parameters>
                    <Parameter>
                        <name>name</name>
                        <kind>String</kind>
                        <defaultValue/>
                        <enumType/>
                        <primitiveType/>
                    </Parameter>
                </parameters>
            </Operation>
        </ownedOperation>
        <ownedAttribute>
            <Property>
                <name>name</name>
                <isDerived/>
                <aggregation/>
                <lower>1</lower>
                <upper>1</upper>
            </Property>
        </ownedAttribute>
        <ownedAttribute>
            <Property>
                <name>svnr</name>
                <isDerived/>
                <aggregation/>
                <lower>0</lower>
                <upper>1</upper>
            </Property>
        </ownedAttribute>
    </Class>
</UML20>
```

Listing 6.6: UML 2.0-medium-Instanz nach Ausführung des Nested-XQuery-Scripts

Nach Ausführung des Nested-XQuery-Scripts entsteht die in Listing 6.6 dargestellte UML-2.0-medium-Instanz. Wie bereits in Abschnitt 6.2.1 wird ein leeres Tag erstellt, falls für ein Element kein Quellwert vorhanden ist. Ansonsten werden die Werte aus dem UML-1.4-Schema korrekt in die Zielinstanz übertragen. Sie enthält eine Klasse *Person* mit der Methode *getSvnr* und den beiden Properties *name* und *svnr*. Die Multiplizität der beiden Properties wurde trotz der strukturellen Differenzen der beiden Schemata korrekt übertragen.

6.2.3 Testmodell 3: *UML-full*

Schlussendlich wird das gesamte UML-Testset aus Abschnitt 6.1 zur Evaluierung herangezogen. Die zu transformierende Beispielinstanz ist eine Erweiterung der UML 1.4-medium-Instanz und wird hier nicht extra abgebildet.

Bereits die Erzeugung der Wertkorrespondenzen erweist sich als problematisch. Die Korrespondenzen *21* bzw. *22* aus Tabelle 6.2, die jeweils eine Fremdschlüsselbeziehung abbilden, rufen eine Fehlermeldung hervor, die bewirkt, dass keine weiteren Wertzuweisungen erstellt werden können. Nach Entfernung der Assoziationen im Zielschema lassen sich zwar sämtliche Wertkorrespondenzen abbilden, das resultierende Nested-XQuery-Script lässt sich allerdings nicht ausführen.

Nachdem die Fremdschlüsselbeziehungen auch im Quellschema entfernt werden, kann die Datentransformation mit Hilfe des Nested-XQuery-Scripts durchgeführt werden. Das Ergebnis entspricht jenem, das mit den UML-medium-Schemata erzielt wurde, und muss nicht weiter interpretiert werden.

6.3 Fazit

Obwohl bei den durchgeführten Tests teils brauchbare Transformationsergebnisse erzielt werden konnten, scheint eine Nutzung von Clio als Mapping-Tool für Metamodelle nicht sehr sinnvoll zu sein. Der Hauptgrund dafür ist nicht Clios Funktionsweise sondern die Tatsache, dass es für Schemata entwickelt wurde, die dem relationalen Datenmodell entsprechen, Ecore-Metamodelle jedoch der objektorientierten Modellierung zugewiesen werden. Bei der Umformung objektorientierter Modelle in relationale Schemata gehen wesentliche Schemainformationen verloren. Auch wenn es möglich ist, Mappings zwischen Metamodellen zu erstellen und deren Instanzen zumindest teilweise zu transformieren, ist es nicht möglich, die mit EcoreToXsd erzeugten XML-Schemata und deren Instanzen zurück in Ecore-Modelle zu transferieren. Aufgrund folgender Fakten scheint der Einsatz Clios in Anwendung auf Ecore-Metamodelle nicht sinnvoll zu sein:

- Eingeschränkte Modellierungsmöglichkeiten: rekursive Komposition und Vererbung können zwar in XML Schema dargestellt, nicht aber von Clio verarbeitet werden.

- Modellierung von Assoziationen: Assoziationen über Fremdschlüsselbeziehungen sind im relationalen Modell ein wesentliches Modellierungselement.

Die Umformung von Assoziationen der Ecore-Metamodelle wie in Kapitel 5 beschrieben scheint für die Anwendung von Clio auf Metamodelle nicht sinnvoll zu sein. Clios Transformationsergebnisse sind wesentlich besser, wenn auf die Darstellung von Assoziationen verzichtet wird.

- Eingeschränkte Zuweisung von Wertkorrespondenzen: Nur Wertkorrespondenzen für Attribute können erstellt werden. Klassen können nicht aufeinander abgebildet werden.

- Clios Engine zur Erzeugung der Transformations-Queries ist sehr unzuverlässig.

Generell lässt sich sagen, dass Clio nur gute Ergebnisse erzielen kann, wenn es relationale Schemata als Inputdaten bekommt. [Lai97] beschreibt eine Methode, wie objektorientierte Daten auf das relationale Modell übertragen werden können. Eine zukünftige Arbeit könnte sich damit beschäftigen, Metamodelle mit dieser Methodik umzuformen und zu evaluieren, wie sich Clio in Anwendung auf die gewonnenen Schemata verhält.

Kapitel 7

Zusammenfassung

In der vorliegenden Arbeit wurde die Transformation von Ecore-Metamodellen mit Clio untersucht. Als Vorbereitung zur eigentlichen Evaluierung beschäftigte sich Kapitel 3 mit Clios Bedienung. Aufgrund der teilweise veralteten Dokumentation und der nicht vorhandenen Bedienungsanleitung stellte sich diese Analyse als schwieriger dar als ursprünglich angenommen. Clios schwer durchschaubares Verhalten sorgte für zusätzliche Hürden. Im Mittelpunkt dieser Untersuchungen stand die Frage, in welcher Form die Inputdaten vorliegen müssen, damit sie von Clio bearbeitet werden können. Dabei wurde herausgefunden, dass diese als XML Schemata vorliegen müssen, die dem relationalen Datenmodell entsprechen.

Entsprechende XML Schemata wurden in Kapitel 4 verwendet, um Clio in Mapping-Situationen zu testen, wie sie in [Leg05] klassifiziert sind. Das Ergebnis ist sowohl ein Vergleich als auch die entsprechende Dokumentation der Abweichungen zwischen den Testergebnissen dieser Arbeit und jenen aus [Leg05].

Kapitel 5 widmete sich der Transformation von Ecore-Metamodellen in von Clio lesbare XML Schemata. Dabei sollte eruiert werden, wie einzelne Ecore-Elemente als XML-Schema-Elemente ausgedrückt werden können. Außerdem sollte ein Algorithmus entwickelt werden, der Metamodelle automatisch in XML Schemata transformiert. Dieses Ziel konnte nicht vollständig erreicht werden: Der Algorithmus *EcoreToXSD* benötigt durch die Umformung der Metamodelle in eine relationale Darstellung und die entsprechende Verflachung der Daten in diversen Situationen eine Interaktion mit der/dem BenutzerIn. Ein wichtiger Teil von Kapitel 5 war die Untersuchung der Datenintegrität bei der Anwendung von EcoreToXsd und die Identifizie-

rung jener Ecore-Elemente, die nicht im relationalen Modell ausgedrückt werden können. Dabei wurde festgestellt, dass Kompromisse eingegangen werden müssen, um die Datenintegrität zu bewahren – unter anderem bei der Umformung von Assoziationen und bei der Modellierung von Vererbung.

Kapitel 6 beschäftigte sich schließlich mit der Evaluierung Clios in Anwendung auf Ecore-Metamodelle. Dazu wurde ein Testset, bestehend aus dem UML-1.4-Metamodell und dem UML-2.0-Metamodell sowie deren Überlappungen, erstellt. Bei der Umformung der Testmodelle mittels EcoreToXsd wurden weitere Konfliktsituationen erkannt, in denen die Datenintegrität gefährdet ist. Zur besseren Nachvollziehbarkeit von Clios Verhalten wurde das UML-Testset in drei Versionen evaluiert, wobei die erzeugten Mappings mit Hilfe einer Beispielinstanz geprüft und eine entsprechende Analyse des Outputs durchgeführt wurde.

Zusammenfassend lässt sich sagen, dass die Erstellung von Schema Mappings und die anschließende Transformation von Beispielinstanzen der Metamodelle in XML-Schema-Instanzen mit Clio teilweise brauchbare Ergebnisse bringt. Die Kompromisse, die bei der Transformation von Ecore-Metamodellen in von Clio lesbare, relationale XML Schemata eingegangen werden müssen, führen jedoch zu Datenverlusten, die meiner Meinung nach nicht akzeptabel sind.

Anhang

Testset aus Kapitel 5 als serialisiertes Ecore-Metamodell

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="Universitaet">
    <eClassifiers xsi:type="ecore:EClass" name="Person" abstract="true">
       <eStructuralFeatures xsi:type="ecore:EAttribute" name="Name" lowerBound="1"
         eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
       <eStructuralFeatures xsi:type="ecore:EAttribute" name="Geburtsdatum"
         unique="false" eType="ecore:EDataType
         http://www.eclipse.org/emf/2002/Ecore#//EDate"/>
       <eStructuralFeatures xsi:type="ecore:EAttribute" name="Geschlecht"
         eType="#//Geschlechter"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="ProfessorIn" eSuperTypes="#//Person">
       <eStructuralFeatures xsi:type="ecore:EAttribute" name="Gehalt"
         eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EDouble"/>
       <eStructuralFeatures xsi:type="ecore:EReference" name="haelt" upperBound="-1"
         eType="#//Lehrveranstaltung"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="StudentIn" eSuperTypes="#//Person">
       <eStructuralFeatures xsi:type="ecore:EAttribute" name="Matrikelnummer"
         eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
       <eStructuralFeatures xsi:type="ecore:EReference" name="besucht" upperBound="-1"
         eType="#//Lehrveranstaltung"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="Lehrveranstaltung">
       <eStructuralFeatures xsi:type="ecore:EAttribute" name="Bezeichnung"
         eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
       <eStructuralFeatures xsi:type="ecore:EAttribute" name="Nummer" lowerBound="1"
         eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
       <eStructuralFeatures xsi:type="ecore:EReference" name="besuchtVon"
         lowerBound="1" upperBound="-1" eType="#//StudentIn"/>
       <eStructuralFeatures xsi:type="ecore:EReference" name="gehaltenVon"
         lowerBound="1" eType="#//ProfessorIn"/>
       <eStructuralFeatures xsi:type="ecore:EReference" name="hatScript" upperBound="-
         1" eType="#//Script" containment="true"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="Script">
       <eStructuralFeatures xsi:type="ecore:EAttribute" name="Titel" lowerBound="1"
         eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
       <eStructuralFeatures xsi:type="ecore:EReference" name="gehoertZu"
         lowerBound="1" eType="#//Lehrveranstaltung"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EEnum" name="Geschlechter">
       <eLiterals name="maennlich"/>
       <eLiterals name="weiblich" value="1"/>
    </eClassifiers>
</ecore:EPackage>
```

Testmodell aus Kapitel 5 als XML Schema nach Anwendung von *EcoreToXsd*

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
  attributeFormDefault="unqualified">
   <xs:element name="Universitaet">
         <xs:complexType>
            <xs:sequence>
               <xs:element name="ProfessorIn" minOccurs="0" maxOccurs="unbounded">
                  <xs:complexType>
                     <xs:complexContent>
                        <xs:extension base="ProfessorIn"/>
                     </xs:complexContent>
                  </xs:complexType>
               </xs:element>
               <xs:element name="StudentIn" minOccurs="0" maxOccurs="unbounded">
                  <xs:complexType>
                     <xs:complexContent>
                        <xs:extension base="StudentIn"/>
                     </xs:complexContent>
                  </xs:complexType>
               </xs:element>
               <xs:element name="Lehrveranstaltung" minOccurs="0"
                 maxOccurs="unbounded">
                  <xs:complexType>
                     <xs:complexContent>
                        <xs:extension base="Lehrveranstaltung"/>
                     </xs:complexContent>
                  </xs:complexType>
               </xs:element>
            </xs:sequence>
         </xs:complexType>
         <xs:key name="ProfessorIn_Key">
            <xs:selector xpath="./ProfessorIn"/>
            <xs:field xpath="Name"/>
         </xs:key>
         <xs:key name="StudentIn_Key">
            <xs:selector xpath="./StudentIn"/>
            <xs:field xpath="Name"/>
         </xs:key>
         <xs:key name="Lehrveranstaltung_Key">
            <xs:selector xpath="./Lehrveranstaltung"/>
            <xs:field xpath="Nummer"/>
         </xs:key>
         <xs:keyref name="ProfessorIn_haelt" refer="Lehrveranstaltung_Key">
            <xs:selector xpath="./ProfessorIn"/>
            <xs:field xpath="haelt"/>
         </xs:keyref>
         <xs:keyref name="StudentIn_besucht" refer="Lehrveranstaltung_Key">
            <xs:selector xpath="./StudentIn"/>
            <xs:field xpath="besucht"/>
         </xs:keyref>
         <xs:keyref name="Lehrveranstaltung_besuchtVon" refer="StudentIn_Key">
            <xs:selector xpath="./Lehrveranstaltung"/>
            <xs:field xpath="besuchtVon"/>
         </xs:keyref>
         <xs:keyref name="Lehrveranstaltung_gehaltenVon" refer="ProfessorIn_Key">
            <xs:selector xpath="./Lehrveranstaltung"/>
            <xs:field xpath="gehaltenVon"/>
         </xs:keyref>
         <xs:keyref name="Script_gehoertZu" refer="Lehrveranstaltung_Key">
            <xs:selector xpath="./Lehrveranstaltung/hatScript/Script"/>
            <xs:field xpath="gehoertZu"/>
         </xs:keyref>
   </xs:element>
   <xs:complexType name="Person" abstract="true">
      <xs:sequence>
         <xs:element name="Name" type="xs:string"/>
         <xs:element name="Geburtsdatum" type="xs:date" minOccurs="0"/>
         <xs:element name="Geschlecht" type="Geschlechter" minOccurs="0"/>
      </xs:sequence>
   </xs:complexType>
```

```
   <xs:complexType name="ProfessorIn">
      <xs:complexContent>
         <xs:extension base="Person">
            <xs:sequence>
               <xs:element name="Gehalt" type="xs:double" minOccurs="0"/>
               <xs:element name="haelt" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
   <xs:complexType name="StudentIn">
      <xs:complexContent>
         <xs:extension base="Person">
            <xs:sequence>
               <xs:element name="Matrikelnummer" type="xs:integer" minOccurs="0"/>
               <xs:element name="besucht" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
   <xs:complexType name="Lehrveranstaltung">
      <xs:sequence>
         <xs:element name="Nummer" type="xs:string"/>
         <xs:element name="Bezeichnung" type="xs:string" minOccurs="0"/>
         <xs:element name="besuchtVon" maxOccurs="unbounded"/>
         <xs:element name="gehaltenVon"/>
         <xs:element name="hatScript" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="Script">
                     <xs:complexType>
                        <xs:complexContent>
                           <xs:extension base="Script"/>
                        </xs:complexContent>
                     </xs:complexType>
                 </xs:element>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="Script">
      <xs:sequence>
         <xs:element name="Titel" type="xs:string"/>
         <xs:element name="gehoertZu"/>
      </xs:sequence>
   </xs:complexType>
   <xs:simpleType name="Geschlechter" final="restriction">
      <xs:restriction base="xs:string">
         <xs:enumeration value="maennlich"/>
         <xs:enumeration value="weiblich"/>
      </xs:restriction>
   </xs:simpleType>
</xs:schema>
```

Literaturverzeichnis

[BM01] Paul V. Biron, Ashok Malhotra, *XML Schema Part 2: Datatypes Second Edition*, W3C Recommendation 28 October 2004, http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

[BPS+06] Tim Bray, Jean Paoli, C. M: Sperberg-McQueen, Eve Maler, François Yergeau, *Extensible Markup Language (XML) 1.0 (Fourth Edition)*, W3C Recommendation 16 August 2006, http://www.w3.org/TR/2006/REC-xml-20060816/

[CD99] James Clark, Steven DeRose, *XML Path Language (XPath) Version 1.0*, W3C Recommendation 16 November 1999, http://www.w3.org/TR/1999/REC-xpath-19991116

[Cla99] James Clark, *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation 16 November 1999, http://www.w3.org/TR/1999/REC-xslt-19991116

[FW04] David C. Fallside, Priscilla Walmsley , *XML Schema Part 0: Primer Second Edition*, W3C Recommendation 28 October 2004, http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/

[HHH+05] Laura M. Haas, Mauricio A. Hernández, Howard Ho, Lucian Popa, Mary Roth, *Clio Grows Up: From Research Prototype to Industrial Tool*, In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), Seiten 805-810, 2005

[Huc00] Michael Hucka, SCHUCHS: *A UML-Based Approach for Describing Data Representations Intended for XML Encoding*, 2000

[Kar08] Horst Kargl, *Smart Matching – An Approach for the Automatic Generation of Executable Schema Mappings*, Ph.D. Thesis, Vienna University of Technology, 2008

[Kay07] Michael Kay, *XSL Transformations (XSLT) Version 2.0*, W3C Recommendation 23 January 2007, http://www.w3.org/TR/2007/REC-xslt20-20070123/

[KST02] Wassilios Kazakos, Andreas Schmidt, Peter Tomczyk, *Datenbanken und XML*, Springer, 2002

[Lai97] Harri Laine, *Transformation of Object Model to the Relational schema*, 1997, http://www.cs.helsinki.fi/u/laine/info/kevat97/trans.html

[Leg05] Frank Legler, *Datentransformation mittels Schema Mapping*, Diplomarbeit an der Humboldt-Universität zu Berlin, 2005

[LN07] Frank Legler, Felix Naumann, *A Classification of Schema Mappings and Analysis of Mapping Tools* (Extended Version), 2007

[MHH00] Renée J. Miller, Laura M. Haas, Mauricio A. Hernández, *Schema Mapping as Query Discovery*, In: Proceedings of the International Conference on Very Large Databases (VLDB), Seiten 77-88, 2000

[MHH+01] Renée J. Miller, Mauricio A. Hernández, Laura M. Haas, Lingling Yan, C. T. Howard Ho, Ronald Fagin, Lucian Popa, *The Clio Project: Managing Hetereogenity*, In ACM SIGMOD Record 30, Seiten 78-83, 2001

[Mil07] Renée J. Miller, *Retrospective on Clio: Schema Mapping and Data Exchange in Practice*, 2007, http://ceur-ws.org/Vol-250/invited_1.pdf

[MOF04] Object Managment Group (OMG), *Meta Object Facility (MOF) 2.0 Core Specification Version 2.0*, 2004, http://www.omg.org/docs/ptc/04-10-15.pdf

[NL07] Felix Naumann, Ulf Leser, *Informationsintegration – Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen*, dpunkt.verlag Heidelberg, 2007

[PHV+02] Lucian Popa, Mauricio A. Hernàndez, Yannis Velegrakis, Renée Miller, Felix Naumann, Howard Ho, *Mapping XML and Relational Schemas with Clio*, In: Proceedings of the 18th International Conference on Data Engineering, 2002, IEEE Computer Society, Washington, DC, USA

[Pro02] Will Provost, *UML For W3C XML Schema Design*, 2002, http://www.xml.com/pub/a/2002/08/07/wxs_uml.html

[RSB+07] Jonathan Robie, Jérôme Siméon, Scott Boag, Daniela Florescu, Mary F. Fernández, Don Chamberlin, *XQuery 1.0: An XML Query Language*, W3C Recommendation 23 January 2007, http://www.w3.org/TR/2007/REC-xquery-20070123/

[RSC+07] Jonathan Robie, Jérôme Siméon, Don Chamberlin, Mary F. Fernández, Michael Kay, Anders Berglund, Scott Boag, *XML Path Language (XPath) 2.0*, W3C Recommendation 23 January 2007, http://www.w3.org/TR/2007/REC-xpath20-20070123/

[TMB+04] Henry S. Thompson, Noah Mendelsohn, David Beech, Murray Maloney, *XML Schema Part 1: Structures Second Edition*, W3C Recommendation 28 October 2004, http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/

[WKS+07] Manuel Wimmer, Horst Kargl, Martina Seidl, Michael Strommer, Thomas Reiter, *Integrating Ontologies with CAR Mappings*, In: The First International Workshop on Semantic Technology Adoption in Business, dapir academic press, Seiten 27 – 38, 2007

[WSK+06] Manuel Wimmer, Andrea Schauerhuber, Elisabeth Kapsammer, Gerhard Kramler, *From Document Type Definitions to Metamodels: The WebML Case Study*, 2006