

**Integration von Ontologien und
modellgetriebener
Softwareentwicklung:
Konzeption und Entwurf eines Eclipse
Ontologie Frameworks (EOF)**

Diplomarbeit

zur Erlangung des Grades eines
Diplom-Informatikers
im Studiengang Informatik

vorgelegt von

Carsten Schneider

Gutachter: Prof. Dr. Steffen Staab, Institut „WeST - Web Science and
Technologies“, Fachbereich 4: Informatik
MSc. Fernando Silva Parreiras, Institut „WeST - Web Science
and Technologies“, Fachbereich 4: Informatik
Dr. Ansgar Scherp, Institut „WeST - Web Science and
Technologies“, Fachbereich 4: Informatik

Koblenz, im November 2010

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ja Nein

Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.

Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

.....
(Ort, Datum)

.....
(Unterschrift)

Zusammenfassung

Modellgetriebene Softwareentwicklung und Ontologien verbreiten sich zunehmend. Allerdings werden Ontologien zurzeit nur selten in der Softwareentwicklung genutzt, obwohl Ontologien den Softwareentwicklungsprozess verbessern könnten. Insbesondere die modellgetriebene Softwareentwicklung könnte von Ontologien profitieren, indem die Ontologien zur Definition von Restriktionen auf dem Softwaremodell verwendet werden. Ein Grund für die seltene Verwendung von Ontologien in der Softwareentwicklung ist, dass die Verwendung von Ontologien kaum von etablierten Softwareentwicklungswerkzeugen unterstützt wird.

In dieser Arbeit stellen wir einen Ansatz für die Integration von Ontologien und modellgetriebener Softwareentwicklung vor. Dieser Ansatz sieht vor, dass die Funktionalitäten eines modellgetriebenen Systems um Funktionalitäten zur Verwendung von Ontologien erweitert werden. Der Ansatz wurde so konzipiert, dass man für ein nach diesem Ansatz erweitertes System die Werkzeuge, die für das ursprüngliche System entwickelt wurden, weiterhin verwenden kann. Um dies zu erreichen, werden die Annotationen des ursprünglichen Systems für die Erweiterung der Softwaremodellierung verwendet. Mit den Annotationen wird es ermöglicht eine Ontologie für ein Modell zu definieren und Anfragen mit Operationen zu verbinden.

Aus dem annotierten Modell und dessen Modellinstanzen kann eine vollständige Ontologie generiert werden. Der generierte Code verwendet die generierte Ontologie, um mithilfe von Anfragen die Einhaltung einiger Ontologierestriktionen zur Laufzeit zu gewährleisten und um mit Konsistenzüberprüfungen zu prüfen, ob zu einem Zeitpunkt alle Ontologierestriktionen eingehalten sind. Das Verhalten von Operationen der Klassen kann mithilfe von Anfragen auf der generierten Ontologie modelliert werden. Es kann mit einer Anfrage festgelegt werden, welche Werte eine Operation zurückliefert.

Somit erlaubt der Ansatz, dass zusätzlich zu den Funktionalitäten eines modellgetriebenen Systems eine Ontologie für die Definition von Restriktionen verwendet werden kann und mit Anfragen auf dieser Ontologie das Verhalten von Operationen der Klassen modelliert werden kann.

Abstract

Model driven software development and ontologies are spreading increasingly. However ontologies are rarely used in the software development, even though ontologies could improve the software development process. In particular model driven software development could benefit from ontologies by using ontologies for the definition of restrictions on the software model. One reason for the rare use of these technologies in the software development is the lack of integration of ontologies and established software development tools.

In this master thesis we present an approach for the integration of ontologies and model driven software development. The functionalities of a model driven system are extended by functionalities for the use of ontologies. Moreover the tools, which were developed for the original system, can also be used for the extended system. This is achieved by using the annotations of the original system for the extension of the software modeling. The annotations can be used to define an ontology for a model and to bind queries to operations.

A complete ontology can be generated from the annotated model and its model instances. The generated program code uses the generated ontology to ensure the compliance with some ontology restrictions at runtime by applying queries on the ontology. In addition the code uses the generated ontology to verify with consistency checks, if all ontology restrictions are met at a particular time. The behaviour of operations can be modeled with queries on the generated ontology. The queries can be used to define, which results are retrieved by an operation.

Therefore the approach permits to use ontologies to define restrictions and to model the behaviour of operations with queries on this ontology in addition to the use of the functionalities of a model driven system.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Gliederung	4
2	Szenario	5
2.1	Bestellvorgangsszenarios	5
2.2	Weitere Modellierungswünsche für das Bestellvorgangsszenario	8
2.3	Herausforderungen des Bestellvorgangsszenarios	9
3	Grundlagen und Verwandte Arbeiten	11
3.1	Modelle	11
3.1.1	UML-artige Modelle	12
3.1.2	Ontologien	12
3.2	Codegenerierung	14
3.2.1	Modellgetriebene Softwareentwicklung	15
3.2.2	Ontologiegetriebene Softwareentwicklung	16
3.3	Vergleich der Konzepte von Ontologien und objektorientierter Konzepte	19
3.4	Nutzung von Ontologien in Anwendungen	20
4	Funktionale Anforderungen	23
5	Integration von Ontologien und modellgetriebener Softwareentwicklung	27
5.1	Lösungsansatz: Eclipse Ontologie Framework (EOF)	27
5.2	Anwendungsfälle für den Lösungsansatz	32
5.3	Ontologie- und Anfrageannotationen zur Erweiterung der Softwaremodellierung	37
5.3.1	Ontologiedefinitionen	39
5.3.2	Ontologiedefinitionen im Modell	39
5.3.3	Ontologiedefinitionen in den Modellinstanzen	41

VIII Inhaltsverzeichnis

5.3.4	Anfrageannotationen für Modelle	42
5.4	Berechnung des Generator-Eingabemodells	42
5.4.1	Herleitung weiterer Ontologieannotationen und Modellanpassungen	43
5.4.2	Behandlung von <code>ClassExpressions</code> und Klassenaxiomen . .	44
5.5	Ontologiegenerierung	46
5.6	Verwendung der Ontologie im generierten Code	48
5.6.1	Einhaltung von Ontologierestriktionen zur Laufzeit mithilfe von Anfragen	50
5.6.2	Verwendung von Konsistenzüberprüfungen zur Validitätsprüfung	51
5.6.3	Verwendung von Anfragen zur Modellierung des Verhaltens von Operationen	51
6	Implementierung	55
6.1	Gewählte Architektur für das EOF System	55
6.2	Annotierte Ecore Modelle als EOF Modelle	57
6.2.1	Ontologiedefinitionen	58
6.2.2	Ontologiedefinitionen im Ecore Modell	59
6.2.3	Ontologiedefinitionen in den Modellinstanzen	59
6.2.4	Anfrageannotationen	59
6.3	Ontologiegenerierung	60
6.4	Codegenerierung des EOF Systems	62
6.4.1	Organisation der Eclipse Plugins	63
6.4.2	Erweiterung der <code>Java Emitter Templates</code>	63
6.4.3	Erwartungen des erweiterten Generators an seine Eingabemodelle	66
6.5	Verwendung der Ontologie im generierten Code	67
6.5.1	Einhaltung von Ontologierestriktionen zur Laufzeit mithilfe von Anfragen	68
6.5.2	Verwendung von Konsistenzüberprüfungen zur Validitätsprüfung	71
6.5.3	Verwendung von Anfragen zur Modellierung des Verhaltens von Operationen	71
7	Prototyp	73
7.1	EOF Modell erstellen	73
7.2	Codegenerierung	81
7.3	Bearbeitung des generierten Codes	81
7.4	Verwendung des generierten Editors	81
8	Zusammenfassung und Ausblick	85
8.1	Zusammenfassung	85
8.2	Ausblick	86

Inhaltsverzeichnis IX

Literaturverzeichnis 89

Einleitung

1.1 Motivation

In der Softwareentwicklung haben Modelle eine bedeutende Rolle. Die Entwickler entwerfen verschiedene Modelle, die sie in Modellierungssprachen wie z.B. UML [20] oder Ecore [36] erstellen. Diese Modelle werden für die Spezifikation und Dokumentation von Software verwendet. Die Modelle beschreiben die Komponenten der Anwendung, deren Beziehungen zueinander und verschiedene Anforderungen an die Komponenten. Zusammen stellen die Modelle eine Art Architektur für die Software dar.

Somit ist ihre Bedeutung bei der Softwareentwicklung mit der Bedeutung von Bauplänen in der Baubranche vergleichbar. Baupläne spezifizieren aus welchen Teilen ein Gebäude besteht und welche Anforderungen diese Teile erfüllen müssen. Zudem werden die Baupläne zur Dokumentation verwendet, um nach der Erstellung des Gebäudes Informationen über das Gebäude zur Verfügung zu haben. Zum Beispiel kann mit dem Bauplan überprüft werden, ob eine Wand tragend ist und wo die Versorgungsleitungen verlegt wurden.

Bei der modellgetriebenen Softwareentwicklung geht man noch einen Schritt weiter. Die Entwickler verwenden die Modelle zusätzlich als Eingabe für einen Codegenerator. Der Generator implementiert auf Basis des Modells ein Grundgerüst für die Software. Der so generierte Code kann vom Entwickler modifiziert und erweitert werden.

In den letzten Jahren ist die Bedeutung von Ontologien zur Repräsentation von Wissen im *Semantic Web* [10, 1] gewachsen. Ontologien eignen sich gut für die Repräsentation von Wissen und das Formulieren von Einschränkungen, da sie auf Logiken basieren. Mit der Verbreitung der Ontologien kam der Wunsch auf, die Ontologien auch für die Softwareentwicklung zu nutzen. Man begann sich mit ontologiegetriebener Softwareentwicklung zu befassen. Bei der ontologiegetriebenen Softwareentwicklung werden Ontologien verwendet, um das Wissen einer Anwendung zu modellieren und um aus dem so modellierten Wissen APIs zur Verwaltung von Daten zu generieren.

Modellgetriebene Softwareentwicklung und ontologiegetriebene Softwareentwicklung haben jeweils verschiedene Vor- und Nachteile. Die modellgetriebene Softwareentwicklung hat den Vorteil, dass die Informationen und Semantik von Modellen sich relativ leicht im Programmcode abbilden lassen. Dadurch lassen sich Modelle gut für die Beschreibung von Anwendungen einsetzen. Zudem sind Softwareentwickler es gewohnt mit Modellen Anwendungen zu beschreiben. Dies erleichtert es modellgetriebene Software zu entwickeln.

Allerdings sind die modellgetriebenen Softwareentwicklungssysteme in der Regel so konzipiert, dass die Arbeit mit komplexen Einschränkungen für Modelle schwierig ist. Häufig müssen diese Einschränkungen über komplizierte Ausdrücke mit Quantoren definiert werden. So wäre zum Beispiel für das Fordern von Transitivität ein Ausdruck mit mehreren Quantoren nötig. Zudem müssen diese Einschränkungen meistens von Hand implementiert werden. Die Einhaltung von Einschränkungen wird selten zur Laufzeit gewährleistet. Häufig werden die Einschränkungen nur überprüft, wenn ihre Einhaltung für das Ausführen einer Aktion notwendig ist. Sind die Einschränkungen zu diesem Zeitpunkt nicht erfüllt, so muss man selbst einen korrekten Zustand herstellen, um die Aktion ausführen zu können.

Die ontologiegetriebene Softwareentwicklung hat den Vorteil, dass sich Ontologien gut für die Repräsentation von Wissen und das Definieren von Restriktionen eignen. Da Ontologien auf Logiken basieren ist es zudem möglich automatische Beweiser für Ontologien zu verwenden. Diese Beweiser können für das Überprüfen von Restriktionen, das Inferieren von Fakten und zur Anfragebearbeitung eingesetzt werden. Dadurch eignen sich Ontologien wesentlich besser für die Arbeit mit Restriktionen als Modelle.

Allerdings befriedigen die ontologiegetriebenen Softwareentwicklungssysteme nicht die Bedürfnisse der meisten Softwareentwickler. Da Ontologien auf Logiken basieren und keine Programmiersprachen sind, unterscheidet sich die Semantik der Ontologien in einigen Punkten stark von der Semantik der verbreiteten Programmiersprachen. Dies erschwert es Software mit Ontologien zu modellieren. Außerdem sind Softwareentwickler in der Regel noch nicht mit Ontologien vertraut, was sie zusätzlich von der Verwendung ontologiegetriebener Softwareentwicklungssysteme abschreckt.

Für die zukünftige Softwareentwicklung wäre es wünschenswert, dass man Modelle für die Beschreibung der Anwendungen und Ontologien für die Arbeit mit Restriktionen verwenden kann. Die Modelle könnten zum Beispiel die Eigenschaften der Klassen der Anwendung und Ontologien die Restriktionen für die Klassen beschreiben. Daher wäre es sinnvoll modellgetriebene Softwareentwicklung und ontologiegetriebene Softwareentwicklung miteinander zu kombinieren, um die Vorteile beider nutzen und ihre Beschränkungen umgehen zu können. Allerdings muss eine solche Kombination auch die Unterschiede von Modellen und Ontologien beachten.

Die Frage ist, wie modellgetriebene und ontologiegetriebene Softwareentwicklung kombiniert werden können, sodass die Funktionalitäten der modellgetriebenen Softwareentwicklung bewahrt werden, dass Entwickler aus dem

modellgetriebenen Bereich eine gewohnte Entwicklungsumgebung und deren Werkzeuge nutzen können und dass zumindest einige Funktionalitäten der ontologiegetriebenen Softwareentwicklung zur Verfügung stehen.

In dieser Arbeit präsentieren wir einen Ansatz für eine solche Kombination aus modellgetriebener und ontologiegetriebener Softwareentwicklung. Diese Arbeit hat zum Ziel ein System für modellgetriebene Softwareentwicklung so zu erweitern, dass zusätzlich Wissen mit Ontologien modelliert werden kann und dass das automatische Beweisen und Inferieren auf diesen Ontologien möglich ist. Das von den Ontologien modellierte Wissen soll ebenfalls Einfluss auf die Codegenerierung haben.

Für die Entwicklung dieses Ansatzes wurde die Sicht eines Entwicklers aus dem modellgetriebenen Bereich angenommen, um eine gute Akzeptanz in der relativ großen Gemeinde der modellgetriebenen Entwickler zu erreichen. Daher wurde der Ansatz so konzipiert, dass die Funktionalitäten des modellgetriebenen Systems erhalten bleiben und die für dieses System entwickelten Werkzeuge auch für das erweiterte System verwendet werden können.

Diese Kompatibilität wird dadurch erreicht, dass für die Erweiterungen der Modellierung die Annotationen des ursprünglichen Systems genutzt werden. Es wurden Annotationen entwickelt, die zusammen mit den Informationen des Modells eine Ontologie für das Modell definieren. Die Ontologie definiert zusätzliche Restriktionen für das Modell. Somit ist es zum Beispiel möglich zu definieren, dass eine Referenz transitiv sein soll. Die Einhaltung einiger Restriktionen der Ontologie wird vom generierten Code zur Laufzeit gewährleistet. Zum Beispiel sorgt der generierte Code dafür, dass die Transitivität einer Referenz nicht verletzt werden kann.

Außerdem bietet das erweiterte System die Annotationen an, die Anfragen auf der definierten Ontologie mit Operationen verknüpfen. Die Anfragen modellieren dann das Verhalten dieser Operationen. So ist es möglich mit einer Anfrage auf der Ontologie festzulegen, welche Werte eine Operation als Ergebnis liefert.

Ein System, das dem vorgestellten Ansatz entsprechend entwickelt wurde, nutzt sowohl Entwicklern aus dem ontologiegetriebenen Bereich als auch Entwicklern aus dem modellgetriebenen Bereich. Entwickler aus dem ontologiegetriebenen Bereich erhalten ein Softwareentwicklungssystem, das ihnen die Möglichkeit bietet Ontologierestriktionen im Code abzubilden. Zudem können sie mit Anfragen auf Ontologien das Verhalten von Operationen dynamisch modellieren.

Entwickler aus dem modellgetriebenen Bereich bietet das erweiterte System eine gewohnte Entwicklungsumgebung für modellgetriebene Softwareentwicklung, die es ihnen erlaubt ihre gewohnten Werkzeuge zu nutzen und die ihnen zusätzlich die Verwendung von Ontologien zur Definition von Restriktionen und die Modellierung des Verhaltens von Operationen ermöglicht.

1.2 Gliederung

Die Arbeit gliedert sich in acht Kapitel. In Kapitel 2 wird ein einfaches Szenario für die Nutzung einer Kombination aus modellgetriebener und ontologiegetriebener Entwicklung vorgestellt. Dieses Szenario wird in den folgenden Kapiteln aufgegriffen, um Sachverhalte zu veranschaulichen. In Kapitel 3 werden die Grundlagen für die Arbeit behandelt und einige verwandte Arbeiten vorgestellt. Die verschiedenen funktionalen Anforderungen für die gewünschte Erweiterung eines Systems für modellgetriebene Softwareentwicklung werden in Kapitel 4 aufgeführt. Der Lösungsansatz dieser Arbeit wird in Kapitel 5 vorgestellt und beschrieben. Wie dieser Ansatz implementiert wurde, wird in Kapitel 6 erläutert. Kapitel 7 stellt den entwickelten Prototyp anhand des Szenarios vor. Schließlich wird in Kapitel 8 der Inhalt der Arbeit zusammengefasst und ein Ausblick auf mögliche Weiterentwicklungen gegeben.

Szenario

In diesem Kapitel wird ein einfaches Szenario, das als Beispiel für die folgenden Kapitel dienen soll, vorgestellt. Das Szenario ist ein Bestellvorgangsszenario, das aus den verschiedenen Szenarien in [36] entwickelt wurde. Das Szenario befasst sich mit der Erstellung von Software zur Verwaltung verschiedener Bestellungen von einem oder mehreren Lieferanten. Dabei soll ein Großteil des zu erstellenden Programmcodes automatisch aus einem Klassendiagramm generiert werden. Im Abschnitt 2.1 wird zunächst das Szenario und dessen grundlegende Modellierung beschrieben. Anschließend werden im Abschnitt 2.2 verschiedene weitere Modellierungswünsche für das Szenario formuliert. Im Abschnitt 2.3 werden schließlich die Herausforderungen des Szenarios erläutert.

2.1 Bestellvorgangsszenarios

Das Szenario befasst sich mit der Erstellung von Software zur Verwaltung verschiedener Bestellungen von einem oder mehreren Lieferanten. Dabei soll ein Großteil des zu erstellenden Programmcodes automatisch aus einem Klassendiagramm generiert werden. Die Abbildung 2.1 zeigt ein Klassendiagramm für das Bestellvorgangsszenario, das ein Modell der zu erstellenden Software zur Verwaltung der Bestellungen darstellt. Das Klassendiagramm modelliert Lieferanten (**Supplier**), Kunden (**Customer**), Bestellungen (**Order**), Bestellungen (Item) und Adressen (**Address**) sowie deren Beziehungen zueinander.

Die Lieferanten werden über ihren Namen, der in ihrem **name** Attribut gespeichert wird, identifiziert. Jeder Lieferant führt eine Liste der Kunden, die bei ihm eine Bestellung aufgegeben haben. Diese Liste kann über die **customers** Referenz aufgerufen werden. Ebenso führt jeder Lieferant eine Liste der Bestellungen, die bei ihm eingegangen sind. Die verschiedenen Bestellungen werden dem Lieferanten über die **orders** Referenz zugeordnet. Zudem

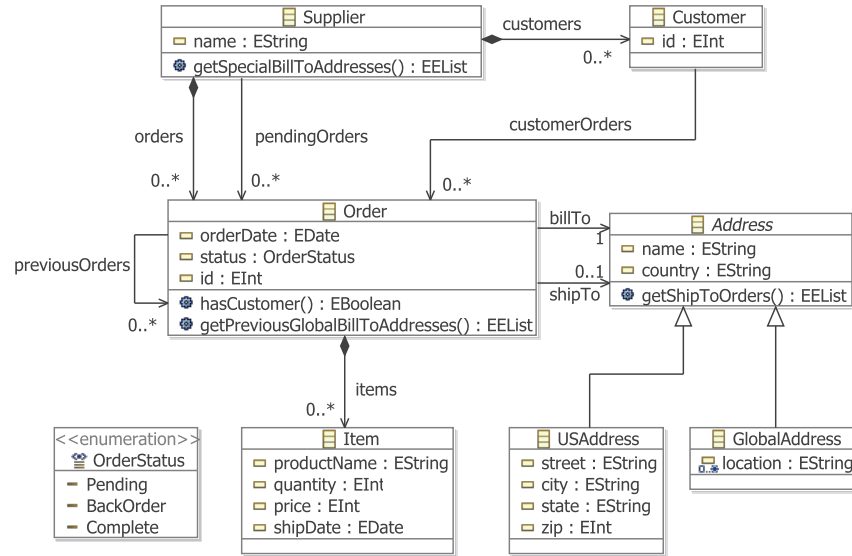


Abb. 2.1. Klassendiagramm des Bestellvorgangsszenarios

pflegt jeder Lieferant eine Liste der unerledigten Bestellungen, um einen guten Überblick über die noch zu erledigenden Aufgaben zu haben. Diese Liste kann über die `pendingOrders` Referenz aufgerufen werden.

Zur Identifizierung der Kunden eines Lieferanten erhält jeder Kunde eine ID, die im `id` Attribut gespeichert wird. Für jeden Kunden eines Lieferanten wird eine Liste der Bestellungen, die von dem Kunden aufgegeben wurden, geführt, um schnell die bisherigen Bestellungen eines Kunden finden zu können. Diese Liste kann über die `customerOrders` Referenz aufgerufen werden.

Die verschiedenen Bestellungen eines Lieferanten erhalten zur Identifizierung eine ID, die im `id` Attribut gespeichert wird. Zudem werden sowohl das Datum, an dem die Bestellung aufgegeben wurde, als auch der aktuelle Status der Bestellung gespeichert. Für das Bestelldatum wird das `orderDate` Attribut und für den Bestelldatum wird das `status` Attribut verwendet. Der Status einer Bestellung kann drei verschiedene Werte annehmen: unerledigt (`Pending`), Lieferrückstand (`BackOrder`) und abgeschlossen (`Complete`). Der Status einer Bestellung ist Lieferrückstand, falls für irgendein Produkt der Bestellung Lieferprobleme existieren.

Um Rückschlüsse aus vorangegangenen Bestellungen schließen zu können, wird für jede Bestellung eine Liste der vorangegangenen Bestellungen, die mit der aktuellen Bestellung assoziiert sind, erstellt. Diese Liste kann über die `previousOrders` Referenz der Bestellung aufgerufen werden. Die einzelnen

Posten (**Item**) einer Bestellung werden in der Postenliste der Bestellung verwaltet. Auf diese Liste kann mit der **items** Referenz zugegriffen werden.

Die einzelnen Posten enthalten den Namen des mit dem Posten assoziierten Produkts. Dieser Name wird im **productName** Attribut gespeichert. Für jeden Posten der Bestellung werden der Produktpreis und die Anzahl der bestellten Produkte erfasst. Der Produktpreis wird im **price** Attribut und die Produktanzahl im **quantity** Attribut des Postens gespeichert. Zudem wird zur Verfolgung des Status für jeden Posten im **shipDate** Attribut gespeichert, wann die entsprechenden Produkte verschickt wurden.

Jeder Bestellung werden ein bis zwei Adressen zugeordnet. Es gibt immer eine Rechnungsadresse. Auf die Rechnungsadresse wird mit der **billTo** Referenz verwiesen. Falls die Produkte der Bestellung nicht an die Rechnungsadresse gesendet werden sollen, verweist die **shipTo** Referenz auf die Lieferadresse. Der Name des Adressaten wird in dem **name** Attribut der Adresse gespeichert. Das Land der Adresse wird in das **country** Attribut kodiert.

Da die Lieferanten ihren Firmensitz in den USA haben, wird bei den Adressen zwischen zwei Arten von Adressen unterschieden: Adressen in den USA (**USAddress**) und andere globale Adressen (**GlobalAddress**). Für die Adressen in den USA werden zusätzlich Straße, Ort, Staat und Postleitzahl gespeichert. Dafür werden die Attribute **street**, **city**, **state** und **zip** verwendet. Für die anderen Adressen wird lediglich eine Liste der weiteren Adressdaten gespeichert. Für die Speicherung dieser Liste wird das **location** Attribut verwendet.

Zusätzlich zu den oben beschriebenen Eigenschaften soll die Software zur Verwaltung der Bestellungen dem Nutzer einige Operationen zur Verfügung stellen. Für die verschiedenen Lieferanten soll eine Operation **getSpecialBillToAddresses** bereitgestellt werden. Diese Operation soll eine Liste aller Adressen zurückliefern, die Rechnungsadresse einer Bestellung dieses Lieferanten sind und deren Attribute ein bestimmtes Kriterium erfüllen. Mit dieser Operation soll es ermöglicht werden, dass ein Lieferant seine Rechnungsadressen gefiltert nach einem bestimmten Kriterium erhält. Zunächst soll diese Operation lediglich die Rechnungsadressen zurückliefern, deren **country** Attribut die Kodierung für Deutschland enthält. Es soll aber auch möglich sein, später ein anderes Kriterium zu wählen.

Für die einzelnen Bestellungen soll die **hasCustomer** Operation es ermöglichen zu überprüfen, ob die jeweilige Bestellung in der Bestellungsliste eines Kunden (**customerOrders**) enthalten ist. Ist dies der Fall, so soll diese Operation **true** zurückliefern. Anderenfalls soll das Ergebnis **false** sein. Zusätzlich soll für die einzelnen Bestellungen die Operation **getPreviousGlobalBillToAddresses** bereitgestellt werden. Diese Operation liefert eine Liste von Rechnungsadressen. In dieser Liste sind Rechnungsadressen von vorangegangenen Bestellungen, die mit der jeweiligen Bestellung assoziiert sind.

Für die verschiedenen Adressen soll die **getShipToOrders** Operation angeboten werden. Diese Operation liefert für eine Adresse eine Liste von Bestel-

lungen. Diese Liste enthält diejenigen Bestellungen, für die als Lieferadresse die jeweilige Adresse angegeben wurde.

2.2 Weitere Modellierungswünsche für das Bestellvorgangsszenario

Zusätzlich zu der im vorangegangenen Absatz beschriebenen Modellierung sollen einige weitergehende Beschränkungen für das Modell festgelegt werden. Es sollen drei OWL Beschränkungen, die Referenzen des Klassendiagramms betreffen, zum Modell hinzugefügt werden. Außerdem sollen Modellierungen für das Verhalten der vier Operationen des Klassendiagramms in das Modell aufgenommen werden.

Als zusätzliche Bedingung soll festgelegt werden, dass für einen beliebigen Lieferanten und eine beliebige Bestellung gelten muss, dass sie auch über die `orders` Referenz miteinander verbunden sein müssen, wenn sie über die `pendingOrders` Referenz miteinander verbunden sind. Daraus resultiert die Bedingung: Die Menge aller Paare (Lieferant, Bestellung), die über die `pendingOrders` Referenz miteinander verknüpft sind, ist eine Teilmenge der Paare (Lieferant, Bestellung), die über die `orders` Referenz miteinander verknüpft sind. Zur Modellierung dieser Bedingung soll die OWL Beschränkung `SubObjectPropertyOf(pendingOrders, orders)` zum Modell hinzugefügt werden.

Da für eine Bestellung nur dann eine Lieferadresse gespeichert werden soll, wenn sich die Lieferadresse von der Rechnungsadresse unterscheidet, soll als zusätzliche Bedingung festgelegt werden, dass eine Bestellung und eine Adresse nicht sowohl über die `billTo` als auch über die `shipTo` Referenz miteinander verbunden werden dürfen. Dies führt zu der Bedingung, dass die Menge der Paare (Bestellung, Adresse), die über die `billTo` Referenz miteinander verknüpft sind, und die Menge der Paare (Bestellung, Adresse), die über die `shipTo` Referenz miteinander verknüpft sind, disjunkt sein müssen. Daraus resultiert die OWL Beschränkung: `DisjointObjectProperties(billTo, shipTo)`.

Für jede Bestellung c soll gelten, dass wenn eine Bestellung a einer Bestellung b vorangegangen ist und die Bestellung b der Bestellung c vorangegangen ist, somit auch die Bestellung a der Bestellung c vorangegangen sein muss ($\forall a \forall b \forall c ((a, b) \wedge (b, c)) \Rightarrow (a, c)$). Daher wird gefordert, dass die `previousOrders` Referenz transitiv ist. Zur Abbildung dieser Bedingung soll die OWL Beschränkungen `TransitiveObjectProperty(previousOrders)` in das Modell aufgenommen werden.

Es wird gewünscht möglichst viel Code der Verwaltungssoftware automatisch zu generieren. Daher soll auch das Verhalten der vier Operationen des Klassendiagramms modelliert werden. Diese Modellierung soll definieren, welche Werte von den Operationen zurückgegeben werden. Zudem sollte das Verhalten der Operationen möglichst dynamisch anpassbar sein, damit das Aus-

wahlkriterium der `getSpecialBillToAddresses` Operation verändert werden kann ohne den Programmcode zu ändern. Dies soll geschehen, indem das Verhalten der Operationen mit Anfragen modelliert wird. Diese Anfragen sollen in einem begrenzten Rahmen zur Laufzeit modifiziert werden können und diese Modifizierungen sollen dann eine Änderung des Verhaltens der Operation bewirken. Dies erlaubt es, das Verhalten der Operationen ohne großen Aufwand anzupassen, indem die modellierende Anfrage angepasst wird.

2.3 Herausforderungen des Bestellvorgangsszenarios

Eine wesentliche Herausforderung des Szenarios ist es, möglichst viel Programmcode automatisch aus einem Modell generieren zu können. Es sollen sowohl die Codegenerierungsfunktionen eines einfachen Systems zur modellgetriebenen Softwareentwicklung als auch einige Bestandteile der ontologiegetriebenen Softwareentwicklung bereitgestellt werden. Dabei werden die ontologiebasierten Bestandteile für die zusätzlichen Modellierungswünsche, die in Abschnitt 2.2 beschrieben sind, benötigt.

Mit reinen modellgetriebenen Systemen ist es nicht möglich diese zusätzlichen Modellierungen in das Modell aufzunehmen und somit auch nicht den entsprechenden Programmcode zu generieren. Dies bedeutet, dass der entsprechende Code manuell eingefügt werden muss, was einen Mehraufwand sowohl bei der erstmaligen Softwareerstellung als auch bei späteren Änderungen darstellt.

Mit reinen ontologiegetriebenen Systemen ist zwar eine Modellierung der Beschränkungen, die die Referenzen betreffen, möglich, aber es können keine Operationen mit Ontologien modelliert werden. Daher kann kein Code für die Operationen erzeugt werden. Somit muss der komplette Code für die Operationen manuell eingefügt werden. Dies bedeutet wiederum einen erheblichen Mehraufwand bei der erstmaligen Softwareerstellung und späteren Änderungen.

Für das Szenario muss also ein System für modellgetriebene Softwareentwicklung erweitert werden, sodass es zusätzlich möglich ist Code, der die Semantik von OWL Restriktionen implementiert, zu generieren und Anfragen zur Modellierung von Operationen so zu verwenden, dass Code für die Operationen erzeugt wird und dass das Verhalten der Operation entsprechend angepasst wird, falls die Anfrage verändert wird.

Grundlagen und Verwandte Arbeiten

In diesem Kapitel werden die Grundlagen für die Arbeit und einige verwandte Arbeiten behandelt. Die Grundlagen werden kurz erläutert und es wird auf weiterführende Quellen zu den Grundlagen verwiesen. Ebenso werden die Ansätze verschiedener verwandter Arbeiten kurz zusammengefasst und Unterschiede zum Ansatz dieser Arbeit aufgezeigt.

Die vorgestellten verwandten Arbeiten befassen sich mit ontologiegetriebener Softwareentwicklung, mit den Unterschieden von Ontologien und objektorientierten Konzepten, mit Abbildungen zwischen Ontologien und objektorientierten Konzepten und mit der Nutzung von Ontologien in Anwendungen. Mir ist keine Arbeit bekannt, die ein System für modellgetriebene Softwareentwicklung um zusätzliche Funktionalitäten aus der ontologiegetriebenen Softwareentwicklung erweitert.

In Abschnitt 3.1 wird zunächst auf Modelle eingegangen. Dabei werden insbesondere Ontologien, deren Nutzen und verfügbare Methoden zur Arbeit mit Ontologien beschrieben. Die Grundlagen zur Codegenerierung werden in Abschnitt 3.2 behandelt. Zudem werden Ansätze für modellgetriebene Softwareentwicklung und für ontologiegetriebene Softwareentwicklung beschrieben. Die Ansätze für ontologiegetriebene Softwareentwicklung definieren ebenfalls Abbildungen zwischen Ontologien und objektorientierten Konzepten. In Abschnitt 3.3 werden die Konzepte von Ontologien und objektorientierte Konzepte miteinander verglichen. Schließlich werden in Abschnitt 3.4 Ansätze, die sich mit der sinnvollen Nutzung von Ontologien in Anwendungen beschäftigen, vorgestellt.

3.1 Modelle

In der Softwareentwicklung haben Modelle eine wichtige Rolle. Bei der Softwareentwicklung werden Modelle häufig dazu verwendet einen Realitätsausschnitt abstrakt darzustellen. Durch die zunehmende Verbreitung der objektorientierten Softwareentwicklung ist ebenfalls die Bedeutung von Modellen im

Softwareentwicklungsprozess stark gewachsen, da Modelle häufig als Bauplan für die objektorientierte Erstellung von Software dienen.

Es gibt viele verschiedene Arten von Modellen, die jeweils unterschiedliche Informationen liefern. Modelle können sowohl Informationen über die Struktur einer Anwendung als auch Informationen über das Verhalten einer Anwendung enthalten. Welche Elemente und Informationen ein Modell enthalten darf, wird in der Regel durch das Metamodell des Modells festgelegt.

Klassendiagramme stellen wohl die am häufigsten verwendete Modellart dar. Klassendiagramme werden in der Regel dazu verwendet die verschiedenen Klassen einer Anwendung zu beschreiben. In Klassendiagrammen sind Informationen zu den Klassen einer Anwendung, den Inhalten der Klassen und den Beziehungen der Klassen zueinander enthalten.

Im Laufe der Zeit wurden viele unterschiedliche Modellierungssprachen zur Definition von Modellen entwickelt. Zur einheitlichen Beschreibung von Entwürfen für Anwendungen wurde die **Unified Modeling Language (UML)**[20, 32] als Standardmodellierungssprache eingeführt. Die UML beinhaltet Definitionen für mehrere Modellarten zur Beschreibung der Struktur und des Verhaltens einer Anwendung bzw. eines Systems.

Im Folgenden sollen zwei Arten von Modellen betrachtet werden: UML-artige Modelle (Abschnitt 3.1.1) und Ontologien (Abschnitt 3.1.2). Dabei liegt der Fokus auf Ontologien. Obwohl Ontologien auch Modelle sind, soll im folgenden Text dieser Arbeit mit dem Begriff Modell stets nur UML-artige Modelle gemeint sein.

3.1.1 UML-artige Modelle

Als UML-artige Modelle sollen UML Modelle und Modelle, die lediglich Informationen enthalten, die auch mit UML Modellen darstellbar sind, bezeichnet werden. Ein Beispiel für ein UML-artiges Modell ist ein **Ecore** Modell [36]. Ein **Ecore** Modell enthält lediglich Informationen, die auch mit UML Modellen darstellbar sind. In der Regel sind die Informationen eines **Ecore** Modells mit Kombinationen von UML Paketdiagrammen und UML Klassendiagrammen darstellbar.

Die meisten Informationen, die zur Erstellung einer Anwendung benötigt werden, lassen sich mit UML-artigen Modellen darstellen. Daher fertigen Entwickler häufig lediglich UML-artige Modelle für die Modellierung einer Anwendung an.

3.1.2 Ontologien

Ontologien sind spezielle Modelle, die häufig zur Wissensrepräsentation verwendet werden. In Ontologien kann Wissen über verschiedene Klassen und Objekte, deren **Properties** und die Beziehungen von Klassen, Objekten und **Properties** repräsentiert werden. Es gibt zwei Arten von **Properties**: **DataProperties** und **ObjectProperties**. Die Werte von **DataProperties** stellen Datenwerte

dar wohingegen die Werte von `ObjectProperties` auf Objekte verweisen. Ontologien können mittels verschiedener Restriktionen Wissen über Klassen und `Properties` formulieren.

Es gibt viele verschiedene Sprachen zur Definition von Ontologien. Im Unterabschnitt 3.1.2 wird auf wichtige Ontologiesprachen kurz eingegangen. Die verschiedenen Ontologiesprachen sind so konstruiert, dass Computer das durch Ontologien repräsentierte Wissen auswerten und in einem gewissen Maß verstehen können. Um dies zu ermöglichen, basieren Ontologiesprachen auf bestimmten Logiken. In der Regel sind dies Beschreibungslogiken.

Beschreibungslogiken [2] sind spezielle Logiken, die für die Wissensrepräsentation gedacht sind. Die meisten Beschreibungslogiken stellen eine entscheidbare Untermenge der Prädikatenlogiken erster Stufe [34] dar. Die Entscheidbarkeit ermöglicht es Schlussfolgerungen aus dem vorhandenen Wissen zu ziehen. Im Unterabschnitt 3.1.2 sind einige Methoden aufgeführt, die für entscheidbare Beschreibungslogiken sowie auf solchen Logiken basierenden Ontologien zur Verfügung stehen.

Bei Beschreibungslogiken und Ontologien wird häufig eine Unterteilung in terminologische Box (TBox) und assertionale Box (ABox) vorgenommen. Die TBox enthält das terminologische Wissen über Konzepte bzw. Klassen und deren Beziehungen. Die ABox hingegen enthält das Wissen über die Instanzen bzw. Objekte und deren Eigenschaften.

Ontologiesprachen

Es gibt viele verschiedene Sprachen zur Definition von Ontologien. Die wichtigsten Ontologiesprachen sind wohl Resource Description Framework (RDF) und Web Ontology Language (OWL).

RDF [17, 15] wurde zur formalen Beschreibung von Informationen über Ressourcen entwickelt. Eine Ressource stellt ein über eine URI bzw. IRI eindeutig identifizierbares Objekt dar. Die Informationen über die Ressourcen werden als Tripel der Form (Subjekt, Prädikat, Objekt) gespeichert. Ein Tripel stellt jeweils eine Aussage dar. Das Subjekt entspricht der Ressource, für die eine Aussage formuliert wird. Das Prädikat ist eine Eigenschaft des Subjekts und das Objekt das Argument des Prädikats. Die Tripel ergeben zusammen einen Graphen, der das modellierte Wissen enthält. Für RDF existiert ein XML Format, das häufig für die Serialisierung von Ontologien und den Austausch von Ontologien zwischen verschiedenen Programmen verwendet wird.

OWL [18, 15] ist eine standardisierte Sprache zur Definition von Ontologien, die auf RDF basiert. OWL stellt zusätzliche Sprachkonstrukte zur Definition von Ontologierestriktionen zur Verfügung. Für OWL existieren verschiedene Sprachebenen, die sich in ihrer Mächtigkeit unterscheiden. Die unterschiedlichen Sprachebenen schränken den Gebrauch der Sprachkonstrukte ein und erreichen so eine Beschränkung der Mächtigkeit. Am gebräuchlichsten sind wohl die Sprachebenen OWL Full und OWL DL. OWL Full ist die mächtigste

Sprachebene. Für diese Sprachebene ist der Gebrauch der Sprachkonstrukte nicht eingeschränkt. Die OWL DL Sprachebene schränkt die Mächtigkeit so ein, dass sie der Mächtigkeit einer entscheidbaren Beschreibungslogik entspricht.

Verfügbare Methoden zur Arbeit mit Ontologien

Für Beschreibungslogiken und Ontologien existieren automatische Beweiser, die es ermöglichen Konsistenzüberprüfungen und Anfragen durchzuführen. Damit der Beweiser ein Ergebnis liefern kann, sollte das an ihn gestellte Problem entscheidbar sein. Bei einer Konsistenzüberprüfung wird geprüft, ob es Widersprüche im vorhandenen Wissen gibt. Für Anfragen auf Ontologien kann die SPARQL Protocol and RDF Query Language (SPARQL)[29] verwendet werden. SPARQL ist eine graphbasierte Anfragesprache für RDF Ontologien. Der automatische Beweiser Pellet¹ erlaubt sowohl das Ausführen von SPARQL Anfragen als auch das Durchführen von Konsistenzüberprüfungen auf Ontologien.

3.2 Codegenerierung

Bei der Codegenerierung wird aus einer Vorlage automatisch Programmcode generiert. Abbildung 3.1 zeigt den verallgemeinerten Ablauf bei der Codegenerierung. In einer Vorlage werden verschiedene Informationen für den zu generierenden Code gespeichert. Der Generator liest die Vorlage ein und verwendet Templates, um festzustellen, wie er aus den Informationen der Vorlage den zu generierenden Code herleiten kann. Anschließend generiert und speichert der Generator den entsprechenden Code.

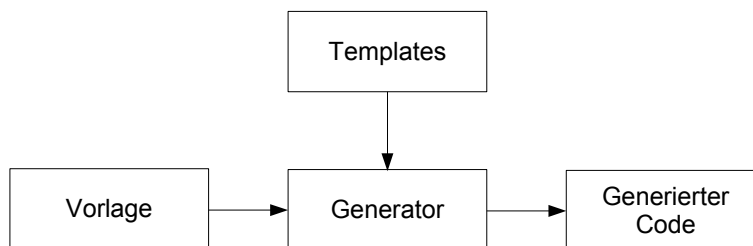


Abb. 3.1. Verallgemeinerten Codegenerierungsablauf

Im Folgenden werden zwei Softwareentwicklungsansätze, die Codegenerierung verwenden, betrachtet: modellgetriebene Softwareentwicklung (3.2.1) und ontologiegetriebene Softwareentwicklung (3.2.2).

¹ Internetseite Pellet: <http://clarkparsia.com/pellet/>

3.2.1 Modellgetriebene Softwareentwicklung

Bei der modellgetriebenen Softwareentwicklung werden Modelle als Vorlage für den zu generierenden Code verwendet. Es gibt mittlerweile viele Systeme für modellgetriebene Softwareentwicklung. Ein Beispiel für ein solches System ist das Eclipse Modeling Framework (EMF)[36, 26, 6].

EMF bietet dem Entwickler das Ecore Metamodell zum Entwerfen von Modellen an. In Ecore Modellen können unter anderem die Informationen eines Klassendiagramms und Informationen über Pakete gespeichert werden. Abbildung 3.2 zeigt den vereinfachten Softwareentwicklungsablauf mit EMF. Zunächst entwirft der Entwickler ein Ecore Modell. Aus diesem Ecore Modell wird ein GenModel erzeugt. Ein GenModel enthält die Informationen eines Ecore Modells und fügt diesen spezielle Informationen für den Generator zur Codegenerierung hinzu. Zum Beispiel werden im GenModel Pfadangaben hinzugefügt, die festlegen, wohin der generierte Code gespeichert wird.

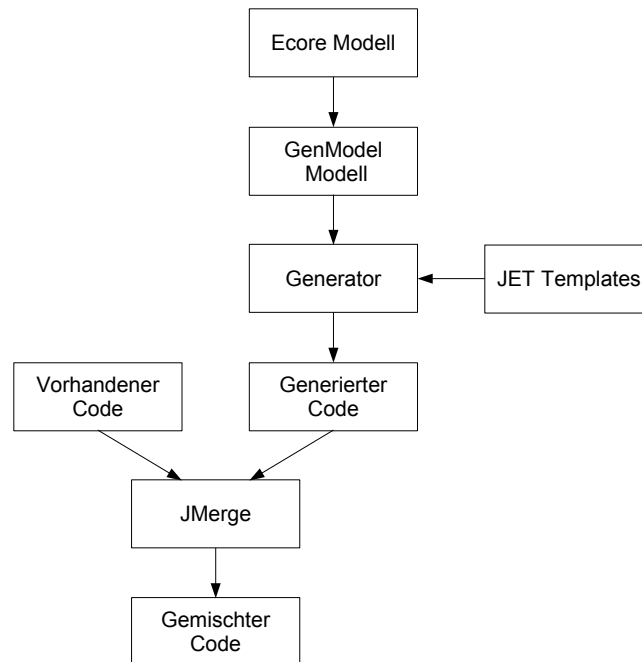


Abb. 3.2. Vereinfachter EMF Softwareentwicklungsablauf

Der Generator liest das GenModel ein und verwendet Java Emitter Templates (JET)[24, 25, 27], um aus dem GenModel Code zu generieren. Sollte bereits Code vorhanden sein, so verwendet der Generator JavaMerge (JMerge)[28] um den vorhandenen Code mit dem generierten Code zu vermischen.

3.2.2 Ontologiegetriebene Softwareentwicklung

Bei der ontologiegetriebenen Softwareentwicklung wird das von Ontologien modellierte Wissen als Vorlage für den zu generierenden Code verwendet. Häufig wird dabei auch von der Generierung von Ontologie APIs geredet. Mittlerweile gibt es mehrere unterschiedliche Ansätze für ontologiegetriebene Softwareentwicklung. Im Folgenden werden einige Ansätze für Systeme für ontologiegetriebene Softwareentwicklung beschrieben.

OntoJava und OntoSQL

Andreas Eberhart stellt in seiner Arbeit [7] zwei Ansätze zur ontologiegetriebenen Softwareentwicklung vor. Als Vorlage werden eine RDF-Schema [5, 15] Ontologie und RuleML [4] Regeln verwendet. Die Codegenerierung wird von speziellen Compilern übernommen. Es stehen zwei verschiedene Compiler zur Verfügung: *OntoJava* und *OntoSQL*. Der *OntoJava* Compiler erzeugt Java Code, wohingegen der *OntoSQL* Compiler SQL Code für eine IBM DB2 Datenbank erzeugt.

Der *OntoJava* Compiler generiert ein System von Java Klassen, die eine Objektdatenbank mit eingebauten Regeln zum Inferieren weiterer Fakten bilden. Für die verschiedenen Klassen der Ontologie wird jeweils eine entsprechende Java Klasse erzeugt. Die Vererbungshierarchie der Ontologien wird für die Java Klassen übernommen. Dabei werden für die Abbildung der Mehrfachvererbung Interfaces verwendet. Entsprechend der Domain Angaben für die Properties der Ontologie werden Instanzvariablen sowie entsprechende Zugriffsoperationen in die Java Klassen eingefügt. Zudem wird eine weitere Klasse erzeugt, die Factory Operationen für das Erzeugen von Instanzen und Operationen zum Übertragen der Assertions der Ontologie enthält.

Die RuleML Regeln werden jeweils durch eine statische Operation im Java Code repräsentiert. Eine solche Operation simuliert das Anwenden einer Regel. Die Operationen für die Regeln werden bei relevanten Änderungen vom generierten Code aufgerufen. Somit werden gegebenenfalls weitere Fakten hergeleitet. Dieser Mechanismus wird ebenfalls für die Abbildung der `subPropertyOf` Ausdrücke verwendet.

Der *OntoSQL* Compiler erzeugt SQL Code für eine IBM DB2 Datenbank. Die Assertions der Ontologie werden in Tripelform (Subjekt, Prädikat, Objekt) in einer Datenbanktabelle gespeichert. Dabei sind die drei Spalten Teil des zusammengesetzten Primärschlüssels. Für Datalog Anfragen werden entsprechende SQL Select Anfragen erzeugt. Die Datalog Regeln werden durch Views, die SQL Ausdrücke zur Repräsentation der Regel enthalten, dargestellt.

Die von Andreas Eberhart vorgestellten Ansätze ermöglichen es das Domänenwissen einer RDF-Schema Ontologie sowie die Assertions der Ontologie im generierten Code abzubilden. Ebenso kann mit RuleML Regeln das Inferieren weiterer Fakten durch den generierten Code veranlasst werden. Allerdings behandelt dieser Ansatz die Codegenerierung eher aus der Sicht eines Ontologieentwicklers als aus der Sicht eines Softwareentwicklers. Die Modellierung des

Codes alleine durch die Ontologie ist für Softwareentwickler unpraktisch, da sie in der Regel kaum Kenntnisse über Ontologien haben und mit UML-artigen Modellen besser ihre Wünsche für den Code modellieren können. Zudem bietet dieser Ansatz keine Modellierungsmöglichkeiten für Operationen an und Softwareentwickler können nicht ihre gewohnten Werkzeuge verwenden.

Abbildung von OWL Ontologien zu Java

In der Arbeit von Kalyanpur et al. [12] wird eine Abbildung von OWL Ontologien zu Java Code vorgestellt. Diese Abbildung kann für die ontologiegelebene Softwareentwicklung genutzt werden. Für diesen Ansatz wird als Einschränkung gefordert, dass die Individuen der Ontologie nicht zu zwei Klassen, die nicht über die Vererbungshierarchie miteinander verbunden sind, gehören können.

Zur Unterstützung der Mehrfachvererbung wird jede Ontologieklass auf ein Java Interface abgebildet. Dieses Interface enthält die Zugriffsoperationen für die Properties, die über Domain Konstrukte dieser Ontologieklass zugeordnet wurden. Die Subklassenbeziehungen der Ontologie werden auf die Java Interfaces übertragen. Zudem wird für jede Ontologieklass eine Java Klasse erzeugt, die von dem entsprechenden Interface erbt. Für äquivalente Ontologieklassen wird ein gemeinsames Interface erzeugt.

Verschiedene ClassExpressions und Klassenaxiome werden in der Struktur der Java Interfaces und Klassen abgebildet. oneOf Konstrukte werden durch Enumerationen repräsentiert. Ebenfalls abgebildet werden intersectionOf, unionOf, complementOf und disjointWith. Diese Abbildungen werden über die Java Interfaces und Klassen, deren Vererbungshierarchie und über einen Blockiermechanismus, der die Einschränkungen für das Überladen von Java Operationen ausnutzt, realisiert.

Für die Abbildung von Property Restriktionen werden verschiedene Listener verwendet, die eine Verletzung der Restriktionen verhindern sollen. Dabei werden zwei Arten von Listnern benutzt. Die eine Art wird genutzt, um Änderungen zu verhindern, falls sie zu einer Verletzung einer Bedingung führen. Die andere Art von Listener nimmt zusätzliche Änderungen vor, die für die Einhaltung einer Bedingung ebenfalls vorgenommen werden müssen. Diese Art wird zum Beispiel für die Einhaltung der Symmetrie verwendet.

Mit einem System, dass diese Abbildung zur Codegenerierung verwendet, können die Klassen, Properties und einige Restriktionen der Ontologie im Code abgebildet werden. Dabei sorgen die Listener und die Strukturabbildung für die Einhaltung einiger Ontologierestriktionen zur Laufzeit. Dieser Ansatz ist ebenfalls schlecht für Softwareentwickler anwendbar, da wiederum eine Ontologie für die Modellierung verwendet wird. Zudem erschweren die Strukturabbildungen einem Softwareentwickler die Arbeit, da er dadurch selten vorhersehen kann, welche Klassen und Interfaces letztendlich erzeugt werden. Außerdem gibt es keine Möglichkeit zusätzliche Operationen zu modellieren.

Transformation von OWL Ontologien in Ecore Modelle mit OCL Restriktionen

Rahmani et al. präsentieren in ihrer Arbeit [31] eine Transformation von OWL Ontologien in Ecore Modelle mit (OCL)[21] Restriktionen. Dieser Ansatz sieht vor, dass aus einer Ontologie mit dieser Transformation ein entsprechendes Ecore Modell mit OCL Restriktionen erzeugt wird. Anschließend wird der EMF Generator für die Generierung von Code aus diesem Ecore Modell verwendet.

Die Transformation bildet die Ontologieklassen auf Ecore Klassen ab. Ebenso wird die Vererbungshierarchie der Ontologie auf das Ecore Modell übertragen. Für anonyme Klassen wird eine Hilfsklasse als Repräsentant dieser anonymen Klasse in das Ecore Modell eingefügt oder es werden OCL Invarianten zur Repräsentation der anonymen Klassen erzeugt. Die Hilfsklassen für anonyme Ontologieklassen werden den ClassExpressions entsprechend in die Vererbungshierarchie eingebunden. Zur Repräsentation von URIs bzw. IRIs und von sameAs und differentFrom Konstrukten werden entsprechende Attribute und Referenzen in die Ecore Klasse eingefügt, die die OWL Thing Klasse repräsentiert.

Die Properties der Ontologie werden auf Referenzen und Attribute abgebildet. Zur Zuordnung der Properties zu den Ecore Klassen werden die Domain Angaben der Properties verwendet. Falls eine Property keine Domain Angabe hat, so wird sie der Repräsentantenklasse von OWL Thing zugeordnet. Die Hierarchie der Properties wird durch OCL Restriktionen repräsentiert. Ebenso werden die Property Restriktionen der Ontologie durch entsprechende OCL Restriktionen dargestellt.

Die Transformation dieser Arbeit sieht vor, dass der Entwickler mithilfe eines Reglers festlegen kann, wie viele Informationen der Ontologie für die Erstellung des Ecore Modells verwendet werden. Somit werden nur die Transformationsschritte durchgeführt, die vom Entwickler gewünscht werden.

Dieser Ansatz bietet die Möglichkeit das Domänenwissen einer OWL Ontologie im generierten Code abzubilden. Zudem stellt das erzeugte Ecore Modell mit OCL Restriktionen ein für Softwareentwickler gewohntes Modell dar, für das einige Entwicklungswerkzeuge verfügbar sind. Es wäre auch möglich in dieses erzeugte Modell zusätzliche Operationen aufzunehmen. Allerdings hat die Darstellung von Ontologierestriktionen als OCL Restriktionen den Nachteil, dass die OCL Restriktionen lediglich zur Überprüfung der Ontologierestriktionen zu einem Zeitpunkt verwendet werden können. Es ist nicht möglich Assertions mithilfe der OCL Restriktionen zu inferieren. Außerdem bietet der Ansatz keine Möglichkeit das Verhalten einer Operation zu modellieren.

Modellgetriebene Generation von Ontologie APIs

Die Arbeit von Scheglmann et al. [33] behandelt einen Ansatz zur schrittweisen Generierung von Ontologie APIs. Dieser Ansatz bietet dem Entwickler

Möglichkeiten die zu generierende API anzupassen, indem der Entwickler bestimmt, welche Teile der Ontologie in der API abgebildet werden sollen.

Als Vorlage für die Codegenerierung wird zunächst eine bereits existierende Ontologie eingelesen und aus dieser Ontologie ein MoOn Modell erzeugt. Das MoOn Modell basiert auf dem Ontology Definition Metamodel [19] und enthält zusätzlich einige weitere Modellierungsmöglichkeiten. Ein MoOn Modell repräsentiert eine Ontologie als ein UML Modell mit OWL Profil. Alle Informationen der Ontologie sind ebenfalls in dem MoOn Modell enthalten. Der Entwickler kann auf dem MoOn Modell Anpassungen auf der Ontologieebene vornehmen, ohne dass er die ursprünglich verwendete Ontologie verändern muss. Zudem wird in diesem Modell festgelegt, welche Ontologie Konzepte durch die API abgebildet werden sollen.

Anschließend wird das angepasste MoOn Modell in ein OAM Modell transformiert. Ein OAM Modell stellt eine objektorientierte Repräsentation der API dar. Alle Eigenschaften der API werden durch das OAM Modell definiert. Somit enthält das OAM Modell alle für die Generierung der API nötigen Informationen. Der Entwickler hat wiederum die Möglichkeit Anpassungen an dem OAM Modell und somit auch an der API vorzunehmen bevor er aus dem OAM Modell den Programmcode für die API generieren lässt.

Der Ansatz von Scheglmann et al. ermöglicht es aus Ontologien APIs zu generieren. Im Gegensatz zu anderen Ansätzen erlaubt es dieser Ansatz, dass Teile der Ontologie nicht durch die API abgebildet werden, da im Normalfall eine exakte Abbildung der Ontologie in der API nicht den Wünschen eines Entwicklers entspricht. Aber auch dieser Ansatz behandelt die Codegenerierung eher aus der Sicht eines Ontologieentwicklers. Allerdings kommen die Zwischendarstellungen in Form von Modellen Softwareentwicklern entgegen. Durch Anpassungen an dem OAM Modell sollte es möglich sein weitere Operationen in das Modell für den Code aufzunehmen. Es gibt aber keine Möglichkeit das Verhalten von Operationen zu modellieren. Zudem ist keine Möglichkeit zur Nutzung von automatischen Beweisern im Programmcode vorgesehen.

3.3 Vergleich der Konzepte von Ontologien und objektorientierter Konzepte

Es gibt bereits einige Arbeiten, die sich teilweise mit dem Vergleich der Konzepte von Ontologien und objektorientierter Konzepte befassen. In diesem Abschnitt werden einige Ergebnisse der Arbeiten [15], [9], [23] und [31] zusammengefasst aufgeführt.

Sowohl Ontologien als auch objektorientierte Sprachen bieten die Möglichkeit Klassen, deren Properties bzw. Referenzen und Attribute sowie Instanzen bzw. Individuen für die Modellierung zu verwenden. Ebenso erlauben Ontologien und objektorientierte Sprachen das Festlegen von Kardinalitätsbeschrän-

kungen. In der Regel ermöglichen objektorientierte Sprachen es auch Operationen in das Modell aufzunehmen.

Die Konzepte von Ontologien und objektorientierter Sprachen sind zwar ähnlich, haben aber eine unterschiedliche Semantik. Diese Semantikunterschiede erschweren das Definieren von Abbildungen zwischen Ontologien und objektorientierten Sprachen. Die Tabelle 3.1 führt einige der Semantikunterschiede auf.

Objektorientierte Sprachen	Ontologien
Attribute und Referenzen werden lokal in den Klassen definiert und von den Oberklassen geerbt.	Properties werden global definiert und können durch Domain Konstrukte an Klassen gebunden werden.
Eine Instanz vom Typ einer Klasse muss die Einschränkungen für diese Klasse erfüllen.	Jedes Individuum, das die Einschränkungen einer Klasse erfüllt, gehört zu dieser Klasse.
Jede Instanz hat genau eine Klasse als Typ.	Individuen können zu mehreren Klassen gehören.
In der Regel sind Instanzen anonym, sodass nicht von außen auf sie referenziert werden kann.	Alle Instanzen besitzen eindeutige IRIs bzw. URIs, über die auf sie referenziert werden kann.
Die Klassenzugehörigkeit von Instanzen kann zur Laufzeit nicht geändert werden.	Die Klassenzugehörigkeit von Individuen kann sich zur Laufzeit verändern.
Alle Klassen sind zur Zeit der Kompilierung bekannt und können danach nicht geändert werden.	Klassen können zur Laufzeit erzeugt und verändert werden.
Closed World Assumption: Ist es nicht möglich herzuleiten, dass eine Aussage wahr ist, so wird angenommen, dass die Aussage falsch ist.	Open World Assumption: Ist es nicht möglich herzuleiten, dass eine Aussage wahr ist, so kann diese wahr oder falsch sein.

Tabelle 3.1. Unterschiede der Semantik von Ontologien und objektorientierter Sprachen

3.4 Nutzung von Ontologien in Anwendungen

Es gibt einige Arbeiten, die sich damit beschäftigen, wie Ontologien sinnvoll in Anwendungen verwendet werden können. Im Folgenden werden ein paar dieser Arbeiten vorgestellt.

Entwicklung einer Semantic Web Anwendung

Holger Knublauch beschäftigt sich in seiner Arbeit [13] mit der Frage, wie eine Semantic Web [10, 1] Anwendung mithilfe von Ontologien und ontologiegetriebener Softwareentwicklung erstellt werden sollte. Anhand eines Beispiels aus

dem Tourismusbereich stellt Holger Knublauch eine Softwarearchitektur und verschiedene Richtlinien für die Entwicklung einer **Semantic Web** Anwendung vor.

Als Basis für die **Semantic Web** Anwendung sollen verschiedene Kernontologien verwendet werden. Zum Beispiel könnten als Kernontologien für eine Anwendung, die Reiseveranstaltungen sucht, eine Ontologie für die Modellierung von geographischen Orten und eine Ontologie für die Modellierung der unterschiedlichen Reiseveranstaltungen verwendet werden. Die verschiedenen Anbieter können ihre Angebote der Anwendung zugänglich machen, indem sie entsprechende Instanzen für die Ontologie erzeugen. Ebenso können die Anbieter Erweiterungen für die Kernontologien veröffentlichen, um spezielle Sachverhalte zu modellieren. Zum Beispiel könnten sie einen zusätzlichen Veranstaltungstyp als Untertyp eines bereits enthaltenen Veranstaltungstyps festlegen. Diese Informationen werden mithilfe eines **Web Services** [16] eingesammelt.

Die von Holger Knublauch vorgestellte Architektur sieht vor, dass die Funktionalität der Anwendung über einen **Web Service** für Softwareagenten bereitgestellt wird. Holger Knublauch nimmt eine Unterscheidung in zwei Schichten vor: **Semantic Web** Schicht und interne Schicht. Die **Semantic Web** Schicht stellt die Ontologien und Schnittstellen öffentlich zur Verfügung. Die interne Schicht hingegen enthält die Kontrollmechanismen und Mechanismen für das automatische Beweisen und Schlussfolgern auf Ontologien.

Über die Ontologie wird das Verhalten der internen Komponenten gesteuert. Für die Kernontologien enthält die interne Schicht eine entsprechende Programmcoderepräsentation. Diese Repräsentation kann durch ein System für ontologiegetriebene Softwareentwicklung automatisch generiert werden. Für die Bearbeitung der restlichen Ontologieinformationen werden die Parser und automatischen Beweiser der internen Schicht verwendet. Holger Knublauch empfiehlt **Protégé** mit dem **OWL Plugin** [14] für die Entwicklung der Ontologie und die Codegenerierung sowie agile Entwicklungsansätze [3] für die Erstellung der internen Schicht zu verwenden.

Die Arbeit von Holger Knublauch stellt eine Möglichkeit zur sinnvollen Nutzung von Systemen für ontologiegetriebene Softwareentwicklung vor. Diese Möglichkeit ist ebenfalls auf Systeme, die ontologiegetriebene und modellgetriebene Softwareentwicklung kombinieren, überführbar. Somit unterstreicht Holger Knublauch mit seiner Arbeit auch den Nutzen solcher Systeme.

Integration von objektorientierten und ontologischen Repräsentationen

Puleston et al. [30] beschäftigen sich damit, wie Ontologien in Softwarearchitekturen für objektorientierte Anwendungen integriert werden können. Sie betrachten drei verschiedene Ansätze für die Integration einer **OWL** Ontologie in eine **Java** Anwendung: einen direkten Ansatz, einen indirekten Ansatz und einen hybriden Ansatz.

Der direkte Ansatz nutzt die Ontologie als Entwurf für die Programmklassen. Für die Ontologie wird eine entsprechende Coderepräsentation erstellt und es können keine dynamischen Änderungen vorgenommen werden. Der indirekte Ansatz stellt das genaue Gegenteil dar. Die Ontologie wird nicht für die Modellierung der Programmklassen verwendet. Stattdessen greifen die Programmklassen auf eine externe Ontologie zu und passen so ihr Verhalten an. Dieser Ansatz erlaubt das dynamische Anpassen der Ontologie und diese Anpassungen beeinflussen das Verhalten des Programmcodes.

Der dritte Ansatz, auf den sich Puleston et al. konzentrieren, stellt eine Kombination des direkten und des indirekten Ansatzes dar. Bei diesem Ansatz repräsentieren die Programmklassen lediglich einen Teil der Ontologie. Dieser modellierte Teil der Ontologie wird dynamisch durch eine externe Ontologie erweitert. Somit können Vorteile des direkten und des indirekten Ansatzes genutzt werden.

Puleston et al. wendeten den hybriden Ansatz für die Entwicklung einer Anwendung für den medizinischen Bereich an und kamen zu der Schlussfolgerung, dass die Verwendung von Ontologien für solche Anwendung hilfreich ist.

Verknüpfung von EMF Anwendungen mit RDF Datenbeständen

Die Arbeit von Hillairet et al. [9] präsentiert einen Ansatz für die Verknüpfung von EMF Anwendungen mit RDF Datenbeständen. Sie präsentieren ein System, das sowohl eine Möglichkeit zur Instanziierung von EMF Objekten aus RDF Daten als auch eine Möglichkeit zur Serialisierung von EMF Objekten in RDF Daten bietet.

Dieser Ansatz sieht vor, dass das EMF Modell der Anwendung mit einer Ontologie verknüpft wird. Für die Definition dieser Verknüpfung wird eine domänenspezifische Sprache angeboten. Diese Sprache erlaubt es die Elemente eines EMF Modells den Elementen einer bereits existierenden Ontologie zuzuordnen oder zu definieren, wie eine neue Ontologie aus dem EMF Modell hergeleitet wird.

Die mit der domänenspezifischen Sprache definierte Zuordnung zwischen Elementen des Modells und Elementen einer Ontologie wird verwendet, um Transformationen zwischen den EMF Objekten und den Ontologie Individuen zu generieren. Diese generierten Transformationen werden für das Laden der EMF Objekte aus den RDF Daten und das Speichern der EMF Objekte in die RDF Daten verwendet.

Mit diesem Ansatz ist es also möglich Ontologien aus EMF Modellen und EMF Objekten zu generieren bzw. Ontologien mit Individuen zu füllen. Allerdings findet lediglich eine Transformation zwischen EMF Objekten und Ontologie Individuen statt, ohne dass auf die Einhaltung von Restriktionen der Ontologie geachtet wird.

Funktionale Anforderungen

Das Szenario aus Kapitel 2 und die in Kapitel 3 vorgestellten verwandten Arbeiten liefern mehrere Anforderungen für die gewünschte Erweiterung eines verbreiteten Systems für modellgetriebene Softwareentwicklung. In diesem Kapitel werden die funktionalen Anforderungen an das erweiterte System erläutert. Die Pflichtanforderungen sind mit einem Pluszeichen (+) gekennzeichnet, während optionale Anforderungen mit einem Minuszeichen (−) markiert sind. Im Folgenden wird das zu erweiternde System für modellgetriebene Softwareentwicklung als MDE System und das erweiterte System als EOF System bezeichnet.

- R1⁺** Das EOF System bietet dem Entwickler die Möglichkeit mit Annotationen an ein Modell und an dessen Modellinstanzen eine Ontologie zu definieren, sodass das annotierte Modell konform zum Metamodell des ursprünglichen MDE Systems ist. Dabei definieren die Annotationen zusätzliche Ontologierestriktionen für das Modell. Diese Annotationen werden im Abschnitt 5.3 näher beschrieben. Im Folgenden wird ein solches annotiertes Modell als EOF Modell bezeichnet.
- R2⁺** Das EOF System ermöglicht die Generierung einer Ontologie, die durch ein EOF Modell und dessen Modellinstanzen definiert wurde. Die generierte Ontologie bzw. die Ontologiegenerierung kann im generierten Code verwendet werden. Die TBox der Ontologie wird aus dem EOF Modell generiert, wohingegen die ABox aus Instanzen des annotierten Modells generiert wird. Die Ontologiegenerierung wird in Abschnitt 5.5 beschrieben.
- R3[−]** Das EOF System ermöglicht es dem Entwickler zu überprüfen, ob die durch das EOF Modell definierte Ontologie eine OWL DL Ontologie ist. Zum Beispiel wird überprüft, dass keine

Property als transitiv und asymmetrisch deklariert wurde, da ansonsten die OWL DL Restriktionen aus [18] nicht eingehalten wären und die Ontologie somit keine OWL DL Ontologie wäre.

- R4⁻** Das EOF System berechnet aus dem EOF Modell, das vom Entwickler eingegeben wurde, ein entsprechendes EOF Modell, das als Eingabe für den erweiterten Generator dient. Das berechnete Modell erfüllt alle Bedingungen, die der Generator an sein Eingabemodell stellt, damit die korrekte Umsetzung des EOF Modells in Code gewährleistet werden kann. Zum Beispiel wird sichergestellt, dass in dem berechneten Modell eine OWL Thing Klasse als Oberklasse aller anderer Klassen enthalten ist. Im Abschnitt 5.4 befindet sich eine detaillierte Beschreibung zur Berechnung dieses Modells.
- R5⁻** Das EOF System berechnet auf Wunsch aus dem EOF Modell, das vom Entwickler eingegeben wurde, ein entsprechendes EOF Modell, dessen Struktur die Semantik einiger Class-Expression Ausdrücke und Klassenaxiome des ursprünglichen EOF Modells abbildet. Diese Berechnung orientiert sich an der von Kalyanpur et al. [12] vorgestellten Abbildung von OWL Ontologien in Java und wird im Abschnitt 5.4 genauer beschrieben.
- R6⁺** Der erweiterte Generator berücksichtigt die Annotationen des Modells bei der Codegenerierung. Dazu werden die Annotationen während der Codegenerierung eingelesen und der generierte Code wird den Annotationen entsprechend angepasst.
- R7⁺** Der generierte Code gewährleistet für einige Ontologieausdrücke zur Laufzeit die Einhaltung der durch die Ausdrücke definierten Restriktionen. Zum Beispiel soll die Einhaltung von TransitiveObjectProperty, SymmetricObjectProperty und ReflexiveObjectProperty Ausdrücken zur Laufzeit gewährleistet werden. Dafür verhindert der generierte Code die Ausführung von Aktionen, die zu einer Verletzung der zur Laufzeit zu gewährleistenden Ontologierestriktionen führen würde, oder der generierte Code inferiert weitere Aktionen, deren zusätzliche Ausführung die Einhaltung der zur Laufzeit zu gewährleistenden Ontologierestriktionen sicherstellt, und wendet diese weiteren Aktionen entsprechend an. Im Abschnitt 5.6.1 wird die Gewährleistung der Einhaltung von Ontologierestriktionen zur Laufzeit genauer beschrieben.

- R8⁺** Der erweiterte Generator erzeugt im generierten Code zusätzliche Operationen, die mittels Konsistenzprüfungen auf einer generierten Ontologie die Konformität mit den Ontologierestriktionen des EOF Modells überprüfen.
- R9⁺** Der erweiterte Generator generiert auf Wunsch aus dem EOF Modell einen einfachen Editor, der dem Nutzer die Arbeit auf Instanzen des Modells bzw. der Ontologie ermöglicht. Dieser einfache Editor bietet die Möglichkeit Instanzen von Modell-elementen zu erzeugen und deren Attribut- und Referenzwerte auszulesen und abzuändern. Zudem erlaubt der Editor das Speichern erzeugter Modellinstanzen und das Laden gespeicherter Modellinstanzen.
- R10⁺** Der generierte Editor bietet die Möglichkeit über eine Validierungsaktion die Korrektheit des Modells zu überprüfen. Die Validierungsaktion prüft sowohl die Konformität des Modells zum Metamodell als auch die Konsistenz des Modells bezüglich der durch die Annotationen definierten Ontologie.
- R11⁺** Das EOF System bietet die Möglichkeit das Verhalten von Operationen mittels annotierter Anfragen auf der durch das EOF Modell definierten Ontologie zu beschreiben. Innerhalb der annotierten Anfragen kann mit `?self` auf das Objekt verwiesen werden, auf dem die Operation ausgeführt wird.
- R12⁺** Der erweiterte Generator fügt für Operationen, die mit Anfragen annotiert sind, Code zum Ausführen der annotierten Anfragen auf einer generierten Ontologie in den generierten Code ein. Im generierten Code werden vor der Ausführung der Anfrage alle Vorkommen von `?self` in der Anfrage durch die IRI des Objekts, auf dem die Operation ausgeführt wird, ersetzt.
- R13⁺** Der Generator implementiert eine mit einer Anfrage annotierte Operation vollständig, falls der Typ des Rückgabewerts dieser Operation im Modell passend zur Anfrage gewählt wurde und kein Code an die Operation annotiert ist. Der generierte Code für solche Operationen besteht aus Code für die Ausführung der Anfrage auf einer generierten Ontologie, Code zur Umwandlung des Ergebnisses in den Typ des Rückgabewertes und Code zum Setzen des Rückgabewerts. Bei der Umwandlung des Ergebnisses werden die IRIs im Ergebnis durch die zugehörigen Objekte ersetzt. Weitere Informationen hierzu befinden sich in Abschnitt 5.6.3.

Integration von Ontologien und modellgetriebener Softwareentwicklung

In diesem Kapitel wird der Lösungsansatz dieser Arbeit für das Erreichen der Anforderungen aus Kapitel 4 beschrieben. Bei diesem Lösungsansatz soll ein System für modellgetriebene Softwareentwicklung um zusätzliche Funktionalitäten aus der ontologiegetriebenen Softwareentwicklung erweitert werden. Im Folgenden wird das zu erweiternde System als MDE System und das nach diesem Lösungsansatz erweiterte System als EOF System bezeichnet. Analog dazu werden die Modelle für diese Systeme als MDE Modell bzw. EOF Modell bezeichnet.

Zuerst wird der Lösungsansatz im Abschnitt 5.1 grob beschrieben. Anschließend werden im Abschnitt 5.2 verschiedene Anwendungsfälle für den Lösungsansatz beschrieben und den Anforderungen aus Kapitel 4 zugeordnet. In den restlichen Abschnitten des Kapitels werden schließlich die verschiedenen Teile des Lösungsansatzes detailliert beschrieben.

5.1 Lösungsansatz: Eclipse Ontologie Framework (EOF)

Bei dem Lösungsansatz dieser Arbeit soll ein System für modellgetriebene Softwareentwicklungen um zusätzliche Funktionalitäten, die auf Ontologien basieren, erweitert werden. Die zusätzlichen Möglichkeiten sollen die Abbildung der Semantik von Ontologierestriktionen im generierten Code sowie die Modellierung von Operationen mit Anfragen auf Ontologien umfassen. Als zu erweiterndes System sollte möglichst ein weit verbreitetes System gewählt werden, um eine gute Akzeptanz in der Gemeinde der modellgetriebenen Softwareentwicklung zu erreichen.

Als Basis für den Lösungsansatz werden spezielle Annotationen für das Metamodell des ursprünglichen Systems definiert. Diese speziellen Annotationen stellen Ontologieausdrücke dar und erlauben es dem Entwickler Ontologierestriktionen für ein Modell zu definieren. Zusätzlich werden Annotationen definiert, mit denen Anfragen mit Operationen verbunden werden können, um das Verhalten von Operationen mit Anfragen zu modellieren. Mit den

Anfragen kann festgelegt werden, welche Werte die Operation als Ergebnis liefert. Im Abschnitt 5.3 werden die speziellen Annotationen für Modelle und Modellinstanzen des EOF Systems erläutert.

Die neu definierten Annotationen sollen konform zum Metamodell des ursprünglichen Systems sein, sodass jedes korrekte Modell des EOF Systems auch ein zum Metamodell des ursprünglichen Systems konformes Modell ist. Durch diese Forderung wird erreicht, dass ein EOF Modell immer zum ursprünglichen System kompatibel ist. Somit können alle Werkzeuge, die auf dem Metamodell des ursprünglichen Systems basieren, auch für die EOF Modelle verwendet werden. Dadurch kann der Entwickler seine gewohnten Werkzeuge verwenden und muss lediglich die Verwendung der wenigen neuen Elemente des erweiterten Systems erlernen. Dies sollte für eine bessere Akzeptanz des EOF Systems in der Gemeinde des ursprünglichen Systems sorgen.

Die Abbildung 5.1 stellt ein vereinfachtes Aktivitätsdiagramm für den Prozess der Softwareentwicklung von der Erstellung des EOF Modells bis zur Ausführung des generierten Codes dar. Im Folgenden werden die verschiedenen Schritte des Diagramms grob beschrieben.

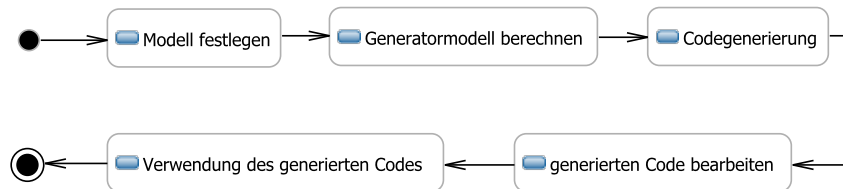


Abb. 5.1. Aktivitätsdiagramm für den Prozess der Softwareentwicklung

Zunächst legt der Entwickler das EOF Modell für die Erstellung der Anwendung fest. In der Regel beinhaltet dieses EOF Modell Ontologieannotationen, aus denen sich weitere Ontologieannotationen, deren Ontologierestriktionen ebenfalls gelten müssen, herleiten lassen. Der Generator muss für die korrekte Behandlung des EOF Modells auch einige herleitbare Annotationen kennen und sowohl auf eingegebene als auch auf herleitbare Annotationen effizient zugreifen können.

Da die herleitbaren Annotationen durchaus auch interessant für den Entwickler sein können, zum Beispiel zur Fehlererkennung, wird das Inferieren der herleitbaren Annotationen in einen Berechnungsschritt vor der Codegenerierung ausgelagert. Dieser Berechnungsschritt erhält als Eingabe das vom Entwickler eingegebene Modell und berechnet daraus ein neues Modell, das als Eingabe für die Codegenerierung dient.

Da dieser Berechnungsschritt ein neues Modell erzeugt und das gesamte eingegebene Modell mit allen Annotationen durchlaufen muss, werden einige andere Berechnungen diesem Schritt hinzugefügt. Zum einen wird die Be-

handlung von einigen `ClassExpression` Ausdrücken und Klassenaxiomen, gemäß Anforderung R5, optional in diesen Berechnungsschritt eingebunden. Zum anderen wird sichergestellt, dass das EOF Modell wohlgeformt ist und somit keine Restriktionen verletzt. Zum Beispiel wird sichergestellt, dass eine Klasse, die OWL `Thing` repräsentiert, im neuen Modell enthalten ist und dass diese Klasse Oberklasse aller anderen Klassen ist. Der Abschnitt 5.4 beschreibt die Berechnung des Eingabemodells für den Generator detaillierter.

Als Vorbedingung für die Codegenerierung wird festgelegt, dass ein Modell, das als Eingabe für den erweiterten Generator verwendet wird, die Gestalt eines Modells, das den in Abschnitt 5.4 beschriebenen Berechnungsschritt durchlaufen hat, haben muss. Der Entwickler kann also auch ein von ihm erstelltes Modell direkt als Eingabemodell für den Generator verwenden. Er muss dann allerdings selbst dafür sorgen, dass dieses Modell alle nötigen Bedingungen erfüllt.

Der erweiterte Generator liest die Ontologieannotationen und Anfrageannotationen während der Codegenerierung ein und passt den generierten Code entsprechend an. Nachdem der Code generiert wurde, fügt der Entwickler den Programmcode ein, der nicht automatisch generiert werden kann. Dazu bietet das EOF System dem Entwickler die Möglichkeiten des MDE Systems zur Bearbeitung des generierten Codes, indem es das MDE System nutzt.

Schließlich kann der generierte Code ausgeführt werden. Der generierte Code verwendet für verschiedene Aufgaben die durch das EOF Modell und dessen Modellinstanzen definierte Ontologie. Das EOF System bietet eine Möglichkeit diese Ontologie zu generieren. Für die Generierung der TBox reicht das EOF Modell aus. Soll zusätzlich die ABox generiert werden, so werden auch die entsprechenden Modellinstanzen benötigt. Um eine Zuordnung zwischen den Objekten der verschiedenen Individuen und den IRLs der Ontologien vornehmen zu können, wird während der Ontologiegenerierung eine entsprechende Abbildung gespeichert. Der Abschnitt 5.5 befasst sich mit der Generierung der Ontologie.

Die Aufgaben, für die der generierte Code eine generierte Ontologie verwendet, sind die Gewährleistung der Einhaltung einiger Ontologierestriktionen zur Laufzeit, die Überprüfung der Einhaltung aller Ontologierestriktionen und die Ausführung von Operationen, die mit Anfragen modelliert wurden. Der Abschnitt 5.6 befasst sich mit der Verwendung von Ontologien zur Erfüllung dieser Aufgaben.

Um die Einhaltung von Ontologierestriktionen zur Laufzeit zu gewährleisten, wird für Aktionen, deren alleinige Ausführung zu einer Verletzung dieser Restriktionen führen würde, entweder deren Ausführung gestoppt oder es werden weitere Aktionen, deren zusätzliche Ausführung die Einhaltung der zur Laufzeit zu gewährleistenden Ontologierestriktionen sicherstellt, inferiert und entsprechend angewendet. Zur Erkennung von zu verhindernden Aktionen und zum Inferieren weiterer Aktionen stellt der generierte Code Anfragen an eine generierte Ontologie.

Um überprüfen zu können, ob alle Ontologierestriktionen zu einem Zeitpunkt eingehalten sind, enthält der generierte Code eine Operation, die eine Konsistenzprüfung auf der für diesen Zeitpunkt generierten Ontologie ausführt.

Der generierte Code für Operationen, die mit Anfragen modelliert wurden, enthält Code der die annotierten Anfragen auf einer generierten Ontologie ausführt. Je nach Art der Anfrage und des Rückgabetyps der Operation wird die Operation vollständig implementiert. Ist dies der Fall fügt der Generator Code zur Umwandlung des Ergebnisses in den Typ des Rückgabewertes und Code zum Setzen des Rückgabewerts in den generierten Code ein. In diesem Fall werden ebenfalls die IRIs des Ergebnisses der Anfrage durch die entsprechenden Objekte ersetzt, indem die Abbildung verwendet wird, die während der Ontologiegenerierung gespeichert wurde.

Die Abbildung 5.2 zeigt ein Komponentendiagramm mit einem Architektorentwurf für ein EOF System. In der Komponente EOF System sind die Komponenten enthalten, die das EOF System zusätzlich erhalten soll. Die anderen Komponenten des Diagramms werden vom EOF System verwendet, um bestimmte Funktionen bereitzustellen.

Die Komponente für die Codegenerierung enthält den erweiterten Generator sowie eine Komponente zur Berechnung eines Generatormodells. Der erweiterte Generator baut auf dem Codegenerator des MDE System auf und generiert zusätzlichen Code für die Ontologierestriktionen und die mit Anfragen modellierten Operationen. Die Komponente zur Berechnung des Generatormodells berechnet aus einem vom Entwickler eingegebenen EOF Modell ein EOF Modell, das alle Anforderungen des erweiterten Generators an Eingabemodelle für die Codegenerierung erfüllt.

Ebenso benötigt ein EOF System eine Komponente zur Generierung einer Ontologie aus einem EOF Modell. Diese Komponente ermöglicht es eine vollständige Ontologie direkt aus den Informationen eines EOF Modells und entsprechenden Modellinstanzen zu gewinnen. Für die Generierung der TBox reichen die Informationen des EOF Modells. Möchte man hingegen eine Ontologie mit ABox, so benötigt man zusätzlich die entsprechenden Modellinstanzen.

Das EOF System enthält eine zusätzliche Komponente, um Transformationen zwischen verschiedenen Ontologieformaten sowie zwischen verschiedenen Anfrageformaten durchführen zu können. Zur Ausführung dieser Transformationen wird auf ein entsprechendes Modelltransformationssystem zurückgegriffen.

Zusätzlich zu den bereits erwähnten Komponenten sollte ein EOF System weitere Komponenten enthalten, die dem Entwickler die Verwendung der zusätzlichen Funktionen des EOF Systems erleichtern. Ein Beispiel dafür sind erweiterte Editoren, die zusätzlich zu den Möglichkeiten der Editoren des MDE Systems die Eingabe der zusätzlichen Informationen eines EOF Modells unterstützen.



Abb. 5.2. Architekturentwurf für ein EOF System

Für die Bereitstellung der Funktionalitäten des MDE Systems nutzt das EOF System das MDE System. Damit die Anfrage auf Ontologien und Konsistenztests für Ontologien vom EOF System durchgeführt werden können, nutzt das EOF System einen externen Beweiser, der entsprechende Funktionalitäten bereitstellt.

5.2 Anwendungsfälle für den Lösungsansatz

In diesem Abschnitt werden die verschiedenen Anwendungsfälle für den Lösungsansatz beschrieben und anschließend den Anforderungen aus Kapitel 4 zugeordnet. Die Abbildung 5.3 enthält ein Anwendungsfalldiagramm mit den Anwendungsfällen für den Lösungsansatz.

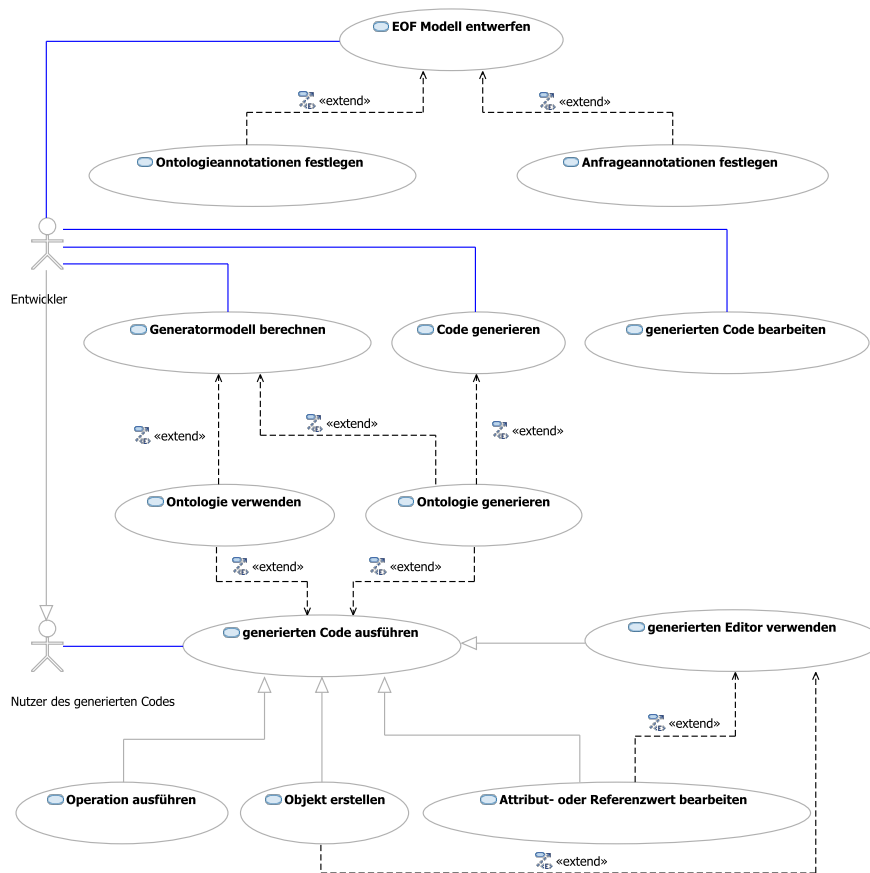


Abb. 5.3. Anwendungsfälle für den Lösungsansatz

1. EOF Modell entwerfen

Beschreibung: Der Entwickler entwirft ein EOF Modell für die zu erstellende Anwendung.

Vorbedingungen: Keine.

Nachbedingung: Es liegt ein vollständiges EOF Modell für die zu erstellende Anwendung vor.

Standardablauf:

1. Der Entwickler erstellt ein neues EOF Modell.
2. Der Entwickler legt die Modellinformationen fest.
3. Entwickler **Ontologieannotationen**.
4. Entwickler **Anfrageannotationen**.
5. Der Entwickler speichert das EOF Modell.

Alternative Abläufe:

1. Der Entwickler lädt ein existierendes EOF Modell.

Erweiterungspunkte:

- **Ontologieannotationen**.
- **Anfrageannotationen**.

2. Ontologieannotationen festlegen

Beschreibung: Der Entwickler legt Ontologierestriktionen für ein EOF Modell fest, indem er Ontologieannotationen in das EOF Modell einfügt.

Vorbedingungen: Das EOF Modell existiert bereits.

Nachbedingung: Das EOF Modell enthält Ontologieannotationen, die Ontologierestriktionen festlegen.

Standardablauf:

1. Der Entwickler öffnet das EOF Modell.
2. Der Entwickler erstellt Ontologieannotationen im EOF Modell.
3. Der Entwickler setzt die Parameterwerte der Ontologieannotationen.
4. Der Entwickler speichert das EOF Modell.

Alternative Abläufe: Keine.

Erweiterungspunkte: Keine.

3. Anfrageannotationen festlegen

Beschreibung: Der Entwickler modelliert das Verhalten von Operationen des EOF Modells, indem er Anfragen entwirft und diese Anfragen über Anfrageannotationen mit den Operationen verbindet.

Vorbedingungen: Das EOF Modell existiert bereits.

Nachbedingung: Das EOF Modell enthält Anfrageannotationen, die auf existierende Anfragen verweisen.

Standardablauf:

1. Der Entwickler öffnet das EOF Modell.
2. Der Entwickler entwirft Anfragen für Operationen des EOF Modells.

3. Der Entwickler erstellt Anfrageannotationen im EOF Modell.
4. Der Entwickler setzt die Werte der Anfrageannotationen.
5. Der Entwickler speichert das EOF Modell.

Alternative Abläufe: Keine.

Erweiterungspunkte: Keine.

4. Generatormodell berechnen

Beschreibung: Der Entwickler lässt aus dem von ihm erstellten EOF Modell ein neues EOF Modell erstellen, das als Eingabe für den Generator dienen soll.

Vorbedingungen: Das übergebene EOF Modell existiert.

Nachbedingung: Es wurde ein neues EOF Modell erzeugt, dass die Erwartungen des Generator an seine Eingabemodelle erfüllt.

Standardablauf:

1. Das System lädt das vom Entwickler übergebene EOF Modell.
2. System TBox.
3. Das System überprüft, ob das übergebene EOF Modell eine valide OWL DL Ontologie definiert (System `Ontologie benötigt`).
4. Das System erstellt ein neues EOF Modell, das eine Kopie des übergebenen EOF Modells ist.
5. Das System leitet weitere Ontologieannotationen für das neue EOF Modell her und passt das neue EOF Modell an.
6. Das neue EOF Modell wird gespeichert und dem Entwickler zur Verfügung gestellt.

Fehlerabläufe:

4. Die Ausführung wird mit einer Fehlermeldung abgebrochen, da das EOF Modell nicht eine valide OWL DL Ontologie definiert.

Alternative Abläufe:

6. Die Semantik einiger `ClassExpressions` und Klassenaxiome wird im neuen EOF Modell abgebildet.
7. Das neue EOF Modell wird gespeichert und dem Entwickler zur Verfügung gestellt.

Erweiterungspunkte:

- TBox.
- `Ontologie benötigt`.

5. Code generieren

Beschreibung: Der Entwickler lässt den Programmcode für ein EOF Modell generieren.

Vorbedingungen: Das übergebene EOF Modell existiert und erfüllt die Erwartungen des Generators an seine Eingabemodelle.

Nachbedingung: Der Programmcode für das EOF Modell wurde generiert.

Standardablauf:

1. Das System lädt das EOF Modell.
2. Der Generator generiert den Programmcode für das EOF Modell.
3. Generator TBox.
4. Der Programmcode wird gespeichert und dem Entwickler zur Verfügung gestellt.

Alternative Abläufe: Keine.

Erweiterungspunkte: TBox.

6. Generierten Code bearbeiten

Beschreibung: Der Entwickler nimmt manuell Anpassungen am generierten Code vor.

Vorbedingungen: Der Programmcode wurde bereits generiert.

Nachbedingung: Keine.

Standardablauf:

1. Der generierte Code wird geöffnet.
2. Der Entwickler bearbeitet den Programmcode.
3. Die Änderungen werden gespeichert.

Alternative Abläufe: Keine.

Erweiterungspunkte: Keine.

7. Ontologie generieren

Beschreibung: Das System generiert die Ontologie, die durch ein EOF Modell bzw. durch ein EOF Modell und dessen Modellinstanzen definiert wird.

Vorbedingungen: Das EOF Modell existiert.

Nachbedingung: Die Ontologie wurde korrekt generiert.

Standardablauf: TBox

1. Das System lädt das EOF Modell.
2. Das System generiert die TBox der von dem EOF Modell definierten Ontologie.
3. Die generierte Ontologie wird gespeichert und bereitgestellt.

Alternative Abläufe: **Ontologie benötigt**

3. Das System lädt die übergebenen Modellinstanzen.
4. Das System generiert die ABox der Ontologie.
5. Die generierte Ontologie wird gespeichert und bereitgestellt.

Erweiterungspunkte: Keine.

8. Ontologie verwenden

Beschreibung: Eine bestehende Ontologie wird verwendet.

Vorbedingungen: Die Ontologie existiert.

Nachbedingung: Keine.

Standardablauf: **Ontologie benötigt**

1. Die Ontologie wird geladen.
2. Die Ontologie wird verwendet.

Alternative Abläufe: Keine.

Erweiterungspunkte: Keine.

9. Generierten Code ausführen

Beschreibung: Der generierte Code wird ausgeführt, um die Anwendung zu verwenden.

Vorbedingungen: Der Code wurde bereits generiert.

Nachbedingung: Der Code wurde ausgeführt.

Standardablauf:

1. Der generierte Code wird geladen.
2. **System Code ausführen.**

Alternative Abläufe: Keine.

Erweiterungspunkte: Keine.

Unterabläufe: **Code ausführen.**

10. Generierten Editor verwenden

Beschreibung: Der Entwickler verwendet den generierten Editor.

Vorbedingungen: Der Code wurde bereits generiert.

Nachbedingung: Der generierte Editor wurde geschlossen.

Unterablauf: **Code ausführen**

1. Der Entwickler öffnet den generierten Editor.
2. Der Entwickler erstellt eine Modellinstanz.
3. Der Entwickler bearbeitet die Modellinstanz (**Objekt, Feature, Editor**).
4. Der Entwickler speichert die Modellinstanz.
5. Der Entwickler schließt den generierten Editor.

Alternative Abläufe:

1. Der Entwickler lädt eine bereits existierende Modellinstanz.

Erweiterungspunkte:

- **Objekt.**
- **Feature.**
- **Editor.**

11. Operation ausführen

Beschreibung: Eine Operation wird aufgerufen.

Vorbedingungen: Der Code wurde bereits generiert.

Nachbedingung: Der Code der Operation wurde ausgeführt.

Unterablauf: **Code ausführen**

1. Der Code der Operation wird ausgeführt (**Ontologie benötigt**).
 Alternative Abläufe: Keine.
 Erweiterungspunkte: **Ontologie benötigt**

12. Objekt erstellen

Beschreibung: Ein Objekt des EOF Modells wird erstellt.
 Vorbedingungen: Der Code wurde bereits generiert.
 Nachbedingung: Ein Objekt wurde erstellt.
 Unterablauf: **Code ausführen**

1. Das Objekt wird erstellt.
2. Der Code des Konstruktors wird ausgeführt (**Ontologie benötigt**).

Alternative Abläufe: Keine.
 Erweiterungspunkte: **Ontologie benötigt**

13. Attribut- oder Referenzwert bearbeiten

Beschreibung: Der Wert einer Referenz oder eines Attributs eines Objekts wird bearbeitet.
 Vorbedingungen: Der Code wurde bereits generiert.
 Nachbedingung: Der Code für das Ändern des Werts wurde ausgeführt.
 Unterablauf: **Code ausführen**

1. Der Code zum Ändern des Werts der Referenz bzw. des Attributs wird ausgeführt (**Ontologie benötigt**).

Alternative Abläufe: Keine.
 Erweiterungspunkte: **Ontologie benötigt**

Zuordnung der Anwendungsfälle

Die verschiedenen Anwendungsfälle werden in der Tabelle 5.1 den einzelnen Anforderungen aus Kapitel 4 zugeordnet.

5.3 Ontologie- und Anfrageannotationen zur Erweiterung der Softwaremodellierung

In diesem Abschnitt wird beschrieben, welche Gestalt ein erweitertes Metamodell haben sollte, damit es als Basis für den vorgestellten Lösungsansatz dienen kann. Als Bedingung für das zu erweiternde Metamodell wird davon ausgegangen, dass das zu erweiternde Metamodell ermöglicht die Informationen eines Klassendiagramms zu modellieren. Sollte dies nicht der Fall sein, so müsste es zusätzlich so erweitert werden, dass dies möglich ist.

Zur Erweiterung des Metamodells werden spezielle Annotationen für das Metamodell definiert, mit denen man eine Ontologie auf einem Modell und

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13
EOF Modell entwerfen	X	X	X	X	X	X	X	X	X	X	X	X	X
Ontologieannotationen festlegen	X	X	X	X	X	X	X	X	X	X	X	X	X
Anfrageannotationen festlegen						X					X	X	X
Generatormodell berechnen			X	X	X								
Code generieren						X						X	X
Generierten Code bearbeiten													
Ontologie generieren		X					X	X				X	X
Ontologie verwenden		X					X	X				X	X
Generierten Code ausführen		X					X	X	X	X	X	X	X
Generierten Editor verwenden		X					X		X	X			
Operation ausführen		X						X			X	X	X
Objekt erstellen		X							X				
Attribut- oder Referenzwert bearbeiten		X					X		X				

Tabelle 5.1. Zuordnung der Anwendungsfälle zu den Anforderungen

dessen Modellinstanzen definieren kann. Diese Annotationen werden in Abschnitt 5.3.1 beschrieben. Außerdem werden Annotationen für die Modellierung von Operationen mit Anfragen hinzugefügt. Der Abschnitt 5.3.4 befasst sich mit diesen Annotationen.

Die neu definierten Annotationen werden als Annotationen des ursprünglichen Metamodells realisiert. Dadurch sind die neu definierten Annotationen konform zum ursprünglichen Metamodell und Modelle für das EOF System sind mit dem ursprünglichen System kompatibel. Diese Kompatibilität ermöglicht es die verschiedenen Werkzeuge, die für das ursprüngliche System existieren und auf dem ursprünglichen Metamodell basieren, auch für Modelle des EOF Systems zu verwenden. Somit kann der Entwickler seine gewohnten Werkzeuge verwenden und muss lediglich die Verwendung der wenigen neuen Elemente des EOF Systems erlernen. Dies sollte für eine bessere Akzeptanz des EOF Systems in der Gemeinde des ursprünglichen Systems sorgen.

Die Wahl der Annotationen zur Erweiterung des Metamodells hat den Vorteil, dass die Kompatibilität zum ursprünglichen System gewährleistet ist. Allerdings hat dieser Ansatz ebenso den Nachteil, dass der Entwickler relativ frei in der Eingabe von Annotationen ist und somit unerwünschte Effekte auftreten können, falls der Entwickler den Editor des ursprünglichen Systems zur Erstellung des Modells verwendet. Zudem wird die Verwendung des Editors des ursprünglichen Systems für die Eingabe der Annotationen ziemlich umständlich sein. Dieser Nachteil kann abgemildert werden, indem dem Nutzer ein entsprechend angepasster Editor bereitgestellt wird, der den Entwickler

bei der Eingabe der Annotationen unterstützt und eventuell die Einhaltung der verschiedenen Einschränkungen für das EOF Modell gewährleistet.

5.3.1 Ontologiedefinitionen

Die OWL Konstrukte werden über das eigentliche Modell oder die speziellen Ontologieannotationen modelliert. Die Annotationen sollten sich an einer OWL Syntax orientieren, um dem Entwickler die Verwendung der Annotationen zu erleichtern. Die verschiedenen Ontologieannotationen sind dadurch gekennzeichnet, dass die Art des Ontologieausdrucks, den sie repräsentieren sollen, über spezielle Werte für die Variablen einer Annotation des ursprünglichen Metamodells kodiert wird.

Die neu definierten Ontologieannotationen werden an die Klassen, Referenzen, Attribute, oder andere Ontologieannotationen geheftet, für die ein Ontologieausdruck definiert werden soll. Außerdem verweisen sie auf die Klassen, Referenzen, Attribute oder andere Ontologieannotationen, die als Parameter an den Ontologieausdruck übergeben werden sollen. Somit ist es möglich alle OWL Konstrukte als solche Annotationen zu beschreiben.

Bei der Modellierung und bei den Ontologieannotationen soll eine Trennung von TBox und ABox bzw. von Modell und Modellinstanz vorgenommen werden. Das Modell und seine Annotationen sollen die Informationen eines Klassendiagramms und der TBox enthalten, wohingegen die Modellinstanzen die Informationen eines Objektdiagramms und der ABox enthalten. Ausgenommen von dieser Trennung sind lediglich die OWL Konstrukte `ObjectOneOf` und `ObjectHasValue` sowie negative `PropertyAssertions`, falls diese unterstützt werden sollen. Diese Konstrukte werden im Modell verwendet und verweisen mit IRIs auf Individuen. In den Modellinstanzen können dann diese IRIs über `SameIndividual` Definitionen mit Objekten bzw. Individuen verbunden werden.

Um die durch das EOF Modell definierte Ontologie sinnvoll nutzen zu können, muss man die Mächtigkeit der Ontologie so einschränken, dass die Entscheidbarkeit der grundlegenden Beweisführungsprobleme gewährleistet ist. Daher soll angenommen werden, dass der Entwickler nur OWL DL Ontologien definiert. Die Einhaltung dieser Einschränkung sowie der weiteren Einschränkungen für das EOF Modell kann automatisch überprüft werden. Dem Entwickler sollte eine entsprechende automatische Überprüfung angeboten werden. Diese Überprüfung sollte in dem in Abschnitt 5.4 beschriebenen Berechnungsschritt durchgeführt werden. Eventuell könnte der Entwickler auch Hilfestellungen zum Beheben des Fehlers erhalten.

Im Folgenden werden zunächst die Ontologieannotationen für das Modell in Abschnitt 5.3.2 und anschließend die Ontologieannotationen für die Modellinstanzen in Abschnitt 5.3.3 beschrieben.

5.3.2 Ontologiedefinitionen im Modell

Die verschiedenen Klassen des Modells stellen ebenfalls OWL Klassen dar. Zum Beispiel würde eine Ontologie für das Modell des Szenarios eine OWL

Klasse `Order` enthalten. Man kann eine Klasse als Repräsentant der OWL Thing Klasse definieren, indem man dieser Klasse den Namen OWL Thing gibt.

Außer den durch das Modell definierten Klassen können lediglich anonyme Klassen definiert werden. Diese anonymen Klassen werden über Ontologieannotationen definiert, die `ClassExpressions` modellieren. Die Annotationen für `ClassExpressions` dürfen als `ClassExpression` Parameter lediglich die Klassen des Modells oder andere `ClassExpression` Annotationen erhalten. Insbesondere soll es nicht möglich sein mit IRIs weitere OWL Klassen zu definieren. Auf die Repräsentation der `ObjectOneOf`, `ObjectHasValue` und `DataHasValue` Konstrukte wird später eingegangen.

Die verschiedenen Referenzen und Attribute einer Klasse definieren OWL `Properties` und somit auch `PropertyExpressions`. Dabei definieren gleichnamige Attribute bzw. Referenzen in verschiedenen Klassen verschiedene OWL `Properties`, falls die gleichnamigen Attribute bzw. Referenzen nicht über die Vererbungshierarchie ineinander überführt werden können. Für die `name` Attribute der `Supplier` Klasse und der `Address` Klasse des Szenarios enthält die Ontologie also zwei verschiedene `Properties`.

Attribute definieren OWL `DataProperties` wohingegen Referenzen OWL `ObjectProperties` definieren. Zum Beispiel würde eine Ontologie für das Modell des Szenarios für das Attribut `country` eine entsprechende `DataProperty` und für die Referenz `previousOrders` eine entsprechende `ObjectProperty` enthalten.

Wie bei Klassen ist es nicht möglich mit IRIs weitere OWL `Properties` zu definieren. Es ist lediglich möglich über eine Ontologieannotation, die das `Inverse` Konstrukt modelliert, weitere `ObjectPropertyExpressions` zu definieren. Diese Annotation erhält als Parameter eine Referenz des Modells. Lediglich die `PropertyExpressions`, die wie in diesem Abschnitt beschrieben definiert wurden, dürfen als Parameter für andere Ontologiekonstrukte verwendet werden.

Die verschiedenen Datentypen des Modells definieren die Datentypen der Ontologie. Wiederum ist es nicht möglich mit IRIs weitere Datentypen zu definieren. Zur Abbildung der `DataRanges` der Ontologie existieren für die anderen OWL Konstrukte zur Bildung von `DataRanges` jeweils entsprechende Annotationen. Diese Annotationen dürfen als Parameter lediglich andere `DataRange` Annotationen oder Datentypen erhalten. Lediglich die so definierten `DataRanges` dürfen als `DataRange` Parameter für andere Ontologiekonstrukte verwendet werden. Die Datentypdefinitionen einer Ontologie werden durch Datentypen des Modells mit `DataRange` Annotationen repräsentiert.

Die `DataOneOf` und `ObjectOneOf` Konstrukte werden durch Enumerationen im Modell repräsentiert. Für `DataOneOf` Konstrukte stellen die Literale der Enumeration Repräsentanten für die Literale der Ontologie dar oder es können mit Annotationen Literale der Ontologie an die Enumerationsliterals gebunden werden. Zum Beispiel könnte man eine `DataOneOf` Annotation an die Enumeration `OrderStatus` des Modells für das Szenario hängen und Annotationen mit Stringrepräsentanten für Ontologie Literale mit den Literalen

der Enumeration, wie zum Beispiel `Pending`, verbinden. Somit hätte man ein entsprechendes `DataOneOf` Konstrukt für die Ontologie definiert.

Für `ObjectOneOf` Konstrukte stellen die Literale der Enumeration IRIs für Individuen dar oder die Literale der Enumeration können mit IRIs annotiert werden. Für die `DataHasValue` und `ObjectHasValue` Konstrukte werden entsprechende Ontologieannotationen definiert. Diese Annotationen müssen die Kodierung des Literals bzw. der IRI in einer Variablen der Annotation erlauben. Um die `ObjectOneOf` und `ObjectHasValue` Konstrukte sinnvoll zu nutzen, sollten die von ihnen verwendeten IRIs in den Modellinstanzen über `SameIndividual` Definitionen mit Objekten bzw. Individuen verbunden werden.

Für die verschiedenen Klassenaxiome, `ObjectProperty` Axiome, `DataProperty` Axiome sowie für `HasKey` Axiome werden entsprechende Annotationen definiert. Diese dürfen als `ClassExpression`, `PropertyExpression` und `DataRange` Parameter, entsprechende `ClassExpressions`, `PropertyExpressions` und `DataRanges`, die wie oben beschrieben definiert wurden, erhalten. Zum Beispiel würde in das Modell für das Szenario eine `TransitiveObjectProperty` Annotation, die als Parameter einen Verweis auf die `previousOrders` `ObjectProperty` erhält, eingefügt.

Zudem definieren die Subklassenbeziehungen des Modells auch Subklassenbeziehungen der Ontologie. Für das Modell des Szenarios würde die Ontologie also eine Subklassenbeziehung zwischen `USAddress` und `Address` enthalten. `Annotation` Axiome können analog zu den anderen Axiomen unterstützt werden, falls dies gewünscht ist. Für `Declaration` Axiome hingegen werden keine entsprechenden Annotationen definiert. Die Deklarationen werden lediglich implizit durch das Modell vorgenommen. Einzige Ausnahme hiervon sind `AnnotationProperties`. Falls diese unterstützt werden sollen, werden sie von einer entsprechenden Annotation deklariert.

Falls gewünscht könnte man zusätzlich zu den bisher erwähnten Ontologie Definitionen festlegen, dass weitere Ontologie Konstrukte aus dem Modell hergeleitet werden. Es wäre möglich `Domains` und `Ranges` der `Properties` aus dem Modell oder auch Restriktionen aus den Kardinalitäten des Modells herzuleiten. So könnten `ClassExpressions` für Kardinalitätsbeschränkungen sowie `FunctionalProperties` und `InverseFunctionalObjectProperties` hergeleitet werden. Ebenso könnte teilweise das Modell dazu verwendet werden um `InverseObjectProperty` Restriktionen herzuleiten.

5.3.3 Ontologiedefinitionen in den Modellinstanzen

Die `Assertion` Axiome werden von den Modellinstanzen definiert. Ausgenommen hiervon sind, wie bereits erwähnt, die negativen `PropertyAssertions`. Falls diese Konstrukte unterstützt werden, werden die entsprechenden Annotationen im Modell an den Repräsentanten des `PropertyExpressions` geheftet. Zudem werden die Parameter IRIs in Variablen der Annotation kodiert. Um die negativen `PropertyAssertions` sinnvoll zu nutzen, sollten die von ihnen verwendeten

IRIs in den Modellinstanzen über `SameIndividual` Definitionen mit Objekten bzw. Individuen verbunden werden.

`PropertyAssertions` werden lediglich durch das Setzen der entsprechenden Werte für die Attribute bzw. Referenzen modelliert. Ebenso werden `ClassAssertions` nur aus der Zugehörigkeit eines Objektes zu einer Klasse hergeleitet und können nicht auf andere Weise definiert werden. Nehmen wir zum Beispiel an, dass wir eine Instanz des Modells für das Szenario haben und diese Instanz ein `Supplier` Objekt hat, dessen `name` Attribut auf den Wert `s1` gesetzt wurde. Dann würde die Ontologie für dieses Objekt eine `ClassAssertion`, die das Objekt der `Supplier` Klasse zuordnet, und eine `PropertyAssertion`, die den Wert der `name` `DataProperty` auf `s1` festlegt, enthalten.

Des Weiteren soll davon ausgegangen werden, dass eine Instanz nicht zu zwei Klassen gehören kann, falls nicht eine dieser Klassen Unterklasse der anderen Klasse ist. Möchte man diese Einschränkung nicht vornehmen, so könnte man jeweils ein Objekt für die beiden Klassen erzeugen und diese mit einer `SameIndividuals` Annotation verbinden. Allerdings sollte man daran denken, dass Referenzen und Attribute, die in beiden Klassen den gleichen Namen haben, auch für diese Objekte verschiedene OWL Properties bezeichnen und dass man zum Abfragen und zum Setzen der `Property` Werte jeweils das passende Objekt verwenden muss.

`SameIndividual` und `DifferentIndividual` Konstrukte werden mittels entsprechender Annotation definiert. Diese Annotationen erhalten als Parameter Objekte bzw. Instanzen eines Modellelements. Sollten im Modell Annotationen für `ObjectOneOf`, `ObjectHasValue` oder negative `PropertyAssertions` verwendet worden sein, so dürfen die `SameIndividual` Annotationen zusätzlich die im Modell definierten IRIs als Parameter erhalten. Diese werden dann in Variablen der `SameIndividual` Annotation kodiert.

5.3.4 Anfrageannotationen für Modelle

Für die Modellierung von Operationen mit Anfragen werden Annotationen zum Verbinden von Operationen und Anfragen bereitgestellt. Diese Annotationen sind dadurch gekennzeichnet, dass der Pfad der Anfrage in Variablen der Annotation kodiert ist. Abhängig von dem zu erweiternden MDE System und der gewünschten Implementierung kann es zudem sinnvoll sein weitere Informationen in diesen Annotationen zu speichern. Zum Beispiel könnte man auf eine Modellrepräsentation der Anfrage verweisen, falls eine solche Repräsentation existiert. Dadurch könnte dem Entwickler auf Wunsch das Modell für eine annotierte Anfrage angezeigt werden.

5.4 Berechnung des Generator-Eingabemodells

Die Wahl der Annotationen zur Erweiterung des Metamodells hat den Nachteil, dass der Entwickler in der Regel relativ frei in der Eingabe von Annotationen ist und somit die Einhaltung von Bedingungen, die die Annotationen

erfüllen sollen, schwierig zu überwachen ist. Um effizient arbeiten zu können, muss sich der Generator aber darauf verlassen, dass alle diese Bedingungen in den Modellen, die dem Generator übergeben werden, eingehalten wurden.

Zudem lassen sich in der Regel aus den Ontologieannotationen eines vom Entwickler eingegebenen EOF Modells weitere Ontologieannotationen, deren Ontologierestriktionen ebenfalls gelten müssen, herleiten. Der Generator muss für die korrekte Behandlung des EOF Modells auch einige herleitbare Annotationen kennen und sowohl auf eingegebene als auch auf herleitbare Annotationen effizient zugreifen können. Die herleitbaren Annotationen können durchaus auch interessant für den Entwickler sein. Zum Beispiel können diese Annotationen ihn bei der Fehlersuche unterstützen.

Aus diesen Gründen sollte dem Entwickler ein Berechnungsschritt angeboten werden, der aus dem vom Entwickler eingegebenen EOF Modell ein neues EOF Modell erzeugt, das als Eingabe für die Codegenerierung dient. Der Generator wird so ausgelegt, dass er davon ausgeht, dass alle an ihn übergebenen Modelle Resultat dieses Berechnungsschritts sind. Ist das bei einem Modell nicht der Fall, so muss der Entwickler selbst dafür sorgen, dass dieses Modell alle nötigen Bedingungen erfüllt.

Da dieser Berechnungsschritt ein neues Modell erzeugt und das gesamte eingegebene Modell mit allen Annotationen durchlaufen muss, wird die Behandlung von einigen `ClassExpressions` und Klassenaxiomen, gemäß Anforderung R5, optional in diesen Berechnungsschritt eingebunden.

Zuerst wird bei der Berechnung des Generator-Eingabemodells überprüft, ob die von dem EOF Modell definierte Ontologie eine valide OWL DL Ontologie ist und somit keine OWL DL Einschränkung verletzt. Sollte dies nicht der Fall sein, so wird die Berechnung mit einer entsprechenden Fehlermeldung abgebrochen. Anderenfalls wird eine Kopie des Modells erstellt. Dieses kopierte EOF Modell wird mit den folgenden Schritten angepasst und dem Entwickler bereitgestellt.

Der erste Schritt besteht aus der Herleitung weiterer Ontologieannotationen und der Anpassung des Modells. Dieser Schritt sorgt dafür, dass alle Erwartungen des Generators an sein Eingabemodell erfüllt sind. Der Unterabschnitt 5.4.1 befasst sich mit diesem Schritt. Als zweiter Schritt wird je nach Wunsch des Entwicklers die Behandlung von einigen `ClassExpressions` und Klassenaxiomen, gemäß Anforderung R5, vorgenommen. Dieser Schritt wird im Unterabschnitt 5.4.2 beschrieben.

5.4.1 Herleitung weiterer Ontologieannotationen und Modellanpassungen

In dem hier beschriebenen Schritt soll das kopierte Modell mit weiteren hergeleiteten Ontologieannotationen versehen und so angepasst werden, dass die Erwartungen des Generators an ein Eingabemodell erfüllt sind. Dafür werden zunächst iterativ weitere Ontologieannotationen hergeleitet. Dabei sollten zumindest die herleitbaren Ontologieannotationen, die eine Ontologierestriktion

definieren, deren Einhaltung zur Laufzeit vom generierten Code gewährleistet werden soll, in das Modell eingefügt werden.

Angenommen das Modell hat zwei Referenzen `r1`, `r2`, an die eine `InverseObjectProperties` Annotation geheftet wurde, und die Referenz `r1` wurde als `TransitiveObjectProperty` deklariert. Dann kann daraus hergeleitet werden, dass die Referenz `r2` ebenfalls transitiv ist und es kann eine entsprechende Annotation hinzugefügt werden.

Zudem wird eine Mustersuche nach Ontologieausdrücken, die sich durch einen einfacheren äquivalenten Ontologieausdruck darstellen lassen, durchgeführt. Die entsprechenden äquivalenten Ontologieannotationen werden in das EOF Modell eingefügt und die komplexeren Ontologieausdrücke werden entfernt.

Ein Beispiel für ein solches Muster, nach dem gesucht werden sollte, sind zirkuläre `SubClassOf` Axiome. Diese Axiome können durch ein `EquivalentClasses` Axiom ersetzt werden. Ebenso könnte ein Axiom der Form `SubObjectPropertyOf(ObjectPropertyChain(OPE OPE) OPE)` durch ein `TransitiveObjectProperty` Axiom ersetzt werden.

Anschließend werden einige Anpassungen am neuen EOF Modell vorgenommen. Sollte das EOF Modell keine Klasse haben, die die OWL `Thing` Klasse repräsentiert, so wird eine solche Klasse hinzugefügt. Zudem wird die Klasse des Modells, die OWL `Thing` repräsentiert, als Oberklasse aller Klassen, die im EOF Modell noch keine Oberklasse haben, definiert. Je nach Implementierung des Generators werden weitere Anpassungen am EOF Modell vorgenommen.

5.4.2 Behandlung von `ClassExpressions` und Klassenaxiomen

In diesem Abschnitt wird die Behandlung von `ClassExpressions` und Klassenaxiomen, gemäß Anforderung R5, beschrieben. Diese Behandlung orientiert sich an der von Kalyanpur et al. [12] vorgestellten Abbildung von OWL Ontologien in Java.

Die Semantik einiger `ClassExpression` Ausdrücke und Klassenaxiome des EOF Modells soll in der Struktur des Generator-Eingabemodells abgebildet werden, sodass keine Objekte erzeugt werden können, die diese Restriktionen verletzen. Damit dies funktioniert muss allerdings gefordert werden, dass wenn zwei Objekte als `SameIndividuals` deklariert sind, der Typ eines Objekts ein Untertyp des anderen Objekts sein muss. Dies hat zur Folge, dass kein Objekt als Instanz zweier nicht verbundener Klassen deklariert werden kann, da `ClassAssertions` nur implizit über die Klassenzugehörigkeit eines Objekts definiert werden können.

Für die Abbildung der Semantik werden iterativ aus den `ClassExpressions` und Klassenaxiomen neue Klassen bzw. neue Interfaces für das EOF Modell hergeleitet und in das EOF Modell eingefügt. Dieser iterative Prozess betrachtet die `ObjectIntersectionOf`, und `ObjectUnionOf` `ClassExpressions` sowie die `SubClassOf`, `EquivalentClasses`, `DisjointClasses` und `DisjointUnion` Klassenaxiome.

Bei einem Schritt des iterativen Prozesses werden lediglich diejenigen `ClassExpressions` und diejenigen Klassenaxiome bearbeitet, die als Parameter nur `ClassExpressions` enthalten, die von einer Klasse oder einem Interface des EOF Modells repräsentiert werden. Für diese `ClassExpressions` und Klassenaxiome werden entsprechende Klassen bzw. Interfaces in das EOF Modell eingefügt.

Für `SubClassOf` Axiome werden entsprechende Subklassenbeziehungen zwischen den entsprechenden Klassen bzw. Interfaces des EOF Modells eingefügt. Für `EquivalentClasses` Axiome werden die entsprechenden Klassen des EOF Modells zu einer einzigen Klasse verschmolzen. Sollten die Parameter der Axiome sowohl Klassen als auch Interfaces enthalten, so wird für die Klassen, die die `ClassExpressions` repräsentieren, jeweils ein neues Interface im Modell erstellt und die Klassen erben von dem entsprechenden Interface. Zudem wird die Repräsentation der `ClassExpressions` von den Klassen auf die entsprechenden Interfaces verlagert. Schließlich werden die oben beschriebenen Schritte zur Umsetzung der Axiome auf den Interfaces durchgeführt.

Für die Erstellung eines Repräsentanten für einen `ObjectIntersectionOf` Ausdruck, sind mehrere Schritte nötig. Angenommen es soll ein Repräsentant A für einen `ObjectIntersectionOf(C1, \dots, CN)` erstellt werden, dann müssen folgende Schritte durchgeführt werden:

- Für die Klassen des EOF Modells, die die `ClassExpressions` $C1, \dots, CN$ repräsentieren, wird jeweils ein neues Interface im Modell erstellt und die Klassen erben von dem entsprechenden Interface.
- Die Repräsentation der `ClassExpressions` wird von den Klassen auf die entsprechenden Interfaces verlagert.
- Als Repräsentant wird ein Interface A erstellt, das von den Interfaces für $C1, \dots, CN$ erbt.
- Es wird dafür gesorgt, dass alle Klassen und Interfaces, die von mehr als einem der neuen Interfaces erben auch von dem Interface A erben.

Die Erstellung eines Repräsentanten für einen `ObjectUnionOf` Ausdruck kann einfacher erreicht werden. Angenommen es soll ein Repräsentant A für einen `ObjectUnionOf(C1, \dots, CN)` erstellt werden. Dann wird als Repräsentant ein Interface A erstellt und festgelegt, dass die Repräsentanten von $C1, \dots, CN$ von diesem Interface erben.

Für die Repräsentation von `DisjointClasses` Axiomen wird ein Blockierungsmechanismus verwendet. Dieser Mechanismus nutzt die Einschränkung für das Überladen von Operationen aus. Diese Einschränkung besagt, dass eine Klasse bzw. ein Interface nicht zwei Operationen haben darf, deren Signatur sich lediglich durch einen unterschiedlichen Rückgabebetyp unterscheidet. Da kein Objekt als Instanz zweier nicht verbundener Klassen deklariert werden kann, reicht dieser Blockiermechanismus aus, um `DisjointClasses` Axiome zu repräsentieren.

Angenommen ein `DisjointClasses(C1, \dots, CN)` Axiom soll in der Struktur des EOF Modells repräsentiert werden. Dann wird in die Klassen bzw.

Interfaces für C_1, \dots, C_N jeweils eine Operation eingefügt. Die eingefügten Operationen haben alle den selben Namen und erwarten keine Parameter. Die Rückgabetypen der eingefügten Operationen sind die Typen der Klasse bzw. des Interfaces, in die die jeweilige Operation eingefügt wurde. Also würde die Operation einer Klasse C_i als Rückgabewert ein Objekt dieser Klasse erwarten.

Die Repräsentation eines `DisjointUnion` Axioms wird durch die Kombination der Schritte für `DisjointClasses` Axiome und `ObjectUnionOf` Ausdrücke erreicht. Angenommen ein `DisjointUnion(C, C_1, \dots, C_N)` soll in der Struktur des EOF Modells repräsentiert werden. Dann wird festgelegt, dass die Repräsentanten von C_1, \dots, C_N vom Repräsentanten von C erben und es werden Blockieroperationen in die Repräsentanten von C_1, \dots, C_N eingefügt.

5.5 Ontologiegenerierung

Modelle und Modellinstanzen mit Annotationen, wie in Abschnitt 5.3 beschrieben, definieren eine Ontologie. In diesem Abschnitt wird beschrieben, wie die so definierte Ontologie aus einem EOF Modell und dessen Modellinstanzen hergeleitet wird. Es soll davon ausgegangen werden, dass die durch das Modell definierte Ontologie eine konsistente OWL DL Ontologie darstellt. Eine entsprechende Überprüfung könnte in dem in Abschnitt 5.4 vorgestellten Berechnungsschritt durchgeführt werden.

Im vorherigen Abschnitt wurde bereits erläutert, dass das EOF Modell bis auf wenige Ausnahmen nur die Informationen der TBox einer Ontologie enthält und dass die Modellinstanzen nur die Informationen für die ABox einer Ontologie enthalten. Daher ist es nicht möglich nur aus den Modellinstanzen eine Ontologie zu generieren. Man benötigt zumindest einen Verweis auf das EOF Modell. Dieser Verweis muss es zudem ermöglichen den Inhalt des gesamten Modells einzulesen. Möchte man hingegen nur aus dem EOF Modell eine Ontologie generieren, so ist dies grundsätzlich möglich. Allerdings enthält eine so generierte Ontologie dann keine ABox Informationen.

Die Präfixdeklarationen der generierten Ontologie bestehen aus den Standard-Präfixdeklarationen und Präfixdeklarationen für die verschiedenen Pakete im Modell. Der Präfixname und die zugehörige IRI einer Präfixdeklaration für ein Paket des Modells werden aus dem Namen und den Variablenwerten des Pakets und seiner Oberpakete hergeleitet. Falls es gewünscht wird, könnte man dem Entwickler auch erlauben zusätzliche Präfixdeklarationen zu definieren oder die Präfixdeklarationen für Pakete teilweise selbst zu bestimmen. Dann müsste allerdings überprüft werden, dass die angegebenen Präfixdeklarationen der OWL Spezifikation entsprechen und dass keine Namenskollisionen auftreten.

Die IRIs der generierten Ontologie werden ebenfalls aus dem Modell und den Modellinstanzen hergeleitet. Für Klassen und Datentypen setzt sich die

IRI aus dem Präfix ihres Pakets und ihrem Namen zusammen. Für die IRI einer Property werden ihr Name und die IRI der obersten Klasse, die im Modell die entsprechende Referenz bzw. das entsprechende Attribut enthält, verwendet. Die IRIs der Individuen werden aus den Werten des zugehörigen Objekts und der IRI der Klasse dieses Objekts zusammengesetzt. Analog zu den Präfixdefinitionen wäre es auch möglich dem Entwickler eine Möglichkeit zur Wahl der IRIs anzubieten und die Eingaben zu überprüfen.

Angenommen die Klassen des Modells für das Szenario lägen alle in einem Paket mit Präfix `po`, dann würde die hergeleitete IRI für die `Order` Klasse `po:Order` lauten. Die zum `name` Attribut einer `USAdresse` gehörende `DataProperty` würde als IRI `po:Address.name` erhalten.

Für die im Modell enthaltenen Klassen und Datentypen werden Klassen- und Datentypdeklarationen in der generierten Ontologie erstellt. Ebenso werden `DataProperty` Deklarationen für Attribute und `ObjectProperty` Deklarationen für Referenzen angelegt. Bei dem Erstellen der `Property` Deklarationen werden nur Attribute und Referenzen behandelt, die nicht von einer Oberklasse geerbt wurden. Für die `name` Attribute der `Address`, `USAddress` und `GlobalAddress` Klassen würde also nur eine `Property` Deklaration erzeugt. Für jedes Objekt der Modellinstanzen wird eine `Individual` Deklaration in die generierte Ontologie eingefügt. Die IRIs für die Deklarationen werden jeweils, wie oben beschrieben, aus dem EOF Modell und den Modellinstanzen hergeleitet.

Zudem werden für die Subklassenbeziehungen des Modells entsprechende `SubClassOf` Axiome in die Ontologie eingefügt. Falls es gewünscht wird, könnten ebenfalls weitere Ontologieausdrücke aus dem Modell hergeleitet werden. Aus den im Modell angegebenen Kardinalitäten für die Attribute bzw. Referenzen könnten `Cardinality ClassExpressions` sowie teilweise `FunctionalProperties` und `InverseFunctionalObjectProperties` hergeleitet werden. Außerdem könnte man `Domains` und `Ranges` der `Properties` aus den Definitionen der Referenzen bzw. der Attribute im Modell gewinnen.

Für die verschiedenen Ontologieannotationen in dem Modell werden entsprechende Ontologieausdrücke in der generierten Ontologie erzeugt. Sollten bei dieser Umwandlung Literale für Objekte benötigt werden, so werden die Literale durch Kombination der Stringrepräsentation des Objekts und der Stringrepräsentation des entsprechenden Datentyps gewonnen. Auf diese Weise können die meisten Axiome der Ontologie erzeugt werden. Ausgenommen hiervon sind die Axiome, die implizit durch die Objekte und deren Werte definiert werden.

Diese Axiome werden erzeugt, indem die Modellinstanzen durchlaufen werden und für jedes Objekt die entsprechenden `Assertion` Axiome hinzugefügt werden. Für jedes Objekt und jede diesem Objekt zugehörige Klasse wird ein entsprechendes `ClassAssertion` Axiom erzeugt. Außerdem werden die Attribute und Referenzen jedes Objekts durchlaufen und entsprechende `DataProperty` Axiome bzw. `ObjectProperty` Axiome erzeugt. Dabei werden mehrere `Assertions` für eine `Property` erzeugt, falls deren zugehörige Referenz bzw. deren zugehöriges Attribut eine Kardinalität größer eins und mehr als einen Wert hat.

In diesem Fall wird eine `Assertion` pro Wert generiert. Sollte also ein `Order` Objekt über die `previousOrders` Referenz drei `Order` Objekte zugewiesen bekommen, so würden auch drei entsprechende `Assertions` in der Ontologie erzeugt.

Mit den in den obigen Absätzen beschriebenen Umwandlungen ist es möglich eine vollständige Ontologie aus einem EOF Modell und den zugehörigen Modellinstanzen zu generieren. Möchte man eine so generierte Ontologie sinnvoll im Programmcode verwenden, so ist es zudem notwendig, dass man nach der Generierung der Ontologie über die IRIs der Ontologie die zugehörigen Entitäten wiederfinden kann. Dazu wird bei der Ontologiegenerierung gespeichert, welche Entitäten zu den einzelnen in der Ontologie verwendeten IRIs gehören. Außerdem benötigt man eine Möglichkeit aus den Literalen der generierten Ontologie die passenden Datenwerte auszulesen.

Hat man eine Abbildung die diese beiden Anforderungen erfüllt, dann kann man Anfragen an die Ontologie stellen und die gelieferten Ergebnisse mit dieser Abbildung so transformieren, dass die transformierten Ergebnisse sinnvoll im Programmcode verwendet werden können. Dies liefert den Vorteil, dass einige zu programmierende Aufgaben über entsprechende Anfragen und eine Transformation des Ergebnisses implementiert werden können. Somit ist es dann auch möglich das Verhalten von Operationen über Anfragen auf der generierten Ontologie zu modellieren.

5.6 Verwendung der Ontologie im generierten Code

Im vorherigen Abschnitt wurde beschrieben, wie aus dem EOF Modell und den zugehörigen Modellinstanzen eine Ontologie generiert werden kann. In diesem Abschnitt wird erläutert, wie die so generierte Ontologie sinnvoll genutzt werden kann. Als Voraussetzung für die Nutzung der Ontologie wird gefordert, dass es sich um eine OWL DL Ontologie handelt, sodass die Entscheidbarkeit der grundlegenden Beweisführungsprobleme gewährleistet ist und somit Anfragen an die Ontologie immer ein Ergebnis liefern.

Für die Behandlung einiger `ClassExpressions` und Klassenaxiome kann ein optionaler Berechnungsschritt vor der Codegenerierung genutzt werden. Dieser Berechnungsschritt erzeugt aus dem vom Entwickler eingegebenen Modell ein neues Modell, in dem die Einhaltung einiger `ClassExpression` Restriktionen und Klassenaxiome durch eine Veränderung der Struktur des Modells eingehalten wird, sodass es unmöglich wird Objekte zu erzeugen, die gegen diese Restriktionen verstoßen. Dieser Berechnungsschritt wurde bereits in Abschnitt 5.4 behandelt.

Zudem soll die Ontologie Einfluss auf die Codegenerierung und den generierten Code haben. Die Einhaltung einiger Ontologierestriktionen soll zur Laufzeit vom generierten Code gewährleistet werden. Für diese Gewährleistung verwendet der generierte Code eine generierte Ontologie. Der Unterabschnitt 5.6.1 legt fest, für welche Ontologieausdrücke die Einhaltung der durch

diese Ausdrücke definierten Restriktionen zur Laufzeit vom generierten Code gewährleistet werden soll, und erläutert, wie diese Gewährleistung erreicht wird.

Es gibt allerdings Ontologieausdrücke, die nicht vom Berechnungsschritt vor der Codegenerierung behandelt werden und deren Einhaltung nicht zur Laufzeit durch den generierten Code gewährleistet wird. Wie der Entwickler diese Ausdrücke dennoch sinnvoll nutzen kann, wird im Unterabschnitt 5.6.2 erläutert.

Zudem verwendet der generierte Code die Ontologie bei der Ausführung von Operationen, die mit Anfragen annotiert wurden. Der Unterabschnitt 5.6.3 befasst sich damit, wie das Verhalten solcher Operationen mit den Anfragen modelliert wird und welche Gestalt der generierte Code haben soll.

Bei der Verwendung der generierten Ontologie gibt es allerdings ein Problem. Zur Laufzeit ändern sich die vorhandenen Objekte und deren Werte kontinuierlich. Das hat auch Auswirkungen auf die von dem EOF Modell und den Modellinstanzen definierte Ontologie. Deren ABox verändert sich bei jedem Erzeugen oder Löschen eines Objekts sowie bei jeder Wertveränderung der im Modell enthaltenen Referenzen und Attribute. Im Folgenden sollen zwei Möglichkeiten zum Umgang mit diesem Problem erläutert werden.

1. Bei der Codegenerierung wird die Ontologie soweit wie möglich generiert und so gespeichert, dass vom Code aus immer darauf zugegriffen werden kann. Zudem wird bei jeder Änderung, die die Ontologie betrifft, die gespeicherte Ontologie aktualisiert. Wird die generierte Ontologie benötigt, so wird die gespeicherte Ontologie verwendet.
2. Die generierte Ontologie wird gar nicht gespeichert oder es wird lediglich der Teil der generierten Ontologie gespeichert, der sich nicht zur Laufzeit ändert. Wird die generierte Ontologie benötigt, so wird die gesamte Ontologie bzw. der noch fehlende Teil der Ontologie neu generiert und die so erhaltene Ontologie verwendet.

Beide Ansätze haben Vor- und Nachteile, sodass keiner von beiden generell zu bevorzugen ist. Beim ersten Ansatz kann der Speicheraufwand für große Datenbestände sehr hoch werden, da die komplette Ontologie gespeichert wird. Ebenso muss bei jeder Erzeugung und jedem Löschen eines Objekts sowie bei jeder Wertveränderung die Ontologie aktualisiert werden, was zusätzlichen Rechenaufwand bedeutet. Allerdings ist der Rechenaufwand immer dann wesentlich geringer als beim zweiten Ansatz, wenn die generierte Ontologie verwendet wird.

Beim zweiten Ansatz ist der Speicheraufwand meistens wesentlich geringer, da im Allgemeinen nur ein relativ kleiner Teil der Ontologie gespeichert wird. Ebenso spart der zweite Ansatz den Rechenaufwand für Aktualisierungen der Ontologie. Der große Nachteil dieses Ansatzes ist allerdings, dass der Rechenaufwand für den Zugriff auf die generierte Ontologie wesentlich höher ist, da zunächst der fehlende Teil neu generiert werden muss. Zudem kann der zweite Ansatz nur dann angewendet werden, wenn man von einem beliebigen

Objekt der Modellinstanz den kompletten Inhalt der Modellinstanz einlesen kann, um daraus den fehlenden Teil der Ontologie zu generieren.

Welcher Ansatz besser ist hängt also stark davon ab, wie der generierte Code hauptsächlich genutzt werden soll und welche Einschränkungen gegeben sind. Daher sollte die Entscheidung, welcher Ansatz verwendet wird, möglichst dem Entwickler überlassen werden. Der erste Ansatz sollte dann bevorzugt werden, wenn genug Speicherplatz für die Ontologie zur Verfügung steht und relativ häufig auf die generierte Ontologie zugegriffen wird, sodass der zusätzliche Rechenaufwand für die Aktualisierungen gerechtfertigt ist.

Der zweite Ansatz hingegen sollte genutzt werden, wenn nicht genügend Speicher zur Speicherung der gesamten Ontologie zur Verfügung steht. Außerdem sollte dieser Ansatz bevorzugt werden, wenn die generierte Ontologie relativ selten verwendet wird, sodass der Rechenaufwand für das neue Generieren der Ontologie weniger ins Gewicht fällt als der Aufwand für die Aktualisierungen der Ontologie es tun würde.

5.6.1 Einhaltung von Ontologierestriktionen zur Laufzeit mithilfe von Anfragen

Die Einhaltung der **Property** Axiome soll zur Laufzeit vom generierten Code gewährleistet werden. Dafür überprüft der generierte Code vor jeder Änderung der Attribut- bzw. Referenzwerte, ob diese Änderung eine Verletzung der relevanten Restriktionen der entsprechenden **Property** bedeuten würde.

Ist dies der Fall so reagiert der generierte Code mit dem Verbot der Änderung oder dem Inferieren und Ausführen weiterer Änderungen, deren Ausführung die Einhaltung der **Property** Axiome sicherstellt. Für die Erfüllung dieser Aufgaben verwendet der generierte Code Anfragen auf der generierten Ontologie. Über die Anfragen wird geprüft, ob das Hinzufügen bzw. Entfernen der entsprechenden **Property Assertion** eine Verletzung einer relevanten Restriktion bedeuten würde bzw. welche **Property Assertions** zusätzlich hinzugefügt bzw. entfernt werden müssen, um die Einhaltung der **Property** Axiome zu gewährleisten.

Anschließend muss zwischen drei verschiedenen Fällen unterschieden werden.

1. Die gewünschte Änderung würde keine relevante Restriktion verletzen.
2. Die gewünschte Änderung würde eine relevante Restriktion verletzen, aber es könnten zusätzliche Änderungen inferiert werden, deren Ausführung eine Verletzung der relevanten Restriktionen verhindert.
3. Die gewünschte Änderung würde eine relevante Restriktion verletzen und es könnten keine zusätzlichen Änderungen inferiert werden, deren Ausführung eine Verletzung der relevanten Restriktionen verhindern würde.

Im ersten Fall und im zweiten Fall wird die gewünschte Änderung vorgenommen. Allerdings werden im zweiten Fall zusätzlich die inferierten Änderungen ausgeführt. Im dritten Fall hingegen wird die gewünschte Änderung

nicht vorgenommen und somit eine Verletzung der relevanten Restriktionen verhindert.

5.6.2 Verwendung von Konsistenzüberprüfungen zur Validitätsprüfung

Es gibt mehrere Ontologieausdrücke, die nicht vom Berechnungsschritt vor der Codegenerierung behandelt werden und deren Einhaltung nicht zur Laufzeit durch den generierten Code gewährleistet wird. Dieser Unterabschnitt erläutert, wie diese Ausdrücke behandelt werden und wie der Entwickler diese Ausdrücke dennoch sinnvoll nutzen kann.

Anstatt zu verhindern, dass diese Restriktionen verletzt werden, wird es gestattet die durch diese Ontologieausdrücke definierten Restriktionen zwischenzeitlich zu verletzen und zu einem Zeitpunkt zu überprüfen, dass keine Ontologierestriktionen verletzt sind. Dazu werden im generierten Code Operationen erzeugt, die eine solche Überprüfung vornehmen. Diese Überprüfung wird durchgeführt, indem der generierte Code unter Annahme einer geschlossenen Welt eine Konsistenzüberprüfung auf der generierten Ontologie anwendet.

Der Entwickler kann dann diese Operationen aufrufen, wenn er überprüfen möchte, dass alle Ontologierestriktionen eingehalten sind. Ist der Entwickler darauf angewiesen, dass alle Restriktionen zu einem Zeitpunkt eingehalten sind, so kann er dies zunächst überprüfen und anschließend angemessen darauf reagieren. Eventuell können zusätzliche Operationen im generierten Code erzeugt werden, die dem Entwickler Informationen darüber geben, wieso Ontologierestriktionen im derzeitigen Datenbestand verletzt sind. Diese zusätzlichen Operationen würden den Entwickler bei der Wahl der geeigneten Reaktion unterstützen.

5.6.3 Verwendung von Anfragen zur Modellierung des Verhaltens von Operationen

Das EOF System bietet die Möglichkeit das Verhalten einer Operation mit Anfragen zu modellieren. In diesem Abschnitt wird beschrieben, wie das Verhalten solcher Operationen mit den Anfragen modelliert wird und welche Gestalt der generierte Code dieser Operationen hat.

Es gibt drei verschiedene Möglichkeiten das Verhalten einer Operation mit Anfragen zu modellieren:

1. Die Operation wird mit genau einer Anfrage und keinem Programmcode annotiert. Diese Anfrage ist eine Select oder eine Ask Anfrage und der Rückgabebetyp der Operation wurde passend gewählt.
2. Die Operation wird sowohl mit einer oder mehreren Anfragen als auch mit Programmcode annotiert.

3. Die Operation wird mit genau einer Anfrage und keinem Programmcode annotiert. Der Anfragetyp oder der Rückgabebetyp der Operation wird nicht passend zu den Bedingungen der ersten Möglichkeit gewählt.

Die erste Möglichkeit ist die zu bevorzugende Möglichkeit. In diesem Fall wird das Verhalten der Operation vollständig durch die Anfrage modelliert und der generierte Code implementiert die Operation vollständig. Der Code einer solchen Operation besteht aus Code zum Ausführen der Anfrage auf einer generierten Ontologie, Code zur Umwandlung des Anfrageergebnisses und Code zum Setzen des Rückgabewerts. Eine solche Operation liefert das Ergebnis der Anfrage in umgewandelter Form als Rückgabewert.

Bei der Umwandlung des Ergebnisses werden die IRIs, die im Ergebnis enthalten sind, durch die Entitäten substituiert, die von den IRIs repräsentiert werden. Für diese Substitution wird die bei der Ontologiegenerierung erstellte Abbildung zwischen IRIs und Entitäten verwendet. Ebenfalls wird bei der Umwandlung der Typ des Anfrageergebnisses an den Typ des Rückgabewerts angepasst.

Bei der zweiten Möglichkeit wird das Verhalten der Operation vollständig durch die Anfragen und den annotierten Programmcode modelliert. In diesem Fall wird in den generierten Code der Code zum Ausführen der Anfragen auf einer generierten Ontologie und anschließend der annotierte Programmcode eingefügt. Dabei wird davon ausgegangen, dass der annotierte Programmcode die Ergebnisse der Anfragen verwendet und den Rest der Implementierung übernimmt. Somit sollte der annotierte Programmcode auch den Rückgabewert setzen. Die Umwandlung des Anfrageergebnisses, die für die erste Möglichkeit verwendet wird, sollte dem Entwickler so bereitgestellt werden, dass er sie im annotierten Programmcode verwenden kann.

Bei der dritten Möglichkeit wird das Verhalten der Operation teilweise durch die Anfrage beschrieben. Der Entwickler muss nach der Codegenerierung das restliche Verhalten manuell implementieren. Für solche Operationen enthält der generierte Code den Code zum Ausführen der Anfragen auf einer generierten Ontologie. Anschließend werden ein Kommentar und Code für das Werfen eines Fehlers in den generierten Code eingefügt. Der eingefügte Kommentar weist den Entwickler darauf hin, dass er den Rest der Methode selbst implementieren muss.

Um bei der Modellierung einer Operation mit Anfragen einfach auf das Objekt verweisen zu können, auf dem die Operation ausgeführt wird, wird festgelegt, dass dieses Objekt innerhalb einer annotierten Anfrage mit `?self` bezeichnet wird. Somit wird zur Laufzeit nicht die annotierte Anfrage ausgeführt, sondern eine entsprechende Anfrage, in der alle Vorkommen von `?self` durch die IRI dieses Objekts substituiert wurden.

Zum Beispiel enthält Listing 5.1 eine SPARQL Anfrage, die das Verhalten der `getShipToOrders` Operation aus dem Szenario modellieren soll. Die Präfixdeklarationen der Anfrage sind nicht im Listing enthalten. Es soll davon ausgegangen werden, dass `po` das Präfix für das Paket ist, in dem die Klassen

des Szenarios enthalten sind. Die Anfrage liefert also als Ergebnis alle Bestellungen, deren `shipTo` Referenz auf die Adresse verweist, auf der die Operation aufgerufen wird.

```
1 SELECT DISTINCT ?order
2 WHERE{
3     ?order po:Order.shipTo ?self.
4 }
```

Listing 5.1. Annotierte SPARQL Anfrage der `getShipToOrders` Operation

Im Folgenden wird der generierte Code für die Ausführung einer annotierten Anfrage auf einer generierten Ontologie beschrieben. Im generierten Code wird der Pfad der Anfrage einer Variablen zugewiesen. Der Pfad der Anfrage lässt sich aus der Anfrageannotation herleiten.

Da innerhalb der annotierten Anfragen mit `?self` auf das Objekt, auf dem die Operation ausgeführt wird, verwiesen werden kann, wird Code zum Erzeugen einer neuen Anfrage erzeugt. Diese neue Anfrage entspricht der annotierten Anfrage mit dem Unterschied, dass alle Vorkommen von `?self` in der annotierten Anfrage durch die IRI des Objekts, auf dem die Operation ausgeführt wird, ersetzt wurden. Die IRI wird über die bei der Ontologiegenerierung erstellte Abbildung zwischen IRIs und Objekten ermittelt. Schließlich folgt Code zum Ausführen dieser neuen Anfrage auf der generierten Ontologie.

Implementierung

In diesem Kapitel wird eine Implementierung des in dieser Arbeit vorgestellten Lösungsansatzes vorgestellt. Diese Implementierung erlaubt es den entsprechenden `Java` Code für ein EOF Modell zu generieren. Als zu erweiterndes MDE System wurde das `Eclipse Modeling Framework (EMF)`[36] gewählt. Diese Wahl bietet sich an, weil `EMF` ein weit verbreitetes System für modellgetriebene Softwareentwicklung ist, das zahlreiche Erweiterungsmöglichkeiten anbietet, und weil für `EMF` bereits viele verschiedene Werkzeuge existieren.

Zunächst wird in Abschnitt 6.1 die Architektur vorgestellt, die für die Implementierung gewählt wurde. Anschließend wird in Abschnitt 6.2 beschrieben, wie `Ecore` Modelle zu EOF Modellen erweitert werden. Wie die Generierung einer Ontologie aus solchen Modellen implementiert wurde, wird in Abschnitt 6.3 erläutert. Der Abschnitt 6.4 befasst sich mit der Implementierung der Codegenerierung des EOF Systems. Schließlich wird im Abschnitt 6.5 die Verwendung der Ontologie im generierten Code behandelt.

6.1 Gewählte Architektur für das EOF System

Für die Implementierung des Lösungsansatzes wurden zusätzliche Plugins, die die `EMF` Codegenerierung erweitern, in das `TwoUse Toolkit`¹ [22, 35] integriert. Das `TwoUse Toolkit` ist ein System, das bereits viele Funktionalitäten für die modellgetriebene Entwicklung, für die Erstellung von Ontologien und für die Arbeit mit Ontologien anbietet. Somit bietet das `TwoUse Toolkit` bereits einen Großteil, der für ein EOF System benötigten Funktionalitäten an. Durch die Integration der Plugins für die EOF Codegenerierung wird es zu einem EOF System erweitert. Die Abbildung 6.1 zeigt ein Komponentendiagramm mit dem Architekturentwurf für das EOF System.

Für das `TwoUse Toolkit` wurden bereits verschiedene Annotationen für `Ecore` Modelle definiert, die es ermöglichen Ontologierestriktionen für `Eco-`

¹ Projektseite des `TwoUse Toolkits`: <http://code.google.com/p/twouse/>



Abb. 6.1. Gewählte Architektur für das EOF System

re Modelle zu formulieren. In Abschnitt 6.2 werden diese Modelle und ihre Verwendung als EOF Modelle beschrieben. Aus solchen Modellen und deren Modellinstanzen lassen sich mithilfe des OWLizers [37] des TwoUse Toolkits bereits Ontologien generieren. Wie diese Ontologiegenerierung funktioniert und welche Erweiterungen für die Nutzung in einem EOF System notwendig sind, wird im Abschnitt 6.3 erläutert. Das TwoUse Toolkit verwendet den Pellet² Beweiser, um Konsistenzüberprüfungen und Anfragen auf Ontologien auszuführen.

Für die Eingabe der annotierten Ecore Modelle enthält das TwoUse Toolkit den textuellen Editor `Text Ecore`. Dieser Editor soll den Umgang mit den Ontologieannotationen erleichtern. Zudem beinhaltet das TwoUse Toolkit einige Erweiterungen für EMF Editoren. Zum Beispiel gibt es eine Aktion, die das einfache Verbinden von Anfragen mit Ecore Operationen ermöglicht.

Das TwoUse Toolkit unterstützt mehrere Sprachen und Syntaxen für Ontologien und enthält mehrere Plugins für Transformationen von Ontologien und Anfragen zwischen diesen Sprachen und Syntaxen. Zur Durchführung dieser Transformationen werden verschiedene externe Modelltransformationssysteme verwendet. Zum Beispiel wurden einige Transformationen mit der ATLAS Transformation Language (ATL)[8] definiert. Zur Ausführung dieser Transformationen wird das ATL Transformationssystem verwendet.

Die Plugins, die für die Implementierung der EOF Codegenerierung in das TwoUse Toolkit integriert wurden, beinhalten Erweiterungen für den EMF Generator und greifen für die Bereitstellung der meisten Codegenerierungsfunktionen auf das EMF System zurück. Im Abschnitt 6.4 werden die zusätzlichen Plugins und die EOF Codegenerierung beschrieben. Auf die Entwicklung einer Komponente für die Berechnung des Generatormodells wurde verzichtet, da sie für die Entwicklung eines EOF Systems zwar wünschenswert, aber nicht notwendig ist. Eine spätere Integration einer solchen Komponente wäre möglich.

6.2 Annotierte Ecore Modelle als EOF Modelle

Für die Implementierung des Lösungsansatzes sollen annotierte Ecore Modelle als EOF Modelle dienen. Mithilfe der Ecore EAnnotations werden Ontologiestriktionen für das Modell festgelegt und Anfragen mit Operationen verbunden. Diese Annotationen wurden von Tobias Walter³ et al. [38] definiert.

Eine EAnnotation wird an genau ein Modellelement angefügt und verfügt über eine Referenz, mit der die Annotation auf dieses Modellelement zugreifen kann. Zudem haben EAnnotations ein Attribut `source`, das das Speichern von Strings ermöglicht, und eine Referenz `references` die es erlaubt auf EObjects zu verweisen. Somit kann diese Referenz genutzt werden, um auf ein beliebiges

² Internetseite Pellet: <http://clarkparsia.com/pellet/>

³ walter@uni-koblenz.de

Modellelement zu verweisen. Insbesondere ist es möglich mit dieser Referenz auf eine andere Annotation zu verweisen. Die Objekte, auf die diese Referenz verweist, werden in einer geordneten Liste gespeichert.

Im Unterabschnitt 6.2.1 wird erläutert, wie mit den annotierten *Ecore* Modellen eine Ontologie definiert wird. Anschließend wird im Unterabschnitt 6.2.4 auf das Verbinden von Anfragen und Annotationen eingegangen.

6.2.1 Ontologiedefinitionen

Die verschiedenen OWL Konstrukte werden über das eigentliche *Ecore* Modell und über spezielle Ontologieannotationen modelliert. Die Ontologieannotationen der Implementierung orientieren sich an der *Manchester Syntax* für OWL Ontologien [11]. Die verschiedenen Ontologieannotationen sind dadurch gekennzeichnet, dass die Art des Ontologieausdrucks, den sie repräsentieren, als Stringwert, gemäß der *Manchester Syntax*, in das `source` Attribut der Annotation kodiert wird.

Die Ontologieannotationen werden an die Modellelemente geheftet, für die ein Ontologieausdruck definiert werden soll. Zudem verweisen sie mit der Referenz `references` auf die Modellelemente, die als Parameter an den Ontologieausdruck übergeben werden. Sollte der Ontologieausdruck weitere Parameter benötigen, so werden diese über die `details` Map der Annotation hinzugefügt. Zum Beispiel wird der Zahlenwert für Kardinalitätsrestriktionen über diese Map als Parameter übergeben, indem ein entsprechender Eintrag in die `details` Map der Annotation für diese Restriktion eingetragen wird. Somit ist es grundsätzlich möglich alle OWL Konstrukte als solche Annotationen zu beschreiben.

Bei der Modellierung und bei den Ontologieannotationen wird eine Trennung von *TBox* und *ABox* bzw. von Modell und Modellinstanz vorgenommen. Das *Ecore* Modell und seine Ontologieannotationen enthalten die Informationen eines Klassendiagramms und der *TBox*. Die Modellinstanzen hingegen enthalten die Informationen eines Objektdiagramms und der *ABox*. Die Ontologiegenerierung der aktuellen Version des *OWLizers* unterstützt nicht die Verwendung von `SameIndividuals` und `DifferentIndividuals` Ausdrücken. Daher werden diese Ausdrücke und die OWL Konstrukte `ObjectOneOf`, `ObjectHasValue`, sowie negative `PropertyAssertions` nicht von den annotierten *Ecore* Modellen unterstützt.

Um die durch das *EOF* Modell definierte Ontologie sinnvoll nutzen zu können, wird gefordert, dass nur OWL DL Ontologien definiert werden. Für die Einhaltung dieser Einschränkung muss der Entwickler sorgen. Man könnte den Entwickler später dabei unterstützen, indem man ihm entsprechend angepasste Editoren bereitstellt oder den im Abschnitt 5.4 beschriebenen Berechnungsschritt implementiert.

Im Folgenden werden zunächst die Ontologiedefinitionen im *Ecore* Modell in Abschnitt 6.2.2 und anschließend die Ontologiedefinitionen in den Modellinstanzen in Abschnitt 6.2.3 beschrieben.

6.2.2 Ontologiedefinitionen im Ecore Modell

Die Ontologiedefinitionen im Ecore Modell erfolgen gemäß der Beschreibung in Abschnitt 5.3.2. Demnach repräsentieren die Ecore Klassen ebenfalls OWL Klassen, die Referenzen und Attribute ebenfalls Properties und die Ecore Datentypen ebenfalls OWL Datentypen. Für die Parameter der Ontologieannotationen wird, wie bereits erwähnt, die Referenz `references` der Annotationen verwendet. Da diese eine geordnete Liste liefert, kann auch die Reihenfolge der Parameter festgelegt werden.

Das Anheften der Annotationen wird analog zu den Frames der Manchester Syntax verwendet. Zum Beispiel würde die Restriktion `SubObjectPropertyOf(pendingOrders, orders)` des Szenarios dargestellt, indem im Ecore Modell an die `pendingOrders` Referenz eine `SubPropertyOf` Annotation angeheftet wird, deren `references` Referenz auf die `orders` Referenz verweist.

Allerdings werden nicht alle OWL Konstrukte von den annotierten Ecore Modellen unterstützt, da die Ontologiegenerierung der aktuellen OWLizer Version einige Konstrukte noch nicht unterstützt. Zu den nicht unterstützten Konstrukten gehören die `AnnotationProperties` sowie die OWL Konstrukte `ObjectOneOf`, `ObjectHasValue`, `DataSomeValuesFrom`, `DataAllValuesFrom`, `DataHasValue`. Außerdem werden keine komplexen `DataRange` Konstrukte unterstützt. Somit können lediglich Datentypen als `DataRanges` verwendet werden.

Aus den `eOpposite` Angaben der Referenzen des Ecore Modells werden zusätzliche `InverseObjectProperty` Restriktionen hergeleitet. Der Entwickler kann zudem über die Optionen für die Ontologiegenerierung festlegen, dass weitere Ontologiekonstrukte aus dem Modell hergeleitet werden. Es wäre möglich `Domains` und `Ranges` der Properties aus den Angaben im Ecore Modell oder auch Restriktionen aus den Kardinalitäten des Ecore Modells herzuleiten. Über die Kardinalitätsangaben können Klassenaxiome für Kardinalitätsbeschränkungen sowie `FunctionalProperties` und `InverseFunctionalObjectProperties` hergeleitet werden.

6.2.3 Ontologiedefinitionen in den Modellinstanzen

Die Modellinstanzen definieren, wie in Abschnitt 5.3.3 beschrieben, die Assertion Axiome der Ontologie. Die OWL Konstrukte `SameIndividual` und `DifferentIndividual`, sowie negative `PropertyAssertions` werden nicht unterstützt, da die Ontologiegenerierung der aktuellen OWLizer Version diese nicht unterstützt. Des Weiteren wird davon ausgegangen, dass ein Individuum nicht zu zwei Klassen gehören kann, falls nicht eine dieser Klassen Unterklasse der anderen Klasse ist.

6.2.4 Anfrageannotationen

Für die Modellierung von Operationen mit Anfragen wurden Annotationen zum Verbinden von Operationen und Anfragen definiert. Diese Annotationen

werden im Modell direkt an die entsprechende Operation angeheftet und sind dadurch gekennzeichnet, dass der Pfad der Anfrage in der `source` Variablen der Annotation als Stringwert kodiert ist.

Das `TwoUse` Toolkit bietet die Möglichkeit Anfragen verschiedener Anfragesprachen als `Ecore` Modelle darzustellen. Daher ist es sinnvoll die Referenz `references` zu verwenden, um auf dieses Modell zu verweisen. Dadurch kann der Entwickler sich das Modell einer Anfrage, die das Verhalten einer Operation modelliert, ansehen, ohne vorher lange suchen zu müssen.

6.3 Ontologiegenerierung

Das `TwoUse` Toolkit bietet mit dem `OWLizer`[37] bereits die Möglichkeit eine Ontologie aus `Ecore` Modellen, deren Modellinstanzen, sowie den in Abschnitt 6.2 beschriebenen Annotationen zu generieren. Für die Implementierung des Lösungsansatzes wurde diese Ontologiegenerierung verwendet und um das Speichern von Abbildungen zwischen Entitäten und IRIs und zwischen Literalen und Datenwerten erweitert.

Für das Arbeiten mit diesen Abbildungen wurde eine eigene Java Klasse `OWLizerMapping` erzeugt. Diese Klasse enthält verschiedene Maps, die für das Speichern der Abbildungen verwendet werden. Zudem bietet diese Klasse verschiedene Operationen an, die es erlauben neue Elemente in die Abbildungen hinzuzufügen, die Abbildungen zu leeren und die Abbildungen für das Auffinden von zugehörigen Elementen zu verwenden. Zum Beispiel ist es möglich eine Entität mit der zugehörigen IRI in die Abbildung aufzunehmen und später mithilfe der IRI die Entität zu erhalten oder mithilfe der Entität eine zugehörige IRI zu erhalten.

Zudem wurde eine Java Klasse `MappingOWLizer` für die Ontologiegenerierung erzeugt, die die Ontologiegenerierung des `OWLizers` verwendet und während der Ontologiegenerierung die verwendeten IRIs, Entitäten, Literale und Datenwerte in die Abbildung eines `OWLizerMapping` Objekts speichert. Vor der Generierung einer neuen Ontologie werden die Abbildungen wieder geleert.

Im Folgenden wird die Ontologiegenerierung des `OWLizers` beschrieben. Der `OWLizer` verwendet ein `Ecore` Modell und dessen Ontologieannotationen für die Generierung der `TBox` der Ontologie. Für die Generierung der `ABox` werden die Informationen der Modellinstanzen des `Ecore` Modells verwendet. Die EMF Codegenerierung erzeugt im generierten Code auch eine Coderepräsentation des `Ecore` Modells. Dadurch ist es in der Regel möglich von einem Objekt aus das gesamte Modell und die mit diesem Objekt verbundenen Modellinstanzen zu durchlaufen. Somit kann der `OWLizer` eine komplette Ontologie generieren, wenn er ein Objekt eines `Ecore` Modells übergeben bekommt.

Probleme gibt es bei der Ontologiegenerierung von einem Objekt aus, wenn die Objekte der Modellinstanzen in verschiedenen Ressourcen liegen und zwischen diesen Ressourcen nur eine Verbindung in eine Richtung existiert. Ein

Beispiel hierfür ist im Szenario enthalten. Im Modell des Szenarios dürfen die **Address** Objekte in einer anderen Ressource als die restlichen Objekte gespeichert werden. Da aber eine Adresse keine Referenz auf ein Objekt der anderen Ressource hat, würde eine von einer Adresse aus generierte Ontologie lediglich Individuen für die Adressen enthalten. Generiert man hingegen die Ontologie von einer Bestellung aus, so sind alle Individuen der Modellinstanz in der Ontologie enthalten.

Zur Lösung dieses Problems könnte man dafür sorgen, dass zwischen den Ressourcen Verbindungen in beide Richtungen existieren oder dass alle Objekte der Modellinstanzen in einer einzigen Ressource gespeichert werden. Dies kann erreicht werden, indem man weitere Referenzen einfügt bzw. verschiedene Referenzen als **containment** Referenzen deklariert. Eine weitere Möglichkeit wäre, dass der Generator dafür sorgt, dass in solchen Fällen zusätzliche Listen für das Auffinden der anderen Ressourcen geführt werden. Der implementierte Prototyp behandelt dieses Problem noch nicht. Der Entwickler muss selbst dafür sorgen, dass es keine Probleme bei der Ontologiegenerierung durch die Speicherung der Objekte in verschiedenen Ressourcen gibt.

Der OWLizer generiert für die Ontologie die Standard-Präfixdeklarationen und Präfixdeklarationen für das **Ecore** Modell. Die aktuelle Version des OWLizers verwendet noch für alle IRIs das gleiche Präfix. Es ist aber geplant später die Präfixe für die IRIs gemäß den Paketen des **Ecore** Modells zu vergeben, so dass für jedes Paket ein entsprechendes Präfix angelegt wird. Der Präfixname und die zugehörige IRI einer Präfixdeklaration könnten aus den **nsURI** und **nsPrefix** Attributen der **EPackages** übernommen werden.

Die IRIs der generierten Ontologie werden vom OWLizer aus dem Modell und den Modellinstanzen hergeleitet. Für Klassen und Datentypen setzt sich die IRI aus ihrem Präfix und ihrem Namen zusammen. Für die IRI einer Property werden ihr Name und die IRI der obersten Klasse, die im Modell die entsprechende Referenz bzw. das entsprechende Attribut enthält, verwendet. Die IRIs der Individuen werden momentan aus dem Hashcode des zugehörigen Objekts und der IRI der Klasse dieses Objekts zusammengesetzt. Die Verwendung des Hashcodes ist nicht ideal und wird eventuell zu einem späteren Zeitpunkt durch eine andere Methode abgelöst.

Der OWLizer generiert für die im Modell enthaltenen **Ecore** Klassen und **Ecore** Datentypen entsprechende Klassen- und Datentypdeklarationen für die Ontologie. Ebenso werden **DataProperty** Deklarationen für Attribute und **ObjectProperty** Deklarationen für Referenzen angelegt. Bei dem Erstellen der Property Deklarationen werden nur Attribute und Referenzen behandelt, die nicht von einer Oberklasse geerbt wurden. Für die **name** Attribute der **Address**, **USAddress** und **GlobalAddress** Klassen würde also nur eine Property Deklaration erzeugt. Zudem werden **Individual** Deklarationen für die Objekte der Modellinstanzen in die generierte Ontologie eingefügt.

Für die Subklassenbeziehungen des Modells generiert der OWLizer entsprechende **SubClassOf** Axiome. Ebenso verwendet der OWLizer die **eOpposite** Angaben der Referenzen des **Ecore** Modells, um zusätzliche **InverseObjectPro-**

property Restriktionen herzuleiten. Außerdem kann der Entwickler über Optionen bestimmen, ob weitere Ontologieausdrücke aus dem Modell hergeleitet werden. Zum Beispiel kann er `Domains` und `Ranges` der `Properties` aus den Angaben im `Ecore` Modell oder auch Restriktionen aus den Kardinalitäten des `Ecore` Modells herleiten lassen.

Die verschiedenen Ontologieannotationen im `Ecore` Modell werden vom `OWLizer` genutzt, um entsprechende Ontologieausdrücke für die Ontologie zu generieren. Sollten bei dieser Umwandlung Literale für Java Objekte benötigt werden, so werden die Literale mithilfe der `toString` Operation des Objekts und der Stringrepräsentation des entsprechenden Datentyps gewonnen.

Die `Assertion` Axiome der Ontologie generiert der `OWLizer` mithilfe der Modellinstanzen. Für jedes Objekt der Modellinstanzen und jede diesem Objekt zugehörige Klasse wird eine entsprechende `ClassAssertion` erzeugt. Außerdem werden die Attribute und Referenzen jedes Objekts durchlaufen und entsprechende `DataPropertyAssertions` bzw. `ObjectPropertyAssertions` erzeugt. Dabei werden mehrere `Assertions` für eine `Property` erzeugt, falls deren zugehörige Referenz bzw. deren zugehöriges Attribut eine Kardinalität größer eins und mehr als einen Wert hat. In diesem Fall wird eine `Assertion` pro Wert generiert. Sollte also ein `Order` Objekt über die `previousOrders` Referenz drei `Order` Objekte zugewiesen bekommen, so würden auch drei entsprechende `Assertions` in der Ontologie erzeugt.

6.4 Codegenerierung des EOF Systems

EMF bietet verschiedene Möglichkeiten an, um die EMF Codegenerierung anzupassen. Für die Implementierung wurde die Erweiterungsmöglichkeit über dynamische Templates gewählt. Diese Möglichkeit reicht für die Umsetzung des Lösungsansatzes aus und der Aufwand für die Anpassung an neue EMF Versionen ist relativ gering.

Bei dieser Möglichkeit werden die `JET` Templates, die der Generator verwendet, um aus dem `GenModel` den zu generierenden Code herzuleiten, dynamisch zu Beginn der Codegenerierung geladen. Über die Optionen des `GenModels` kann man das dynamische Laden der Templates einschalten und angeben, welche Templates geladen werden sollen. Somit ist es möglich den EMF Generator mit eigenen angepassten `JET` Templates zu verwenden. Allerdings müssen die eigenen Templates einige Bedingungen erfüllen und es werden lediglich die Dateien erzeugt, die auch von den EMF Templates erzeugt worden wären.

Um zusätzliche Generatoroptionen für die eigenen Templates festzulegen, können die `GenAnnotations` für `GenModels` verwendet werden. Zum Beispiel kann man festlegen, dass ein bestimmter Wert für das `source` Attribut einer `GenAnnotation` bedeutet, dass eine bestimmte Generatoroption gewählt wurde. In den eigenen Templates überprüft man dann die `GenAnnotations` des `Gen-`

Models und passt die Codegenerierung den in diesen Annotationen kodierten Generatoroptionen entsprechend an.

Im Folgenden wird die implementierte Codegenerierung des EOF Systems beschrieben. Dazu wird im Abschnitt 6.4.1 die Organisation der Plugins, die für die Erweiterung des Generators und für die Unterstützung des generierten Codes erstellt wurden, erläutert. Anschließend wird in Abschnitt 6.4.2 beschrieben, wie die Templates erweitert wurden und welche Änderungen der Codegenerierung daraus resultieren. Schließlich werden in Abschnitt 6.4.3 die Erwartungen des erweiterten Generators an seine Eingabemodelle aufgeführt.

6.4.1 Organisation der Eclipse Plugins

Für die Erweiterung des Generators und für die Unterstützung des generierten Codes wurden drei Plugins entwickelt: ein Generator-Plugin (`west.twouse.eof.codegen.ecore`), ein Generator-UI-Plugin (`west.twouse.eof.codegen.ecore.ui`) und ein Common-Plugin (`west.twouse.eof.common`). Diese Plugins greifen auf verschiedene Plugins des TwoUse Toolkits zu, um bereits vorhandene Funktionen des TwoUse Toolkits zu nutzen.

Das Generator-Plugin ist für die Erweiterungen für den Generator vorgesehen. Dieses Plugin enthält die erweiterten Templates für den Generator und Hilfsklassen, die die Templates oder der Generator für die Codegenerierung verwenden. Diese Hilfsklassen beinhalten verschiedene Operationen zum Umgang mit dem EMF GenModel und den EOF Annotationen. Die Berechnung des Generatormodells könnte in dieses Plugin integriert oder in ein eigenes Plugin ausgelagert werden.

Das Generator-UI-Plugin enthält Erweiterungen der Benutzerschnittstelle, die für den Umgang mit dem erweiterten Generator angeboten werden. In dieses Plugin sollten zum Beispiel Erweiterungen der Benutzerschnittstelle eingebunden werden, die die Arbeit mit den GenAnnotationen erleichtern, die für das Festlegen der Optionen des erweiterten Generators genutzt werden. Ebenfalls in diesem Plugin sind Aktionen, die das Umschalten zwischen EOF Codegenerierung und EMF Codegenerierung erleichtern.

Das Common-Plugin enthält Hilfsklassen, die die Durchführung verschiedener Aufgaben unterstützen und von dem generierten Code genutzt werden können. Die Hilfsklassen bieten Unterstützung für die Generierung einer Ontologie aus einem EOF Modell, für die Transformation und Ausführung von Anfragen, sowie für die Umwandlung der Anfragenergebnisse an. Zudem sind in diesem Plugin Klassen für die verschiedenen Listen enthalten, die die Einhaltung von bestimmten Property Axiomen zur Laufzeit gewährleisten.

6.4.2 Erweiterung der Java Emitter Templates

Für die EOF Codegenerierung wurden, die EMF Templates angepasst. EMF verwendet mehrere Templates zur Generierung des Codes. Dabei wird jedes

Template für die Erzeugung einer bestimmten Dateiart verwendet. Zum Beispiel erzeugt das **Manifest** Template des Modellordners die Manifestdatei für das generierte Plugin. Das **Class** Template des Modellordners hingegen erzeugt die Interfaces und Klassen für die **Ecore** Klassen des Modells.

Diese Templates bieten verschiedene **Insertion Points** und **Override Points** zur Erweiterung der Templates an. Die **Insertion Points** verweisen auf den Pfad für eine Ergänzungsdatei für das Template. Der von dieser Datei definierte Code wird an die Stelle des **Insertion Points** eingefügt. Wurde dieser **Insertion Point** als optional deklariert, so muss die Datei nicht vorhanden sein. Ist die Datei eines optionalen **Insertion Points** nicht vorhanden, so wird einfach kein Code an dieser Stelle eingefügt. Sollte der **Insertion Point** nicht optional sein, so muss die verlinkte Datei für die Codegenerierung vorhanden sein.

Override Points erlauben es einen Teil des Templates durch den Code einer Überschreibungsdatei für das Template zu ersetzen. Die Definition eines **Override Points** beinhaltet einen Verweis auf den Pfad für diese Datei und markiert den Code der ersetzt wird, falls diese Datei existiert. Die **Insertion Points** und **Override Points** wurden genutzt und es wurden neue **Insertion Points** in den erweiterten Templates definiert, um den Aufwand für das Anpassen an neue EMF Versionen möglichst gering zu halten.

Für die Implementierung der EOF Codegenerierung reicht es aus das **Manifest** Template und das **Class** Template des Modellordners zu erweitern, da die EMF Codegenerierung des Editors ohne Änderung verwendet werden kann. Möchte man die Validierungsaktion für den Editor, gemäß Anforderung R10, implementieren, so müssten zusätzlich das **Manifest** Template und das **ActionBarContributer** Template des Editorordners erweitert werden.

Die Änderungen an den **Manifest** Templates sorgen nur dafür, dass die für den zusätzlichen Code benötigten Pluginabhängigkeiten in die generierten Manifeste eingefügt werden. Die Änderungen am **ActionBarContributer** Template ersetzen das Erzeugen der EMF Validierungsaktion im Konstruktor durch das Erzeugen einer EOF Validierungsaktion. Die Erweiterungen für das **Class** Template des Modellordners sind hingegen umfangreicher. Diese Erweiterungen erzeugen den Code für die Einhaltung einiger **Property** Axiome zur Laufzeit (6.4.2), den Code für die Operationen für Konsistenzüberprüfungen (6.4.2), sowie den Code für die mit Anfragen modellierten Operationen (6.4.2).

Erweiterungen für die Einhaltung von Property Axiomen zur Laufzeit

Die Einhaltung der **Property** Axiome wird dadurch erreicht, dass für Attribute und Referenzen mit **Property** Axiomen, die zur Laufzeit eingehalten werden sollen, spezielle Listen verwendet werden, die gewährleisten, dass diese Axiome nicht verletzt werden. Die Arbeitsweise dieser Listen wird in Abschnitt 6.5.1 beschrieben.

Bei der EMF Codegenerierung werden die Listen für die Attribute und Referenzen in den entsprechenden Getter-Operationen erzeugt. Die Änderungen

für die EOF Codegenerierung sorgen dafür, dass für Attribute und Referenzen mit `Property` Axiomen, die zur Laufzeit eingehalten werden sollen, die speziellen Listen für die jeweiligen `Property` Axiome gewählt werden und für die anderen Attribute und Referenzen die EMF Codegenerierung verwendet wird.

Zudem muss zur Einhaltung der Reflexivität einer Referenz bzw. eines Attributs der Konstruktor der entsprechenden Klasse das neu erzeugte Objekt in die entsprechende Liste einfügen. Daher wird die Codegenerierung so angepasst, dass Code zum Einfügen des neu erzeugten Objekts in alle Listen, die reflexive `Properties` repräsentieren, in dem Code der Konstruktoren enthalten ist.

Erweiterungen für Operationen für Konsistenzüberprüfungen

Für die Implementierung der Codegenerierung für die in Abschnitt 6.5.2 beschriebenen Operationen wird der `Insertion Point Class/genOperation.insert.javajetinc` genutzt. Der Code für die Generation dieser Operation befindet sich in der entsprechenden Ergänzungsdatei und generiert für jede Java Klasse, die eine `Ecore` Klasse repräsentiert eine solche Operation.

Erweiterungen für mit Anfragen modellierte Operationen

Die Implementierung der Codegenerierung für mit Anfragen modellierte Operationen wird mithilfe des `Override Points Class/genOperation.override.javajetinc` realisiert. Die Abbildung 6.2 zeigt ein Aktivitätsdiagramm, das den Kontrollfluss für die Codegenerierung der `Ecore` Operationen modelliert.

Der erweiterte Generator überprüft während der Generierung des Codes für eine Operation, ob an diese Operation Anfragen annotiert sind. Ist keine Anfrage annotiert, so wird die EMF Codegenerierung für die Generierung der Operation verwendet. Anderenfalls wird der Code für die Ontologiegenerierung und für die Ausführung der annotierten Anfragen generiert.

Nach der Generierung des Codes für die Ausführung der annotierten Anfragen wird überprüft, um welchen der folgenden drei Fälle es sich handelt.

1. Die Operation wurde mit genau einer Anfrage und keinem Programmcode annotiert. Diese Anfrage ist eine `Select` oder eine `Ask` Anfrage und der Rückgabebetyp der Operation wurde passend gewählt.
2. Die Operation wurde sowohl mit einer oder mehreren Anfragen als auch mit Programmcode annotiert.
3. Die Operation wurde nicht mit Programmcode annotiert und die Bedingungen der ersten Möglichkeit sind nicht erfüllt.

Je nach festgestelltem Fall fährt der erweiterte Generator unterschiedlich fort. Beim ersten Fall wird die vollständige Implementierung der Operation unterstützt und es wird der Code zum Setzen des Rückgabewerts generiert. Dieser Code beinhaltet Code, der das Anfrageergebnis so umwandelt, dass das umgewandelte Ergebnis zum Rückgabebetyp passt. Beim zweiten Fall wird der annotierte Programmcode an das Ende des generierten

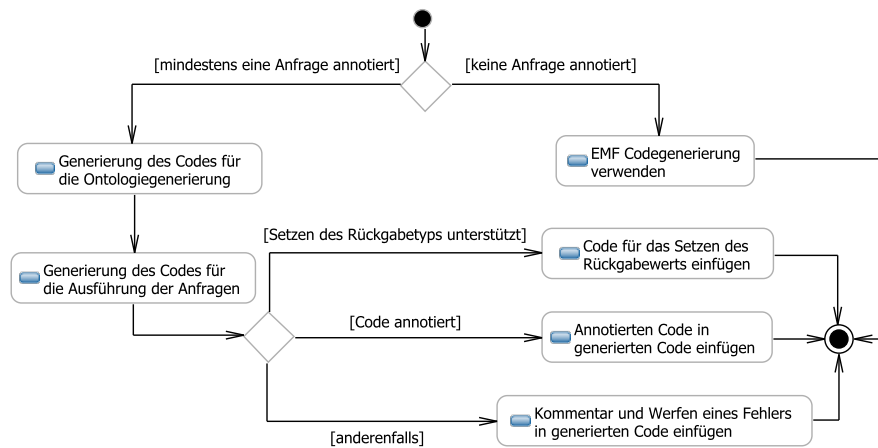


Abb. 6.2. Aktivitätsdiagramm für die Codegenerierung der Ecore Operationen

Codes eingefügt. Beim dritten Fall wird ein Kommentar eingefügt, der den Entwickler darauf hinweist, dass er den Rest der Operation selbst implementieren muss. Nach diesem Kommentar wird Code für das Werfen einer `UnsupportedOperationException` eingefügt. In Abschnitt 6.5.3 wird der generierte Code für mit Anfragen modellierte Operationen genauer beschrieben.

6.4.3 Erwartungen des erweiterten Generators an seine Eingabemodelle

Um die Konformität des generierten Codes mit dem Ecore Modell sicherzustellen und eine effiziente Arbeitsweise des erweiterten Generators zu ermöglichen, stellt der erweiterte Generator einige Bedingungen an seine Eingabemodelle. Diese Erwartungen des implementierten Generators werden in diesem Abschnitt aufgezählt. Da die Berechnung des Generator-Eingabemodells noch nicht implementiert wurde, muss der Entwickler selbst für die Einhaltung dieser Bedingungen sorgen.

Für die sinnvolle Verwendung der durch ein EOF Modell definierten Ontologie ist es notwendig, dass die grundlegenden Beweisführungsprobleme entscheidbar sind. Daher verlangt der Generator, dass die EOF Modelle, die ihm als Eingabemodelle übergeben werden, eine OWL DL Ontologie definieren.

Die EMF Codegenerierung verwendet für Attribute und Referenzen nur dann Listen, wenn sie eine maximale Kardinalität größer eins bzw. ungebun-

dene maximale Kardinalität haben. Die implementierte EOF Codegenerierung verwendet jedoch Listen für alle Attribute und Referenzen, die mit Property Axiomen, deren Einhaltung zur Laufzeit gewährleistet werden soll, verbunden sind. Zur Einhaltung der Konformität mit der EMF Codegenerierung und zur Vermeidung von Problemen durch die Ecore Kardinalität wird daher gefordert, dass alle Attribute und Referenzen mit Property Axiomen im EOF Modell eine ungebundene maximale Kardinalität haben müssen.

Um eine effiziente Arbeitsweise des erweiterten Generators zu ermöglichen, wird folgendes gefordert: Falls ein Attribut bzw. eine Referenz mit Property Axiomen als Parameter für eine Ontologieannotation verwendet wird, so muss eine spezielle Annotation, die auf diese Ontologieannotation verweist, im EOF Modell enthalten sein. Zum Beispiel müsste im EOF Modell für das Szenario an die `orders` Referenz eine Annotation, die auf die Ontologieannotation für die Restriktion `SubObjectPropertyOf(pendingOrders, orders)` verweist, geheftet werden.

Für die korrekte Behandlung von äquivalenten Properties wird gefordert, dass alle zu äquivalenten Properties gehörenden Attribute bzw. Referenzen in den gleichen Klassen enthalten sind. Angenommen es gibt zum Beispiel zwei Referenzen `r1` und `r2` im Modell und `r1` ist in der Klasse `C` enthalten, so muss auch die Referenz `r2` in der Klasse `C` enthalten sein, falls die zu `r1` und `r2` gehörenden Properties als äquivalent deklariert wurden. Außerdem müssen äquivalente Properties im EOF Modell den gleichen Typ und somit auch den gleichen Range haben.

Zudem muss der Entwickler, wie bereits in Abschnitt 6.3 erläutert, selbst dafür sorgen, dass keine Probleme bei der Ontologiegenerierung durch die Speicherung von Objekten in verschiedenen Ressourcen auftreten. Diese Probleme kann der Entwickler dadurch verhindern, dass er zusätzliche Referenzen einfügt oder dafür sorgt, dass alle Objekte einer Modellinstanz in einer Ressource gespeichert werden.

6.5 Verwendung der Ontologie im generierten Code

In diesem Abschnitt wird erläutert, wie die in Abschnitt 5.6 vorgestellte Verwendung der Ontologie implementiert wurde und es wird die Arbeitsweise des zusätzlich generierten Codes beschrieben. Für die Implementierung wurde der zweite Ansatz für die Ontologieverwendung gewählt. Es wird also vor jeder Verwendung der Ontologie die gesamte Ontologie neu generiert. Dies führt allerdings zu einem recht hohen Rechenaufwand für die Aktionen, die die Ontologie verwenden.

Im Unterabschnitt 5.6.1 wird erläutert, wie die Einhaltung von Ontologierestriktionen zur Laufzeit vom generierten Code gewährleistet wird. Der Unterabschnitt 5.6.2 beschäftigt sich mit den zusätzlichen Operationen, die Konsistenzüberprüfungen vornehmen. Schließlich behandelt der Unterabschnitt 5.6.3 den generierten Code für mit Anfragen modellierte Operationen.

6.5.1 Einhaltung von Ontologierestriktionen zur Laufzeit mithilfe von Anfragen

Die in Abschnitt 5.6.1 beschriebene Gewährleistung der Einhaltung der Property Axiome zur Laufzeit, wurde mithilfe von speziellen Listen implementiert. Auf die Attribute und Referenzen mit Property Axiomen kann nur über die Getter-Operationen zugegriffen werden. Diese Getter-Operationen liefern die entsprechende Liste für das Attribut bzw. die Referenz.

Die speziellen Listen werden über das Common-Plugin bereitgestellt. Die `add` bzw. `remove` Operationen dieser Listen verändern nur dann die Liste, wenn dies nicht zu einer Verletzung der Property Axiome führt bzw. wenn die Verletzung durch das Ausführen weiterer Änderungen verhindert wird.

Für die Implementierung dieser Listen wurde die Klasse `PREList` erzeugt. Diese Liste implementiert die EMF Interfaces `EList` und `InternalEList`. Dadurch kann mit den speziellen Listen problemlos eine andere `EList` ersetzt werden. Die `PREList` Klasse enthält Standardimplementierungen für alle in ihr enthaltenen Operationen. Für die verschiedenen Arten von Property Axiomen und für Kombinationen von Property Axiomen wurden Unterklassen von `PREList` erzeugt, die für die Einhaltung der entsprechenden Ontologierestriktionen sorgen und dafür teilweise die Standardimplementierungen verwenden.

Die `PREList` Klasse wurde so entworfen, dass sie eine andere Liste übergeben bekommt, an die sie verschiedene Aufgaben delegiert. Zum Beispiel wird die Speicherung der Listenelemente an diese Liste delegiert. Die `add` und `remove` Operationen der `PREList`en verwenden die Ontologie, um zu bestimmen, ob eine Änderung verboten werden muss, und um weitere Änderungen zu inferieren, die zusätzlich ausgeführt werden müssen. Dafür verwenden die Listen `Construct` Anfragen auf der generierten Ontologie. Die Ergebnisse dieser Anfragen legen fest, welche Änderungen vorgenommen werden. Dabei wird davon ausgegangen, dass zur Zeit des Aufrufs der `add` bzw. `remove` Operation kein Property Axiom verletzt war.

Zur Verdeutlichung enthalten Listing 6.1 und Listing 6.2 Muster für Anfragen, die die `add` und `remove` Methoden einer `TransitiveEList` verwenden. Listing 6.1 stellt eine Musteranfrage für eine `add` Methode und Listing 6.2 eine Musteranfrage für eine `remove` Methode dar.

```

1  CONSTRUCT {
2      *owner *property *object .
3      ?x *property *object .
4      *owner *property ?y .
5      ?x *property ?y .
6      ?a *property ?c .
7      ?a *property *object .
8      ?a *property ?y .
9      *owner *property ?c .
10     ?x *property ?c
11 }
```

```

12 WHERE {
13   OPTIONAL {
14     ?a *property *owner
15   } OPTIONAL {
16     *object *property ?c
17   } OPTIONAL {
18     ?x owl:sameAs *owner
19   } OPTIONAL {
20     ?y owl:sameAs *object
21   }
22 }

```

Listing 6.1. Musteranfrage für eine add Methode einer TransitiveEList

```

1  CONSTRUCT {
2    *owner *property *object .
3    ?x *property *object .
4    *owner *property ?y .
5    ?x *property ?y
6  }
7  WHERE {
8    OPTIONAL {
9      *owner *property ?v .
10     ?v *property *object
11     OPTIONAL {
12       ?ow owl:sameAs ?v
13       FILTER ( *owner = ?ow )
14     } OPTIONAL {
15       ?ob owl:sameAs ?v
16       FILTER ( *object = ?ob )
17     }
18     FILTER ( ! ( *owner = ?v || *object = ?v ||
19                 bound ( ?ow ) || bound ( ?ob ) ) )
20   }
21   FILTER ( ! ( bound ( ?v ) ) )
22   OPTIONAL {
23     ?x owl:sameAs *owner
24   } OPTIONAL {
25     ?y owl:sameAs *object
26   }
27 }

```

Listing 6.2. Musteranfrage für eine remove Methode einer TransitiveEList

In den Musteranfragen steht `*owner` für die IRI des Objekts, dessen Property die Liste darstellt, `*property` für die IRI der Property und `*object` für die

IRI des Objekts, das hinzugefügt bzw. entfernt werden soll. Die IRIs und deren zugehörige Entitäten werden jeweils über die bei der Ontologiegenerierung erstellte Abbildung zwischen Entitäten und IRIs gewonnen.

Die Anfrage der `add` Methode liefert als Ergebnis immer, dass das Objekt hinzugefügt werden kann. Zudem inferiert diese Anfrage die `PropertyAssertions`, die zusätzlich hinzugefügt werden müssen, um die Einhaltung der Transitivität zu gewährleisten. Die Anfrage der `remove` Methode prüft, ob das Objekt entfernt werden darf, indem es untersucht, ob die entsprechende `PropertyAssertion` für die Einhaltung der Transitivität benötigt wird. Zudem inferiert diese Anfrage, welche `PropertyAssertions` ebenfalls entfernt werden müssen.

Nachdem die Ergebnisse der Anfragen feststehen, nimmt die `PREList` die Änderungen vor oder wirft einen Fehler, falls die Aktion nicht ausgeführt werden darf. Sollten für das Ausführen der zusätzlichen Aktionen andere Listen verändert werden müssen, so wird mithilfe der Reflektionsmöglichkeiten von EMF auf diese Listen zugegriffen. Die `PREList`en bieten für solche Zugriffe auch `add` und `remove` Operationen, die keine erneute Überprüfung über Anfragen ausführen, an. Um die Reflexivität einer `Property` einhalten zu können, sorgen die Konstruktoren entsprechender Klassen dafür, dass ein neu erzeugtes Objekt in alle reflexiven `Properties` eingefügt wird.

Mithilfe der bisher vorgestellten Methoden wird die Einhaltung der `Property` Axiome, die nur eine `Property` betreffen, und der `DisjointObjectProperties` Axiome gewährleistet. `EquivalentProperties` Axiome werden dadurch eingehalten, dass Änderungen stets an allen äquivalenten `Properties` vorgenommen werden. Analog dazu werden `InverseObjectProperty` Axiome dadurch eingehalten, dass Änderungen auch an den inversen `Properties` vorgenommen werden. Für `SubObjectPropertyOf` Axiome werden in die `SubProperty` eingefügte `Assertions` auch in die `SuperProperty` eingefügt und aus der `SuperProperty` entfernte `Assertions` auch aus der `SubProperty` entfernt.

Für Kombinationen mehrerer `Property` Axiome werden entweder Listen verwendet, die für diese Kombination entwickelt wurden und entsprechend kombinierte Anfragen enthalten, oder es werden für die Kombination vorgesehene Listen entsprechend miteinander verkettet. Diese verketteten Listen arbeiten so, dass jeweils eine Liste die Gewährleistung der Einhaltung bestimmter `Property` Axiome übernimmt und die Gewährleistung der Einhaltung der restlichen `Property` Axiome an die anderen Listen der Kette delegiert.

Zum Beispiel wird die Einhaltung einer Kombination von `EquivalentObjectProperties` und `TransitiveObjectProperty` gewährleistet, indem die Liste, die die Einhaltung des `EquivalentObjectProperties` Axioms gewährleistet, die Berechnung der nötigen Änderungen an eine `TransitiveEList` delegiert, und anschließend die nötigen Änderungen für alle äquivalenten `Properties` vornimmt.

Die Einhaltung der Ontologierestriktionen für `containment` Referenzen wird noch nicht vom implementierten Prototyp zur Laufzeit gewährleistet, da der Mechanismus für `containment` Referenzen eventuell zu dem Entfernen

von Objekten führen kann. Um dies zu verhindern müsste dieser Mechanismus angepasst werden.

6.5.2 Verwendung von Konsistenzüberprüfungen zur Validitätsprüfung

Es gibt mehrere Ontologieausdrücke, deren Einhaltung nicht zur Laufzeit durch den generierten Code gewährleistet wird. Anstatt zu verhindern, dass diese Restriktionen verletzt werden, wird es gestattet die durch diese Ontologieausdrücke definierten Restriktionen zwischenzeitlich zu verletzen und zu einem Zeitpunkt zu überprüfen, dass keine Ontologierestriktionen verletzt sind.

Dafür enthält der generierte Code Operationen, die überprüfen, ob alle Ontologierestriktionen eingehalten sind. Eine solche Operation generiert unter Annahme einer geschlossenen Welt eine Ontologie für den aktuellen Zustand und führt mithilfe des Pellet Beweisers eine Konsistenzüberprüfung auf dieser Ontologie durch. Das Ergebnis dieser Überprüfung wird von der Operation zurückgeliefert.

Möchte man überprüfen, ob alle Ontologierestriktionen eingehalten sind, so ruft man eine solche Operation auf. Ist der Entwickler darauf angewiesen, dass alle Restriktionen zu einem Zeitpunkt eingehalten sind, so kann er dies zunächst mit diesen Operationen überprüfen und anschließend angemessen darauf reagieren.

6.5.3 Verwendung von Anfragen zur Modellierung des Verhaltens von Operationen

In diesem Abschnitt wird der generierte Code für mit Anfragen modellierte Operationen beschrieben. Der implementierte Prototyp bietet drei verschiedene Möglichkeiten für die Modellierung des Verhaltens einer Operation mit Anfragen an:

1. Die Operation wird mit genau einer Anfrage und keinem Programmcode annotiert. Diese Anfrage ist eine Select oder eine Ask Anfrage und der Rückgabotyp der Operation wurde passend gewählt.
2. Die Operation wird sowohl mit einer oder mehreren Anfragen als auch mit Programmcode annotiert.
3. Die Operation wird mit genau einer Anfrage und keinem Programmcode annotiert. Der Anfragetyp oder der Rückgabotyp der Operation wird nicht passend zu den Bedingungen der ersten Möglichkeit gewählt.

Bei allen drei Möglichkeiten enthält der generierte Code der Operation Code für die Ontologiegenerierung und Code zum Ausführen der Anfragen auf einer generierten Ontologie. In dem Code für die Ausführung einer annotierten Anfrage wird der Pfad der Anfrage aus der Annotation einer Variablen zugewiesen.

Da innerhalb der annotierten Anfragen mit `?self` auf das Objekt, auf dem die Operation ausgeführt wird, verwiesen werden kann, ist Code zum Erzeugen einer neuen Anfrage im generierten Code enthalten. Diese neue Anfrage entspricht der annotierten Anfrage mit dem Unterschied, dass alle Vorkommen von `?self` in der annotierten Anfrage durch die IRI des Objekts, auf dem die Operation ausgeführt wird, ersetzt wurden. Dabei wird die IRI über die bei der Ontologiegenerierung erstellte Abbildung zwischen IRIs und Objekten ermittelt. Diese neue Anfrage wird auf der generierten Ontologie ausgeführt.

Der Code nach der Ausführung der Anfrage hängt von der gewählten Modellierungsmöglichkeit ab. Bei der zweiten Möglichkeit folgt lediglich der annotierte Programmcode. Bei der dritten Möglichkeit folgt ein Kommentar, der den Entwickler darauf hinweist, dass er den Rest der Operation selbst implementieren muss, und zudem Code für das Werfen einer `UnsupportedOperationException`. Bei der ersten Möglichkeit folgt der Code zum Setzen des Rückgabewerts.

Der Code zum Setzen des Rückgabewerts beinhaltet lediglich das Zurückliefern des Anfrageergebnisses als Rückgabewert, falls keine Umwandlung des Anfrageergebnisses notwendig ist. Anderenfalls besteht der Code zum Setzen des Rückgabewerts aus Code zum Umwandeln des Anfrageergebnisses und aus dem Zurückliefern des umgewandelten Ergebnisses als Rückgabewert.

Für die Umwandlung des Ergebnisses von `Select` Anfragen stellt die `QueryHelper` Klasse des Common-Plugins mehrere Operationen bereit. Diese Operationen substituieren die IRIs, die im Ergebnis enthalten sind, durch die Entitäten, die von den IRIs repräsentiert werden. Dafür wird wiederum die IRI über die bei der Ontologiegenerierung erstellte Abbildung zwischen IRIs und Objekten ermittelt. Zudem wandeln diese Operationen das Ergebnis der `Select` Anfrage in eine Liste um. Dabei handelt es sich entweder um eine Liste von Objekten oder um eine Liste von Maps.

Die Operationen, die eine Liste von Objekten liefern, sind für `Select` Anfrage mit genau einer `Select` Variable vorgesehen. Diese Listen enthalten die Belegungen dieser Variable im Anfrageergebnis. Die Operationen, die eine Liste von Maps liefern, sind für alle `Select` Anfragen vorgesehen. In den Maps sind die Variablenbelegungen eines Ergebnissatzes gespeichert. Für den Zugriff auf die Variablenbelegungen werden die Stringrepräsentationen der `Select` Variablen als Schlüssel für die Maps verwendet.

Der Code für die Umwandlung ruft eine dieser Umwandlungsoperationen auf. Zudem kann der Code für die Umwandlung Code enthalten, der das Ergebnis der Umwandlungsoperation zu dem Rückgabebetyp der Operation castet. Auf diese Weise wird zum Beispiel das Ergebnis der Anfrage für die `getShipToOrders` Operation des Szenarios mithilfe einer Umwandlungsoperation und mithilfe von Casts in eine Liste von `Order` Objekten umgewandelt.

Prototyp

In diesem Kapitel wird der entwickelte Prototyp vorgestellt. Dazu wird gezeigt, wie der Prototyp für das Szenario aus Kapitel 2 genutzt werden kann. Zunächst wird in Abschnitt 7.1 erläutert, wie mit dem Prototyp ein EOF Modell für das Szenario erstellt wird. Anschließend wird in Abschnitt 7.2 beschrieben, wie aus einem EOF Modell mit dem Prototyp Code generiert wird. Abschnitt 7.3 behandelt die Bearbeitung des Codes. Schließlich wird in Abschnitt 7.4 die Verwendung des generierten Editors erläutert.

7.1 EOF Modell erstellen

Zur Erstellung eines EOF Modells ist es sinnvoll zunächst ein Ecore Modell für den generierten Code zu erstellen. Dafür kann man zum Beispiel zunächst ein Ecore Diagramm erstellen. Die Abbildung 7.1 zeigt einen Screenshot von der Erstellung eines Ecore Diagramms für das Szenario. Um die in Abschnitt 6.4.3 beschriebenen Erwartungen des Generators zu erfüllen wurden ein paar kleinere Änderungen an dem Modell für das Szenario vorgenommen. Die maximale Kardinalität der `billTo` und `shipTo` Referenzen wurde von eins auf ungebunden gesetzt.

Zudem wurde der `Supplier` Klasse eine weitere Referenz `addresses` hinzugefügt. Diese weitere Referenz ordnet einem Lieferanten verschiedene Adressen zu. Es wird davon ausgegangen, dass die `billTo` und `shipTo` Referenzen einer Bestellung nur auf Adressen des zugehörigen Lieferanten der Bestellung verweisen. Diese Referenz wurde eingefügt, um das in Abschnitt 6.3 beschriebene Problem bei der Ontologiegenerierung für Objekte verschiedener Ecore Ressourcen zu vermeiden.

Nachdem wir nun ein Ecore Modell für das Szenario erstellt haben, erzeugen wir die Anfragen, die das Verhalten der Operationen modellieren sollen. Die Modellierung von Operationen mit Anfragen wurde in Abschnitt 5.6.3 konzeptionell beschrieben. Wir wählen die erste Möglichkeit für die Modellierung der Operationen. Bei dieser Möglichkeit wird das Verhalten der Ope-

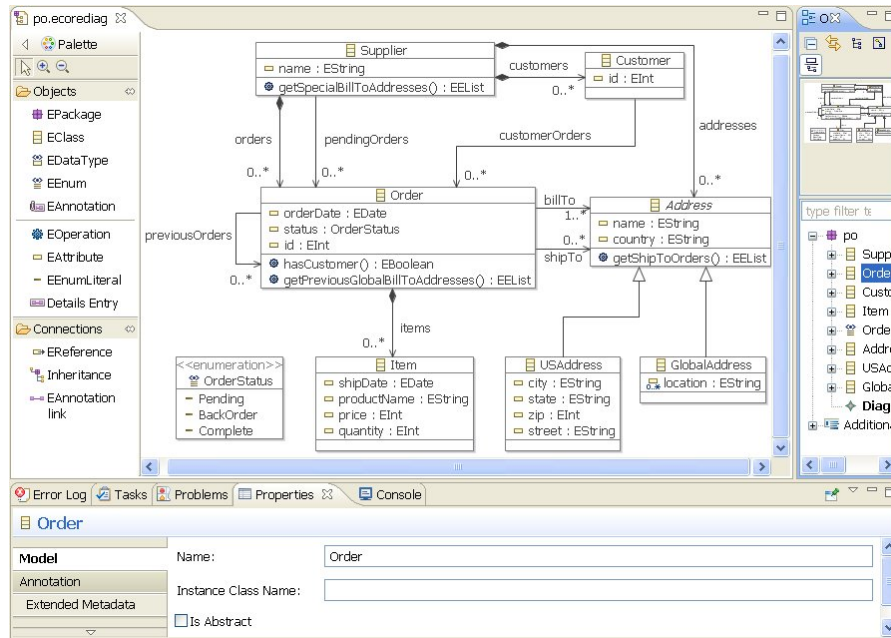


Abb. 7.1. Ecore Diagramm für das Szenario

ration vollständig durch die Anfrage modelliert und der generierte Code implementiert die Operation vollständig. Die Rückgabetyperen der Operationen wurden bereits bei der Erstellung des Ecore Modells passend gewählt. So liefert zum Beispiel die `getShipToOrders` Operation eine Liste von Bestellungen und die `hasCustomer` Operation einen booleschen Wert.

In den Anfragen, die die Operationen modellieren, verweist die `?self` Variable jeweils auf das Objekt, auf dem die Operation aufgerufen wurde. Die Abbildung 7.2 zeigt einen Screenshot mit der in einem Editor geöffneten SPARQL Anfrage für die `getShipToOrders` Operation. Für diese Operation wurde eine `Select` Anfrage verwendet, die alle Bestellungen zurückliefert, deren `shipTo` Referenz auf das Adressenobjekt verweist, auf dem die Operation aufgerufen wurde.

Die folgenden Anfragen verwenden die gleichen Präfixdeklarationen, wie die `getShipToOrders` Anfrage. Daher werden die Präfixdeklarationen nicht erneut aufgeführt. Die SPARQL Anfrage für die `hasCustomer` Operation ist in Listing 7.1. Für diese Operation verwenden wir eine `Ask` Anfrage, mit der wir prüfen, ob die `customerOrders` Referenz eines Kunden auf die aktuelle Bestellung verweist.

```

1  ASK
2  WHERE {
3      ?customer :Customer.customerOrders ?self

```

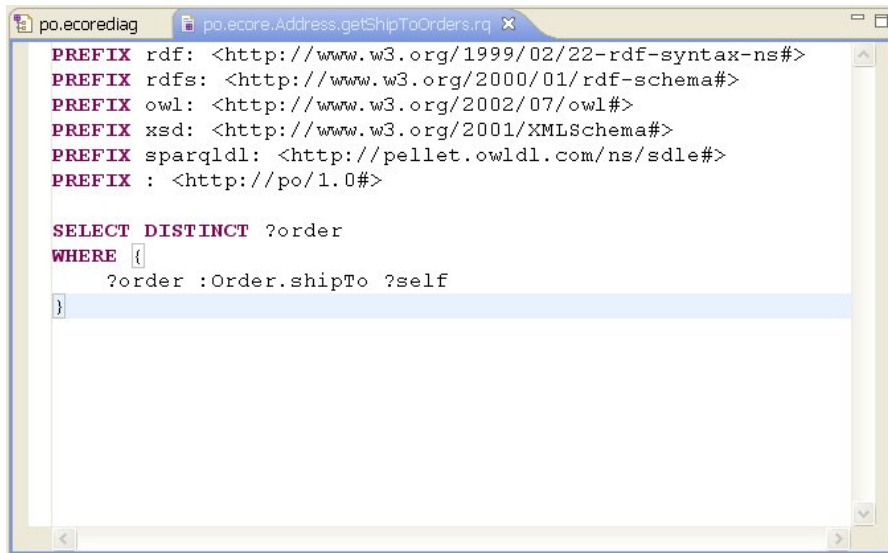


Abb. 7.2. SPARQL Anfrage für die getShipToOrders Operation

```
4 }
```

Listing 7.1. SPARQL Anfrage für die hasCustomer Operation

Nun fahren wir mit der Erstellung der SPARQL Anfrage für die Operation `getPreviousGlobalBillToAddresses` fort. Für diese Operation verwenden wir wiederum eine Select Anfrage. In der Anfrage werden mit der dritten Zeile die Bestellungen, auf die die `previousOrders` Referenz der aktuellen Bestellung verweist, ermittelt und an die `?prevOrder` Variable gebunden. Die Anfrage liefert alle Adressen, auf die die `billTo` Referenz einer `?prevOrder` Bestellung verweist und die vom Typ `GlobalAddress` sind.

```

1 SELECT DISTINCT ?address
2 WHERE {
3   ?self :Order.previousOrders ?prevOrder.
4   ?prevOrder :Order.billTo ?address.
5   ?address rdf:type :GlobalAddress
6 }

```

Listing 7.2. SPARQL Anfrage für die getPreviousGlobalBillToAddresses Operation

Die SPARQL Anfrage für die `getSpecialBillToAddresses` Operation ist in Listing 7.3 dargestellt. Die Zeilen 3 und 4 sorgen dafür, dass die Anfrage lediglich die Adressen zurückliefert, auf die die `billTo` Referenz einer Bestellung, die über die `orders` Referenz einem Lieferanten zugeordnet ist, verweist. Zeile 5 fordert als zusätzliches Kriterium, dass die `country` Referenz

der Bestellung die Kennung für Deutschland enthalten muss. Dieses Kriterium kann, wie in Abschnitt 2.2 gefordert, zur Laufzeit durch ein anderes Kriterium ersetzt werden, indem die Anfrage angepasst wird. Es muss lediglich gewährleistet sein, dass die veränderte Anfrage weiterhin nur die Ergebnisse einer Variablen liefert und dass diese Variable immer nur mit Adressen belegt ist.

```

1  SELECT DISTINCT ?address
2  WHERE {
3      ?self :Supplier.orders ?order .
4      ?order :Order.billTo ?address .
5      ?address :Address.country "GER" ^^ xsd:string
6  }
```

Listing 7.3. SPARQL Anfrage für die `getSpecialBillToAddresses` Operation

Nachdem wir das Ecore Modell und die Anfragen für die Operationen erstellt haben, wollen wir mit dem Erstellen der Ontologieannotationen fortfahren, um die in Abschnitt 2.2 geforderten Ontologierestriktionen in das Modell aufzunehmen. Die unterstützten Ontologieannotationen wurden in Abschnitt 6.2.1 beschrieben. Für die Erstellung dieser Annotationen können wir sowohl den Text Ecore Editor des TwoUse Toolkits als auch den Ecore Editor verwenden.

Die Abbildung 7.3 zeigt den Text Ecore Editor. In diesem Editor werden die Annotationen in eckigen Klammern notiert. Auf dem Screenshot sind zwei Ontologieannotationen zu sehen: eine `transitive` Annotation und eine `subPropertyOf` Annotation.

Die `transitive` Annotation wurde an die `previousOrders` Referenz geheftet und fordert, dass diese Referenz transitiv ist. Die `subPropertyOf` Annotation fordert, dass die `pendingOrders` Referenz eine `SubProperty` der `orders` Referenz darstellt. Diese `subPropertyOf` Annotation wurde an die `pendingOrders` Referenz des Modells geheftet und verweist mit ihrer `references` Referenz auf die `orders` Referenz des Modells.

Zusätzlich zu den bereits erwähnten Annotationen erstellen wir analog noch einige weitere Annotationen für das Modell. Wir erstellen noch eine `disjointproperties` Annotation, die fordert dass die Referenzen `billTo` und `shipTo` disjunkte `Properties` darstellen. Außerdem benötigen wir wegen der Erwartungen des Generators (Abschnitt 6.4.3), noch jeweils eine `refAnnotation` Annotation für die Referenzen `orders` und `shipTo` die auf die `subPropertyOf` Annotation bzw. die `disjointproperties` Annotation verweist.

Da wir die maximalen Kardinalitäten der Referenzen `billTo` und `shipTo` auf ungebunden abändern mussten, fügen wir zudem für diese Referenzen jeweils eine `functional` Annotation ein. Diese Annotation soll die maximale Kardinalität eins abbilden, die eigentlich für die Referenzen gewählt wurde.

Die Abbildung 7.4 zeigt das EOF Modell im Ecore Editor. In dem Screenshot kann man sehen, wie die im Text Ecore Editor erstellten Annotationen

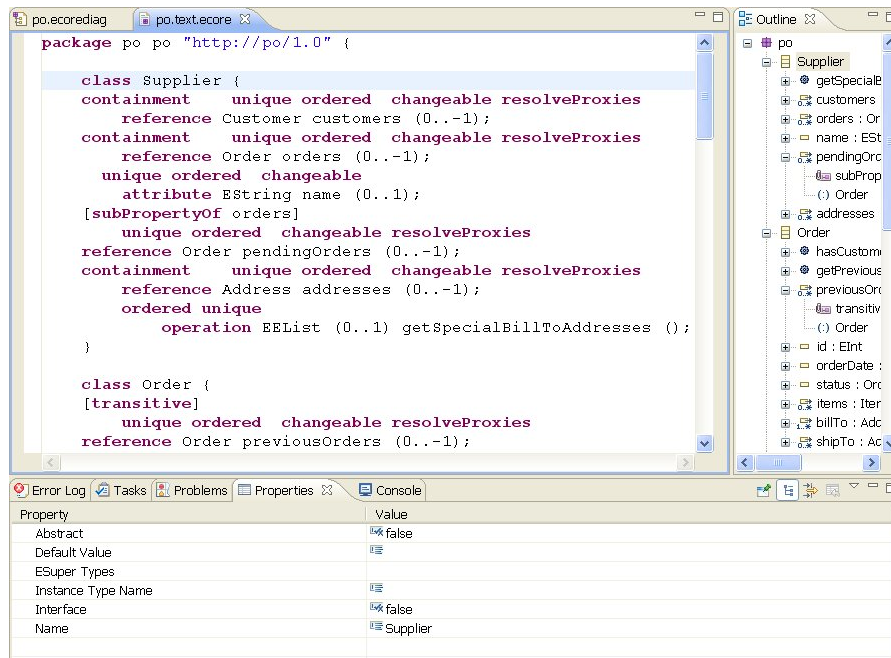


Abb. 7.3. Erzeugen des EOF Modells mit dem Text Ecore Editor

im Ecore Modell dargestellt werden. Möchte man lieber den Ecore Editor als den Text Ecore Editor verwenden, so erstellt man die Annotationen analog im Ecore Editor.

Für die Fertigstellung des EOF Modells müssen nur noch die Anfragen, die die Operationen modellieren sollen, über Anfrageannotationen mit den entsprechenden Operationen verbunden werden. Die Anfrageannotationen wurden in Abschnitt 6.2.4 beschrieben. Dementsprechend heften wir an die vier Operationen jeweils eine Anfrageannotation. In die `source` Referenz der Annotation kodieren wir den Pfad der entsprechenden Anfragen. Da die Anfragen in den Ordner erstellt wurden, in dem auch das EOF Modell ist, genügt es den Namen der Anfrage in diese Referenz zu schreiben. Wäre dies nicht der Fall so müsste der Pfad angegeben werden.

Die Abbildung 7.5 zeigt einen Screenshot, auf dem die Anfrageannotation für die `getSpecialBillToAddresses` Operation zu sehen ist. Die `references` Referenz dieser Annotation verweist auf das Modell für die erstellte Anfrage. Dadurch wird die Ressource der Anfrage, wie im Screenshot zu sehen ist, automatisch geladen, wenn die Anfrageannotation angeklickt wird. Dies ermöglicht es, dass man sich direkt das Modell der Anfrage ansehen kann.

Nachdem wir nun auch die Anfrageannotationen eingegeben haben ist unser EOF Modell für das Szenario vollständig. Dieses EOF Modell definiert bereits die TBox der Ontologie für das Szenario. Listing 7.4 enthält einen Teil

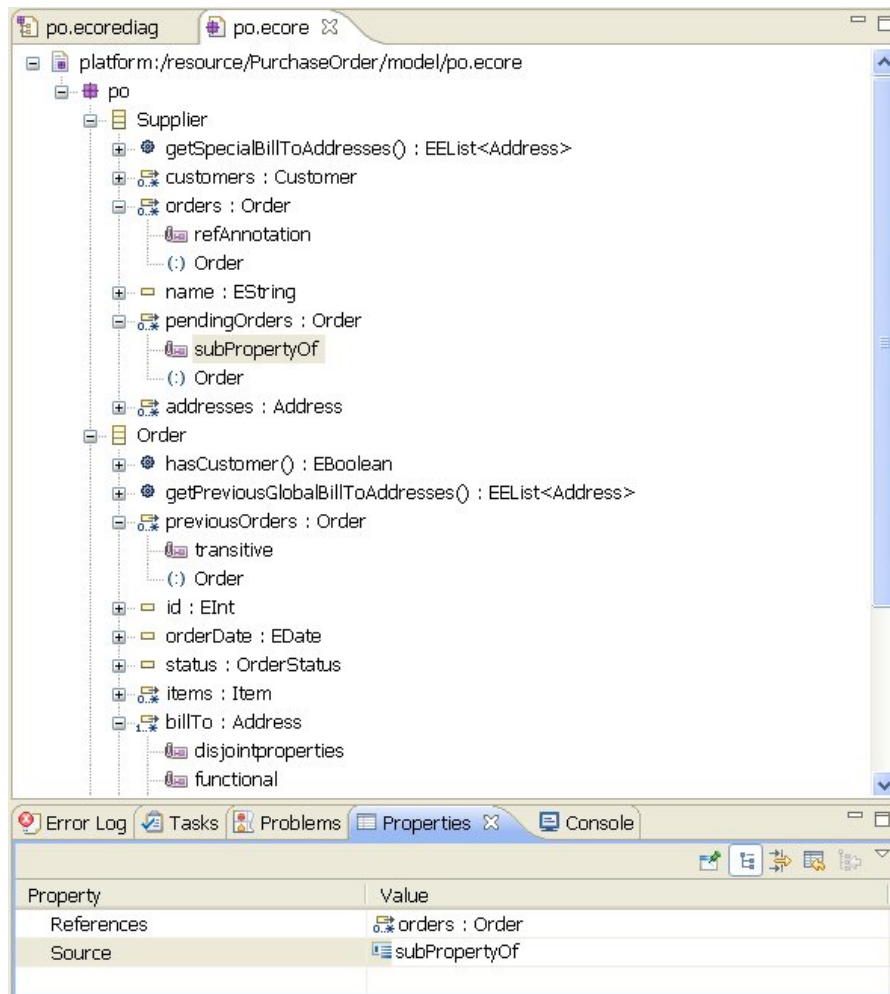


Abb. 7.4. EOF Modell im Ecore Editor

der TBox der für das EOF Modell generierten Ontologie. Wie diese Ontologie aus dem EOF Modell generiert wird, wurde in Abschnitt 6.3 beschrieben.

```

1 Declaration(Class(<http://po/1.0#Order>))
2 Declaration(
3   DataProperty(<http://po/1.0#Order.id>))
4 SubClassOf(<http://po/1.0#Order>
5   DataMaxCardinality(1 <http://po/1.0#Order.id>
6     xsd:int))
7 FunctionalDataProperty(<http://po/1.0#Order.id>)
8 DataPropertyDomain(<http://po/1.0#Order.id>

```

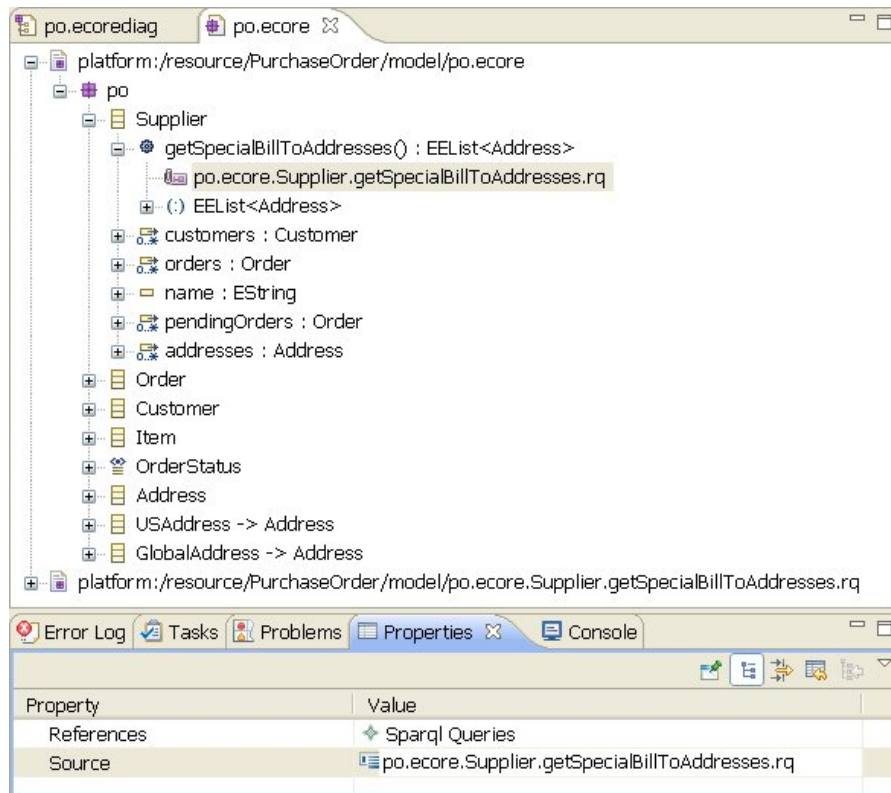


Abb. 7.5. Anfrageannotation für die getSpecialBillToAddresses Operation

```

9      <http://po/1.0#Order >)
10  DataPropertyRange (<http://po/1.0#Order.id>
11      xsd:int)
12  ObjectPropertyDomain (<http://po/1.0#status >
13      <http://po/1.0#Order >)
14  ObjectPropertyRange (<http://po/1.0#status >
15      <http://po/1.0#OrderStatus >)
16  FunctionalObjectProperty (<http://po/1.0#status >)
17  TransitiveObjectProperty (
18      <http://po/1.0#Order.previousOrders >)
19  Declaration (
20      ObjectProperty (
21          <http://po/1.0#Order.previousOrders >))
22  ObjectPropertyDomain (
23      <http://po/1.0#Order.previousOrders >
24      <http://po/1.0#Order >)
25  ObjectPropertyRange (

```

```

26     <http://po/1.0#Order.previousOrders>
27     <http://po/1.0#Order>)
28 Declaration (
29     ObjectProperty (<http://po/1.0#Order.items>))
30 ObjectPropertyDomain (<http://po/1.0#Order.items>
31     <http://po/1.0#Order>)
32 ObjectPropertyRange (<http://po/1.0#Order.items>
33     <http://po/1.0#Item>)
34 DisjointObjectProperties (
35     <http://po/1.0#Order.billTo>
36     <http://po/1.0#Order.shipTo>)
37 FunctionalObjectProperty (
38     <http://po/1.0#Order.billTo>)
39 Declaration (
40     ObjectProperty (<http://po/1.0#Order.billTo>))
41 SubClassOf (<http://po/1.0#Order>
42     ObjectMinCardinality (1
43     <http://po/1.0#Order.billTo>
44     <http://po/1.0#Address>))
45 ObjectPropertyDomain (<http://po/1.0#Order.billTo>
46     <http://po/1.0#Order>)
47 ObjectPropertyRange (<http://po/1.0#Order.billTo>
48     <http://po/1.0#Address>)
49 FunctionalObjectProperty (
50     <http://po/1.0#Order.shipTo>)
51 Declaration (
52     ObjectProperty (<http://po/1.0#Order.shipTo>))
53 ObjectPropertyDomain (<http://po/1.0#Order.shipTo>
54     <http://po/1.0#Order>)
55 ObjectPropertyRange (<http://po/1.0#Order.shipTo>
56     <http://po/1.0#Address>)
57 )

```

Listing 7.4. Teil der für das EOF Modell generierten Ontologie

Bei der Ontologiegenerierung wurden die Optionen so gewählt, dass auch zusätzliche Ontologieaxiome aus dem EOF Modell hergeleitet werden. So wurde beispielsweise die `FunctionalDataProperty` Restriktion für das `id` Attribut (Zeile 7) der Bestellungen aus der maximalen Kardinalität dieses Attributs im EOF Modell hergeleitet. Ebenso wurde die Restriktion in den Zeilen 41 – 44 aus der minimalen Kardinalität der `billTo` Referenz hergeleitet.

Somit kann man sich die Eingabe einiger Ontologierestriktionen ersparen, indem man die Optionen für die Ontologiegenerierung passend wählt. Der größte Teil der Ontologie wird implizit durch das `Ecore` Modell angegeben, sodass wir insgesamt nur einen kleinen Teil der Ontologie explizit durch die Ontologieannotationen angeben mussten.

7.2 Codegenerierung

Die Codegenerierung mit dem Prototyp erfolgt analog zur EMF Codegenerierung. Man muss lediglich beachten, dass das EOF Modell alle Erwartungen des Generators (Abschnitt 6.4.3) erfüllen muss. Wie bei der EMF Codegenerierung erstellt man zunächst aus der Ecore Datei für das Modell eine GenModel Datei. In dem GenModel setzt man die Optionen für den Generator. Zum Beispiel kann man über die Optionen angeben, wohin der generierte Code geschrieben wird.

Für das Wechseln zwischen EOF Codegenerierung und EMF Codegenerierung werden zwei Aktionen angeboten. Diese Aktionen verändern die Werte der GenModel Optionen so, dass die EOF Codegenerierung bzw. die EMF Codegenerierung verwendet wird. Die Abbildung 7.6 zeigt das GenModel und das Kontextmenü mit diesen Aktionen.

In dieser Abbildung ist ebenfalls zu sehen, dass für die Codegenerierung die üblichen EMF Aktionen zur Verfügung stehen. Der Model Code, der Edit Code, der Editor Code und der Test Code kann jeweils separat generiert werden. Zudem ist es möglich diesen Code komplett mit einer Aktion zu generieren.

Falls die Codegenerierung auf die EOF Codegenerierung gestellt wurde, wird der Code, wie in Abschnitt 6.4 beschrieben, generiert. In diesem Fall wird, wie in Abschnitt 6.5.1 beschrieben, die Einhaltung einiger Ontologierestriktionen zur Laufzeit vom generierten Code gewährleistet. Außerdem enthält der Code Operationen für Konsistenzüberprüfungen, gemäß Abschnitt 6.5.2, und implementiert die mit Anfragen modellierten Operationen, wie in Abschnitt 6.5.3 beschrieben.

7.3 Bearbeitung des generierten Codes

Der Prototyp bietet für die Bearbeitung des generierten Codes die Bearbeitungsmöglichkeiten des EMF Systems an. So ist es möglich Code einzufügen, zu löschen, oder auch anzupassen. Zudem werden die Coderegenerierungsmöglichkeiten von EMF auch vom Prototyp unterstützt. Somit ist es möglich den bearbeiteten Code mit neu generiertem Code zu vermischen.

7.4 Verwendung des generierten Editors

Abschließend wollen wir die Arbeit mit dem generierten Editor betrachten. Die Abbildung 7.7 zeigt einen Screenshot von dem für das EOF Modell des Szenarios generierten Editor.

Der generierte Editor bietet grundsätzlich die gleichen Funktionen an, die auch die mit der EMF Codegenerierung erstellten Editoren anbieten. Der generierte Editor ermöglicht es Instanzen von Modellelementen zu erzeugen und

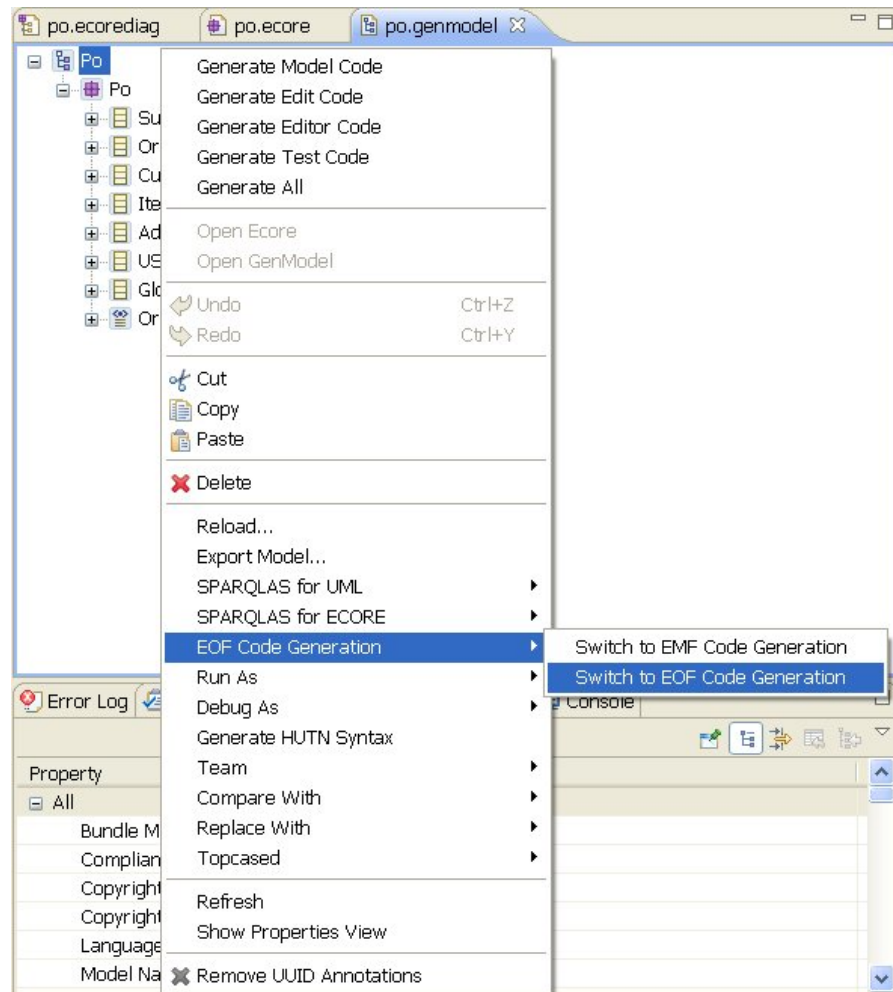


Abb. 7.6. Codegenerierung für das Szenario

deren Attribut- und Referenzwerte auszulesen und abzuändern. Zudem erlaubt der Editor das Speichern erzeugter Modellinstanzen und das Laden gespeicherter Modellinstanzen.

Das Verhalten des Editors ist allerdings unterschiedlich, wenn die EOF Codegenerierung gewählt wurde, da sich in diesem Fall der Model Code unterscheidet. Dadurch ist es zum Beispiel nicht möglich im Generator die Transitivität der `previousOrders` Referenz zu verletzen.

In Abbildung 7.7 ist zu sehen, dass gerade der Liste der `previousOrders` Referenz der Bestellung 5 die Bestellung 4 hinzugefügt wird. Die Abbildung 7.8 zeigt das Ergebnis dieser Aktion. Es wurde nicht nur die Bestellung 4

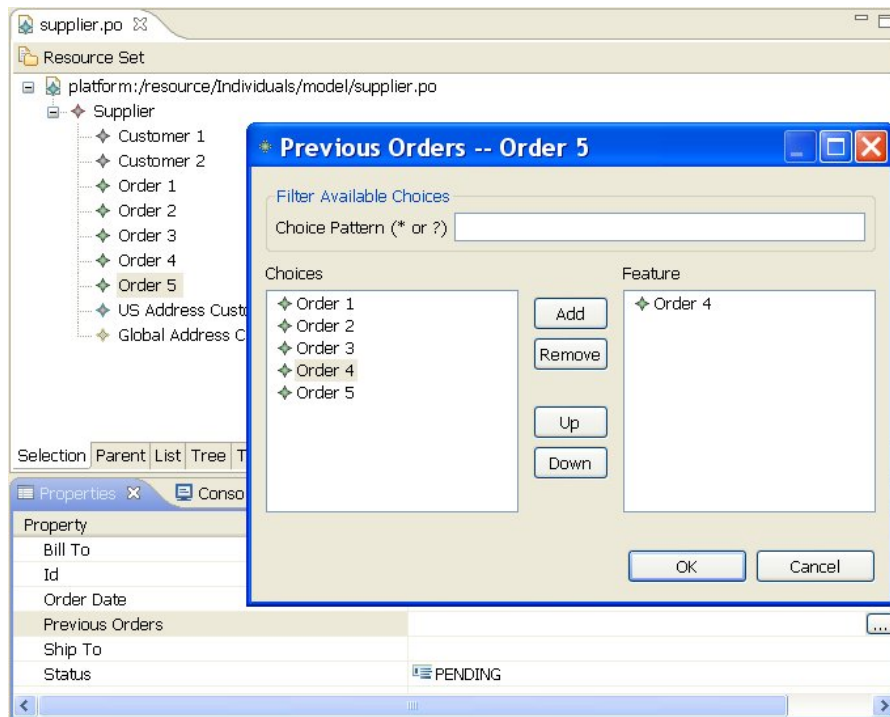


Abb. 7.7. Hinzufügen einer PropertyAssertion für die `previousOrders` Referenz

hinzugefügt, sondern auch die Bestellung 3. Diese Bestellung wurde zusätzlich hinzugefügt, weil die `previousOrders` Referenz der Bestellung 4 bereits auf diese Bestellung verwies. Somit musste zur Einhaltung der Transitivität auch die Bestellung 3 zur Liste der `previousOrders` Referenz der Bestellung 5 hinzugefügt werden.

Ebenso ist es nach dieser Aktion nicht möglich nur die Bestellung 3 aus der Liste der `previousOrders` Referenz der Bestellung 5 zu entfernen. Der Editor würde diese Änderung nicht vornehmen, da diese Änderung die Transitivität verletzen würde. Um die Bestellung 3 wieder zu entfernen muss zuerst entweder die Bestellung 4 aus der Liste der Bestellung 5 oder die Bestellung 3 aus der Liste der Bestellung 4 entfernt werden. Die in Abschnitt 6.5.1 beschriebene Gewährleistung der Einhaltung einiger Ontologierestriktionen durch den generierten Code sorgt für dieses Verhalten. Dadurch kann man den generierten Editor verwenden, um die ABox einer Ontologie vom Editor unterstützt zu füllen, sodass man sich um die Einhaltung einiger Ontologierestriktionen nicht selbst kümmern muss.

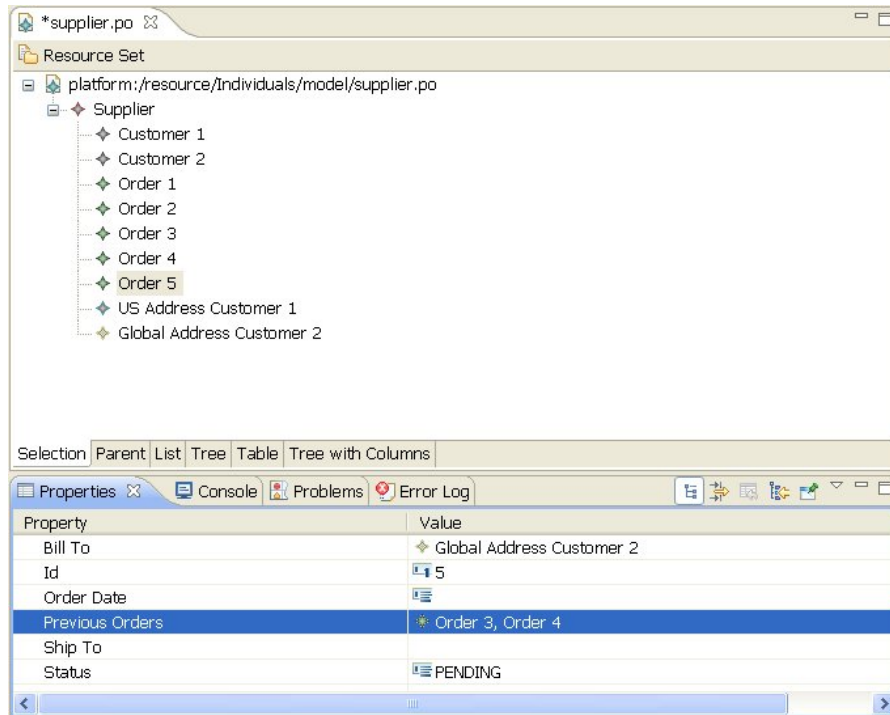


Abb. 7.8. Ergebnis des Hinzufügens der PropertyAssertion

Zusammenfassung und Ausblick

8.1 Zusammenfassung

In dieser Arbeit haben wir einen Ansatz für eine Kombination aus modellgetriebener und ontologiegetriebener Softwareentwicklung präsentiert. Für die Entwicklung dieses Ansatzes wurde die Sicht eines Entwicklers aus dem modellgetriebenen Bereich angenommen, um eine gute Akzeptanz in der relativ großen Gemeinde der modellgetriebenen Entwickler zu erreichen. Daher wurde der Ansatz so konzipiert, dass die Funktionalitäten des modellgetriebenen Systems erhalten bleiben und die für dieses System entwickelten Werkzeuge auch für das erweiterte System verwendet werden können.

Diese Kompatibilität wird dadurch erreicht, dass für die Erweiterungen der Modellierung die Annotationen des ursprünglichen Systems genutzt werden. Dazu wurden in Abschnitt 5.3 Ontologieannotationen und Anfrageannotationen vorgestellt und es wurde erläutert, wie mit diesen Annotationen, dem Modell und dessen Modellinstanzen eine Ontologie definiert werden kann und wie Anfragen auf einer solchen Ontologie mit Operationen verknüpft werden können. Wie aus solchen annotierten Modellen eine Ontologie generiert werden kann, wurde in Abschnitt 5.5 erläutert.

Die Verwendung der Annotationen für die Erweiterung der Modellierung hat den Nachteil, dass für die Verwendung von Annotationen in der Regel fast keine Restriktionen existieren. Daher wird empfohlen, dass ein Berechnungsschritt, der für die Einhaltung der wichtigen Restriktionen sorgt, angeboten wird und der Entwickler mithilfe von angepassten Editoren bei der Eingabe der speziellen Annotationen unterstützt wird. Das Konzept für einen solchen Berechnungsschritt wurde in Abschnitt 5.4 vorgestellt. Dieser Berechnungsschritt sollte auch optional eine Behandlung der `ClassExpressions`, wie in Abschnitt 5.4.2 beschrieben, anbieten.

Welchen Einfluss die Ontologieannotationen und die Anfrageannotationen haben und wie die durch annotierte Modelle und deren Modellinstanzen definierte Ontologie vom generierten Code genutzt werden sollen, wurde in Abschnitt 5.6 behandelt. Der Ansatz sieht vor, dass die Einhaltung einiger

Ontologierestriktionen zur Laufzeit gewährleistet wird (5.6.1), und dass mithilfe von Konsistenzüberprüfungen die Einhaltung aller Ontologierestriktionen zu einem Zeitpunkt geprüft wird (5.6.2). Ebenso wurde erläutert, wie mithilfe von der Ontologie und von annotierten Anfragen das Verhalten einer Operation dynamisch modelliert werden kann (5.6.3).

In Kapitel 6 wurde eine Implementierung des Ansatzes, die das Eclipse Modeling Framework erweitert, vorgestellt. Dazu wurden Komponenten für die erweiterte Codegenerierung in das TwoUse Toolkit integriert. Durch die zahlreichen Funktionen, die das TwoUse Toolkit bereits anbietet, wurde so eine mächtige Entwicklungsumgebung für eine Kombination aus modellgetriebener und ontologiegetriebener Softwareentwicklung erzielt. Der so entwickelte Prototyp wurde in Kapitel 7 vorgestellt. Zudem wurde in diesem Kapitel gezeigt, wie dieser Prototyp für das Bestellvorgangsszenario (Kapitel 2) genutzt werden kann.

Ein System, das dem vorgestellten Ansatz entsprechend entwickelt wurde, nutzt sowohl Entwicklern aus dem ontologiegetriebenen Bereich als auch Entwicklern aus dem modellgetriebenen Bereich. Entwickler aus dem ontologiegetriebenen Bereich erhalten ein Softwareentwicklungssystem, das ihnen die Möglichkeit bietet Ontologierestriktionen im Code abzubilden. Des Weiteren können sie mit Anfragen auf Ontologien das Verhalten von Operationen dynamisch modellieren. Durch die Generierung eines Editors können sie das System verwenden, um Ontologien zu erstellen und existierende Ontologien, unterstützt von einem Editor, mit Individuen zu füllen.

Entwicklern aus dem modellgetriebenen Bereich bietet das erweiterte System eine gewohnte Entwicklungsumgebung für modellgetriebene Softwareentwicklung, die es ihnen erlaubt ihre gewohnten Werkzeuge zu nutzen und die ihnen zusätzlich die Verwendung von Ontologien zur Definition von Restriktionen und die Modellierung des Verhaltens von Operationen ermöglicht. Durch die Möglichkeit der Nutzung der Ontologien in einer gewohnten Umgebung, könnte vielleicht eine bessere Akzeptanz dieser Technologien in der Gemeinde dieser Entwickler erreicht werden.

8.2 Ausblick

Konzeptionell sind einige Erweiterungen denkbar. Naheliegend wäre eine Transformation von OWL Ontologien zu EOF Modellen, die sich an dem Ansatz von Scheglmann et al. [33] orientiert. Durch eine solche Transformation könnten Entwickler aus dem ontologiegetriebenen Bereich Ontologien verwenden, um einen Großteil eines EOF Modells zu definieren. Dann müssten sie an dem EOF Modell lediglich Ergänzungen und Anpassungen vornehmen. Eine weitere naheliegende konzeptionelle Erweiterung wäre es, die Modellierung von abgeleiteten Referenzen und Attributen mit Anfragen zu ermöglichen. Eine solche Modellierung könnte sich an der vorgestellten Modellierung des Verhaltens von Operationen orientieren.

Für den implementierten Prototyp sind einige Verbesserungen denkbar. Neben der Implementierung des in Abschnitt 5.4 vorgestellten Berechnungsschritts sollten auch graphische Editoren zur Unterstützung des Anwenders bei der Eingabe des EOF Modells erstellt werden. Zudem sollten mehr Anpassungsmöglichkeiten für die Codegenerierung angeboten werden. So sollte es zum Beispiel möglich sein auszuwählen, dass eine bestehende Ontologie kontinuierlich aktualisiert wird anstatt die Ontologie immer wieder neu zu generieren.

Literaturverzeichnis

1. ALLEMANG, Dean ; HENDLER, Jim: *Semantic Web for the Working Ontologist: Effective Modeling in RDF, RDFS and OWL*. Morgan Kaufmann Publishers/Elsevier, 2008 (Safari Books Online). – ISBN 978-0-12-373556-0
2. BAADER, F. ; CALVANESE, D. ; MCGUINNESS, D.L. ; NARDI, D. ; PATEL-SCHNEIDER, P.F.: *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2010. – ISBN 978-0521150118
3. BLEEK, Wolf-Gideon ; WOLF, Henning: *Agile Softwareentwicklung: Werte, Konzepte und Methoden*. Dpunkt.Verlag GmbH, 2008. – ISBN 978-3898644730
4. BOLEY, Harold ; GROSOF, Benjamin ; TABET, Said: *RuleML Tutorial*. <http://www.ruleml.org/papers/tutorial-ruleml.html>. Version: 2004. – Abruf: 17.11.2010
5. BRICKLEY, Dan ; GUHA, Ramanathan V. ; MCBRIDE, Brian (Hrsg.): *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation. <http://www.w3.org/TR/rdf-schema>. Version: 2004. – Abruf: 17.11.2010
6. DAMUS, Christian W. ; HUSSEY, Kenn ; MERKS, Ed ; STEINBERG, Dave: *Effective Use of the Eclipse Modeling Framework*. <http://eclipsezilla.eclipsecon.org/php/attachment.php?bugid=3619>. Version: 2007. – Abruf: 17.05.2010
7. EBERHART, Andreas: Automatic Generation of Java/SQL Based Inference Engines from RDF Schema and RuleML. In: HORROCKS, Ian (Hrsg.) ; HENDLER, James A. (Hrsg.): *International Semantic Web Conference Bd. 2342*, Springer, 2002 (Lecture Notes in Computer Science). – ISBN 3-540-43760-6, S. 102-116
8. ECLIPSE FOUNDATION: *ATL User Guide*. http://wiki.eclipse.org/ATL/User_Guide. – Abruf: 17.11.2010
9. HILLAIRET, Guillaume ; BERTRAND, Frédéric ; LAFAYE, Jean Y.: Bridging EMF applications and RDF data sources. In: *SWESE 2008, 4th Workshop on Semantic Web Enabled Software Engineering, workshop at ISWC 2008, the 7th International Semantic Web Conference 2008*, 2008, 26-40
10. HITZLER, Pascal ; KRÖTZSCH, Markus ; RUDOLPH, Sebastian ; SURE, York: *Semantic Web: Grundlagen*. Springer, 2007 (EXamen. press Series). – ISBN 978-3540339939
11. HORRIDGE, Matthew ; PATEL-SCHNEIDER, Peter F.: *OWL 2 Web Ontology Language Manchester Syntax*. <http://www.w3.org/TR/owl2-manchester-syntax/>. Version: Oktober 2009. – Abruf: 19.08.2010

12. KALYANPUR, Aditya ; PASTOR, Daniel J. ; BATTLE, Steve ; PADGET, Julian A.: Automatic Mapping of OWL Ontologies into Java. In: MAURER, Frank (Hrsg.) ; RUHE, Günther (Hrsg.): *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004), Banff, Alberta, Canada, June 20-24, 2004*, 2004. – ISBN 1-891706-14-4, S. 98-103
13. KNUBLAUCH, Holger: Ontology-Driven Software Development in the Context of the Semantic Web: An Example Scenario with Protege/OWL. In: FRANKEL, David S. (Hrsg.) ; KENDALL, Elisa F. (Hrsg.) ; MCGUINNESS, Deborah L. (Hrsg.): *1st International Workshop on the Model-Driven Semantic Web (MDSW2004)*, 2004
14. KNUBLAUCH, Holger ; FERGERSON, Ray W. ; NOY, Natalya F. ; MUSEN, Mark A.: The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. In: MCILRAITH, Sheila A. (Hrsg.) ; PLEXOUSAKIS, Dimitris (Hrsg.) ; HARMELEN, Frank van (Hrsg.): *International Semantic Web Conference* Bd. 3298, Springer, 2004 (Lecture Notes in Computer Science). – ISBN 3-540-23798-4, S. 229-243
15. KNUBLAUCH, Holger ; OBERLE, Daniel ; TETLOW, Phil ; WALLACE, Evan: *A Semantic Web Primer for Object-Oriented Software Developers*. Version: 2006. <http://www.w3.org/2001/sw/BestPractices/SE/ODSD/> Abruf: 17.11.2010
16. KUSCHKE, Michael ; WÖLFEL, Ludger: *Web Services kompakt*. Spektrum-Akademischer Vlg, 2002. – ISBN 978-3827413758
17. MANOLA, Frank ; MILLER, Eric ; MCBRIDE, Brian (Hrsg.): *RDF Primer*. W3C Recommendation. <http://www.w3.org/TR/rdf-primer/>. Version: 2004 (W3C Recommendation). – Abruf: 17.11.2010
18. MOTIK, Boris ; PATEL-SCHNEIDER, Peter F. ; PARSIA, Bijan ; BOCK, Conrad ; FOKOUE, Achille ; HAASE, Peter ; HOEKSTRA, Rinke ; HORROCKS, Ian ; RUTTENBERG, Alan ; SATTLER, Uli ; SMITH, Michael: *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax*. <http://www.w3.org/TR/owl2-syntax/>. Version: Oktober 2009. – Abruf: 24.10.2010
19. OBJECT MANAGEMENT GROUP: *Ontology Definition Metamodel*. <http://www.omg.org/spec/ODM/1.0/PDF>. Version: 2009. – Abruf: 22.04.10
20. OBJECT MANAGEMENT GROUP: *Unified Modeling Language: Superstructure*. <http://www.omg.org/cgi-bin/doc?formal/09-02-02.pdf>. Version: 2, 2009
21. OBJECT MANAGEMENT GROUP: *Object Constraint Language*. <http://www.omg.org/spec/OCL/2.2/PDF>. Version: 2010. – Abruf: 17.11.2010
22. PARREIRAS, Fernando S.: *Marrying Model-Driven Engineering and Ontology Technologies: The TwoUse Approach*, Universität Koblenz-Landau, Diss., Oktober 2010
23. PARREIRAS, Fernando S. ; STAAB, Steffen ; WINTER, Andreas: On Marrying Ontological and Metamodeling Technical Spaces. In: CRNKOVIC, Ivica (Hrsg.) ; BERTOLINO, Antonia (Hrsg.): *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*. New York, NY, USA : ACM, 2007 (ESEC-FSE '07). – ISBN 978-1-59593-811-4, 439-448
24. POPMA, Remko: *JET Tutorial Part 1 (Introduction to JET)*. http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html. Version: 2004. – Abruf: 17.11.2010

25. POPMA, Remko: *JET Tutorial Part 2 (Write Code that Writes Code)*. http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.html. Version: 2004. – Abruf: 17.11.2010
26. POWELL, Adrian: *Model with the Eclipse Modeling Framework, Part 1: Create UML models and generate code*. <http://www.ibm.com/developerworks/library/os-ecemf1/>. Version: 2004. – Abruf: 17.11.2010
27. POWELL, Adrian: *Model with the Eclipse Modeling Framework, Part 2: Generate code with Eclipse's Java Emitter Templates*. <http://www.ibm.com/developerworks/library/os-ecemf2/>. Version: 2004. – Abruf: 17.11.2010
28. POWELL, Adrian: *Model with the Eclipse Modeling Framework, Part 3: Customize generated models and editors with Eclipse's JMerge*. <http://www.ibm.com/developerworks/library/os-ecemf3/>. Version: 2004. – Abruf: 17.11.2010
29. PRUD'HOMMEAUX, Eric ; SEABORNE, Andy: *SPARQL Query Language for RDF (Working Draft)*. <http://www.w3.org/TR/rdf-sparql-query/>. Version: 2008. – Abruf: 17.11.2010
30. PULESTON, Colin ; PARSIA, Bijan ; CUNNINGHAM, James ; RECTOR, Alan L.: Integrating Object-Oriented and Ontological Representations: A Case Study in Java and OWL. In: SHETH, Amit P. (Hrsg.) ; STAAB, Steffen (Hrsg.) ; DEAN, Mike (Hrsg.) ; PAOLUCCI, Massimo (Hrsg.) ; MAYNARD, Diana (Hrsg.) ; FININ, Timothy W. (Hrsg.) ; THIRUNARAYAN, Krishnaprasad (Hrsg.): *International Semantic Web Conference Bd. 5318*, Springer, 2008 (Lecture Notes in Computer Science). – ISBN 978-3-540-88563-4, S. 130–145
31. RAHMANI, Tirdad ; OBERLE, Daniel ; DAHMS, Marco: An Adjustable Transformation from OWL to Ecore. In: PETRIU, Dorina C. (Hrsg.) ; ROUQUETTE, Nicolas (Hrsg.) ; HAUGEN, Øystein (Hrsg.): *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part II Bd. 6395*, Springer, 2010 (Lecture Notes in Computer Science). – ISBN 978-3-642-16128-5, S. 243–257
32. RUPP, Chris ; HAHN, Jürgen ; QUEINS, Stefan ; JECKLE, Mario ; ZENGLER, Barbara: *UML 2 glasklar*. Hanser, 2005. – ISBN 3-446-22952-3
33. SCHEGLMANN, Stefan ; SCHERP, Ansgar ; STAAB, Steffen: Model-driven Generation of APIs for OWL-based Ontologies / Institut WeST, Universität Koblenz-Landau. 2010 (07/2010). – Arbeitsberichte aus dem Fachbereich Informatik
34. SCHÖNING, Uwe: *Logik für Informatiker*. Spektrum, Akad. Verl., 2000. – ISBN 3-8274-1005-3
35. STAAB, Steffen ; WALTER, Tobias ; GRÖNER, Gerd ; PARREIRAS, Fernando S.: Model Driven Engineering with Ontology Technologies. In: ASSMANN, Uwe (Hrsg.) ; BARTHO, Andreas (Hrsg.) ; WENDE, Christian (Hrsg.): *Reasoning Web Bd. 6325*, Springer, 2010 (Lecture Notes in Computer Science). – ISBN 978-3-642-15542-0, S. 62–98
36. STEINBERG, David ; BUDINSKY, Frank ; PATERNOSTRO, Marcelo ; MERKS, Ed ; GAMMA, Erich (Hrsg.) ; NACKMAN, Lee (Hrsg.) ; WIEGAND, John (Hrsg.): *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2008. – ISBN 978-0-321-33188-5
37. WALTER, Tobias ; PARREIRAS, Fernando S. ; GRÖNER, Gerd ; WENDE, Christian: OWLizing - Transforming Software Models to Ontologies. In: *2nd International Workshop on Ontology-Driven Software Engineering (ODiSE 2010)*, 2010
38. WALTER, Tobias ; PARREIRAS, Fernando S. ; STAAB, Steffen: OntoDSL: An Ontology-Based Framework for Domain-Specific Languages. In: SCHÜRR, Andy

(Hrsg.) ; SELIC, Bran (Hrsg.): *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings* Bd. 5795, Springer, 2009 (Lecture Notes in Computer Science). – ISBN 978-3-642-04424-3, S. 408–422