DIPLOMARBEIT

Integritätsprüfung von Diskursmodellen, Transformationsregeln und strukturellen Modellen von graphischen User Interfaces

ausgeführt zur Erlangung des akademischen Grades
eines Diplom-Ingenieurs unter der Leitung von

Univ.Prof. Dipl.-Ing. Dr.techn. Hermann Kaindl
Proj.Ass. Dipl.-Ing. Dr.techn. Jürgen Falb
Proj.Ass. Dipl.-Ing. David Raneburger

am

Institut für Computertechnik (E384)
der Technischen Universität Wien

durch

Alexander Schörkhuber
Matr.Nr. 0025376
Mühlbachstrasse 21, 4451 Garsten

Wien, am 3.11.2010 _____

Kurzfassung

User Interfaces können mit Hilfe von Tools auch automatisch oder semi-automatisch generiert werden. Ein derartiges Tool ist das *Unified Communication Plattform* (UCP)-Framework, welches es ermöglicht aus Diskursmodellen *User Interfaces* zu generieren. Im Zuge dieser Diplomarbeit wurden Eigenschaften untersucht und Modellintegritätsbedingungen definiert, welchen die Diskursmodelle sowie weitere Modelle im Rahmen der Generierung (*Structural UI*-Modelle und Regelmodelle) genügen müssen, damit aus diesen mit dem UCP-Framework *User Interfaces* generiert werden können. Unter Verwendung des sog. *Validation Frameworks* des *Eclipse Modeling Frameworks* (EMF), welches eine einfache Einbindung von Modellprüfungen ermöglicht, wurden entsprechende Modellprüfungs-Plugins realisiert. In diesen wurden die Modellintegritätsbedingungen in Form von *Constraints* implementiert. Zur Formulierung dieser *Constraints* wurde Java und die *Object Constraint Language* (OCL) verwendet. OCL ist eine Erweiterung der *Unified Modelling Language* (UML), welche eine eigens für den Zweck der Formulierung von Gültigkeitsbedingungen geschaffene Sprache darstellt. Diese Überprüfungen sollen dazu führen, dass (in diesem Sinn) korrekte Modelle rascher erstellt werden können und damit ein schnelleres Erstellen eines *User Interfaces* mit diesem Ansatz möglich wird.

Abstract

User interfaces can be generated by tools in an automated or semi-automated way. One of these tools is the Unified Communication Platform (UCP)-Framework, which allows user interface generation based on discourse-models. This thesis evaluates properties of UCP-models and defines model integrity criteria. These must be met by discourse-models and other models used for generating user interfaces (structural UI-models and rule-models), to serve as a basis for the UCP-Framework for user interface generation. Model check plugins were created using the so-called Validation Framework of the Eclipse Modeling Framework (EMF), which allows an easy inclusion of model checks. These plugins contain model integrity criteria implemented in the form of constraints. These constraints were written in Java and the Object Constraint Language (OCL). OCL is an addition to the Unified Modeling Language (UML) and was created for the definition of integrity criteria. These checks should allow – in this context – a faster creation of correct models, and subsequently a faster generation of user interfaces.

Danksagung

Ich möchte allen danken, die mich bei dieser Diplomarbeit unterstützt haben.

Allen voran meinen Eltern für ihre Engelsgeduld, ihre Liebe und ihre finanzielle Unterstützung.

Des Weiteren möchte ich all meinen Freunden für ihren unerschütterlichen Glauben an mich danken und für das Erfüllen der Bitte, um einen Tritt in den Allerwertesten, falls ich mit meiner Leistung unzufrieden war. An meine Karatekas wurde diese Bitte ob der Gefahr einer allzu wörtlichen Auslegung nicht gerichtet, doch auch sie standen mir mit Motivation zur Seite.

Ich möchte mich bei meinen Betreuern und dem gesamten UCP-Projektteam bedanken für die Hilfe und prompte Betreuung, die sie mir angedeihen ließen. Allen voran David, der zu jeder Tages und Nachtzeit ein offenes Ohr und eine Lösung für all meine Probleme hatte.

Terry Pratchett und Gunkl, die mich lehrten den Spaß in der Wissenschaft zu sehen, welcher die Basis bildet, sich ein Leben lang mit Freude weiterzubilden.
Professor Fasching, Lissmann und Nitzsche, deren Ideen meine Sicht auf die Welt umkrempelten und auf eine neue Basis stellten.
Shihan Funakoshi der als oberstes Ziel die Perfektion des Charakters definierte.

Inhaltsverzeichnis

Abkürzungen

AST	Abstract Syntax Tree
EMF	Eclipse Modeling Framework
GUI	Graphical User Interface
OCL	Object Constraint Language
RST	Rhetorical Structure Theorie
SQL	Structured Query Language
SWT	Standard Widget Toolkit
UI	User Interface
UML	Unified Modeling Language
UCP	Unified Communication Platform
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1 Einleitung

Motivation

Die automatische Generierung von *User Interfaces* (UI) unterstützt deren rasches Erstellen und erlaubt dem Benutzer eigene *User Interfaces* zu entwickeln, ohne sich zuvor langjährige Programmiererfahrung angeeignet zu haben. Dafür gibt es zahlreiche Ansätze. Einer von diesen ist das *Unified Communication Platform (UCP)-Framework*, welches einen modellbasierten Ansatz zur semiautomatischen *User Interface*-Generierung darstellt. Ausgangspunkt ist hierbei ein Kommunikationsmodell, welches durch Transformationsregeln in ein *Structural UI*-Modell transformiert wird, aus welchem dann der Quellcode für *User Interfaces* generiert werden kann. Das *UCP-Framework* hat sich in den letzten Jahren von einer ersten Idee zu einem funktionierenden Tool entwickelt.

Wiewohl es funktioniert, ist es allerdings derzeit schwer zu handhaben, da in den Editoren weit mehr modelliert werden kann als erlaubt ist. Ungültige Modelle führen zu Programmabstürzen oder einer fehlerhaften Darstellung im *User Interface*. Modellfehler werden derzeit als Laufzeitfehler mit langen und unübersichtlichen Java Fehlermeldungen zurückgemeldet, wodurch eine Modellierung zu einer langwierigen und schwierigen Aufgabe wird, welche erhebliche Kenntnis über Java, *Eclipse Modeling Framework* (EMF) und das *UCP-Framework* erfordert.

Durch eine Definition von Modellintegritätskriterien und die Überprüfung der Modelle anhand dieser, soll der Benutzer von der derzeit notwendigen Kenntnis über den inneren Aufbau und Funktion des *UCP-Frameworks* befreit werden, sodass sich dieser auf die Modellierung konzentrieren kann. Durch das Auffinden von Fehlern, gute Fehlermeldungen und Markierung der fehlerhaften Objekte soll eine raschere Modellierung und damit ein schnelleres Erstellen eines *User Interfaces* möglich werden.

Ziel

Ziel der Arbeit war die Entwicklung und Implementierung von Modellintegritätskriterien, d.h. die Definition von Gültigkeitsbereichen für Objekteigenschaften und gültigen Objekt-Beziehungen für die Diskursmodelle, Transformationsregeln und *Structural UI*-Modelle des *UCP-Frameworks*.

Es sollen so viele Fehler wie möglich abgefangen werden, die schwerwiegendsten und häufigsten Fehler erkannt und durch gute Fehlermeldungen eine rasche Behebung ermöglicht werden.

Hierbei kann und soll nicht 100%-ige Fehlererkennung erreicht werden. Möglich ist allerdings eine Reihe von Fehlerklassen zu erkennen und damit die Zahl der möglichen Fehlerquellen für unentdeckte Fehler signifikant einzuschränken. Dadurch wird auch eine raschere Fehleridentifikation und Behandlung von unentdeckten Fehlern möglich.

Zur Unterstützung der Implementierung gibt es im EMF ein *Validation Framework*, durch dessen Verwendung der Fokus auf die Entwicklung von geeigneten *Constraints* gelegt werden konnte.

Aufbau der Arbeit

Nach diesem einleitenden Kapitel 1, in welchem das Ziel der Arbeit, die Motivation und der Aufbau kurz dargelegt werden, folgt in Kapitel 2 eine Einführung in die Diskursmodellierung. Dabei wird zuerst das *UCP-Framework* dargelegt und an einem kurzen Beispiel dessen Funktionsweise erläutert. Dann werden die Möglichkeiten erörtert, wie eine Fehlerüberprüfung auf den Modellen realisiert werden kann, und welche Möglichkeiten das EMF dazu bietet. Auf diesem Wissen aufbauend wird dann in Kapitel 3 beschrieben, wie die konkrete Realisierung vorgenommen wurde. Ausgehend von den verwendeten Konzepten und Strategien wird dann die Implementierung der *Validation Plugins* beschrieben. Hierbei wird vor allem erläutert, warum die in diesen *Plugins* enthaltenen *Constraints* entwickelt wurden und welche Fehlerklassen diese abdecken. Abschließend wird erklärt, wie diese evaluiert wurden und die Ergebnisse der Evaluierung präsentiert. In Kapitel 4 werden die erreichten Ergebnisse zusammengefasst und Ansatzpunkte für eine weitere Entwicklung dargelegt. Der Anhang enthält schlussendlich die implementierten *Constraints*.

Um dies alles übersichtlich zu erörtern, werden die folgenden Schriftarten für die angegebenen Elemente verwendet:

- *Fremdwörter* sind kursiv gestellt.
- `Quellcode` ist in Typewriter ausgeführt.
- KAPITÄLCHEN markieren Klassen.
- **Constraints** sind fett dargestellt.

2 Basis der Arbeit

Den Beginn bilden die Theorien der Modellierung von Kommunikation, welche die Grundlage für das *Unified Communication Plattform* (UCP)-*Framework* bilden. Darauf folgt ein Überblick über die Funktionsweise des *Frameworks*, welche anschließend an einem Beispiel verdeutlicht wird. Abschließend folgt eine Beschreibung von Modellintegritätskriteriumsprüfungen im Allgemeinen und es werden die Randbedingungen betrachtet, wie diese mit Hilfe des *Eclipse Modelling Framework(EMF)-Validation Framework* im *UCP-Framework* eingesetzt werden können.

2.1 Diskursmodellierung

Im Diskursmodell werden alle möglichen Kommunikationsabläufe modelliert und es bildet die Basis für eine semiautomatische Generierung eines *User Interfaces* durch das *UCP-Framework*.

Den Kern der Diskursmodellierung bilden *Communicative Acts*, welche auf der *Speech Act Theorie* basieren [Sea69]. Diese stellen die Basiselemente der Kommunikationsmodellierung dar und modellieren einzelne Kommunikationselemente, wie z.B. Fragen oder Aussagen. Zusammengehörige *Communicative Acts* werden über *Adjacency Pairs* aus der *Conversation Analysis* verknüpft [LFG90], z.B. zu einem Frage-Antwort Paar. Über Relationen aus der *Rhetorical Structure Theorie* (RST) und prozedurale Relationen werden diese schließlich zu einem Baum aller möglichen Kommunikationsabläufe verbunden [MT88]. Hierbei werden prozedurale Relationen dazu benutzt, den Kommunikationsablauf zu steuern, und RST-Relationen verbinden zusammengehörige Dialogelemente.

Eine detailliert Beschreibung, wie diese Elemente dazu benutzt werden, um Kommunikationsabläufe zu modellieren, findet sich in [BFK+08].

2.2 UCP-Framework

Das *UCP-Framework* bietet die Möglichkeit, ein Modell der Kommunikation in einem graphischen Editor zu erstellen und dieses anschließend in ein *Structural User Interface* (Structural UI)-Modell zu transformieren, aus welchem dann konkrete *User Interfaces* generiert werden können.

2.2.1 Kommunikationsmodell

Das Kommunikationsmodell besteht aus drei Modellen: Diskursmodell, *Domain of Discourse*-Modell und *Action*-Modell. Diese beinhalten die Elemente, welche für die Beschreibung aller möglichen Kommunikationsabläufe des Modells im *UCP-Framework* benötigt werden.

Das Diskursmodell beinhaltet die Information, welche *Communicative Acts* ausgeführt werden und wie diese über Relationen, prozedurale Relationen und *Adjacency Pairs* zu Dialogen verknüpft werden. Im *Domain of Discourse*-Modell wird der Teil der Domäne, welche für die Kommunikation bzw. das *User Interface* notwendig sind, modelliert. Das *Action*-Modell modelliert die Aktionen, welche in der Interaktion möglich sind.

Die Struktur dieser Modelle d.h. die in ihnen enthaltenen Objekte und deren Eigenschaften werden in einem Metamodell beschrieben. Diese übergeordnete Beschreibung eines Modells legt auch die Gültigkeit der Modellelemente fest.

2.2.1.1 Diskursmodell

Das Diskursmodell beschreibt alle möglichen Kommunikationsabläufe.

Dazu muss definiert werden, welcher Aktor welche Entscheidung trifft und welche Aktionen wann ausgeführt werden.

In Bild 2.1 ist das Diskurs-Metamodell zu sehen, welches die Rahmenbedingungen dargestellt, wie ein Diskursmodell auszusehen hat. Ein konkretes Beispiel findet sich in Kapitel 2.2.3. Die einzelnen Elemente haben folgende Bedeutung:

- Discourse dient als Ausgangspunkt, auf dem das Diskursmodell aufbaut und als Container, in welchem alle Diskursmodellelemente enthalten sind.
- RSTRelation dient zur Verknüpfung von Dialogelementen. Von dieser Klasse werden als zahlreiche Spezialisierungen die Relationen abgeleitet.
- ProceduralRelation dient zur Steuerung des Kommunikationsablaufs.
- Link modelliert die Verbindung von RST-Relationen, prozeduralen Relationen und *Adjacency Pairs*.
- SenderAgent modelliert einen Aktor, welchem *Communicative Acts* bzw. Relationen oder prozedurale Relationen zugeordnet werden. Im Fall eines *Communicative Act* wird dadurch definiert, welcher Kommunikationsteilnehmer diesen ausführt. Durch die Zuordnung von prozeduralen Relationen wird definiert, welcher Kommunikationsteilnehmer die eventuell notwendigen Entscheidungen zu treffen hat.
- CommunicativeAct ist ein in der Kommunikation vorkommendes Dialogelement.
- AdjacencyPairs verknüpfen *Communicative Acts*. Ein *Communicative Act* kann hierbei vom Typ *Opening* oder *Closing* sein.
- Content eines *Communicative Acts* beschreibt, welche *Domain of Discourse*-Objekte übergeben und welche Aktionen benutzt werden. Dies definiert, was damit in der Applikationslogik passieren soll und stellt die Schnittstelle zu dieser dar.

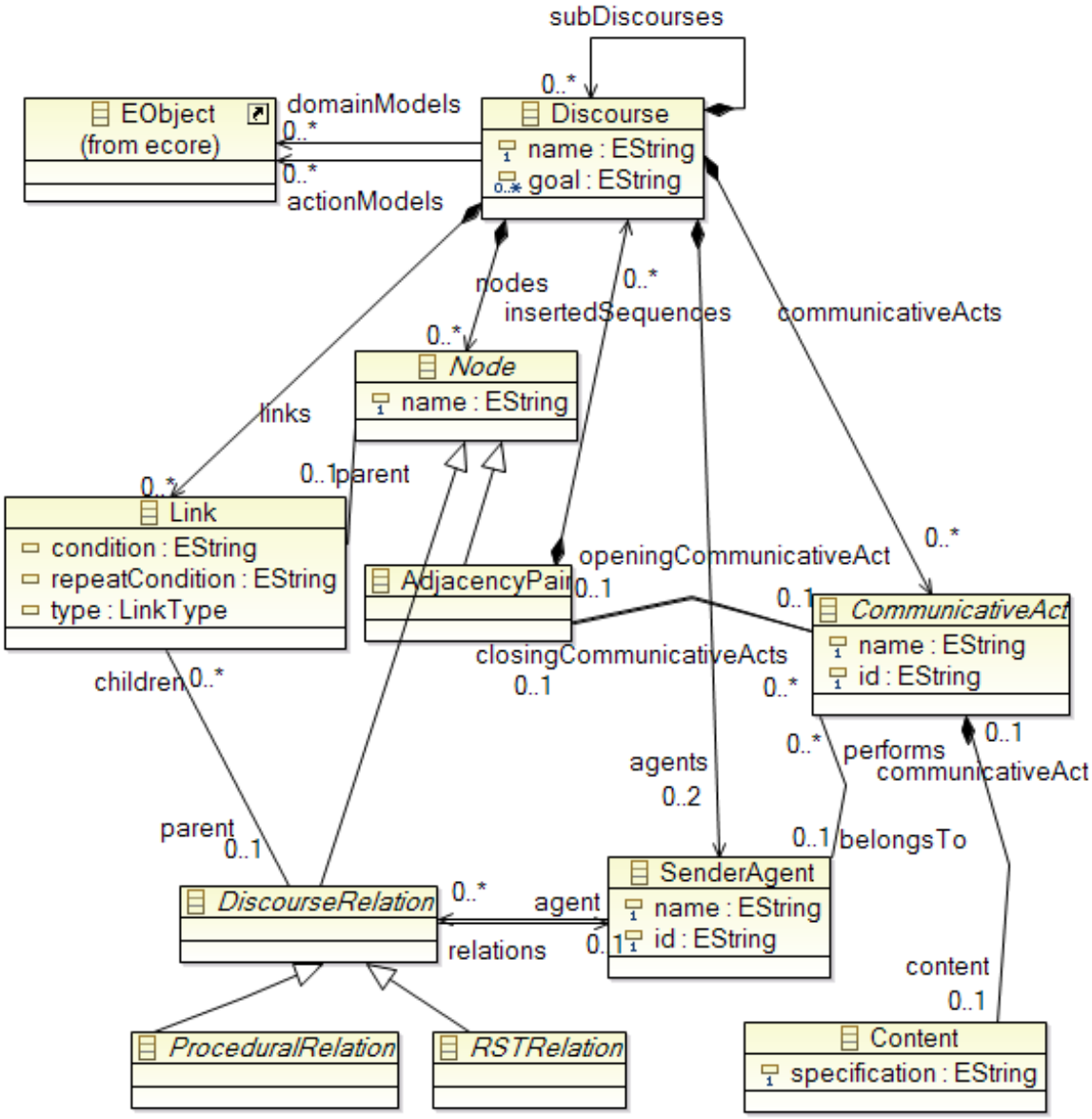

Abbildung 2.1: Diskurs-Metamodell

Discourse

Die Klasse DISCOURSE stellt einen Container dar, welcher ein Diskursmodell enthält. Sie wird auf mehrere Arten verwendet, welche streng unterschieden werden müssen. Jeder Diskurs kann beliebig viele SUBDISCOURSES haben und in *Adjacency Pairs* können INSERTEDSEQUENCES enthalten sein, welche wiederum einen kompletten Diskurs modellieren. Im Hauptdiskurs müssen die *Action*-Modelle und *Domain of Discourse*-Modelle festgelegt sein, des Weiteren werden hier die beiden Aktoren vom Typ *SenderAgent* festgelegt.

Jeder Diskurs benötigt genau einen *Root-Node*, von welchem der Ablauf der Kommunikation startet. Dieser ist daran erkennbar, dass er keinen *Parent* besitzt. Die Topologie aller Elemente eines Diskurses muss ein Baum sein, welcher von einem *Root-Node* ausgeht. Dies ist bei *Adjacency Pairs* und *Communicative Act* implizit gegeben, da *Adjacency Pairs* und *Communicative Acts* keine Relationen, prozedurale Relationen oder *Adjacency Pairs* als Nachfolger haben können und

in *Adjacency Pairs* immer ein oder mehrere *Opening* oder *Closing Communicative Acts* eingetragen werden.

Prozedurale Relationen
Prozedurale Relationen enthalten die Bedingungen, welche den Ablauf der Kommunikation festlegen. Durch diese lassen sich auch komplexe und iterative Kommunikationsabläufe modellieren. Zu ihren Nachfolgern besitzen sie *Links*, welche die Bedingungen enthalten, wann diese Zweige ausgeführt werden. Jede Relation hat eine bestimmte semantische Bedeutung und benötigt eine bestimmte Anzahl und bestimmte Typen von *Links*. Alle Spezialisierungen der abstrakten Klasse ProceduralRelation, welche konkret in einem Diskurs vorkommen können, werden im Folgenden näher erläutert und sind in Abbildung 2.2 auf der linken Seite dargestellt.

- Sequence
  Ziel: Dient dazu, Kommunikationselemente in einer festgelegten Reihenfolge auszuführen.
  Funktion: Die angehängten *Nuclei-Links* werden der Reihe nach ausgeführt.
  Zu beachten: Die Relation darf nur *Links* vom Typ *Nucleus* besitzen, welche in deren Eigenschaft `condition` eindeutig nummeriert sind. Es müssen zwei oder mehr davon vorhanden sein.

- IfUntil
  Ziel: Dient einerseits dazu Abfragen zu realisieren, oder andererseits einen Kommunikationszweig so lange zu wiederholen, bis eine Abbruchbedingung erfüllt ist.
  Funktion: Realisiert eine Wenn-Dann Abfrage mit einer bedingten Schleife. Der *Tree-Link* wird ausgeführt, bis die `condition` des *Then-Links* erfüllt ist. Optional kann auch ein *Else-Link* vorhanden sein, welcher ausgeführt wird, falls die `condition` nicht zutrifft.
  Zu beachten: Durch das Vorhanden sein von *Then* und *Else* wird ein Verlassen der Schleife in jedem Fall erzwungen. Dies führt dazu, dass zumindest ein *Tree* und ein *Then* vorhanden sein müssen, und im *Then* eine `condition` angegeben ist, wann dieser ausgeführt werden soll. Dadurch besteht auch die Möglichkeit Endlosschleifen zu realisieren, indem die `condition` des *Then-Links* leer bleibt. Dies kann prinzipiell ein erwünschter Effekt sein, aber auch durch einen vergessenen Eintrag einer Abbruchbedingung oder das Löschen des die Abbruchbedingung beinhaltenden *Links* zustande kommen.

- Condition
  Ziel: Stellt eine Entscheidung zwischen zwei Zweigen anhand einer Bedingung dar.
  Funktion: Anhand einer Bedingung wird der Zweig, in dem der Kommunikationsprozess fortgeführt wird, ausgewählt.
  Zu beachten: Dazu wird ein *Then-Link* und ein *Else-Link* benötigt, wobei im *Then-Link* die `condition` eingetragen wird, wann dieser Zweig ausgeführt wird. Ist diese nicht erfüllt, wird der *Else-Link* ausgeführt.

RST-Relationen
Jede Relation hat eine bestimmte Aufgabe und benötigt eine bestimmte Anzahl und bestimmte Typen der ihr zugehörigen *Links*. Alle Spezialisierungen der abstrakten Klasse RSTRelation, welche konkret in einem Diskurs vorkommen können, werden im Folgenden näher erläutert und sind in Abbildung 2.2 auf der rechten Seite dargestellt.

- Result
  Ziel: Aus dem Abschluss eines seiner Zweige resultiert die Ausführung des anderen Zweiges.

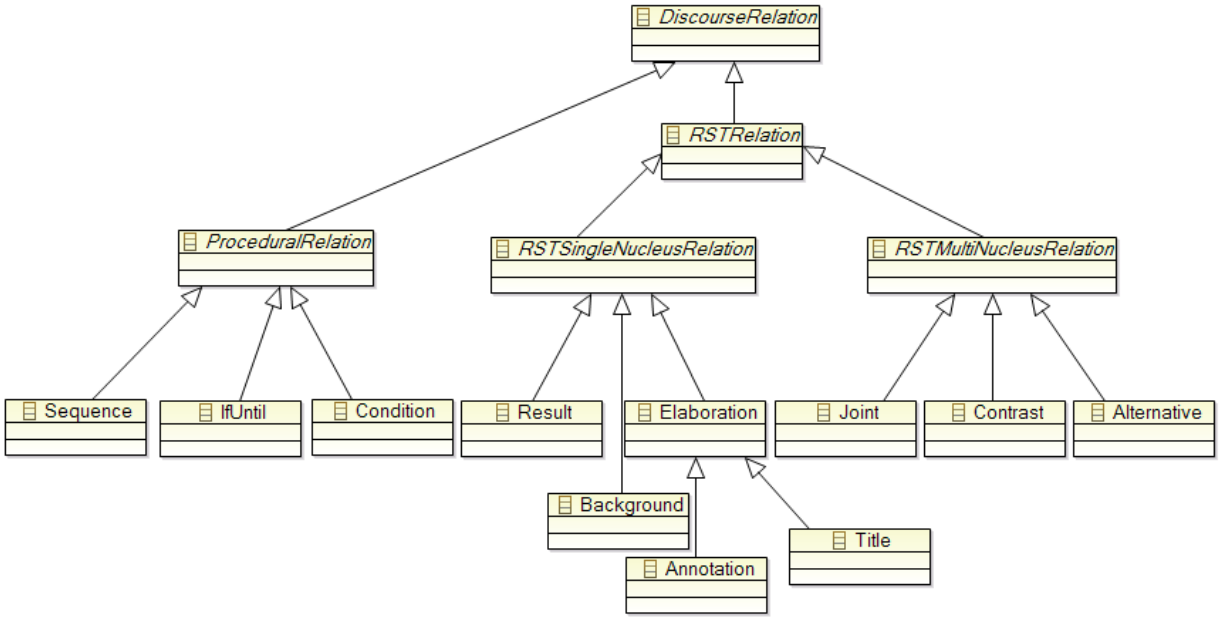

Abbildung 2.2: RST-Relationen und prozedurale Relationen

Funktion: Nach der erfolgreichen Ausführung des *Satellite*-Zweiges, wird der *Nucleus*-Zweig ausgeführt.
Zu beachten: Die Ausführung des *Nucleus*-Zweiges wird durch ein externes Ereignis ausgelöst.

- ELABORATION
  Ziel: Zusätzliche Information zu einem *Nucleus*-Zweig zu liefern.
  Funktion: Im *Satellite*-Zweig werden zusätzliche Informationen über den *Nucleus*-Zweig geliefert. Die zugehörigen *Communicative Acts* können auch *Questions* sein, über welche die zusätzliche Information, welche angezeigt werden soll, spezifiziert und ausgewählt wird.
  Zu beachten: Es ist dazu ein *Nucleus*- und ein *Satellite*-Zweig erforderlich.

- BACKGROUND
  Ziel: Zusätzliche Hintergrundinformation zu dem *Nucleus*-Zweig zu liefern.
  Funktion: Hier bietet wie in der ELABORATION der *Satellite*-Zweig zusätzliche Information zum *Nucleus*-Zweig, wobei hier im Unterschied zu einer ELABORATION die *Communicative Acts* nur vom Typ INFORMING sein sollten.
  Zu beachten: Es sind dazu ein *Nucleus*- und ein *Satellite*-Zweig erforderlich.

- TITLE
  Ziel: *Title* gibt einem Modellelement eine Überschrift.
  Funktion: Eine Spezialisierung der ELABORATION, wobei hier sowohl *Nucleus* als auch *Satellite* vom Typ *Informing* sein sollten. Diese Relation kann z.B. dazu benutzt werden, um einem Infotext eine Überschrift zu geben, oder Bildern eine kurze Beschreibung zur Seite zu stellen.
  Zu beachten: Es ist dazu ein *Nucleus*- und ein *Satellite*-Zweig erforderlich.

- ANNOTATION
  Ziel: Metadaten zur Verfügung stellen.

Funktion: Eine weitere Spezialisierung der ELABORATION, welche dazu benutzt wird, um im *Satellite*-Zweig Metadaten zum *Nucleus* anzuzeigen.
Zu beachten: Es ist dazu ein *Nucleus*- und ein *Satellite*-Zweig erforderlich.

- JOINT
  Ziel: Verbindet mehrere gleichwertige Kommunikationsabläufe.
  Funktion: Im Fall der Relation *Joint* müssen alle *Nuclei*-Zweige abgeschlossen sein, damit die Relationen beendet ist und im Kommunikationsablauf weitergegangen wird.
  Zu beachten: Es werden zwei oder mehr *Nuclei*-Zweige benötigt.

- ALTERNATIVE
  Ziel: Gleichwertige Entscheidungsmöglichkeiten anbieten.
  Funktion: Die Entscheidung wird so realisiert, dass die Relation beendet ist, sobald einer ihrer Zweige abgeschlossen ist.
  Zu beachten: Es werden zwei oder mehr *Nuclei*-Zweige benötigt.

- CONTRAST
  Ziel: Zwei gegensätzliche Entscheidungsmöglichkeiten anbieten.
  Funktion: Hier hat jeder *Nucleus* eine `condition`, wobei sich diese gegenseitig ausschließen müssen. In der `condition` können auch *Domain of Discourse*-Modellobjekte mit logischen Ausdrücken evaluiert werden. Dadurch kann der Kommunikationsfluss ähnlich wie durch IFUNTIL gesteuert werden.
  Zu beachten: Es werden zwei oder mehr *Nuclei*-Zweige benötigt, von denen jeder eine `condition` besitzen muss, welche die anderen ausschließt.

An dieser Stelle könnte man sich fragen, warum genau diese Anzahl an Relationen realisiert wurde, wo einerseits alles auch mit weniger Relationen modelliert werden könnte und andererseits auch zahlreiche weitere Relationen denkbar wären. All diese Relationen werden bei der Transformation in ein *Structural UI*-Modell entsprechend den Regeln umgewandelt und diese Regeln beinhalten Darstellungsmuster für jede Relation. Vor allem ist es auch möglich, für jede Relation mehrere Regeln zu erstellen, wodurch zum Einen eine einheitliche Darstellung erreicht werden kann, welche zum Anderen auch von anderen Parametern abhängen kann, wie z.B. das Vorhandensein von speziellen Modellelementen. Des Weiteren entspricht die Verwendung mehrerer verschiedener Relationen eher dem natürlichen Sprachgebrauch, wodurch die Verständlichkeit und Lesbarkeit der Diskursmodelle verbessert wird.

Links
Wie aus dem Diskursmetamodell, siehe Abbildung 2.1, ersichtlich ist verknüpfen LINKS Relationen untereinander oder eine Relation mit einem *Adjacency Pair*. Sie können vom Typ *Nucleus, Satellite, Tree, Then* oder *Else* sein. Jede Relation benötigt eine bestimmte Anzahl dieser Linktypen. Je nach Funktion können diese Bedingungen besitzen, in welchem Fall der an ihnen hängende Zweig ausgeführt wird.

Agents
Agents modellieren die Kommunikationsteilnehmer und führen als solche *Communicative Acts* aus. Dazu muss jedem *Communicative Act* ein Agent zugewiesen werden. Des Weiteren muss für einige Relationen ein Agent definiert sein, der die Entscheidung trifft, in welchem Zweig der Kommunikationsprozess fortgesetzt wird.

Communicative Acts
Communicative Acts stellen die Dialogbasiselemente, z.B. Fragen oder Antworten, dar. Die in den

Modellen vorkommenden Spezialisierungen der Klasse CommunicativeAct sind in Abbildung 2.3 dargestellt.

Adjacency Pairs
Adjacency Pairs verknüpfen einzelne Dialogbasiselemente z.B. in der Form Frage-Antwort, welche durch die zugehörigen *Communicative Acts* verkörpert werden. Jedes *Adjacency Pair* hat einen *Opening Communicative Act*, und einen oder mehrere *Closing Communicative Acts*.

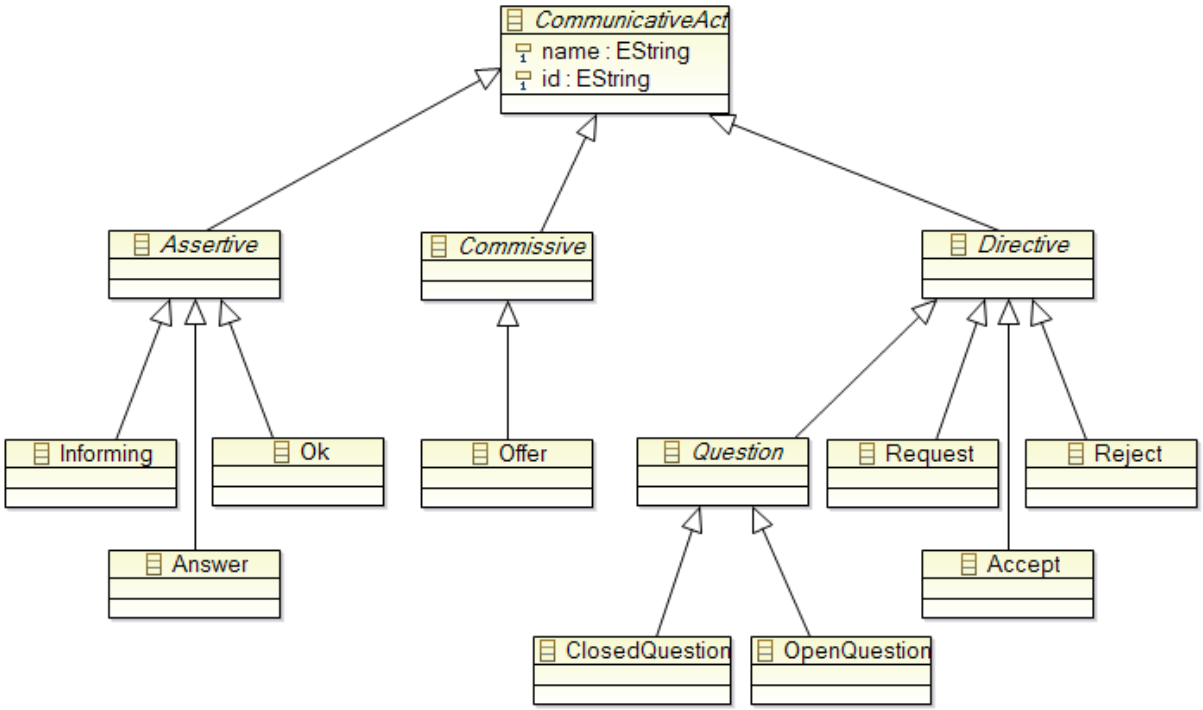

Abbildung 2.3: Communicative Act Taxonomie

Die einzelnen *Communicative Acts* können nicht beliebig kombiniert werden, sondern es sind nur spezielle Kombinationen erlaubt und sinnvoll. *Opening* bzw. *Closing* können nur bestimmte *Communicative Acts* sein. Die erlaubten *Opening Communicative Act*-*Closing Communicative Act* Paare sind in Tabelle 2.1 dargestellt. Eine Ausnahme stellt hier der *Communicative Act Informing* dar. Dieser dient der Darstellung von Information und muss keinen *Closing Communicative Act* besitzen.

Im Allgemeinen wird ein *Adjacency Pair* durch das Ausführen eines seiner *Closing Communicative Acts* geschlossen und eine Relation wird dadurch abgeschlossen, dass je nach Relation eine bestimmte Anzahl seiner *Child*-Zweige abgeschlossen werden. In Falle des *Communicative Act Informing* wird das zugehörige *Adjacency Pair* mit der Erfüllung der Relation, an welcher das *Adjacency Pair*-*Communicative Act* Konstrukt hängt, geschlossen.

Content
Ein *Communicative Act* kann einen *Content* haben, in welchem beschrieben wird, mit welcher Funktion der Applikationslogik dieser *Communicative Act* assoziiert wird und wie die Daten zur Laufzeit verarbeitet werden. Dies geschieht in einer der *Structured Query Language*(SQL) ähnlichen Sprache, wobei bei deren Entwicklung versucht wurde, diese so weit wie möglich an den natürlichen Sprachgebrauch anzupassen. Im *Content* können *Action*-Modell und *Domain of Discourse*-Modell Objekte enthalten sein.

Opening CommAct Types	Closing CommAct Types
Open Question	Answer
Closed Question	Answer
Request	Accept
Request	Reject
Request	Accept & Reject
Request	Ok
Request	Informing
Offer	Accept
Offer	Reject
Offer	Accept & Reject
Offer	Ok
Informing	
Informing	Ok

Tabelle 2.1: Beziehungen zwischen Communicative Acts [BEF+10]

Ob ein *Content* nach dessen Eingabe vom Editor geparst werden konnte, sieht man an Hand der `contentASTspecification`, in welcher alle *Action*- und *Domain of Discourse*-Elemente in aufgelöster Form, d.h. mit deren komplettem Pfad zu dem Objekt im zugehörigen Modell, enthalten sind. Dies ist im Allgemeinen dann der Fall, wenn diese in den im Diskurs definierten *Action*- und *Domain of Discourse*-Modellen enthalten sind. Einige *Communicative Acts* fordern, dass in ihrem *Content* bestimmte *Actions* oder Spezialisierungen davon enthalten sind, welche im Folgenden aufgelistet sind.

- Eine *ClosedQuestion* muss `select` und `for` enthalten.
- Ein *Informing* muss `presenting` enthalten.
- Eine *OpenQuestion* muss `get` oder `update` und `for` enthalten.
- Ein *Request* benötigt eine *Action* im *Content*.

2.2.1.2 Action-Modell

Das *Action*-Modell bildet die Schnittstelle zur Applikationslogik.

Im *UCP-Framework* ist ein Basisset von *Actions* und *Notifications* im *Action*-Modell `basic` definiert. Diese werden dazu verwendet, den *Content* von *Communicative Acts* zu definieren. Das Modell `basic` und alle verwendeten *Action*-Modelle müssen dazu im Hauptdiskurs definiert werden.

Für eine konkrete Implementierung eines Diskurses ist es möglich bzw. notwendig, dieses Basisset um die benötigte Funktionalität zu erweitern. Dies kann einerseits durch die Einführung neuer *Actions* und *Notifications* oder andererseits durch Spezialisierung bestehender geschehen.

Actions und *Notifications* stellen die Schnittstelle zur Applikationslogik dar. In der finalen Codegenerierung werden sie in Dummyfunktionen umgewandelt, welche dann vom Programmierer mit der Interfacelogik befüllt werden, durch welche die Kommunikation mit der Applikationslogik

besorgt wird. Durch die Spezifikation der `parameter`, `attributes` und `representations` kann die Schnittstelle näher definiert und den Erfordernissen des Kommunikationsablaufs angepasst werden.

2.2.1.3 Domain of Discourse-Modell

Im *Domain of Discourse*-Modell wird der Teil der Applikationsdomäne modelliert, welcher für die Kommunikation relevant ist.

Dieses Modell kann als *Unified Modeling Language* (UML)- oder ECore-Diagramm erstellt und visualisiert werden, wofür graphischen Editoren zur Verfügung stehen. Es wird als *XML Metadata Interchange* (XMI)-Datei gespeichert und bietet die Objektbeziehungen bei der UI-Generierung. Die konkreten Instanzen werden diesem Modell entsprechend in einer eigenen XMI-Datei abgelegt.

2.2.2 GUI-Generierung

Durch das *UCP-Framework* kann aus einem Kommunikationsmodell der Code für *User Interfaces* generiert werden.

Das Rendern eines Kommunikationsmodells zu einem *Graphical User Interface* (GUI) ist ein zweistufiger Prozess. Zuerst wird aus den Kommunikationsmodellen ein *Stuctural UI* generiert, welches die Struktur des *User Interfaces* auf Basis von abstrakten *Widgets* beschreibt. Dieses *Structural UI* ist unabhängig von dem verwendeten Ausgabe-*Toolkit*, welches schließlich benutzt wird, um das *User Interface* darzustellen. Allerdings ist es sehr wohl abhängig vom Zielgerät, auf welchem das *User Interface* dargestellt werden soll. Informationen über dieses, wie z.B. Bildschirmauflösung, werden entweder direkt in das *Structural UI*-Modell eingegeben oder dem Code-Generator als CSS-Datei übergeben. [KRR+10]

Diese zweistufige Variante hat zum einen den Vorteil, dass das generierte *Structural UI* platformunabhängig ist und im zweiten Schritt durch beliebige *GUI-Toolkit* Sprachen dargestellt werden kann. Zum Anderen kann das Design direkt beeinflusst und an das Zielgerät angepasst werden. Abbildung 2.4 zeigt die Renderingarchitektur, wobei die Modelle hervorgehoben sind an denen eine Überprüfung notwendig bzw. sinnvoll und möglich ist.

Ein *Discourse To Structural UI Transformer* führt eine Modell-zu-Modell Transformation durch, welche auf Regeln basiert. Diese Regeln beschreiben, wie aus Mustern von Diskursmodellelementen ein *Structural UI* generiert und transformiert wird. Solch eine Regel kann zum Beispiel besagen, dass jeder *Informing-Communicative Act*, welcher sich im Diskursmodell befindet, in das *Structural UI* zu einem *Panel* mit einem *Label*-Widget transformiert wird. Dieses *Label* beinhaltet im Weiteren die Information, welche im entsprechenden Diskursobjekt referenziert ist.

Den *Input* für diesen Transformator bilden die folgenden Modelle:

- Kommunikationsmodell, bestehend aus
  - Diskursmodell
  - *Action*-Modell
  - *Domain of Discourse*-Modell

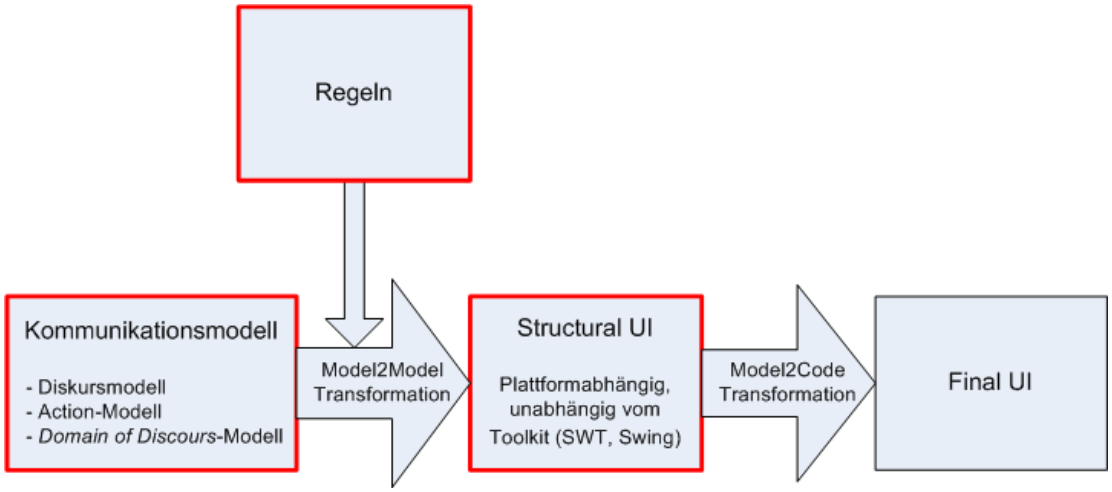

Abbildung 2.4: Der UCP-Transformationsprozess

- Regelmodell
- Einschränkungen durch die Spezifizierung der Zielplattform

Die Generierung eines *UI* aus dem *Structural UI* erfolgt dann aus:

- *Structural UI*-Modell
- Formatierungsinformationen, z.B. CSS

Im *Action*-Modell sind keine Parameter zwingend erforderlich bzw. sind diese bereits durch das ECore-Modell definiert. Das *Domain of Discourse*-Modell modelliert jenen Teil der Applikationsdomäne, welcher für die Kommunikation relevant ist. Im *Domain of Discourse*-Modell können korrekte Objektbeziehungen nicht im Allgemeinen definiert werden. Die korrekte Erstellung und Verwendung der *Action*- und *Domain of Discourse*-Modellobjekte kann erst überprüft werden, wenn diese im Diskursmodell verwendet werden. Erst durch deren Verwendung in den *Content*-Spezifikationen wird im Kontext des Diskursmodelles eine Aussage über deren Korrektheit möglich. Bei der Evaluierung der *Content*-Objekte kann allerdings nicht a priori festgestellt werden, ob ein *Action*- oder *Domain of Discourse*-Modellelement korrekt erstellt und verwendet wurde. Es kann nur eine Aussage darüber getroffen werden, ob die in der `contentSpezification` beschriebene Verwendung der Modellelemente möglich ist. Falls bei der Überprüfung von *Content*-Objekten ein Fehler auftritt muss der Benutzer entscheiden, ob das Modellobjekt fehlerhaft ist oder es im *Content* falsch verwendet wurde.

Einschränkungen durch die Zielplattform und Formatierungsinformationen wären auf Plausibilität überprüfbar. Vor der großen Vielfalt an möglichen Zielplattformen wären dies allerdings sehr schwache Einschränkungen. Einzig ein Abfangen von Fehlern, welche durch leere Eigenschaftsfelder hervorgerufen werden könnten, wäre möglich. Dies ist allerdings obsolet da derartiges vom Code Generator abgefangen wird.

Damit bleiben Modellintegritätsprüfungen an Diskursmodellen, den Regelmodellen und den *Structural UI*-Modellen durchzuführen.

2.2.2.1 Structural UI-Modell

Das *Structural UI*-Modell ist ein Zwischenschritt zwischen dem Diskursmodell und dem generierten Code eines *User Interfaces*.

In Abbildung 2.5 ist ein Ausschnitt aus dem zugehörigen Metamodell abgebildet, welcher die für die Struktur und das Verhalten relevanten Elemente zeigt. Es ist ein auf Widgets basierender Baum, welcher die *User Interface*-Struktur repräsentiert. Basierend auf der abstrakten Klasse WIDGET werden alle im *Structural UI*- Modell vorkommenden Elemente von dieser abgeleitet. Diese Elemente sind entweder INPUTWIDGETS, welche zum Sammeln von Information benutzt werden, OUTPUTWIDGETS, welche zur Präsentation von Informationen dienen, oder PANELS, welche weitere WIDGETS beinhalten können und zur Strukturierung des *User Interfaces* benötigt werden. Einen Sonderfall stellt die Klasse LISTWIDGET dar, welches sowohl von der Klasse INPUTWIDGET als auch von der Klasse PANEL abgeleitet ist.

Jedes dieser Widgets kann Layoutinformationen beinhalten, welche spezifizieren, wie es dargestellt werden soll. Im Fall der *Input Widgets* ist auch angegeben, welche *Events* durch deren Betätigung ausgelöst werden.

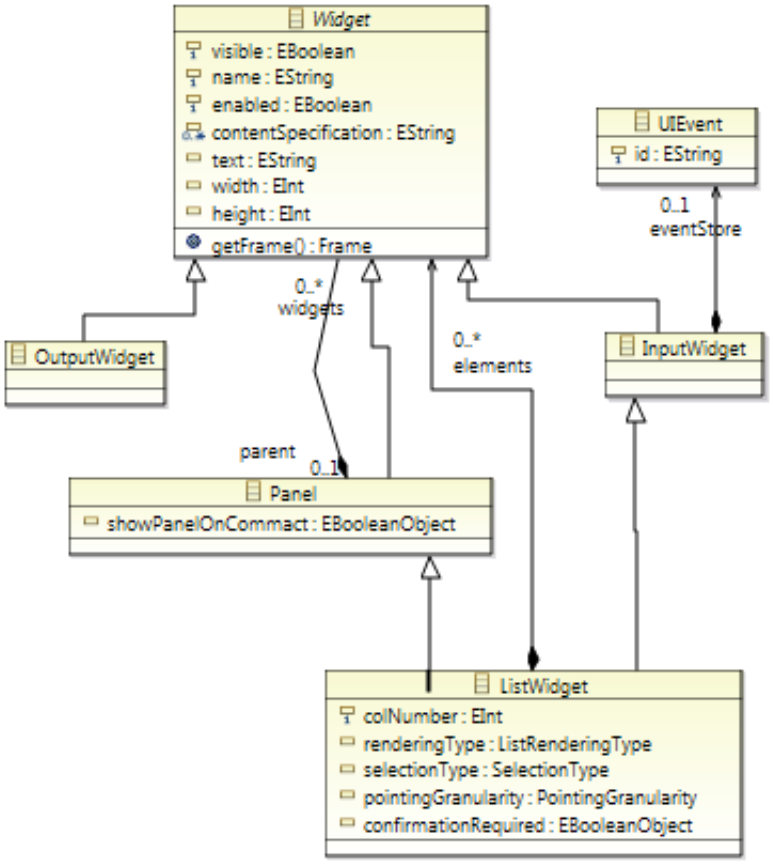

Abbildung 2.5: Structural UI-Metamodell

Diese abstrakten Widgets sind nicht zu verwechseln mit den gleichnamigen Java Widgets, wie sie z.B. im *Standard Widget Toolkit* (SWT) verwendet werden.

2.2.2.2 Transformationsregeln

Transformationsregeln werden für die Modell-zu-Modell Transformation von Diskursmodellen zu *Structural UI*-Modellen verwendet. Sie beschreiben, welche Objektbeziehungen eines Diskursmodells auf welche Art und Weise im *Structural UI*-Modell dargestellt werden.

Dieser Transformationsprozess besteht aus zwei Schritten, wie in [KFK09, Kapitel 3] näher beschrieben ist.

1. Im ersten Schritt werden Transformationsregeln auf Diskursmodellelemente angewandt. Diese Transformationsregeln generieren *Input*- und *Output*-Widgets bzw. Spezialisierungen von diesen wie *Buttons* oder *Labels*. Diese dienen als Platzhalter für *Content*-Objekte.
2. Im zweiten Schritt werden *Content Transformation Rules* im Kontext der Transformationsregeln des ersten Schrittes ausgeführt. Diese Einbettung erlaubt die Auswahl von Widgets für das schlussendliche *Structural UI*-Modell basierend auf dem *Content* Typ, dem diesem *Content* zugeordneten *Communicative Act* Typ und dem gegenwärtigen Kontext, in dem der *Communicative Act* eingebettet ist. Dieser Kontext wird von der übergeordneten Regel definiert.

Am Ende dieser Transformationen dürfen im *Structural UI*-Modell keine *Input*- oder *Output*-Widgets mehr vorhanden sein.

Wie aus dem Regel-Metamodell in Abbildung 2.6 ersichtlich, besteht eine Regel im Kern aus einem **Source-** und einem **Target**-Objekt und hat im Parameter `type` seine Aufgabe spezifiziert. Sie kann je nach `type` verschiedenes Verhalten zeigen und neue *Structural UI*-Objekte kreieren, bestehende *Structural UI*-Objekte modifizieren oder löschen.

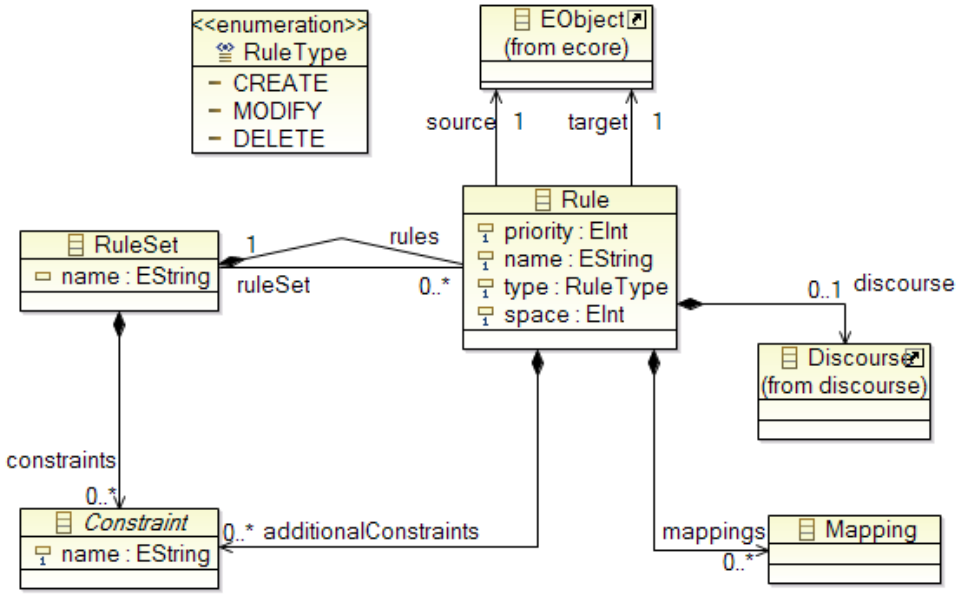

Abbildung 2.6: Transformationsregel-Metamodell

Das Diskursmodell wird rekursiv durchlaufen und eine Regel wird dann angewandt, wenn deren *Source*-Objekt im Diskursmodell gefunden wird. Falls ein Diskursmuster angegeben ist, wird überprüft, ob dieses in dem gefundenen Objekt im Diskursmodell ebenfalls vorliegt. Wenn dies alles

zutrifft, wird das im *Target* angegebene Objekt kreiert, mit allen im *Structural UI*-Teil der Regel spezifizierten Strukturen.

Dieser Prozess wird dadurch erweitert, dass mehrere Transformationsregeln für jedes Diskursmuster vorliegen dürfen. Durch die Anwendung der sogenannten *Conflict Resolution* wird die Regel ausgewählt, welche verwendet wird. Dieser *Conflict Resolution*-Mechanismus muss die Regeln nach bestimmten Kriterien auswählen. Diese Auswahl kann z.B. anhand des Platzes, welchen die Widgets im generierten *User Interface* benötigen, geschehen. Dazu müssen alle Regeln, welche dasselbe Diskursmuster haben, vom Designer hinsichtlich ihres Platzverbrauches bewertet werden. [KRR+10]

Sind **Mappings** vorhanden, so werden Diskursmodellelemente mit *Structural UI*-Elementen in Beziehung gebracht und Eigenschaften des Diskursmodellobjektes auf die Eigenschaften des *Structural UI*-Objekts übertragen.

Constraints stellen zusätzliche Überprüfungen dar, welche erfüllt sein müssen, damit diese Regel angewandt wird. Sie können auch Werte enthalten, welche die Darstellung auf einer konkreten Zielplattform spezifizieren, wie z.B. die Bildschirmgröße, oder referenzieren auf eine *Cascading Style Sheet* (CSS)-Datei, welche die benötigten Formatierungsinformationen enthält. Hierbei sei nochmal darauf verwiesen, dass das *Structural UI*-Modell nur von der Zieltechnologie, d.h. dem verwendeten Ausgabe-*Toolkit* unabhängig ist, nicht allerdings von der Zielplattform und darum bereits alle Informationen über das Layout enthält.

Ist ein **Discourse** angegeben, so stellt dieses ein Diskursmuster und auch eine Einschränkung dar, da die Regel nur angewandt wird, wenn dieses Muster im Diskursmodell gefunden wird.

2.2.3 Beispiel - Shop

Am Beispiel einer Shop-Applikation sollen diese Modelle nochmals verdeutlicht werden. Ausgehend von einem Diskursmodell, siehe Anhang B, wird die Transformation in ein *Structural UI*-Modell anhand einer Transformationsregel erläutert.

Das zu diesem Diskursmodell gehörige *Domain of Discourse*-Modell ist in Abbildung 2.7 dargestellt.

Damit ein UI dargestellt werden kann, müssen noch entsprechende Instanzen der *Domain of Discourse*-Modellobjekte den Modellen beigelegt werden. Diese sind in der Datei OnlineShopPreviewData.xmi gespeichert. Sie dienen nur der Illustration dieses Beispiels, in einem richtigen *User Interface* müssten diese von der Applikation bereitgestellt werden. Diese Shop-Applikation kommt ohne die Definition eines eigenen *Action*-Modells aus, da alle benötigten *Actions* im Modell `basic` bereits vorhanden sind.

Der Kommunikationsablauf beginnt im Diskursmodell in dem unteren *IfUntil*. Dessen *Tree*-Zweig wird solange ausgeführt, bis er abgeschlossen ist. An diesem hängt ein weiteres *IfUntil*, an dessen *Tree*-Zweig ein *Adjacency Pair*, durch eine Raute dargestellt, hängt. Dieses *Adjacency Pair* besitzt einen *Opening Communicative Act* (`ClosedQuestion - AskForProductCategory`) und einen *Closing Communicative Act* (`Answer - SelectProductCategory`). Diese *Closed Question* repräsentiert eine Liste, aus welcher eine Produktkategorie ausgewählt werden muss. Die Auswahl der Produktkategorie wird so lange fortgesetzt, bis die Bedingung `productCategory.products>0` des *Then*-Zweiges erfüllt ist, d.h. bis eine Produktkategorie, in welcher Produkte vorhanden sind, ausgewählt wird.

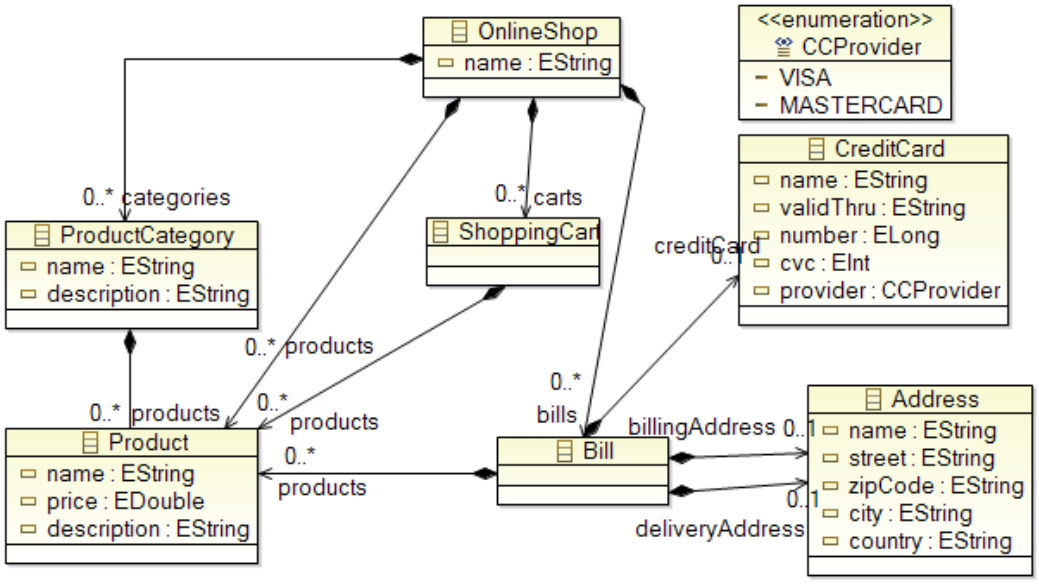

Abbildung 2.7: *Domain of Discourse*-Modell Shop

Dann führt der Kommunikationsablauf über den *Then*-Zweig zur Relation *Background*. Hier wird zum Einen über den *Communicative Act Informing* die Produktkategorie angezeigt, zum Anderen werden durch die `ClosedQuestion - AskForProduct` die Produkte der gewählten Kategorie angezeigt, von denen eines durch `Answer - SelectProduct` ausgewählt wird, welches in den `shoppingCart` gegeben wird. Dadurch ist der *Then*-Zweig des zweiten IfUntil erfüllt und diese prozedurale Relation ist beendet. Dann wird im ersten IfUntil überprüft, ob die Bedingung seines *Then*-Zweiges `shoppingCart.count(products)>0` erfüllt ist, d.h. ob ein Produkt ausgewählt wurde und somit im `shoppingCart` gespeichert ist. Ist dies nicht der Fall, so werden ein weiteres Mal die Produktkategorien angezeigt und nach deren Auswahl die Produkte dieser Kategorie, solange bis ein Produkt ausgewählt wurde.

Sobald ein Produkt ausgewählt wurde, springt der Kommunikationsablauf zur prozeduralen Relation *Joint*. Um diese zu erfüllen, müssen die drei gleichwertigen Fragen, nach der Kreditkartennummer, der Rechnungs- und Zustelladresse beantwortet werden. Hierbei handelt es sich um *Open Questions*, welche eine Eingabe z.B. in ein Textfeld erfordern.

Nach Eingabe dieser Daten ist das *Joint* beendet, damit auch der *Then*-Zweig und das *IfUntil*. Da das *IfUntil* den *Root-Node* darstellt, ist somit der gesamte Diskurs beendet. Dadurch wird entweder der gesamte Kommunikationsablauf beendet, oder, falls dieser Diskurs eine *inserted sequence* in einem anderen Diskursmodell darstellt, in diesem der Kommunikationsablauf weitergeführt.

Anhand der Transformationsregeln wird dieses Kommunikationsmodell in ein *Structural UI*-Modell transformiert. Dieser Prozess wird anhand der Regel in Abbildung 2.8 exemplarisch gezeigt. Das *Adjacency Pair* mit *Question* und *Answer* stellt hierbei das Suchmuster dar. Wird es gefunden, so wird ein *Panel* mit den entsprechenden Objekten erstellt.

Dazu werden alle Diskurselemente auf der Suche nach einem *Adjacency Pair* durchlaufen. Wird eines gefunden, so wird überprüft, ob an diesem dem Suchmuster entsprechend eine *Closed Question* und eine *Answer* hängt, wie in Abbildung 2.8 unter *Discourse ClosedQuestion Discourse*

ersichtlich. Daraufhin werden ein *List Widget*, der *Button Select* und das *Panel ListPanel* mit all ihren Unterstrukturen im *Structural UI*-Modell kreiert.

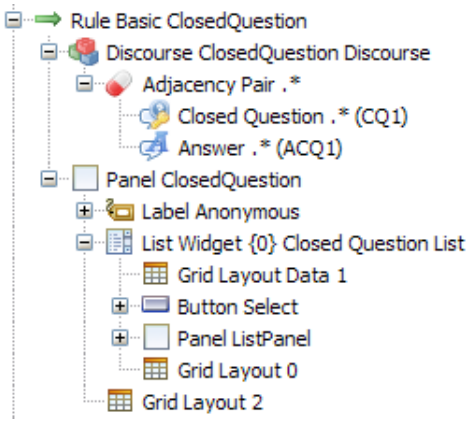

Abbildung 2.8: Regel für eine Closed Question

Dieser Prozess wird für alle Diskursmodellelemente durchgeführt und führt zu einem *Structural UI*-Modell. Durch die oben beschriebene Transformationsregel wird im *Structural UI*-Modell ein *Output Widget* kreiert, welches in einem weiteren Zyklus von einer anderen Regel durch das *Label* ersetzt wurde, wie in Abbildung 2.9 dargestellt. Dargestellt ist das gesamte *Structural UI*-Modell, wobei nur der Ausschnitt, welcher von der oben genannten Regel generiert wurde, entfaltet ist, um die Details zu zeigen.

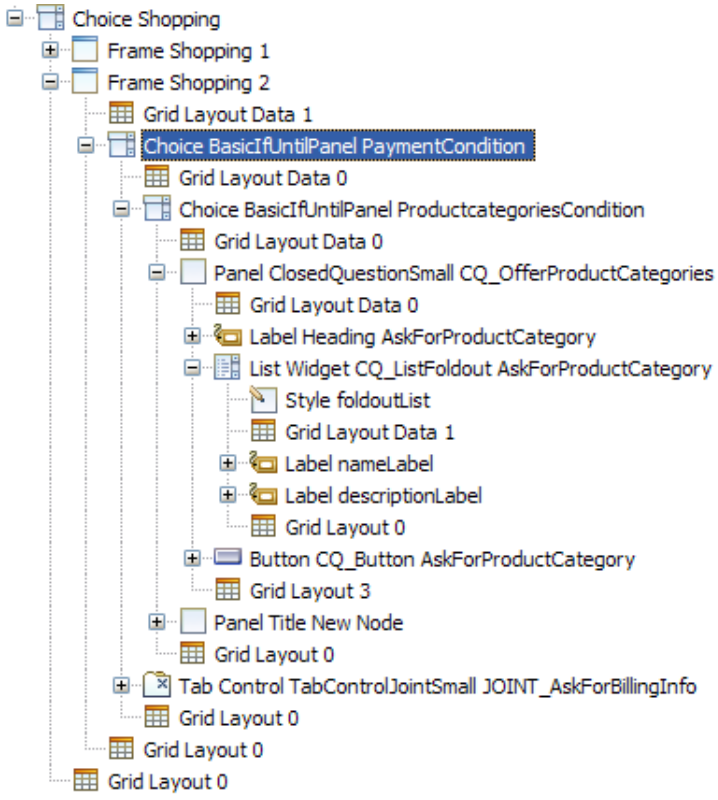

Abbildung 2.9: Structural UI - Ergebnis aus Closed Question

Von diesem *Structural UI*-Modell ausgehend ist es nun möglich *User Interfaces* zu generieren. Dies könnte wie in den Abbildungen 2.10, 2.11 und 2.12 aussehen. Für eine andere Plattform, z.B. eine Applikation für ein Mobiltelefon, kann aus demselben *Structural UI* ein anderes *User Interface* generiert werden.

Abbildung 2.10: Online Shop - Auswahl der Produktkategorie [Ran08]

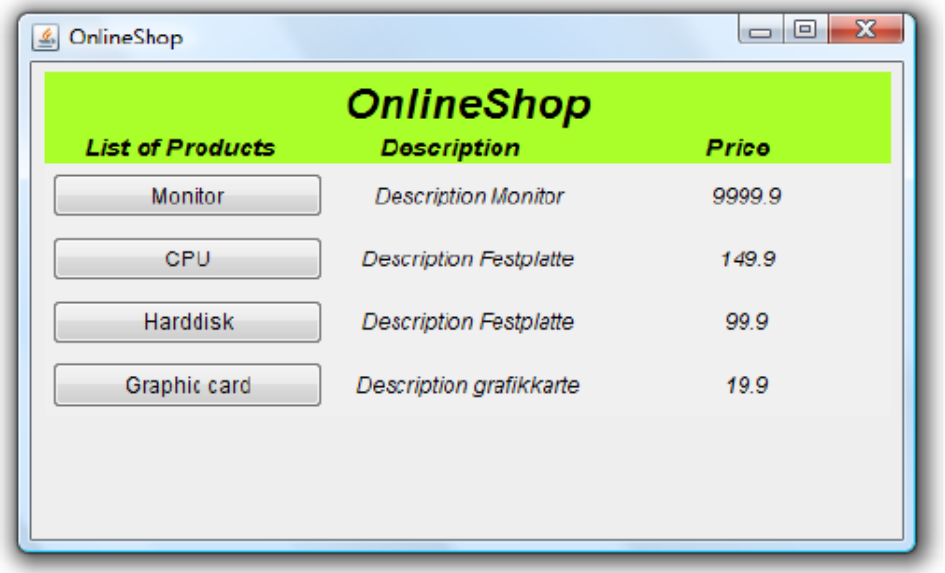

Abbildung 2.11: Online Shop - Produktauswahl [Ran08]

2.3 Modellintegritätskriteriumsprüfung

Durch die Überprüfung eines Modells anhand definierter Gültigkeitsbedingungen können Fehler in Modellen gefunden werden.

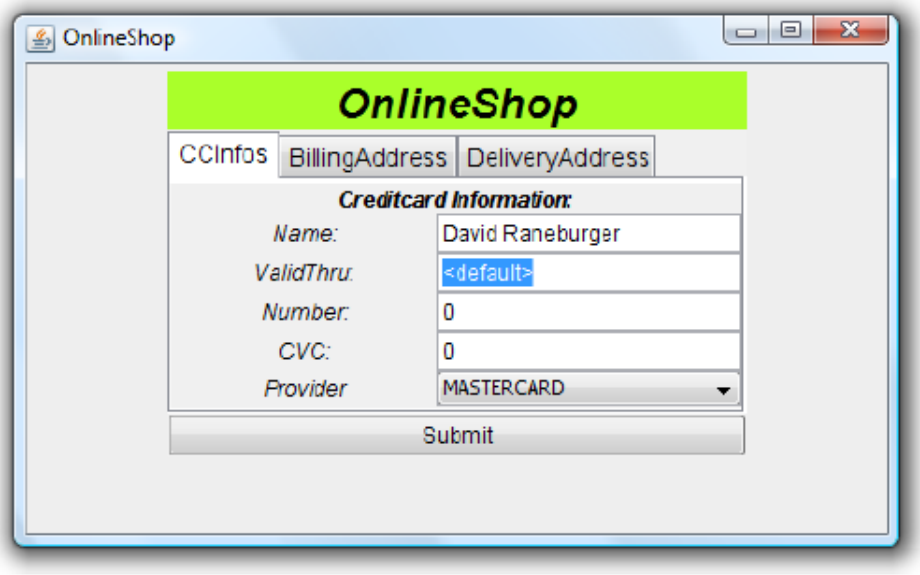

Abbildung 2.12: Online Shop - Eingabe der Kundendaten [Ran08]

Das Wort Validierung hat seine Wurzeln im englischen *valid*, was sich mit gültig oder zulässig übersetzen lässt, auch der Begriff plausibel könnte gebraucht werden. Die Gültigkeit von Variablen ist leicht ersichtlich. Zum Beispiel darf eine Variable welche das Alter eines Menschen in Jahren beschreibt, keine negativen Werte annehmen und Werte über 120 sind wohl unrealistisch. Schwieriger ist die Definition von Gültigkeit in Modellen. Hier müssen die Objektbeziehungen betrachtet werden. Nehmen wir zum Beispiel ein Objekt Baum, welcher 0..* Frucht-Objekte besitzt. Wenn diese Früchte vom Typ Äpfel oder Kirschen sind erscheint es plausibel, sind diese Früchte vom Typ Hamburger liegt wahrscheinlich ein Fehler vor.

Der Begriff Validation wird in einigen Wissensgebieten unterschiedlich verwendet. Dies liegt wohl in der allgemeinen Verwendung von valid als gültig begründet. Während im EMF valid bzw. Validation für die Gültigkeitsprüfung der Modelle hinsichtlich deren Modellintegritätskriterien verwendet wird, hat er im *Software-Engineering* eine gänzlich andere Bedeutung.

Im *Software-Engineering* ist der Begriff Validation eng mit dem Begriff Verifikation verwandt. Beide beschäftigen sich mit der Überprüfung auf Richtigkeit. Man kann sagen, Validation stellt ein „*building the right system*“ dar, während Verifikation ein „*building the system right*“ bezeichnet. Eine dezidierte Unterscheidung gestaltet sich manchmal insofern als schwierig, als Validation und Verifikation versuchen die Richtigkeit sicherzustellen, wobei Überlappungen möglich sind, wodurch oft einfach der Begriff „Valdiation und Verifikation“ benutzt wird.

So steht man hier vor dem Problem, dass die Entwicklung von Modellintegritätskriterien eine Modellierung im Rahmen eines *Software-Engineerings* darstellt und im Rahmen des EMF implementiert wird. Wobei in diesen beiden Gebieten der Begriff valid bzw. Validation eine gänzlich andere Bedeutung besitzt. Darum wird in der vorliegenden Arbeit so weit als möglich auf diese Begriffe verzichtet und stattdessen Synonyme aus den beiden Domänen verwendet.

Als gültig wird ein Modell bezeichnet, wenn es den Integritätskriterien seines Metamodelles genügt. Diese Integritätskriterien werden im *Software-Engineering* gefunden und müssen vom schlussendlich daraus entwickelten System erfüllt werden. In [KKHH04] ist ein *Tool* zur Überprüfung der Übereinstimmung von *Requirements* mit deren Metamodell beschrieben.

Zur Prüfung von Modellintegritätskriterien gibt es im EMF ein *Validation Framework*, durch dessen Verwendung der Fokus auf die Entwicklung von geeigneten *Constraints* konzentriert werden kann. *Constraints* sind Einschränkungen bzw. Bedingungen, denen Objekte genügen müssen, um als richtig bzw. gültig angesehen zu werden. Oder anders gesehen wird richtig oder gültig durch das Erfüllen dieser Kriterien definiert. Zur Implementierung von *Constraints* gibt es drei Möglichkeiten, welche von EMF unterstützt werden. Diese haben unterschiedliche Vor- und Nachteile und unterscheiden sich in deren Mächtigkeit und in ihrem Implementierungsaufwand.

1. **ECore-Modell**
   Beim Durchführen einer Modellprüfung im EMF wird auch überprüft, ob ein Modell den Definitionen seines Metamodelles genügt. Anwendung findet dies z.B. bei der Definition von Kardinalitäten oder der Definition einer Eigenschaft als einzigartig durch das Schlüsselwort *unique*.

   Der Vorteil liegt darin, dass die Implementierung derartiger Constraints durch wenige Klicks erreicht werden kann.

   Der Nachteil ist, dass sich damit nur sehr wenige Objekteigenschaften überprüfen lassen und jede Änderung Auswirkungen auf die gesamte Software haben kann, welche auf diesem Modell basiert.

2. **Object Constraint Language (OCL)**
   Für die Implementierung der *Constraints* bietet sich OCL an, welche UML um die Möglichkeit erweitert Gültigkeitsbereiche von Objekteigenschaften zu überprüfen und ungültige Objektbeziehungen zu erkennen.

   Der Vorteil von OCL liegt vor allem in der Übersichtlichkeit und der geringen Länge von OCL-*Constraints*. Damit lassen sich *Constraints* in wenigen Zeilen realisieren, wodurch diese übersichtlich und schnell zu erstellen sind.

   Ein Nachteil liegt in den kaum vorhandenen *Debug*-Möglichkeiten, welche Eclipse und EMF für OCL mit sich bringen. Viel schwerer wiegt allerdings, dass OCL keine imperative Programmiersprache darstellt und keine Möglichkeit der Programmflusssteuerung bietet. Des Weiteren ist beachtenswert, dass nur seiteneffektfreie Funktionen verwendet werden können, da durch die Überprüfung die überprüften Objekte nicht verändert werden dürfen.

3. **Java**
   Für die Überprüfung komplexerer Objektbeziehungen besteht die Möglichkeit, *Constraints* in Java zu implementieren.

   Der Vorteil hierbei ist, dass der gesamte Java-Sprachumfang mit all seinen Möglichkeiten zur Verfügung steht.

   Dies wird um den Preis erkauft, dass Java-*Constraints* im Allgemeinen länger sind und für deren Erstellung mehr Zeit aufgewendet werden muss.

2.3.1 ECore

ECore-Modelle stellen die Metamodelle des EMF dar. Diese beschreiben wie die Modellelemente auszusehen haben und welche Eigenschaften diese besitzen.

Das *Eclipse Modeling Framework* (EMF) ist ein Modellierungs *Framework* und Codegenerierungs *Tool* zur Erstellung von Applikationen. Ausgehend von einer Modellspezifikation in XMI stellt es

Werkzeuge und Laufzeitunterstützung zur Verfügung, welche benötigt werden, um ein Set von Javaklassen für das Modell zu generieren. Es unterstützt den Entwicklungsprozess durch einen Editor und Adapterklassen, welche das Betrachten und Bearbeiten eines Modells erlauben. [4]

Ein *Domain*-Modell repräsentiert die Daten, welche in einem Programm verwendet werden sollen. Einen generellen Vorteil bietet die Modellierung der Daten unabhängig von der Applikationslogik. Dazu bietet sich das Architekturmuster *Model-View-Control* (MVC) an, in welchem die Software Entwicklung in drei Einheiten strukturiert wird: Datenmodell, Darstellung und Steuerung.

Mit EMF können derartige *Domain*-Modelle modelliert werden. Hierbei wird zwischen Metamodell und dem eigentlichen Modell unterschieden, wobei das Metamodell die Struktur des Modells beschreibt und das eigentliche Modell eine Instanz dieses Metamodelles ist. Diese Modelldefinition wird in XML Metadata Interchange (XMI) implementiert und kann auf Basis von UML, XML Schemata, oder einem XMI Dokument definiert werden. Sobald das EMF-Metamodell spezifiziert ist, können daraus entsprechende Java-Klassen generiert werden. [6]

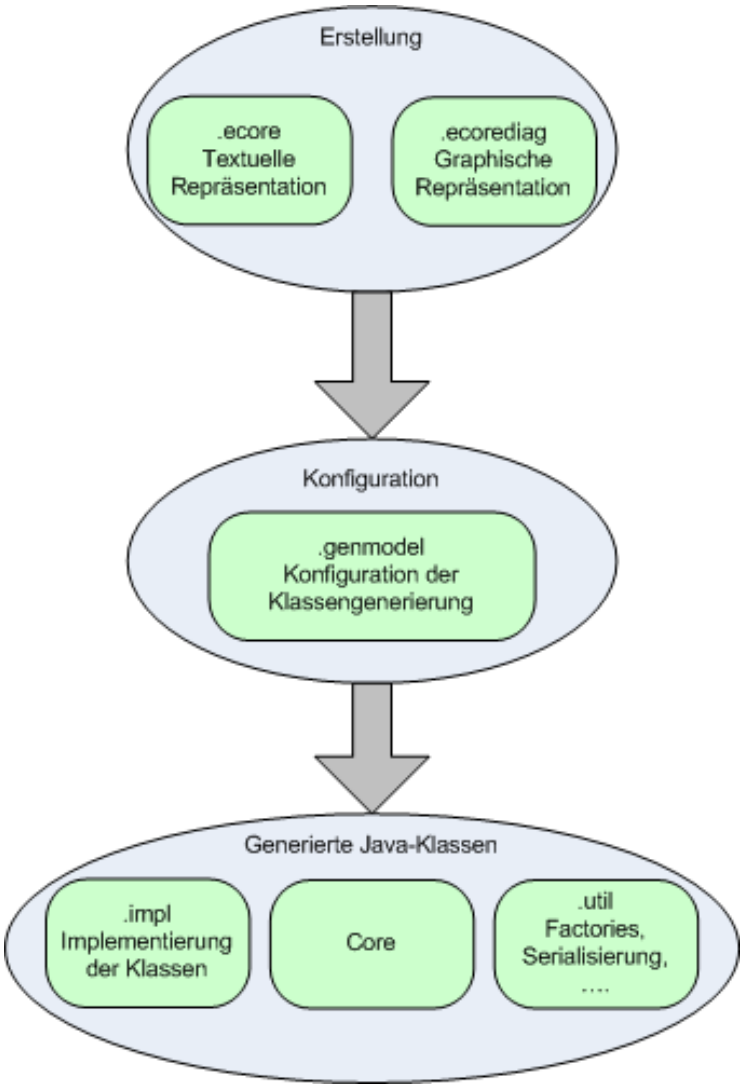

Abbildung 2.13: EMF Modell Generierung [7]

Zu jedem EMF-Modell gibt es zwei Metamodelle, das ECore-Modell und das Genmodel-Modell.

Das ECore-Modell enthält die Information über die definierten Klassen. Es besteht auch die Möglichkeit das ECore-Modell in einem graphischen ECore-Diagrammeditor zu zeichnen und zu editieren. ECore-Modell und ECore-Diagramm besitzen denselben Informationsgehalt. Im Genmodel-Modell können zusätzliche Parameter und Informationen für die automatische Codegenerierung spezifiziert werden.

In Abbildung 2.13 nach [7] ist der Ablauf der EMF Modell Generierung veranschaulicht.

Ein ECore-Modell kann, wie in [6] dargelegt, aus den folgenden vier Datentypen bestehen:

- EClass: Repräsentiert eine Klasse, welche wiederum Attribute und Referenzen enthalten kann.
- EAttribute: Repräsentiert ein Attribut, welches einen Namen und einen Typ hat.
- EReference: Repräsentiert ein Ende einer Assoziation zwischen zwei Klassen.
- EDataType: Repräsentiert den Typ eines Attributes, wie z.B. int, float. Es sind aber auch komplexe Java-Datentypen erlaubt.

Beim Starten einer Modellintegritätsprüfung im EMF wird überprüft, ob ein Objekt den Definitionen in seinem Metamodell genügt. Hier lässt sich z.B. durch die Eigenschaft `Upper Bound` und `Lower Bound` einstellen, wie oft ein Objekt vorkommen darf, und durch Setzten des `Lower Bound` auf 1 erzwingen, dass dieses vorhanden sein muss. Durch die Eigenschaft *Unique* wird festgelegt, dass der Wert einer Eigenschaft in allen Instanzen dieser Klasse unterschiedlich sein muss. Durch die Definition eines *Default Value Literals* kann einem Objekt ein *Default*-Wert gegeben werden. Dieser wird dann bei der Objekterstellung automatisch eingetragen.

Zu beachten ist, dass die Einführung von *Constraints* über die Metamodelle Auswirkungen auf die gesamte Software haben kann, sodass dies nur nach reiflicher Überlegung geschehen sollte. Bei Änderungen an diesen empfiehlt sich ein anschließender Regressionstest um sicherzustellen, dass durch die Einführung eines *Constraints* nicht die Funktionsweise der Software beeinträchtigt wird. Dies kann durch die Verwendung von Objekten in verschiedenen Kontexten passieren.

Das *Constraint* **input widget must have event**, welches in Kapitel 3.3.4 ausführlich erläutert ist, stellt ein Beispiel für einen Fall dar, in welchem die Implementierung eines *Constraints* über das ECore-Modell nicht möglich ist. Da ECore-Modell-*Constraints* nicht abgeschaltet werden können und ein *Input Widget* im Regelmodell anderen Bedingungen genügen muss als im *Structural UI*-Modell, ist hier eine Realisierung als ECore-Modell-*Constraint* nicht möglich.

2.3.2 OCL

Die *Object Constraint Language* (OCL) stellt eine Erweiterung von *Unified Modeling Language* UML dar, mit welcher Einschränkungen für UML-Modelle definiert werden können.

1997 wurde *Unified Modeling Language* (UML) als Standard für objektorientierte Analyse und Design eingeführt. Durch die zahlreichen darin spezifizierten Diagramme lassen sich auf einheitliche Weise Objektstrukturen beschreiben und Verhalten modellieren. Damit lassen sich die Konstrukte eines Systems visualisieren, spezifizieren und dokumentieren. [Son03]

Es können allerdings nicht alle Aspekte eines Modells mit UML modelliert werden. Speziell zusätzliche Einschränkungen für Objektbeziehungen und Objekteigenschaften, werden oft in natürlichsprachlicher Form hinzugefügt. Die Praxis hat gezeigt, dass dies häufig in Zweideutigkeiten der Formulierung resultiert. Um dies zu verhindern wurden schon früher formale Sprachen definiert, welche allerdings ein großes mathematisches Hintergrundwissen erforderten und für den durchschnittlichen UML-Modellierer ein Hindernis darstellten. Um diese Lücke zu füllen, wurde die *Object Constraint Language* (OCL) entwickelt, als formale Sprache, welche dennoch einfach zu lesen und zu schreiben ist. [OMG06]

Dies führt dazu, dass Zweideutigkeiten vermieden werden können, Einschränkungen automatisch überprüft werden können und eine automatische Codegenerierung möglich wird.

Es folgt eine kurze Zusammenfassung der wichtigsten Eigenschaften und Einschränkungen, welche beim Arbeiten mit OCL beachtet werden müssen.

OCL basiert auf einer dreiwertigen Logik. Das heißt, Ausdrücke werden auf die Werte *true, false, undefined* abgebildet. *Undefined* ist dabei der Rückgabewert einer Operation, wenn diese fehlschlägt. Dies kann zum Beispiel passieren durch den Zugriff auf ein Element einer leeren Menge, Fehler beim *Typecasting* oder dem Aufruf einer Funktion auf einem null-Objekt. Es führt dazu, dass dreiwertige Wahrheitstabellen, wie in Tabelle 2.2 dargestellt, benötigt werden.

not

0	1
1	0
?	?

and

	0	1	?
0	0	0	0
1	0	1	?
?	0	?	?

or

	0	1	?
0	0	1	?
1	1	1	1
?	?	1	?

Tabelle 2.2: Wahrheitstabellen - dreiwertige Logik (0:false 1:true ?:undefined) [Son03]

Zu beachten ist, dass OCL keine Programmiersprache darstellt. Deshalb ist es nicht möglich Programmlogik zu implementieren oder Programmflusskontrolle zu realisieren. [OMG06] Zum Bearbeiten und Überprüfen von mengenwertigen Typen, sogenannten *Collections*, stehen allerdings Iteratoren zur Verfügung, welche diese einer Überprüfung zugänglich machen.

Das Aufrufen von in Klassen definierten Methoden ist möglich, es dürfen allerdings nur solche verwendet werden, welche seiteneffektfrei sind. [OMG06]

2.3.3 Java Constraints

Einschränkungen für Modelle können auch in Java implementiert werden, wobei es durch die Mächtigkeit der Sprache mehrere Möglichkeiten gibt diese zu realisieren. Hier werden einige dieser Konzepte betrachtet, welche sich für diese Arbeit als relevant erwiesen.

Es gibt zahlreiche Möglichkeiten in Java *Constraints* zu implementieren, wie in [FGOG07] dargelegt. Prinzipiell wäre es möglich alle *Constraints* direkt im Quellcode zu realisieren z.B. durch bedingte Anweisungen. Dies hat allerdings die folgenden Nachteile:

1. Die Überprüfung eines Constraints an mehreren Stellen im Programm kann zu einer inkonsistenten Implementierung führen.

2. Des Weiteren ist es schwierig zu verifizieren, dass *Constraints*, welche in einem Analyse und Design Prozess gefunden wurden, ihre Umsetzung in den Quellcode gefunden haben.

3. *Constraints* können auch Kontrakte zwischen verschiedenen Systemmodulen betreffen. Eine implizite *Constraint*-Definition unterstützt z.B. nicht das *Design-by-Contract* Prinzip. Dies ist ein Konzept aus dem Bereich der Softwareentwicklung mit dem Ziel, das reibungslose Zusammenspiel einzelner Programmmodule durch die Definition formaler Verträge zur Verwendung von Schnittstellen zu ermöglichen. Diese Verträge gehen über eine statische Definition hinaus [8].

4. Manche Systeme können eine explizite Behandlung der Integritätsbedingungen zur Laufzeit benötigen.

Die Implementierung einer *Constraint*-Überprüfung in dedizierten Java-Klassen ist ein Ansatz, welcher den Code zur Modellintegritätskriterumsprüfung von dem der Applikationslogik trennt. Der *Constraint-Code* kann hierbei in *validate()*-Methoden vorliegen, welche mit entsprechenden Argumenten aufgerufen werden, wann immer ein spezielles *Constraint* überprüft werden soll. Dieser Ansatz erfordert einen Mechanismus, welcher die *validate()*-Methode zum passenden Zeitpunkt aufruft. [FGOG07]

Eine derartige Kapselung des *Constraint*-Codes in unterschiedliche Klassen erlaubt eine flexible Handhabung der Integritätsbedingungen. Diese können in einem *Constraint-Repository* registriert werden. Wann immer benötigt kann dieses *Repository* nach *Constraints* durchsucht werden, basierend auf unterschiedlichen Kriterien, wie zum Beispiel die Klasse des aufgerufenen Objektes oder die Signatur der aufrufenden Methode. Ein derartiges *Constraint-Repository* erlaubt des Weiteren das Hinzufügen, Entfernen, Aktivieren und Deaktivieren von *Constraints* zur Laufzeit. [FGOG07]

2.3.4 Validation Framework

Das *Validation Framework* von EMF unterstützt die Entwicklung von *Constraints* und die Realisierung einer Modellintegritätsprüfung, indem es die Kapselung der *Constraints* in eigene Klassen erlaubt und ein *Constraint-Repository* bietet. Durch die Verwendung dieses *Frameworks* kann der Fokus auf die Definition von geeigneten *Constraints* konzentriert werden.

Durch einen *Extension Point* sieht ein *Plugin* eine Registrierung vor, an der sich Instanzen vormerken lassen können, um ein *Plugin* zu ergänzen. Die registrierende Stelle ist dabei der *Extension Point* und die Erweiterung die *Extension*. Wird in einem *Plugin* eine Stelle erreicht, die ergänzt werden kann oder soll, so wird überprüft ob Instanzen registriert sind und, wenn dies der Fall ist, wird diese Funktionalität ausgeführt. Damit lässt sich ein *Plugin* erweitern, ohne dass es verändert werden muss. [HS08]

Im Folgenden eine Zusammenfassung von [5] über die Klassen und *Extension Points*, welche bei der Implementierung von *Constraints* mit diesem *Framework* zur Verfügung stehen und die Möglichkeiten, die diese eröffnen.

Der *Extension Point* `org.eclipse.emf.validation.constraintProviders` wird verwendet, um *Constraints* bereitzustellen. Es gibt zwei Arten von *Constraints*: statische und dynamische. Statische *Constraints* werden in der Datei plugin.xml deklariert und können in hierarchisch strukturierte Kategorien gruppiert werden. Diese *Constraint Provider* zielen auf ein oder mehrere EPackages ab, welche durch ihre *namespace-URI* identifiziert werden. Dynamische *Constraints* zielen auf

Situationen ab, in denen *Constraints* nicht statisch deklariert werden können, z.B. wenn diese in Modellen oder anderen Ressourcen definiert sind. Dynamische *Provider* deklarieren eine Klasse, welche das Interface *IModelConstraintProvider* implementiert. Diese Klasse sorgt dafür, dass *Constraints* zur Verfügung stehen, wenn entsprechende Situationen diese auslösen.

Über den *Extension-Point* `org.eclipse.emf.validation.traversal` können die *model-traversal*-Algorithmen angepasst werden. Dies ist nur für *Batch*-Validation relevant, da in der *Live*-Validation diese nicht ausgeführt werden und beschreibt, wie ein Teilbaum, ausgehend von der Auswahl durch den Benutzer, durchlaufen wird. Falls kein anderer Algorithmus angegeben wird, wird über den gesamten Teilbaum mit der Funktion `eAllContents()` iteriert.

Durch den *Extension-Point* `org.eclipse.emf.validation.constraintParsers` können weitere *Constraint*-Sprachen eingebunden werden. Das *Validation Framework* unterstützt von Haus aus zwei Sprachen: Java und OCL.

Der *Extension-Point* `org.eclipse.emf.validation.constraintBindings` erlaubt die Definition von `client contexts`, welche die Objekte definieren, auf denen eine Modellprüfung durchgeführt werden soll und bindet diese an *Constraints*. Der `client context` kann durch einen `enablement`-Ausdruck oder durch ein spezielles `selector`-Element, welches in einer Selector-Klasse definiert wird, gebildet werden. Dabei werden alle Modellelemente, welche die spezifizierten Bedingungen erfüllen, dem `client context` hinzugefügt. Der `client context` kann an *Constraints* oder *Constraint*-Kategorien gebunden werden, wobei in zweitem Fall jedes *Constraint* in der Kategorie an den `context` gebunden wird. Dies hat den Vorteil dass neue *Constraints* in einer Kategorie automatisch an den `context` gebunden werden, sogar wenn das *Constraint* in einem *Plugin* definiert wurde, welches diesen `context` nicht kennt.

Der *Extension-Point* `org.eclipse.emf.validation.validationListeners` wird verwendet, um *Validation Listener* für das *Validation Service* `org.eclipse.emf.validation.service.ModelValidationService` zu definieren. Das *Validation Service* benachrichtigt diesen *Listener* jedesmal, wenn eine Validation vorgenommen wurde. Dies kann dazu verwendet werden, wenn *client-Plugins* Informationen über die Validation benötigen, bevor sie geladen werden. Dieser Listener kann auch im Code zur Laufzeit durch die Methode `ModelValidationService.addValidationListener()` registriert werden.

Das *ModelValidationService* koordiniert den Aufruf der Validation. Es definiert eine *single-factory* Methode zur Implementierung des `IValidatior` für die *Batch-* und *Live-Evaluation-Modi*. Die Validatoren prüfen ein oder mehrere Objekte auf einmal. Welche Objekte als *Input* akzeptiert werden, hängt vom Evaluierungsmodus ab. Je nach Konfiguration melden sie die erfolgreiche Überprüfung von *Constraints* oder auch das Auftreten von Fehlern, wobei diese Ergebnisse vom Typ `IValidationStatus` sind. Der `ILiveValidator` prüft EMF-Notifications, während der `IBatchValidator` EObjects prüft und eine Fortschrittsanzeige unterstützt.

Das *Framework* stellt mit `org.eclipse.emf.validation.xml.IXmlConstraintParser` eine Implementierung eines *XML-Constraint-Parser-API* zur Verfügung, welche *XML-Constraints* in OCL unterstützt. Die Klasse *OclConstraintParser* ist eine *Constraint Parser* Implementierung, welche Instanzen der Klasse *OclModelConstraint* aus XML-Constraint-Deskriptoren erstellt. Unter Verwendung der *Query*-Klasse werden Modellelemente gegen OCL-*Constraint*-Ausdrücke getestet.

Des Weiteren ist es möglich die Modellprüfung direkt aus dem Code aufzurufen.

```
ValidationClientSelector.setRunning(true);
```

```
IBatchValidator validator = (IBatchValidator)
     ModelValidationService.getInstance().newValidator(EvaluationMode.BATCH);
validator.setIncludeLiveConstraints(true);
IStatus status = validator.validate(projectSpace);
ValidationClientSelector.setRunning(false);
```

In `status` ist das Ergebnis der Prüfung enthalten, welches vom Editor benutzt wird, um die Regelverletzungen in der *Problems-View* anzuzeigen und entsprechende Fehlermeldungen auszugeben. [3]

2.3.4.1 OCL im EMF

OCL-*Constraints* werden direkt in die Datei plugin.xml geschrieben oder in einer eigenen Datei abgelegt, mit einem Verweis auf diese in der plugin.xml.

Constraints können auch fehlerhaften Code enthalten. Zur Laufzeit werden Fehler, die in OCL-*Constraints* auftreten, nicht zurückgemeldet, einzig ein Hinweis des Typs *Informing* weißt darauf hin, dass ein Fehler aufgetreten ist und dieses *Constraint* deaktiviert wurde. Dieses lässt sich auch nicht wieder aktivieren und nach einer Änderung am *Constraint* muss die Testinstanz neu gestartet werden, um dieses erneut auszuführen. Damit ist eine relativ unkomfortable Variante des Erstellens gegeben. Besser ist es *Constraints* in der OCL-Konsole zu entwickeln und die fertigen *Constraints* dann in die Datei plugin.xml einzufügen.

Ein Debuggen von OCL-*Constraints* kann anhand der Fehlermeldungen in der Konsole, nach aktivieren der *Traces* in der *Runtime*-Konfiguration, durchgeführt werden.

2.3.4.2 OCL-Konsole

Mit der OCL-Konsole können OCL-*Constraints* eingegeben und auf den im Editor ausgewählten Objekten ausgeführt werden.

Die Konsole unterstützt *Code-Completion*, indem es für Objekte, die in diesem enthaltenen Objekte anzeigt und für Objekte die seiteneffektfreien Methoden angibt, welche in OCL verwendet werden dürfen. Dies stellt eine wesentlich bequemere Form der Erstellung von OCL-*Constraints* dar, da diese Funktionalität im *Plugin*-Editor nicht unterstützt wird.

Die OCL-Konsole ist im OCL Beispiel enthalten. Sie kann durch Window → Show View → Console und Auswahl von „Interactive OCL“ gestartet werden.

Als genereller Ansatz bietet es sich an, zuerst die zu überprüfenden Objekte zu sammeln und in einem zweiten Schritt dann auf diesen *Collections* die erforderlichen Integritätsprüfungen durchzuführen.

3 Realisierung von Modellintegritätsprüfungen für das UCP-Framework

In diesem Kapitel werden zuerst die angewandten Strategien zur Implementierung dargelegt. Dann folgt eine Beschreibung der Implementierung der Modellintegritätsprüfungs-*Plugins* und der implementierten *Constraints*. Abschließend werden die Ergebnisse der Evaluierung präsentiert.

3.1 Konzepte und Strategien

Aus den in Kapitel 2.3.4 genannten Möglichkeiten für die Implementierung im Rahmen des *EMF Validation Frameworks* ließen sich einige Strategien für die Realisierung ableiten.

Bei der Implementierung wurden nur *Batch-Constraints* verwendet und die Möglichkeit von *Live-Constraints* aus mehreren Gründen vernachlässigt. *Live-Constraints* benötigen einen höheren Realisierungsaufwand, da für jedes *Constraint* definiert werden muss, durch welche Benutzeraktionen im Editor eine Integritätsprüfung gestartet wird. Für Transformationsregel- und *Structural UI-Constraints* sind *Live-Constraints* ohnehin unnötig, da diese Modelle nicht dazu gedacht sind in einem graphischen Editor bearbeitet zu werden. Im Diskurseditor zeigt sich, dass im Entwicklungsprozess eines Diskursmodells nahezu jede Aktion zahlreiche Fehler auslösen würde. So würde das Hinzufügen einer prozeduralen Relation *Joint* dazu führen, dass dieses zum Erstellungszeitpunkt keine *Links* zu anderen Objekten hat und entsprechende Fehlermeldungen auslösen. Diese *Links* werden erst in den nächsten Arbeitsschritten hinzugefügt. Aus diesem Beispiel ist ersichtlich, dass das Erstellen und Bearbeiten von Diskursmodellen gleichsam immer eine zwischenzeitliche Verletzung der Modellintegrität darstellt. Erst wenn das Modell fertig erstellt ist oder wenn im Entwicklungsprozess einzelne Objekte überprüft werden sollen, ist eine Prüfung sinnvoll. Hierbei ist dann auch die Fehleranzahl signifikant geringer und auf solche beschränkt, welche tatsächlich einer Behandlung bedürfen und nicht im nächsten Bearbeitungsschritt ohnehin behoben werden würden. Des Weiteren ist die Ausführung der *Live-Validation* von den Benutzeraktionen abhängig, je nach Arbeitsstil und benötigten *Constraints* führt dies zu einem unterschiedlich hohen Aufwand an Rechenleistung und Speicher.

Aus diesen Gründen wurde auf eine *Live-Validation* verzichtet. Die Überprüfung der Modelle wird vom Benutzer gestartet. Auch eine automatische Überprüfung der Modelle vor einer Modelltransformation ist möglich.

Damit das Modell im Editor ständig alle Modellintegritätskriterien erfüllt, wäre ein anderer Ansatz für den Editor von Nöten. Dabei würde der Benutzer nicht nur einzelne Elemente erstellen, sondern einen ganzen Zweig, als atomare Aktion kreieren und Modellelemente nicht verändern, sondern in andere Elemente transformieren, entsprechend den Modellintegritätskriterien. Dies könnte allerdings die Erstellung eines Modells komplizierter gestalten als die derzeitige Lösung.

Für die Modellintegritätsprüfung von Diskurs-, *Structural UI*- und Transformationsregel-Modellen wurde jeweils ein eigener *Validation*-Adapter implementiert. Dadurch lässt sich jeder Adapter auf alle Elemente eines Modells anwenden, und der Konfigurationsaufwand bleibt gering.

So weit es sinnvoll und realisierbar war, wurden die vorhandenen *Constraints* in OCL erstellt, da deren Entwicklung im Allgemeinen schneller zu bewerkstelligen war als entsprechende Java-*Constraints* zu implementieren. Im Folgenden darf daher angenommen werden, dass die konkrete Implementierung in OCL vorliegt, sofern nicht anders angegeben.

Eine komplette Liste aller implementierten *Constraints* findet sich in Anhang A. Im Falle von OCL-*Constraints* ist deren Code direkt eingefügt, bei Java-*Constraints* ist der Name der Klasse angegeben, in welcher diese implementiert sind. Des Weiteren ist festgehalten, auf welchen Objekten diese Überprüfung ausgeführt wird und welche Fehlermeldung durch dieses *Constraint* ausgegeben wird, falls die Überprüfung einen Fehler findet.

Die Reihenfolge der Erklärung im Text weicht von der im Anhang ab, da diese die Darstellung im Programm wiederspiegelt, welche auf deren Verwendung im Code zugeschnitten ist. Die Erklärung der *Constraints* in diesem Kapitel entspricht einem Diskursmodell besser und der logische Ablauf ist auf diese Art und Weise besser zu veranschaulichen.

3.2 Validation Framework

Das EMF besitzt ein *Validation Framework*, dessen grundlegende Funktionen bereits in Kapitel 2.3.1 dargelegt wurden. Hier wird nun gezeigt, wie dieses verwendet wurde, um die Modellintegritätsprüfung zu realisieren.

3.2.1 Implementierung des Validation Framework

Für die Verwendung des *Validation Frameworks* sind einige Einstellungen notwendig, welche im Folgenden erläutert werden.

Die Implementierung des EMF *Validation Frameworks* kann an den folgenden drei Beispielen studiert werden:

- OCL Example
- General Validation Example
- Validation Adapter Example

Deren Code kann über File → New → Example → in ein Eclipse-Projekt eingebunden werden. Um mit diesen Beispielen zu arbeiten, muss auch das *Library Example* eingebunden werden. Von

diesen Beispielen ausgehend gibt es einige Tutorials im Internet, welche das Einbinden einer Modellintegritätsprüfung in ein eigenes Projekt erleichtern. Anhand von [1] und [2] wurden die beiden Möglichkeiten der Implementierung studiert und auf deren Basis schließlich die drei *Validation*-Adapter für die zu prüfenden Modelle implementiert.

Der *Validation*-Adapter implementiert hierbei die Kernfunktionalität, welche durch die in dem *General Validation Example* gezeigten Möglichkeiten erweitert wird, in einem Editor ein Kontext Menü zu erstellen und an die Bedürfnisse anzupassen. Damit kann zum Beispiel die *Live-Validation* ein- oder ausgeschaltet werden oder eine Validation des gesamten Modelles gestartet werden. Da dies nicht benötigt wird, wurde nur der *Validation*-Adapter verwendet.

Ausgehend vom *Validation Adapter Example* muss dazu in der Datei Startup.java die Methode `earlyStartup()` modifiziert werden, indem das `EXTLibraryPackage` durch den Packagenamen des verwendeten Metamodelles ersetzt wird. Dann muss in der Datei plugin.xml die `NS-URI` an das eigene Modell angepasst werden. Nach einer Anpassung der *Dependencies* und der *Imports* ist der Adapter bereit und kann nach Definition von *Constraints* verwendet und an das Projekt angepasst werden. Eine ausführliche Anleitung dazu findet sich in [1]. Gestartet wird die Integritätsprüfung durch Rechtsklick auf ein zu prüfendes Modellelement in der Baumansicht und Klick auf den Kontextmenüpunkt Validate.

3.2.2 Validation-Plugins

Die Definition eines *Plugins* erfolgt in der Datei plugin.xml durch die Konfiguration der *extension points*.

Die *Constraints* werden in der Datei plugin.xml registriert, in Kategorien geordnet und ihrem Kontext zugewiesen. Dazu stehen zwei Möglichkeiten zur Verfügung. Man kann die Datei plugin.xml per Texteditor schreiben oder über die Ansicht „extensions“ die benötigten Eigenschaften in einem graphischen Editor eintragen.

Es werden zwei *extension points* benötigt:

```
extension point= “org.eclipse.emf.validation.constraintBindings“
extension point= “org.eclipse.emf.validation.constraintProviders“
```

In den `constraintBindings`, siehe Abbildung 3.1, wird zuerst der `clientContext` definiert, wobei im Feld `enablement` festgelegt wird, für welche Objekte eine Modellintegritätsprüfung, mit den, im `constraintProvider` definierten *Constraints*, durchgeführt wird. In diesem Fall muss es eine Instanz vom Typ EObject sein, welche in dem im Feld `value` angegebenen `namespace` vorkommt.

Anschließend werden dem `Context` im Feld `binding` *Constraints* zugewiesen. Dies kann, wie im Beispiel gezeigt, durch die Angabe einer oder mehrerer *Constraint*-Kategorien geschehen. Genauso wäre auch das Eintragen von einzelnen *Constraints* möglich. Die `categories` sind in einem hierarchischen Baum organisiert, wobei die einzelnen Hierarchieebenen durch einen Schrägstrich getrennt werden. Wird eine Kategorie angegeben, sind auch automatisch alle Unterkategorien an den Kontext gebunden.

Im `extension point constraintProviders`, siehe Codebeispiel Abbildung 3.2, werden die Kategorien definiert, in welche die *Constraints* anschließend eingeteilt werden. In der `Id` wird der gesamte Pfad eingetragen, unter welchem diese *Constraint*-Kategorie später in den Preferences der Testinstanz zu finden sein wird.

```
<extension point="org.eclipse.emf.validation.constraintBindings">
  <clientContext id="org.ontoucp.discourse.validation.adapter">
      <enablement>
         <and>
            <instanceof value="org.eclipse.emf.ecore.EObject"/>
              <test
               property="org.eclipse.emf.validation.examples.adapter.ePackage"
               value="http://www.ontoucp.org/discourse/1.1.0"/>
         </and>
      </enablement>
   </clientContext>
   <binding
      context="org.ontoucp.discourse.validation.adapter"
      category="ontoucp-discourse"/>
</extension>
```

Abbildung 3.1: Extension Point - Constraint Bindings

Im *Constraint Provider* wird zuerst der `namespace` definiert, auf welchem das *Constraint* `target` sensitiv ist. Im Punkt *Constraints* werden Unterkategorien definiert, wo die im weiteren Verlauf implementierten *Constraints* einzuordnen sind. Es besteht auch die Möglichkeit *Constraints* als `mandatory` zu definieren, was dazu führt, dass diese immer ausgeführt werden. Dann erfolgt die eigentliche *Constraint* Definition:

- **id**
  Jedes *Constraint* benötigt eine eindeutige Id.
- **name**
  Dieser spezifiziert, unter welchem Namen es später in den Preferences der Testinstanz angezeigt wird.
- **lang**
  Für ein jedes *Constraint* wird unter `lang` die Sprache definiert, in welcher das *Constraint* implementiert ist. Dies wird im Allgemeinen entweder OCL oder Java sein.
- **class**
  Im Fall von Java-*Constraints* muss die Klasse angegeben werden, in welchem sich der *Constraint*-Code befindet. Bei OCL-*Constraints* fehlt diese.
- **mode**
  Im Feld `mode` wird angegeben, ob es sich um ein *Live*- oder *Batch-Constraint* handelt. *Live-Constraints* werden zur Laufzeit überprüft, wenn eines der angegebenen Features aufgerufen wird. *Batch-Constraints* werden nach dem expliziten Aufruf der Modellintegritätsprüfung überprüft.
- **severity**
  Es gibt mehrere `severity`-Grade, durch welche angeben wird, wie schwerwiegend die Verletzung dieses *Constraints* ist. Diese reichen von `ERROR` über `WARNING` bis `INFO`.
- **statusCode**
  Schlussendlich muss für jedes Constraint ein `statusCode` definiert werden.

```
<extension
      point="org.eclipse.emf.validation.constraintProviders">
  <category
        name="Discourse Model Constraints"
        id="ontoucp-discourse/discourse">
      Constraints for validating discourse models
  </category>
    <constraintProvider cache="true">
      <package namespaceUri="http://www.ontoucp.org/discourse/1.1.0"/>
      <constraints categories="ontoucp-discourse/warnings">
         <constraint
              id="org.ontoucp.discourse.validation.constraint0101"
              lang="OCL"
              mode="Batch"
              name="name not empty"
              severity="WARNING"
              statusCode="0101">
           <message>
              {0} has no name
           </message>
           <target class="CommunicativeAct"/>
           <target class="Discourse"/>
           <description>element should have a name</description>
           <![CDATA[
             name.size() > 0
           ]]>
         </constraint>
      </constraints>
   </constraintProvider>
</extension>
```

Abbildung 3.2: Extension Point - Constraint Provider

- **message**
  In `message` wird für jedes *Constraint* die auszugebende Fehlermeldung definiert. {0} wird hier verwendet, um das Objekt, auf welchem die Überprüfung durchgeführt wurde, anzuzeigen.

- **description**
  Im Feld `description` kann eine Beschreibung des *Constraints* angegeben werden, welche auch in den *Preferences* angezeigt wird.

- **target**
  In `Target` wird die Objekt-Klasse angegeben, auf welcher das *Constraint* angewendet werden soll. Hier können für *Live-Constraints* auch Ereignisse durch das Feld `events` angegeben werden, durch welche eine Modellintegritätsprüfung zur Laufzeit ausgelöst wird.

Die *Constraint*-Namen wurden möglichst so vergeben, dass aus diesen hervorgeht, was diese überprüfen. Die Ids bestehen aus `constraint` und einer vierstellig gewählten Nummer, wobei

die ersten beiden Ziffern die Gruppe identifizieren, in welche dieses *Constraint* eingeordnet ist und die restlichen Ziffern die *Constraints* einer Gruppe durchnummerieren. Der `statusCode` ist gleich dieser Id-Nummer. Zum Speichern der Java-*Constraints* wird in jedem Adapter ein Package `org.ontoucp.{Modellname}.validation.adapter.constraints` verwendet. Als Modus wird immer `Batch` verwendet, aus den in Kapitel 3.1 genannten Gründen. Es werden nur die `severity` Grade `ERROR` und `WARNING` verwendet. `ERROR` wird für die Kennzeichnung von kritischen Modellfehlern und `WARNING` wird für Elemente, die möglicherweise einen Fehler enthalten, verwendet. Der *severity*-Grad *INFO* wird nicht verwendet. Dies stellte sich bei der Implementierung als Vorteil heraus, da Laufzeitfehler in *Constraints* bei der Modellintegritätsprüfung eine Fehlermeldung der *severity* `INFO` nach sich ziehen, und dadurch leicht zu identifizieren sind.

OCL-*Constraints* werden direkt in der Datei plugin.xml abgelegt, wobei es sich empfiehlt den *Constraint-Code* mit `<![CDATA[ OCL constraint ]]>` zu umgeben, da für *OCL-Constraints* häufig Sonderzeichen benötigt werden, welche im Standarttext eines plugin.xml nicht erlaubt wären und ungeklammert zu Fehlern führen würden.

Im Gegensatz dazu muss für jedes Java-*Constraint* eine eigene Klasse implementiert werden, welche das Interface `IValidate` implementiert und eine Fehlerliste bzw. eine Erfolgsmeldung zurück gibt. Diese Klasse muss auch in der Datei plugin.xml durch das Attribut `class= „org.ontoucp.{Modellname}.validation.myConstraint“` im Feld `Constraint` registriert werden.

Welche *Constraints* überprüft werden, hängt zum Einen davon ab, welcher Kontext für die Modellintegritätsprüfung angegeben ist, und zum Anderen hängt es davon ab, welche *Constraints* im Menüpunkt *Preferences* unter *Modelvalidation* aktiviert sind.

Verletzungen von *Constraints* werden nach der Prüfung in einem eigenen Fenster angezeigt bzw. wird die Meldung „Validation completed successfully“ ausgegeben. Fehlermeldungen bleiben in der *Problems-View* gespeichert, von wo aus durch Doppelklick auf den Fehler direkt zu dem fehlerhaften Objekt im Editor gesprungen werden kann.

Auch ein Aufruf der Modellintegritätsprüfung aus dem Code, z.B. vor der Transformation eines Modells ist, wie in Kapitel 2.3.4 angesprochen, möglich.

3.3 Implementierung der Constraints

Ausgangspunkt für die Entwicklung der *Constraints* stellte eine Liste von bekannten Fehlerquellen dar, zwei Bespielmodelle und natürlich das *UCP-Framework* mit Editoren für seine Modelle, auf welchem diese getestet werden konnten.

3.3.1 Implementierungs-Strategien

Im Zug der Erstellung der *Constraints* erwiesen sich einige Vorgehensweisen als zielführend.

Die Implementierung von OCL-*Constraints* erfolgte in der OCL-Konsole. Hier ließen sich diese wesentlich schneller und komfortabler erstellen und testen, als dies in der plugin.xml möglich war. Als zielführend erwies es sich, zuerst die Objekte einzusammeln und zu filtern, bis genau die benötigte Submenge übrig war, auf welcher dann die Überprüfung implementiert wurde. Schlussendlich wurde ein fertiges *Constraint* dann auf den fehlerfreien Beispielmodellen und einem Modell, welches genau den durch das *Constraint* abzufangenden Fehler enthält, getestet.

Für die Implementierung der Java-*Constraints* wurde eine Muster `validate()`-Methode erstellt, welche dann als Basis für alle weiteren *Constraints* verwendet wurde. Diese ist in Abbildung 3.3 dargestellt. Hierbei wird die Liste `problems` erstellt, welche dann mit den Verletzungen der Modellintegritätskriterien befüllt wird. Die Objekte, welche in jedem Fall bei der Implementierung an das *Constraint* angepasst werden müssen, um ein lauffähiges *Constraint* zu erhalten, sind rot unterwellt. Falls diese schlussendlich leer ist, wird eine Erfolgsmeldung zurückgegeben, ansonsten eine Liste der aufgetretenen Fehler.

```
public IStatus validate(IValidationContext ctx) {
    Content c = (Content)ctx.getTarget();
    List<IStatus> problems = new java.util.ArrayList<IStatus>();
    List<Modellobjekt> locus = new java.util.ArrayList<Modellobjekt>();

    locus.add(x);
    IStatus problem = ConstraintStatus.createStatus(
            ctx,
            locus,
            4,
            1234,
            errormessage,
            c);
    problems.add(problem);
    locus.clear();
    if (problems.size() > 0)
        return ConstraintStatus.createMultiStatus(ctx, problems);
    else
        return ctx.createSuccessStatus();

}
```

Abbildung 3.3: validate()-Methode

3.3.2 Diskursmodell-Constraints

Im Folgenden werden die Modellelement-Klassen eines Diskurses betrachtet, zuerst deren Eigenschaften dargelegt und hinsichtlich notwendiger Überprüfungen evaluiert. Anschließend werden die Beziehungen eines jeden Elements zu anderen Diskurselementen hinsichtlich notwendiger Bedingungen für die Modellintegrität betrachtet.

Der Ausgangspunkt für die Entwicklung der Diskursmodell-*Constraints* war zum Einen eine Liste von bekannten Fehlerquellen des Entwicklerteams, zum Anderen eine Evaluierung des Diskursmodelleditors, hinsichtlich der Möglichkeiten dessen, was modelliert werden kann, gegen das, was nach den eingangs erwähnten Theorien erlaubt ist und transformiert werden kann. Diese Constraints finden sich im Anhang A.

Die Gruppierung und Implementierung der *Constraints* wurde so vorgenommen, dass ein möglichst komfortables Arbeiten ermöglicht wird. Dazu wurden die *Warnings* in eine Gruppe zusammengefasst, sodass diese in den *Preferences* → *Model-Validation* durch ein einziges Häkchen abgeschaltet

werden können. Die restlichen *Constraints* wurden thematisch gruppiert. Dadurch können z.B. Relationen überprüft werden, ohne dass auch sämtliche *Adjacency Pairs, Communicative Acts* und *Inserted-Sequences* überprüft werden. Dadurch soll ein möglichst einfaches Debuggen eines Modelles möglich werden.

3.3.2.1 Diskurs

Ein DISCOURSE-Objekt hat folgende Eigenschaften:

- **goal** String Array
  Beschreibt das Ziel, welches durch diesen Diskurs erreicht werden soll.
- **name** String
  Gibt dem Diskurs einen Namen.
- **Action-Models** EObject Array
  Definiert die *Action*-Modelle, welche in diesem Diskurs Verwendung finden.
- **Domain-Models** EObject Array
  Definiert die *Domain of Discourse*-Modelle, welche in diesem Diskurs Verwendung finden.

Des Weiteren kann ein Diskurs die folgenden Objekte beinhalten:

- **nodes** Node Array
  Nodes stellen die Generalisierung von Relationen und *Adjacency Pairs* dar.
- **agents** SenderAgent Array
  Definiert die Aktoren dieses Diskurses.
- **communicativeActs** communicativeAct Array
  Beinhaltet alle *Communicative Acts* dieses Diskurses.
- **subDiscourses** Discourse Array
  Weitere Diskurse, welche in diesem Diskurs enthalten sind.
- **links** Link Array
  Diese verbinden Relationen und *Adjacency Pairs*.

Die Überprüfung, ob ein Diskurs-Objekt gültig ist, erfolgt durch das *Constraint* **check AM, DM, Agent, Roots**. Dieses wurde in Java realisiert, da die Überprüfung, ob ein Diskurs-Objekt die oberste Instanz darstellt, durch die Überprüfung `discourse.eContainer() == null` geschieht, welche in OCL nicht realisiert werden konnte. Im Hauptdiskurs-Element müssen zumindest ein *Action*-Modell, in den meisten Fällen das Basismodell, welches in der Datei basic.anm vorliegt, ein *Domain of Discourse*-Modell und zwei Agenten definiert sein. In *Inserted-Sequences* müssen eben diese Felder leer sein. Da Subdiskurse derzeit nicht implementiert sind, wird von deren Überprüfung abgesehen und ihnen zugehörige Objekte werden beim Erstellen von Objektmengen vernachlässigt.

Jeder Diskurs benötigt genau einen *Root-Node*, welcher den Ausgangspunkt für den Kommunikationsablauf in diesem Diskurs darstellt und dadurch definiert ist, dass er keinen *Parent* besitzt.

Die Funktion `discourse.getRootNodes()` liefert all jene Objekte, auf welche dies zutrifft und wäre auch seiteneffektfrei, sodass einer Lösung als *OCL-Constraint* nichts im Wege stehen würde. Der Doppelklick auf eine Fehlermeldung `just one root node allowed`, würde im Diskurseditor allerdings auf das Diskurs-Objekt springen, und vom Benutzer eine Suche verlangen, welches von all den diesem Diskurse zugehörigen Objekten das Fehlerhafte ist. Bei der Realisierung als *Java-Constraint*, wurde von der Möglichkeit der Definition eines `Locus` in der Fehlermeldung Gebrauch gemacht, sodass für jeden *Root-Node* nun eine Fehlermeldung generiert wird und der Editor auf Doppelklick zu dem entsprechenden Objekt springt.

Des Weiteren sollten in einem Diskurs noch durch die Eigenschaften `name` und `goal` ein Name und ein Ziel definiert sein. Diese sind allerdings nicht zwingend erforderlich und haben auf die Modelltransformation keinen Einfluss. Wohl aber dienen sie der Übersichtlichkeit und Verständlichkeit. Aus diesem Grund stellt ihre Abwesenheit und damit die Verletzung der *Constraints* **name not empty** und **goal not empty** keinen *Error* sondern lediglich eine *Warning* dar.

3.3.2.2 Modellstruktur

Die Topologie aller Objekte in einem Diskurs muss ein Baum sein, was durch das *Constraint* **check for loops** überprüft wird. Dies ist bei *Adjacency Pair*, *Communicative Act* und *Content-Objekten* implizit gegeben, da *Adjacency Pairs* und *Communicative Acts* keine Relationen als Nachfolger haben können und in *Adjacency Pairs* immer ein oder mehrere *Communicative Acts* eingetragen werden.

Bei RST-Relationen und prozeduralen Relationen besteht allerdings die Möglichkeit, diese zu einem Zyklus zu formen, in dem jede Relation den *Parent* einer anderen Relation darstellt, siehe Abbildung 3.4.

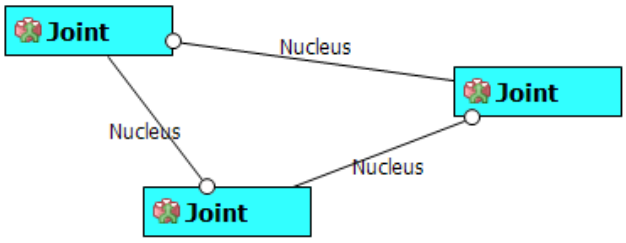

Abbildung 3.4: Zyklus aus Relationen

Diese hätten keinen *Root-Node* und wären damit in einem normalen Diskurs nie zu erreichen. Sie können im graphischen Diskurseditor gezeichnet werden, werden allerdings nicht angezeigt, da sie keinen Baum darstellen. Sie sind allerdings in der XML-Datei des Modells gespeichert und werden an die Transformationsmethode übergeben, sofern sie nicht abgefangen werden.

Für die Generierung von Endlosschleifen oder den Aufbau von Iteratorbasierten-Strukturen, welche wiederholt ausgeführt werden, steht die prozedurale Relation *IfUntil* zur Verfügung. Diese *IfUntil*-Konstrukte können transformiert und in ein *User Interface* umgewandelt werden.

Die Vorgehensweise hierbei ist, in einem ersten Schritt alle RST-Relationen und prozeduralen Relationen eines Diskurses „einzusammeln“. Dann wird für jede Relation versucht, im Baum „aufwärts zu gehen“, *Parent* für *Parent*, und jede Relation in einer Liste abzulegen, wobei bei jeder

neuen Relation überprüft wird, ob sich diese bereits in dieser Liste befindet. Ist dies der Fall, so bewegt sich der Algorithmus im Kreis und bricht mit einer entsprechenden Fehlermeldung ab. Wird hingegen eine Relation ohne *Parent*-Relation erreicht, d.h. ein *Root-Node*, so kann daraus geschlossen werden, dass eine Baum-Topologie vorliegt.

Durch das Ausgeben von Fehlermeldungen mit einem `Locus` auf das Element, welches beim „Aufwärtsgehen“ im Baum zweimal erreicht wird, findet sich schlussendlich eine Liste von Fehlermeldungen mit den Relationen, welche Teil der Schleife sind und von denen jede im Diskurseditor über die Problem-Ansicht von Eclipse erreicht werden kann.

Dieses *Constraint* wäre prinzipiell auch in OCL realisierbar. Zwar steht in OCL keine „Schleifenfunktionalität“ zur Verfügung, es wäre allerdings eine Lösung bis zu einer gewissen fix festgelegten Tiefe denkbar. Dadurch wäre aber die maximale Tiefe, bis zu welcher das Modell nach Schleifen durchsucht werden kann, fix festgelegt und die Implementierung wäre signifikant schwieriger als bei einer Implementierung in Java. Aus diesem Grund wurde dieses *Constraint* in Java realisiert.

3.3.2.3 Agenten

Ein Agent hat folgende Eigenschaften:

- **Id** String
  Gibt einem Agenten eine eindeutige Id.
- **Name** String
  Gibt einem Agenten einen Namen.
- **Performs** Communicative Act Array
  Definiert die *Communicative Acts*, welche von diesem Agenten ausgeführt werden.
- **Relations** Relation Array
  Definiert die Relationen und prozeduralen Relationen, welche von diesem Agenten ausgeführt werden.

Ein jeder Agent benötigt eine eindeutige Id. Diese muss entweder „A“ oder „B“ sein. Dies wird durch das Constraint **id not empty and A oder B** überprüft, welches in Kapitel 3.3.2.7 näher beschrieben ist.

Ein Agent kann einen Namen haben. Da dies lediglich der Verständlichkeit des Modells dient, ist diese Überprüfung als `Warning` durch das Constraint **name not empty** realisiert.

Da in die Felder `Performs` und `Relations` nur *Communicative Acts* , Relationen oder prozedurale Relationen eingetragen werden können und dies durch den Editor automatisch vorgenommen wird, sobald einem *Communicative Act*, einer Relation oder einer prozeduralen Relation ein Agent zugewiesen wird, ist eine Überprüfung dieser beiden Eigenschaften nicht notwendig.

3.3.2.4 Relationen

RST-Relationen und prozedurale Relationen haben folgende Eigenschaften:

- **Agent** Agent
  Gibt, wo benötigt, den Agent an, welcher die Entscheidung trifft.
- **Children** Link Array
  Definiert die *Links*, welche von dieser Relation ausgehen.
- **Parent** Link
  Definiert den *Parent-Link*.
- **Name** String
  Definiert einen Namen.

Die Relationen IfUntil, Condition und Elaboration benötigen einen Agenten. Dieser definiert welcher Kommunikationsteilnehmer über den weiteren Kommunikationsablauf entscheidet. Dass diese einen Agenten besitzen, wird durch das *Constraint* **some relations need an agent** überprüft.

Jede konkrete Spezialisierung einer Relation benötigt bestimmte *Children-Links*, diese werden im Folgenden ausführlich dargelegt. Bei den Modellintegritätsbedingungen für diese gibt es zahlreiche Gemeinsamkeiten, sodass sie in verhältnismäßig wenigen *Constraints* zusammen gefasst werden können. Tabelle 3.1 gibt an, welche Relation welche *Link*-Typen und wie viele davon haben kann bzw. haben muss.

Der *Parent-Link* kann nicht in den Eigenschaften eingetragen werden, dies wird vom Editor automatisch vorgenommen, wenn zu einer Relation ein *Link* gezogen wird bzw. wenn zu einem *Link* eine *Child*-Relation erstellt wird. Relationen ohne *Parent* stellen *Root-Nodes* dar. Die Überprüfung, welche sicherstellt, dass genau einer vorhanden ist, ist in Kapitel 3.3.2.1 beschrieben. Eine Relation kann einen Namen haben. Da dies lediglich der Verständlichkeit des Modelles dient, ist diese Überprüfung als `Warning` durch das *Constraint* **name not empty** realisiert.

RSTSingleNucleusRelationen müssen im Allgemeinen genau einen *Nucleus* und einen *Satellite* besitzen. Dazu werden in einem ersten Schritt alle *Children* überprüft, ob deren *LinkType Nucleus* oder *Satellite* ist und in einem zweiten Schritt sichergestellt, dass jeweils eines von jedem Typ vorhanden ist. Davon ausgespart bleibt die Relation Result, da es Sonderfälle gibt, in welchen diese nur einen *Nucleus* aber keinen *Satellite* besitzen darf. Da dies in den meisten Fällen eher einen Fehler denn ein gewünschtes Verhalten modelliert, wird vom *Constraint* **result 1 n** eine Warnung ausgegeben. In keinem Fall allerdings darf ein *Result* mehr als einen *Satellite* besitzen, was durch ein weiteres *Constraint* **result 1 n s=2..*** sichergestellt wird.

Multinucleusrelationen und die prozedurale Relation Sequence benötigen zwei oder mehr *Nuclei*. Diesem Umstand wird in dem *Constraint* **MultiNucleusRelation and Sequence 2..* n** analog dem der *RSTSingleNucleusRelationen* Rechnung getragen.

Eine Sequence benötigt eine eindeutige Ordnung ihrer *Nuclei*. Diese Ordnung der Zweige erfolgt anhand der Werte der Eigenschaft `condition`. Die *Conditions* aller *Nuclei* einer Sequence müssen eindeutig von einander unterscheidbar sein, was durch das *Constraint* **sequence ordered** überprüft wird.

Die prozedurale Relation Condition benötigt zwingend einen *Then-Link* und einen *Else-Link*, was durch **Condition 1 then 1 else** überprüft wird.

Ein IfUntil fordert einen *Tree*, die Zweige *Then* und *Else* sind optional und auf maximal einen limitiert. Falls die *Repeat-Condition* eines *Tree-Links* leer ist, wird dieser *Tree*-Zweig ständig

Relation	Nuclei	Satellite	Tree	Then	Else
Joint Contrast Alternative Sequence	2..*	0	0	0	0
IfUntil	0	0	1	0..1	0..1
Condition	0	0	0	0..1	0..1
Background Elaboration Annotation Title	1	1	0	0	0
Result	1	0..1	0	0	0

Tabelle 3.1: Links für Relationen und prozedurale Relationen

wiederholt. Da ein endlosschleifenbildendes IfUntil manchmal schwer zu finden sein kann, wird vom *Constraint* **tree has empty repeat condition** eine Warnung ausgegeben, um den User auf diesen Umstand hinzuweisen.

Falls ein *Else*-Zweig aber kein *Then*-Zweig vorhanden ist, muss angenommen werden, dass dieser vergessen oder gelöscht wurde. Dieser *Else*-Zweig verursacht keine Fehler bei der Transformation, er wird allerdings niemals ausgeführt werden, daher wird vom *Constraint* **else needs a then** nur eine entsprechende Warnung ausgegeben.

Die Überprüfung von Elaboration, Titel, Annotation und Background hinsichtlich der Typen ihrer *Satellite*-Zweige, wie in Kapitel 2.2.1.1 beschrieben, wurde nicht realisiert. Ein Grund dafür ist, dass dies nur der Darstellung dient, und ein Fehler hier sofort im fertigen *User Interface* ersichtlich ist. Ein weiterer Grund ist, dass dies keinen Absturz des Programms nach sich zieht, und das schlussendliche Aussehen des *User Interfaces* hauptsächlich durch die definierten Formatierungen beeinflusst wird.

3.3.2.5 Links

Links besitzen die folgenden Eigenschaften:

- **Condition**String
  Definiert die Bedingung, unter welcher der durch diesen *Link* repräsentierte Kommunikationszweig weiterverfolgt wird.
- **Condition *Abstract Syntax Tree* (AST)** String
  Enthält die Bedingung in geparster Form.
- **Repeat Condition** String
  Definiert die Bedingung, unter welcher der *Tree*-Zweig eines IfUntil wiederholt wird.
- **Repeat Condition AST** String
  Enthält die Wiederholungsbedingung in geparster Form.
- **Child** Relation oder Adjacency Pair
  Definiert das Zielobjekt, mit welchem die *Parent*-Relation verbunden wird.

- **Parent**-Relation
  Definiert die *Parent*-Relation.
- **Type** LinkType
  Definiert den Typ dieses *Links*.

`Condition` und `Repeat Condition` werden in einer speziellen Sprache formuliert, näheres dazu in Kapitel 3.3.2.8. Diese beiden Felder werden geparst und ihr Ergebnis in den zugehörigen AST Feldern abgelegt.

Jeder *Link* muss einen `LinkType` besitzen, welcher einen der folgenden Typen sein kann:

- **NUCLEUS**
- **SATELLITE**
- **THEN**
- **ELSE**
- **TREE**

Die von der Klasse ProceduralRelation abgeleiteten prozeduralen Relationen Sequence, IfUntil und Condition dürfen nur *Links* vom Typ `Tree`, `Then` und `Else` besitzen, während von der Klasse RSTRelation abgeleitete Relationen nur *Links* vom Typ `Nucleus` und `Satellite` besitzen dürfen.

Ein *Tree-Link* darf nur an IfUntil angehängt werden, was durch das *Constraint* **tree parent ifuntil** überprüft wird. *Then-* und *Else-Links* dürfen nur an IfUntil oder Conditions angehängt werden, dies wird durch **then, else parent ifuntil or condition** überprüft.

Ein *Then-Link* benötigt eine Bedingung, wann dieser ausgeführt werden soll, ansonsten wird der *Else-Link* gewählt, sofern dieser vorhanden ist. Durch **then must have a condition** wird überprüft, ob diese definiert ist. Damit wird allerdings nicht sichergestellt, dass diese Bedingung auch korrekt ist, lediglich deren Vorhandensein wird überprüft.

Ein *Tree-Link* darf eine *Repeat-Condition* besitzen, allerdings keine *Condition*. Dies wird durch das *Constraint* **in tree condition must not be set** überprüft.

Die Überprüfung von *Links* ist insofern wichtig, als diesen keine Namen zugewiesen werden können, wodurch eine Fehlersuche im Diskurseditor zu einer zeitraubenden und mühsamen Suche wird. Da *Links* einen maßgeblichen Einfluss auf das Modellverhalten haben, ist das Überprüfen ihrer Integrität ein relevanter Faktor, damit aus einem Diskursmodell ein *User Interface* generiert werden kann.

Links verknüpfen Relationen untereinander bzw. diese mit *Adjacency Pairs*. Dazu werden sie in die Eigenschaften `parent` und `child` eingetragen. Dies wird im graphischen Editor automatisch vorgenommen, sobald eine Verbindung erstellt wird. Wird einer Relation oder einem *Adjacency Pair*, welche bereits einen *Parent* haben, ein neuer *Parent* zugewiesen, so bleibt vom alten *Link* eine „Leiche“ ohne `parent` zurück, welche erst später bei einem Update gelöscht wird. Dies führt zu einer zwischenzeitlichen Verletzung der Modellintegrität, welche durch das *Constraint* **parent child set** erkannt wird, welches das Vorhandensein dieser beiden Parameter für jeden *Link* überprüft.

3.3.2.6 Adjacency Pairs

- **Opening Communicative Act** Communicative Act
  Gibt an, welcher *Communicative Act* dieses *Adjacency Pair* eröffnet.
- **Closing Communicative Acts** Communicative Act Array
  Enthält die *Communicative Acts*, welche dieses *Adjacency Pair* schließen.
- **Parent**-Link
  Definiert den *Parent-Link*.
- **Name** String
  Gibt dem *Adjacency Pair* einen Namen.

Ein *Adjacency Pair* stellt ein Dialogelement dar, welches typische Kombinationen von *Communicative Acts* verknüpft z.B. Frage-Antwort. Deshalb wird im *Constraint* **AP must have an opening act** überprüft, ob jedes *Adjacency Pair* einen *Opening Communicative Act* hat, welcher eine Frage oder ein Informing darstellt.

check closing CA stellt sicher, dass *Closing Communicative Acts* vorhanden sind, sofern der *Opening Communicative Act* nicht vom Typ Informing ist. Welche *Communicative Acts* vom Typ *Opening* oder *Closing* sind, ist aus Tabelle 2.1 ersichtlich.

Ein *Adjacency Pair* kann einen Namen haben. Da dies lediglich der Verständlichkeit des Modelles dient, ist diese Überprüfung als *Warning* durch das *Constraint* **name not empty** realisiert.

Der *Parent-Link* kann nicht in den Eigenschaften eingetragen werden, dies wird vom Editor automatisch vorgenommen, wenn zu einem *Adjacency Pair* ein *Link* gezogen wird bzw. wenn zu einem *Link* ein *Child-Adjacency Pair* erstellt wird. *Adjacency Pairs* ohne *Parent* stellen *Root-Nodes* dar. Die Überprüfung, welche sicherstellt, dass genau einer vorhanden ist, ist in Kapitel 3.3.2.1 beschrieben.

3.3.2.7 Communicative Acts

- **belongs To** Agent
  Gibt den Agent an, welcher diesen *Communicative Act* durchführt.
- **Opening Communicative Act Parent** Adjcency Pair
  Gibt das *Adjacency Pair* an, welches durch diesen *Communicative Act* eröffnet wird.
- **Closing Communicative Act Parent** Adjacency Pair
  Gibt das *Adjacency Pair* an, welches durch diesen *Communicative Act* beendet wird.
- **Id** String
  Gibt dem *Communicative Act* eine Id.
- **Name** String
  Gibt dem *Communicative Act* einen Namen.

Ein *Communicative Act* kann einen Namen haben. Da dies lediglich der Verständlichkeit des Modells dient, ist diese Überprüfung als *Warning* durch das *Constraint* **name not empty** realisiert.

Die *Communicative Acts* eines *Adjacency Pairs* repräsentieren die Dialogelemente. Diesen *Communicative Acts* muss ein Agent zugewiesen werden, das heißt, es muss angeben werden, welcher Kommunikationsteilnehmer die Frage stellt und welcher die Antwort darauf gibt. Falls Frage und Antwort demselben Agent zugewiesen sind, stellt dies ein Selbstgespräch dar, welches dem Kommunikationsprozess nicht dienlich ist. Um dies zu verhindern, wird durch **different opening and closing CA - Agent** überprüft, ob der *Opening Communicative Act* eines *Adjacency Pair* einen Agenten hat, welcher sich von den Agenten der *Closing Communicative Acts* unterscheidet.

Es ist zu beachten, dass ein *Adjacency Pair* nur 1 *Opening Communicative Act* haben kann, wohl aber mehrere *Closing Communicative Acts*. *Opening Communicative Acts* müssen vom Typ `Question, Request, Offer` oder `Informing` sein, dies wird durch das *Constraint* **openingCA of type cQ oQ Req Off Inf** überprüft, entsprechend Tabelle 2.1.

Closing Communicative Acts müssen vom Typ `Answer, Accept, Reject` oder `Ok` sein, was durch **closingCA of type Answ Acc Rej ok** überprüft wird.

Ein *Communicative Act* könnte durch Definition sowohl eines *Closing-* als auch eines *Opening Communicative Act-Parent* zwei *Adjacency Pairs* bzw. demselben *Adjacency Pair* zugeordnet werden. Damit würde zum Einen die Modelltopologie keinen Baum mehr darstellen und zum Anderen ein Selbstgespräch modelliert. Dieser Fehler wird auch durch die Überprüfung des Typs im vorhergehenden Absatz abgefangen, durch das *Constraint* **opening xor closing** wird aber die Fehlerursache besser aufgezeigt.

Des Weiteren wird durch **check content not empty** überprüft, ob *Communicative Acts* vom Typ `Accept, Answer, Ok` und `Reject` ein *Content*-Objekt besitzen und in diesem eine Spezifikation vorhanden ist.

Die Überprüfung, ob korrekte, das heißt der Tabelle 2.1 entsprechende, *Communicative Act*-Paare vorliegen, wird von den drei *Constraints* **Q adjacent to Answ, req adjacent to inf acc rej** und **Off adjacent to acc rej Ok** besorgt.

Die Eigenschaft Id soll *Communicative Acts* und Agenten eindeutig identifizieren, und wird modellübergreifend auch im *Structural UI*-Modell verwendet, wo über die Id die *Communicative Acts* eingetragen werden, welche spezifiziert, woher die Daten für ein spezielles Widget kommen. Dies ließe sich durch das „Einsammeln“ aller *Communicative Acts* und Agenten, welche im Diskurs und sämtlichen *insertedDiscourses* vorhanden sind, und einem Vergleich unter all diesen bewerkstelligen. Eine schönere Lösung ist allerdings, im ECore-Modell die Eigenschaft Id verpflichtend und *unique* zu machen. Den Agenten, von denen es immer zwei geben muss, werden die Ids A und B zugewiesen, während *Communicative Acts* alle anderen Ids besitzen dürfen. Dies wird durch das *Constraint* **id not empty and A oder B** für Agenten überprüft und durch **id not empty and not A and not B** für *Communicative Acts*.

Ein im ECore-Modell als verpflichtend definiertes Stringobjekt muss nur vorhanden sein, d.h. ungleich `null` sein, wodurch auch ein Leerstring gültig wäre, darum wird überprüft, ob dessen Länge größer Null ist.

3.3.2.8 Content

Im *Content* wird die Aufgabe eines Dialogelements beschrieben und, was mit den *Action*- und *Domain of Discourse*-Modellelementen im Weiteren geschehen soll. Dies geschieht in einer SQL-ähnlichen Sprache, welche so weit wie möglich an den natürlichen Sprachgebrauch angepasst ist. Im *Content* können *Action*- und *Domain of Discourse*-Modellobjekte enthalten sein. Der *Content* bildet die Schnittstelle zwischen dem *User Interface* und der Applikationslogik. Ein *Content*-Objekt besitzt die folgenden Eigenschaften:

- **Specification** String
  Beschreibt den *Content*.
- **Specification AST** String
  Darstellung des *Content* in geparster Form.

Das *Content*-Objekt besitzt die `Specification`, welche zur Eingabe dient, und die `Specification AST`, welche die geparste Form der `Specification` anzeigt. Die für den User komfortabelste Lösung wäre, einen eigenen Parser zu schreiben, welcher die `Specification` auf semantische und syntaktische Korrektheit prüft, und im Fall eines Fehlers genau dessen Ort und Art angibt. Dies würde allerdings einen signifikanten Aufwand bedeuten, da hier für jeden *Content* zahlreiche *Action*- und *Domain of Discourse*-Modellelemente geprüft werden müssten.

Ein einfacherer und bedeutend schnellerer Weg ist, die `SpecificationAST` anhand formaler Parameter zu prüfen. Dazu wurde das *Constraint* **verify content objects versus the domain and action modell** in der Klasse ContentCheck.java implementiert.

Dadurch wird im ersten Schritt überprüft ob das Objekt `SpecificationAST` überhaupt vorhanden ist und nicht den String „mismatched token“ beinhaltet. Die Fehlermeldung „mismatched token“ beschreibt sowohl die Art des Fehlers als auch dessen Position im String. So wird zum Beispiel „no text“ bei der Eingabe von Leerzeichen ausgegeben.

Bei der Eingabe eines Strings kann es vorkommen, dass dieser nur zum Teil geparst werden kann und dann ein Abbruch erfolgt. In diesem Fall wird die alte *SpecificationAST* beibehalten. Dies stellt allerdings ein Manko des Parsers dar, welcher in absehbarer Zukunft weiterentwickelt wird, sodass kein *Constraint* implementiert wurde, um diesen Fehler zu entdecken.

Der zweite Schritt ist die Überprüfung, ob die Definition der `SpecificationAST` entweder eine Instanz der Interfaces `IAction` oder `INotification` ist. Wenn dies der Fall ist, darf angenommen werden, dass der Basisbefehl korrekt geparst werden konnte.

In einem dritten Schritt wird überprüft, ob der Typ des den *Content* beinhaltenden *Communicative Acts* zum Typ der Definition passt. In der Tabelle 3.2 sind die erlaubten Beziehungen angegeben.

Im vierten Schritt wird der Typ des *Communicative Acts* mit der verwendeten *Action* oder *Notification* aus den *Action*-Modellen verglichen, da bestimmte *Communicative Act*-Typen die Verwendung bestimmter *Actions* oder *Notifications* erfordern. Dabei ist zu beachten, dass diese nicht notwendigerweise aus dem Modell Basic stammen müssen, allerdings von dessen *Actions* und *Notifications* abgeleitet sein müssen. So muss z.B. eine *Closed Question* die *Action select* aus dem Basismodell oder eine Spezialisierung davon im *Content* haben. Welche *Communicative Acts* welche *Actions* oder *Notifications* aus dem Basismodell bzw. deren Spezialisierungen enthalten müssen, ist der Tabelle 3.2 zu entnehmen

Communicative Act - Typ	Content Typ	Basic Action
Informing	Notification	
ClosedQuestion	Action	Select
OpenQuestion	Action	Get oder Update
Accept/Reject	Notification	
Request	Action	
Offer	Action	

Tabelle 3.2: Bedingungen für den Content abhängig vom *Communicative Act* Typ

3.3.3 Transformationsregelmodell-Constraints

Im Folgenden werden die Modellelement-Klassen von Transformationsregeln betrachtet. Zuerst werden deren Eigenschaften dargelegt und hinsichtlich notwendiger Überprüfungen evaluiert. Anschließend werden die Beziehungen eines jeden Elements zu anderen Transformationsregelelementen hinsichtlich notwendiger Bedingungen für die Modellintegrität betrachtet.

Diese Transformationsregeln werden für die Modell-zu-Modell Transformation von Diskursmodellen zu *Structural UI*-Modellen verwendet. Sie beschreiben, welche Objektbeziehungen eines Diskursmodells auf welche Art und Weise im *Structural UI*-Modell dargestellt werden. Da Regeln vom User selbst erstellt werden können bzw. für jede Zielplattform erstellt werden müssen, existieren zahlreiche Regeln, welche angewendet werden können. Des Weiteren ist deren Erstellung ein schwieriges und detailreiches Unterfangen, welches noch durch den Umstand erschwert wird, dass zumeist viele Regeln zusammenspielen. Durch eine Überprüfung der Modellintegritätskriterien der Regeln kann zumindest eine formale Korrektheit sichergestellt und die Häufigkeit von Fehlern bei der Transformation aufgrund von Fehlern in *Structural UI*-Modellen, welche durch fehlerhafte Regeln erzeugt werden, reduziert werden.

Regeln besitzen die folgenden Teile:

- **name** String
  Gibt der Regel einen Namen.
- **priority** Integer
  Gibt die Wichtigkeit der Regel an, wenn entschieden werden muss, welche Regel verwendet wird, wenn ansonsten gleichwertige Regeln zutreffen.
- **source** EObject
  Gibt das zu transformierende Objekt im Diskursmodell an.
- **target** EObject
  Gibt das Zielobjekt an, welches kreiert oder modifiziert werden soll.
- **space** Integer
  `Space` definiert die Größe eines Elements. Diese wird als Kriterium der Entscheidungsfindung, wo dieses Element schlussendlich dargestellt werden soll, verwendet.
- **type** RuleType
  Gibt den Regeltyp an und bestimmt das grundlegende Verhalten. Eine Regel kann vom Typ `Create, Modify` oder `Delete` sein.

- **ruleSet** RuleSet
  Definiert die Regelgruppe, der diese Regel angehört.
- **discourse** Discourse
  Definiert ein Muster, welches im Diskursmodell vorhanden sein muss, damit die Regel angewandt wird.
- **structuralUI** Widget
  Definiert das *Structural UI*-Muster, welches erstellt wird.
- **additionalConstraints** Constraint Array
  Hier können zusätzliche Einschränkungen definiert werden, welche erfüllt sein müssen, damit die Regel ausgeführt wird.
- **mappings** Mapping Array
  Setzt bestimmte Diskurselemente und *Structural UI*-Elemente in Beziehung.

Jede Regel benötigt ein `Source-` und ein `Target`-Objekt, welche definieren, von welchem Diskursmodell-Objekt ausgehend welches Objekt im *Structural UI*-Modell bearbeitet wird. Deren Vorhandensein, wird im ECore-Modell durch die Eigenschaft `Lowerbound` 1 eingestellt. Durch `Upperbound` 1 wird erreicht, dass genau ein Objekt vorhanden ist.

Eine Regel besitzt des Weiteren ein *Source-Pattern* in der Form eines Diskursmodells, welches das Muster enthält, nach welchem im Diskursmodell gesucht werden soll, und ein *Target-Pattern* in der Form eines *Structural UI*-Modells, welches das Muster enthält, welches im *Structural UI*-Modell erstellt werden soll.

Im *Source-Pattern* vorkommende *Adjacency Pairs* können *Communicative Acts* besitzen. Wie im Diskursmodell sind nur solche *Communicative Act*-Kombinationen erlaubt, welche in Tabelle 2.1 aufgelistet sind. Da diese Diskursmodelle nur Muster darstellen, nach welchen gesucht werden soll, ist hier nicht zwingend gefordert, dass diese komplett vorhanden sein müssen. Auch ein *Adjacency Pair* mit nur einem *Communicative Act* stellt ein gültiges Muster dar, welches gesucht werden kann. Deshalb sind diese *Constraints*, **check openquestion answer, check closedquestion answer, check offer request - accept reject, check informing** und **check question answer** als *Warnings* ausgeführt.

`Source-` und `Target`-Objekte müssen sich in den *Source-* und *Target-Patterns* befinden, was durch das *Constraint* **source und target in rule** sichergestellt wird. Dazu werden die beiden Muster rekursiv durchgegangen und jedes Element der Modelle mit `Source` oder `Target` verglichen.

Dieselbe Vorgehensweise wird auf `Mappings` im *Constraint* **mapping only on elements of rule** angewandt. Mappings besitzen auch `Source-` und `Target`-Objekt, die in den Mustern vorhanden sein müssen.

Die Widgets aus dem *Structural UI*-Muster können wie alle Widgets die Eigenschaft `tracesTo` definiert haben. Diese beschreibt, für welches Element des Diskursmodelles das Widget im *Structural UI*-Modell erzeugt wurde. Im Fall einer Regel müssen diese `TracesTo`-Objekte auf Elemente des Diskursmusters dieser Regel verweisen. Dazu wird im *Constraint* **traces to ca/rel in rule** zuerst das Teil *Structural UI*-Modell nach `tracesTo` durchsucht und im Erfolgsfall das gesamte Diskursmodellmuster nach dem Objekt, auf welches `tracesTo` verweist.

Im Folgenden werden die *Constraints* zur Überprüfung von häufig vorkommenden Mustern in Regeln beschrieben. Diese müssen im Allgemeinen erfüllt sein, wenn der Agent des *Opening Communicative Acts* die Applikation ist. Es gibt allerdings Sonderfälle, sodass diese *Constraints* als *Warnings* ausgeführt sind.

- **check openquestion answer**
  Ein *Open Question - Answer* Paar sollte ein *InputWidget* bzw. eine Spezialisierung davon und einen Button mit der *Answer* im *Event* Feld erzeugen.
- **check closedquestion answer**
  Ein *Closed Question - Answer* Paar sollte ein *Listwidget* und einen *Button*, welcher die *Answer* im *Event* Feld enthält, erzeugen.
- **check offer request - accept reject**
  Ein *Request* oder *Offer*, welches *Accept* oder *Reject* besitzt, sollte einen *Button* mit diesem *Accept* und einen *Button* mit diesem *Reject* im *Event* Feld erzeugen.
- **check question answer**
  Ein *Adjacency Pair* mit einem *Question-Answer* Paar sollte ein *InputWidget* bzw. eine Spezialisierung davon mit dem *Event* der Antwort erzeugen.
- **check informing**
  Ein *Informing* sollte keine *InputWidgets* bzw. Spezialisierungen davon erzeugen und sollte zumindest ein *OutputWidget* bzw. eine Spezialisierung davon erzeugen.

3.3.4 Structural UI Constraints

Im Folgenden werden die Modellelement-Klassen eines *Structural UI*-Modells betrachtet. Zuerst werden deren Eigenschaften dargelegt, welche anschließend hinsichtlich notwendiger Überprüfungen evaluiert werden. Dann werden die Beziehungen eines jeden Elements zu anderen *Structural UI*-Elementen hinsichtlich notwendiger Bedingungen für die Modellintegrität betrachtet.

Das *Structural UI*-Modell ist das Ergebnis einer Modell-zu-Modell-Transformation eines Diskursmodelles zu einem *Structural UI*-Modell unter Berücksichtigung einer Auswahl von Transformationsregeln. Es kann dabei vorkommen, dass zahlreiche einfache Regeln auf ein einzelnes Modellelement wirken. Dies stellt einerseits einen Faktor der Mächtigkeit des *UCP-Frameworks* dar, da hier durch die Kombination einfacher und nachvollziehbarer Regeln komplexe Effekte erreicht werden können, andererseits stellt es eine immanente Fehlerquelle dar. Derartige Fehler können auf mannigfaltige und unvorhersehbare Weise eine Codegenerierung unmöglich bzw. fehlerhaft machen.

Structural UI-Modelle basieren auf WIDGETS, welche folgende Eigenschaften haben.

- **name** EString
  Gibt dem Widget einen Namen.
- **visible** EBoolean
  Gibt an, ob dieses Widget sichtbar ist.
- **enabled** EBoolean
  Gibt an, ob dieses Widget verwendet wird.

- **contentSpecification** EString
  Dieses Feld wird während dem ersten Transformationsschritt (Diskursmodell zu *Structural UI*-Modell) ausgewertet. Hier kann eine OCL-*Expression* angegeben werden, die auf das *Content*-Objekt des *Communicative Acts*, auf welchen das *tracesTo*-Feld verweist, angewendet wird. Dieses Objekt ist gegebenenfalls im *Domain of Discourse*-Modell spezifiziert.

- **content** EObject
  Im Fall, dass eine OCL-*Expression* im *contentSpecification*-Feld enthalten ist, enthält dieses Feld das Attribut des *Content*-Objekts, das von diesem Widget repräsentiert werden soll. Dieses Feld enthält ebenfalls einen Teil des Resultats der Auswertung der OCL-*Expression* aus dem ersten Transformationsschritt und ist für den zweiten Transformationsschritt (*Structural UI*-Modell zu *User Interface*-Quellcode) von Bedeutung.

- **contentReference** EReference
  Im Fall, dass die OCL-*Expression* im *contentSpecification*-Feld eine Referenz auflöst, wird in diesem Feld der Name der Referenz eingetragen. Dieses Feld enthält einen Teil des Resultats der OCL-*Expression*-Auswertung aus dem ersten Transformationsschritt und ist für den zweiten Transformationsschritt (*Structural UI*-Modell zu *User Interface*-Quellcode) von Bedeutung.

- **text** EString
  Enthält statischen Text oder eine Formattierungsvorgabe für das Widget.

- **width** EInt
  Definiert die Breite des Widgets.

- **height** EInt
  Definiert die Höhe des Widgets.

Des Weiteren können Widgets die folgenden Elemente beinhalten:

- **style** Style
  Gibt den zu verwendenden Style an.

- **tracesTo** EObject
  Verweist auf ein Element des Diskursmodelles.

- **layoutData** LayoutData
  Enthält Daten zur Darstellung dieses Widgets.

- **parent** Panel
  Definiert das *Panel*, welches dieses Widget enthält.

Von dieser Klasse WIDGET werden dann die Klassen INPUTWIDGET, OUTPUTWIDGET und PANEL abgeleitet, welche wiederum zahlreiche Spezialisierungen besitzen.

Um sicherzustellen, dass aus einem *Structural UI*-Modell ein *User Interface* generiert werden kann, muss es einigen formalen Regeln genügen, welche in den *Stuctural UI*-Modell *Constraints* implementiert sind. Es ist hier ausreichend, die Spezialisierungen der abstrakten Klasse WIDGET

auf Modellintegritätskriteriumskonformheit zu überprüfen. In den *Layout*-Klassen sind Werte spezifiziert, welche, sofern sie vorhanden sind, das Aussehen bestimmen. Für den Fall, dass ein *Layout*-Wert nicht angegeben ist, wird dieser bei der Codegenerierung berechnet, näheres dazu findet sich in [Lei10].

Ein *Structural UI*-Modell stellt einen auf Widgets basierenden Baum dar, welcher auf einem *Choice*-Widget basiert und dessen zweiter Level nur aus *Frames* bestehen darf. Die Integrität dieser Topologie wird durch die beiden *Constraints* **first level must be choice** und **second level must be frame** sichergestellt.

Die vier Parameter `content, contentReference, contentSpecification` und `text` spezifizieren die Aufgabe eines Widgets. `TracesTo` gibt an, auf welches Objekt des Diskursmodells dieses Widget verweist. Dazu müssen bzw. dürfen allerdings nicht alle gesetzt sein, sondern es sind nur spezielle Kombinationen davon zulässig, wie in Tabelle 3.3 dargestellt. Um dies sicherzustellen, wird durch das *Constraint* **check structural UI elements** ein jedes Widget auf das Vorhandensein eines gültigen Musters überprüft. Die Tabelleneinträge bedeuten dabei folgendes:

- **Ja**
  Element muss vorhanden sein.
- **Nein**
  Element darf nicht vorhanden sein.
- **X**
  Es ist egal, ob dieses Element vorhanden ist.
- **Self**
  Feld muss einen Basisdatentyp enthalten.
- **Formatter**
  Element muss Formatierungsinformation enthalten.

Besondere Relevanz hat, dass eine gesetzte `contentReference` immer ein Datum in einem `content`-Feld verlangt. Durch die allgemeine Gültigkeit dieser Einschränkung und den Umstand, dass sich dieses *Framework* weiterentwickelt und die definierten Muster ihre Gültigkeit auch verlieren können und dadurch abgeschaltet bzw. angepasst werden müssen, wurde hierfür ein eigenes *Constraint* **contentReference set implies content set** erstellt, obwohl dieses eine Modellelementbeziehung überprüft, welche auch durch das zuvor genannte *Constraint* überprüft ist.

InputWidgets und OutputWidgets stellen konkrete Spezialisierungen der Klasse Widget dar und werden als solche bei der Transformation eines Diskursmodells in ein *Structural UI*-Modell verwendet. Sie dienen bei der Transformation Basisregeln als zwischenzeitliche Platzhalter und werden später durch speziellere Transformationsregeln in weitere Spezialisierungen von InputWidgets und OutputWidgets umgewandelt. Schlägt diese Umwandlung fehl oder wurde vergessen, eine derartige Transformationsregel zu implementieren, kann es vorkommen, dass sich ein InputWidget oder ein OutputWidget im *Structural UI*-Modell befindet. Da aus diesen kein Code generiert werden kann, werden sie durch das *Constraint* **no input/output Widgets** aufgezeigt.

Es liegt auf der Hand, dass durch jede Eingabe des Benutzers eine Aktion im Programm ausgelöst werden muss. Diese wird im *Structural UI*-Modell durch ein `Event`-Objekt repräsentiert

Content	Ja
ContentReference	Ja
ContentSpecification	X
Text	Nein
TracesTo	Ja
Content	Nein
ContentReference	Nein
ContentSpecification	Self
Text	Nein
TracesTo	Ja
Content	Nein
ContentReference	Nein
ContentSpecification	Self
Text	Formatter
TracesTo	Ja

Content	Nein
ContentReference	Nein
ContentSpecification	Nein
Text	Ja
TracesTo	Ja
Content	Ja
ContentReference	Nein
ContentSpecification	Ja
Text	Nein
TracesTo	Ja

Tabelle 3.3: Gültige Widget-Muster

und ein jedes InputWidget muss ein solches besitzen. Da dieses im *Structural UI*-Muster der Transformationsregeln im Allgemeinen nicht vorhanden ist, schließt sich hier eine Realisierung dieses *Constraints* durch einen entsprechenden Eintrag im ECore-Modell aus und wurde in **input widget must have event** realisiert. Hier sei nochmals auf den Umstand verwiesen, dass ECore-Modell-*Constraints* immer überprüft werden, und auch in den *Preferences* nicht abgeschaltet werden können. Da in den Transformationsregeln *Structural UI*-Elemente verwendet werden, würden diese auch in den Transformationsregeln hinsichtlich der Erfüllung ihrer Modellintegritätskriterien geprüft werden.

Einen Sonderfall stellt die Klasse ListWidget dar, da sich dieses sowohl von der Klasse InputWidget als auch von Klasse Panel ableitet. Es wird zum Einen als Liste, aus welcher ein oder mehrere Einträge ausgewählt werden können, verwendet. Zum Anderen kann es auch ein Panel darstellen. Auf diese Weise kann mit ListWidgets z.B. eine Liste von Produkticons, von denen eines durch Klicken ausgewählt und verarbeitet wird, realisiert werden, aber auch eine Liste von Checkboxen, deren Auswahl mit einem *Button* abgeschickt wird. Welche Darstellungsform verwendet wird, hängt vom Typ des *Communicative Acts* ab, aus welchem das *Listwidget* kreiert wird. So soll zur Darstellung eines *Informing* mit mehreren Werten eine Liste erstellt werden, während aus einer *Closed Question* eine Auswahl aus einer Liste und eine Möglichkeit, diese abzusenden, vorgesehen wird.

Die Darstellungsform wird durch die Eigenschaft `renderingType` festgelegt, welcher die Werte `LIST`, `PANEL` und `FOLDOUT` annehmen kann. Je nach Verwendungsform sind andere Parameter erforderlich. Als *Panel* muss es ein *InputWidget* besitzen, was durch **list panel must have input widget** überprüft wird. Als Liste muss ihm ein *Event* zugeordnet sein, durch **list must have event except panel** überprüft und darf nur eine Spalte besitzen, welche in der Eigenschaft `colNumber` festgelegt und durch das *Constraint* **list panel must have input widget** überprüft wird.

3.4 Evaluierung der Modellprüfung

Eine Meta-Prüfung, d.h. eine Überprüfung der *Constraints*, welche die Modelle prüfen, erfolgte auf den Modellen bestehender Applikationen. Diese sollten als funktionierende Projekte keine Fehler bei der Modellintegritätsprüfung erzeugen. In den Fällen, wo dies doch passierte, wurden die Fehler überprüft und deren Ursache ermittelt. Anschließend wurde das *Constraint*, welches den Fehler ausgelöst hat, an Sonderfälle angepasst oder das Modell richtig gestellt.

Falsch positive Ergebnisse, d.h. wenn durch die Modellprüfung ein *Error* entdeckt wird, obwohl kein Fehler vorhanden ist, stellen ein schwerwiegendes Problem dar. Dies kann zum Einen dazu führen, dass Zeit auf die „Fehlerbehebung“ verwendet wird, obwohl das Modell korrekt ist. Zum Anderen kann durch die „Fehlerbehebung“ bzw. *Workarounds* das Modell komplizierter und schwerer verständlich werden. Allerdings würde eine vollständige Meta-Prüfung der implementierten *Constraints* die Prüfung aller möglichen gültigen Kombinationen von Modellelementen erfordern, was Aufgrund der hohen Anzahl nicht möglich ist.

Auf der anderen Seite stehen falsch negative Ergebnisse, wenn also die Überprüfung einen Fehler „übersieht“. Dies muss nicht notwendigerweise zu einem Absturz des Programmes führen, es kann auch in der falschen Darstellung im UI resultieren oder gar nicht bemerkbar sein. Eine vollständige Meta-Prüfung durchzuführen ist hier nicht möglich, da dazu alle möglichen Objektkombinationen hinsichtlich ihrer Gültigkeit evaluiert werden müssten. Es können allerdings Modelle kreiert werden, welche bestimmte Fehler enthalten. Wenn diese durch die Modellintegritätskriteriumsprüfung gefunden werden, kann das korrekte Erkennen von Fehlern in einem gewissen Umfang angenommen werden.

Vom Online Shop Beispiel wurden für jede Gruppe von *Constraints* Modelle abgeleitet und in diese den *Constraints* entsprechende Fehler eingebaut, welche dann durch eine Modellintegritätsprüfung gefunden wurden. Damit kann ein grundlegendes Funktionieren der implementierten *Constraints* angenommen werden, wobei allerdings nicht alle möglichen Fälle abgedeckt werden.

Für die Evaluierung wurden sechs bestehende Modelle verwendet:

- Online Shop
- Verhandlungsunterstützung
- Flightbooking
- Commrob-Kassa
- Commrob-Roboter
- Bike Rental

Die ersten fünf Modelle wurden bereits zu *User Interfaces* gerendert. Deren Kommunikationsmodelle, Regelmodelle und *Structural UI*-Modelle wurden überprüft.

Das Kommunikationsmodell Bike Rental stellt einen ersten Entwurf eines Kommunikationsmodells durch zwei erfahrene Benutzer dar. Dieses wurde ohne die Modellprüfung zu benutzen erstellt und wurde noch nicht in ein *User Interface* gerendert. An diesem Modell lässt sich sehen, welche Fehler selbst erfahrenen Benutzern machen und wie diese durch die Modellprüfung unterstützt werden können.

3.4.1 Evaluierung bereits gerenderter Modelle

Dies folgenden fünf Modelle wurden bereits zu *User Interfaces* gerendert, daher sollten diese Modelle keine Fehler enthalten. Damit aus diesen Modellen *User Interfaces* generiert werden konnten, wurden in diesen Modellen bereits Fehler behoben. Deren Kommunikationsmodelle, Regelmodelle und *Structural UI*-Modelle wurden überprüft.

- Online Shop
- Verhandlungsunterstützung
- Flightbooking
- Commrob-Kassa
- Commrob-Roboter

Kommunikationsmodelle
Die Modellprüfung hat die folgenden *Errors* und *Warnings* aufgezeigt:

- 1× vergessene Attribute und Parameter in *Content*-Objekten
  Es wurde ein fehlerhaftes *Content*-Objekt aufgezeigt, welches richtig gestellt wurde.
- 1× Verwendung falscher *Link*-Typen
  Dies ist durch die Änderung von Anforderungen an Modelle bzw. die Änderung der Software zu erklären.
- 1× *Alternative* ohne Agent
  Durch eine Änderung des *UCP-Frameworks* wurde das neue Konzept der *Mixed Initiative* eingeführt, sodass die Relation ALTERNATIVE keinen Agent mehr benötigt. Das Modell wurde an die neue Gegebenheit angepasst.
- 13× *Inserted Sequences* ohne *Action*- und *Domain of Discourse*-Modelle
  Durch eine Änderung des *UCP-Frameworks* müssen *Inserted Sequences* jetzt alle für sie relevanten *Action*- und *Domain of Discourse*-Modelle eingetragen haben. Durch eine Änderung des entsprechenden *Constraints* wird diesem Umstand entsprochen.

Structural UI-Modelle
Die Prüfung ergab folgende *Errors* und *Warnings*:

- 11× *Style* Name nicht *unique*
  Das `style heading`-Objekt war im Metamodell als `unique` und `id` definiert. Diese Eigenschaften wurden auf `false` gesetzt, da dies nicht den Anforderungen entspricht.
- 1× *Radiobutton*-Eigenschaft `selected` nicht gesetzt
  Die verpflichtende Eigenschaft *selected* von *Radiobuttons* in Kombination mit dem Fehlen eines *default values* war Ursache dieses Fehlers. Er konnte durch den Eintrag eines *default values* im Metamodell behoben werden.

- 1× *Adjacency Pair* mit den *Communicative Acts Closed Question* und *Answer* kreierte kein *List*-Widget.
  Ein *Communicative Act-Answer* kreierte kein *List-Widget*. Durch eine spezielle Regel wurde jedoch eine andere Darstellung einer Liste erzeugt.

- 3× *Do_not_render*-Regeln kreieren keine Widgets
  Do_not_render-Regeln sind für Spezialfälle geschaffen worden, in welchen keine Widgets erzeugt werden dürfen.

- 3× *Output Widget* im *Structural UI*-Modell
  Nach einer Transformation in ein *Structural UI*-Modell dürfen sich keine *Output Widgets* mehr in diesem befinden. Es sind nur Spezialisierungen der Klasse OutputWidget erlaubt. Deren Vorhandensein zeigt einen Fehler bei der Transformation auf und lässt darauf schließen, dass eine Regel im zweiten Transformationsschritt nicht angewandt wurde, wodurch ein *Target*-Objekt nicht korrekt erstellt wurde.

- 2× *Listwidget* ohne *Input Widget*
  Wenn wie in diesen beiden Fällen ein *List*-Widget lediglich zur Darstellung einer Liste und nicht zur Auswahl aus einer solchen verwendet wird, dann ist das Nichtvorhandensein eines *Input-Widgets* korrekt.

- 21× *TracesTo* verweist auf kein Objekt oder ein falsches Objekt
  Dies zeigt einen Fehler bei der Transformation auf und lässt darauf schließen, dass eine Regel angewandt wurde, allerdings beim Auflösen der Referenzen auf das Diskursmodell ein Fehler passiert ist. Zum Teil ist dies durch eine Änderung des *UCP-Frameworks* zu erklären, da sich durch die Einführung eines des *Screen Models* eine neue *Top-Level*-Struktur von *Structural UI*-Modellen ergibt. Dieses *Screen Model* wird automatisch erstellt und seine Objekte verweisen nicht auf Diskursmodell-Objekte. Daher wurde das *Constraint* geändert, welches überprüft, ob jedes *TracesTo*-Objekt im Diskursmodell enthalten ist und es liefert nun keinen *Error* mehr sondern eine *Warning*.

Regelmodelle
Es besteht die Möglichkeit neben den allgemeinen Transformationsregeln auch spezielle Transformationsregeln zu verwenden, welche für ein bestimmtes Kommunikationsmodell erstellt wurden. Fehler in Transformationsregeln müssen nicht zwangsläufig zu fehlerhaften *Structural UI*-Modellen führen, da nicht alle angewendet werden, siehe Kapitel 2.2.2.2. Eine Prüfung der allgemeinen Transformationsregeln und der speziellen Transformationsregeln, sofern welche vorhanden waren, ergab folgende *Errors* und *Warnings*:

- 33× keine Namen für *Constraints* in *Second-Level*-Regeln
  Namen sind optionale Eigenschaften, daher sind diese Modellintegritätsverletzungen lediglich *Warnings*. Den betroffenen Regeln wurde ein Name zur besseren Verständlichkeit gegeben.

- 1× kein *Mapping-Source*-Objekt
  Fehlende *Mapping-Source*-Objekte sind schwerwiegende Fehler, daher wurde diese Regel richtig gestellt.

- 2× kein *Mapping-Target*-Objekt
  Fehlende *Mapping-Target*-Objekte sind schwerwiegende Fehler, daher wurde diese Regel richtig gestellt.

- 2× *Mapping-Source*-Objekt oder *Mapping-Target*-Objekt nicht Teil des *Source*-Diskursmusters oder *Target-Structural UI*-Musters
  Eine Regel darf nur Beziehungen zwischen Diskursmodellobjekten und *Structural UI*-Modellobjekten herstellen, welche in ihren Diskurs- und *Structural UI*-Mustern angegeben sind. Daher wurden diese Fehler richtiggestellt.

3.4.2 Evaluierung des Bike Rental Kommunikationsmodells

Das Kommunikationsmodell Bike Rental stellt einen ersten Entwurf eines Kommunikationsmodells durch zwei erfahrene Benutzer dar. Dieses wurde ohne die Modellprüfung zu benutzen erstellt und wurde noch nicht in ein *User Interface* gerendert. An diesem Modell lässt sich sehen, welche Fehler selbst erfahrenen Benutzern machen und wie diese durch die Modellprüfung unterstützt werden können. Ein Ausschnitt des Bike Rental-Diskursmodells ist in Abbildung 3.5 dargestellt. Bei der Modellprüfung ergaben sich folgende *Errors*:

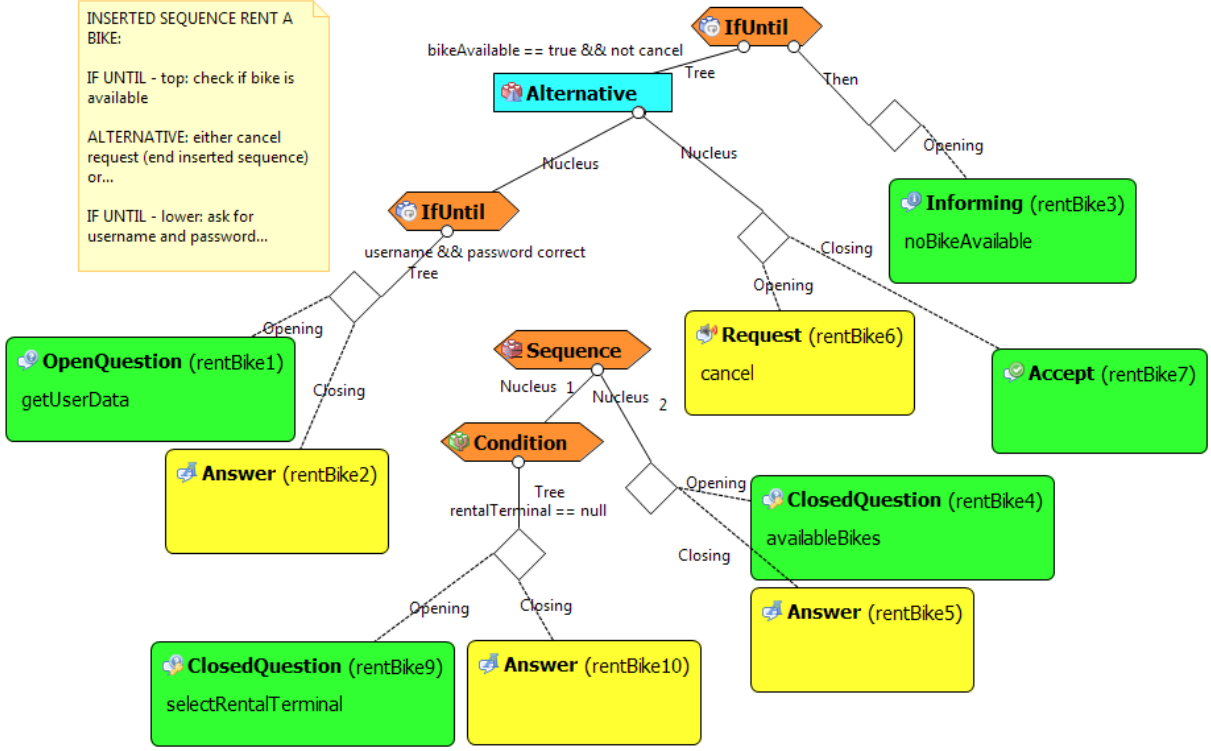

Abbildung 3.5: Ausschnitt aus dem Diskursmodell Bike Rental

- 2× *Opening Communicative Act Request* hat keinen entsprechenden *Closing Communicative Act*. Erlaubte Kombinationen sind in Tabelle 2.1 angegeben.
- 1× *Conditon*, siehe Abbildung 3.5, hat weder *Then*- noch *Else*-Zweig, besitzt aber einen *Tree*-Zweig.
- 4× Eine *Action* oder eine *Notification* befindet sich nicht in den *Action*-Modellen oder wurde nicht vom Basic-Modell abgeleitet. Ein Beispiel dafür ist die *Action* `selectRentalTerminal` in der `ClosedQuestion(rentBike9)`, welche sich nicht im *Action*-Modell `bikeRental` befindet.

- 4× *Open Question* hat kein `get` oder `update` im *Content*. Dies ist zum Beispiel bei der *Action* `getUserData` in der `OpenQuestion(rentBike1)` der Fall, da diese nicht von der Basic-Modell *Action* `get` abgeleitet wurde.
- 4× Es traten Fehler in *Content*-Objekten auf, welche in diesem Ausschnitt des Bike Rental Diskurses nicht ersichtlich sind.
- 1× *IfUntil* hat keinen Agent. Dieser Fehler befindet sich in einem anderen Teil des Diskursmodells.
- 1× Zwei *Root-Nodes* befinden sich in einem Diskurs. Diese sind das obere *IfUntil* und die *Sequence*.
- 3× *Conditon* eines *Then-Links* ist leer. Dies ist zum Beispiel beim oberen *IfUntil* der Fall.
- 1× *Then-Link* hat kein *Child*-Objekt. Dieser Fehler ist im graphischen Diskursedtior nicht ersichtlich.
- 1× *Tree-Link* befindet sich nicht an einem *IfUntil*. Dies ist der *Tree-Link* an der *Condition*.

Bei der Modellprüfung ergaben sich folgende *Warnings*:

- 30× Es wurde kein Name für ein Modellobjekt vergeben. Der Name ist nicht zu verwechseln mit der Id eines Modellobjektes, welche durch runde Klammern gekennzeichnet ist.
- 4× Die Eigenschaft *Goal* eines Diskurses oder einer *inserted Sequence* enthält einen Leerstring.
- 4× Es wurde eine mögliche Endlosschleife gefunden. Dies könnte zum Beispiel im unteren *IfUntil* der Fall sein, da dieses keinen *Then-Link* besitzt.
- 5× Es wurde in einem *Tree-Link* keine Wiederholungsbedingung definiert.

3.4.3 Ergebnisse der Evaluierung

Die *Errors* und *Warnings* aus allen Modellen sind in Abbildung 3.6 zusammengefasst. Hierbei wurden nur Überprüfungsergebnisse beachtet, welche tatsächlich Modellfehler darstellen und Überprüfungsergebnisse, welche zu einer Änderung von *Constraints* führten oder durch Änderungen des *Frameworks* bedingt sind vernachlässigt.

Die größte Fehlergruppe bilden mit 34,8% die Fehler im *Structural UI*-Modell in den *TracesTo*-Objekten. Diese entstehen ähnlich wie die Fehlergruppe der nicht aufgelösten Widgets (6,5%) bei der Transformation eines Diskursmodelles zu einem *Structural UI*-Modell, wenn eine Regel nicht oder nicht korrekt ausgeführt wird. Damit machen Transformationsfehler 41,3% der gesamten Fehler aus. Leider lässt sich nach der Transformation nicht mehr sagen, welche Regeln zu einem konkreten *Structural UI*-Modellobjekt geführt haben. Hier wäre es wünschenswert, mehr Möglichkeiten zur Fehlerbehandlung und Fehlersuche zur Verfügung zu haben, um nicht auf Logging- oder Debugergebnisse angewiesen zu sein. Es wäre hier zum Beispiel vorstellbar, in jedem *Structural UI*-Objekt zu speichern, welche Regel dieses kreiert bzw. modifiziert hat. Damit würde ein einfacheres Nachvollziehen der Transformation möglich sein.

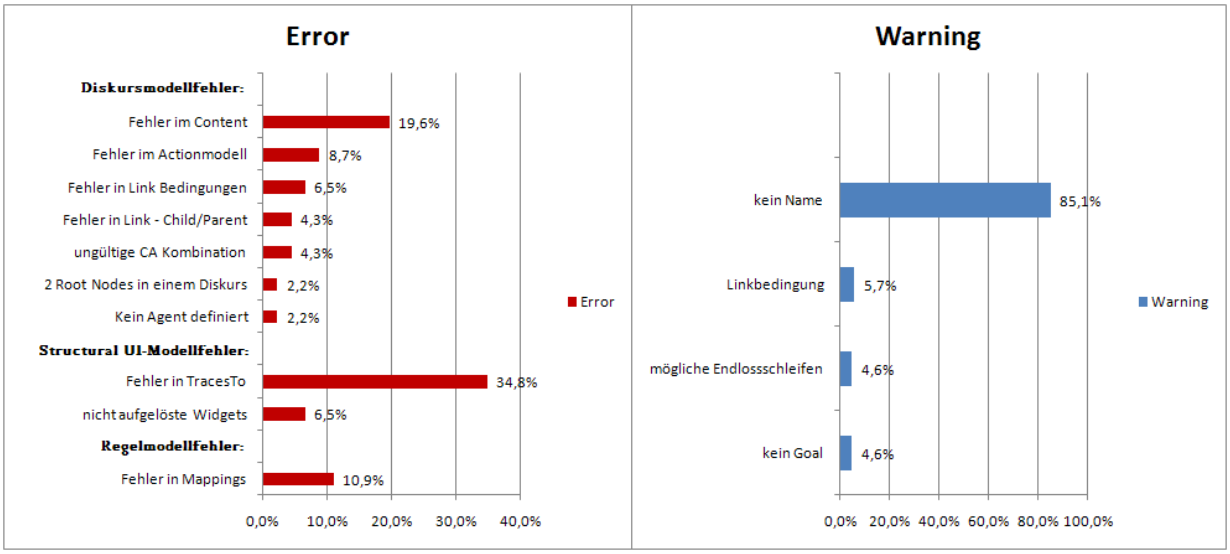

Abbildung 3.6: Evaluierungsergebnisse

Die zweitgrößte Fehlergruppe bilden mit 19,6% die Fehler in *Content*-Objekten. Diese sind verwandt mit den Fehlern im *Action*-Modell mit 8,7%. Durch die Prüfung werden allerdings nur strukturelle Fehler im *Content* erkannt, sodass die tatsächliche Fehleranzahl noch höher liegen kann.

Die drittgrößte Fehlerguppe sind mit 10,9% *Mapping*-Fehler in Regeln. Diese können zu fehlerhaften *Structural UI*-Widgets führen, wenn diese Regel verwendet wird. Die nicht aufgelösten *Input*- oder *Output-Widgets* können so entstanden sein.

Diskursmodellfehler sind Fehler, welche bei der Diskursmodellerstellung passiert sind. Obwohl es schwerwiegende Fehler sind, können sie rasch behoben werden, da die Modellprüfung die fehlerhaften Modellelemente angibt. Damit sind 47,8% aller Fehler im Diskursmodell entdeckt worden.

Die Gruppe der *Warnings* wird mit 85,1% angeführt von der nicht vorhandenen Benennung von Modellobjekten. Dies ist der Definition eines *Goals* (4,6%), d.h. eines Ziels, welches durch einen Diskurs erreicht werden soll, ähnlich. Obwohl diese keine kritischen Fehler darstellen, so sind die Vergabe guter Namen und die Definition von Diskurszielen der Verständlichkeit eines Modells sehr zuträglich.

Die restlichen *Warnings* sind durch *Links* bedingt. Zum Einen werden mögliche „Endlosschleifen“ (4,6%) aufgezeigt, welche in den vorliegenden Modellen durchaus gewollt sind. Zum Anderen wurde in *Tree-Links* keine Bedingung angegeben (4,6%).

3.4.4 Interpretation der Ergebnisse

Es können fehlerhafte Diskursmodelle modelliert werden, da der Diskurseditor die Modellierung von nicht transformierbaren Objekten und Objektstrukturen erlaubt, aus welchen kein *Structural UI*-Modell erstellt werden kann. Wenn die Modellprüfung keinen Fehler liefert und ein Diskurs dennoch nicht transformiert werden kann, so befinden sich die Fehler mit hoher Wahrscheinlichkeit in den *Link*-Bedingungen oder in den *Content*-Objekten, welche durch diese Modellprüfung nur oberflächlich auf das Vorhandensein bestimmter Strukturen geprüft werden.

Ohne Modellprüfung könnten sich *Input-* und *Output-*Widgets im *Structural UI-*Modell befinden, welche nicht dargestellt werden können. *TracesTo-*Objekte könnten nicht aufgelöst worden sein, wodurch Widgets auf falsche Elemente des Diskursmodells zeigen. Falsch eingetragene *Target-*Objekte können dadurch entstehen, dass Regeln nicht angewendet wurden.

Diese entdeckten Fehler können dazu führen, dass das generierte *User Interface* Fehler enthält, oder ein *Structural UI-*Modell nicht zu einem *User Interface* gerendert werden kann. Durch die Modellprüfung werden häufig vorkommende Modellfehler erkannt und die Modellobjekte auf das Vorhandensein einer gewissen Struktur hin geprüft. Eine Fehlerfreiheit kann allerdings nicht sichergestellt werden.

Alle *Structural UI-*Modelle, welche bereits gerendert werden konnten, passierten die Modellprüfung. Es wurden zwar einige *Errors* und *Warnings* ausgeben, allerdings sind diese durch die Änderung des *UCP-Frameworks* zu erklären. Die entsprechenden *Constraints* wurden an die neuen Gegebenheiten angepasst. Es ist zu beachten, dass diese Modelle von Personen erstellt wurden, welche mit dem *UCP-Framework* vertraut sind. Durch das Rendern und dem Vergleich mit dem intendierten Ergebnis wurden bereits viele Modellfehler behoben.

Im Zuge der Evaluierung und der Meta-Prüfung der Modelle habe ich selbst einige Modelle erstellt. Obwohl ich mit den Regeln vertraut bin, sind mir doch wesentlich mehr Fehler relativ zur Anzahl der modellierten Elemente unterlaufen, als im *Bike Rental-*Modell vorgefunden wurden. Daraus schließe ich, dass die Modellierung zu einem gewissen Teil auf Erfahrungswerten beruht und denke, dass Anfänger noch in viel stärkerem Ausmaß von der Modellprüfung profitieren können.

Zur Fehlersuche muss nun nicht jedes mal eine komplette Transformation der Modelle vorgenommen werden. Die Modellprüfung stellt ein Werkzeug dar, um *Copy-Paste-*Fehler zu finden.

Das Vorhandensein dieser Modellprüfung hat die Entwickler motiviert, eine automatisierte Modellprüfung vor der Transformation zu implementieren, welche als neue Funktionalität ins *UCP-Framework* integriert wurde.

Es wurden ca. drei Fehler pro Modell entdeckt, wobei dies sehr häufig Fehler in *Links* waren. Eine Umwandlung der entsprechenden *Batch-Constraints* in *Live-Constraints* und damit deren Prüfung zur Laufzeit wäre eine Möglichkeit, diese Fehler sofort bei der Diskursmodellerstellung abzufangen.

Die Weiterentwicklung des *UCP-Frameworks* führte dazu, dass einige *Constraints* angepasst werden mussten. Da *Inserted Sequences* nun *Action-* und *Domain of Discourse-*Modelle haben müssen, wurde das *Constraint*, welches zuvor deren Vorhandensein verbat, zu einer Überprüfung, ob diese beiden Modelle definiert sind.

Das Konzept der *Mixed Initiative* wurde eingeführt, welches erlaubt, dass in der Relation `Alternative` nun beide Agenten Entscheidungen treffen dürfen, während diese zuvor einen Agenten zugeordnet haben mussten.

Als gute Vorgehensweise empfiehlt es sich, dass parallel zur Erstellung in gewissen Zeitabständen Modellprüfungen durchgeführt werden. Dadurch ist eine ständige Anpassung und eine frühzeitige Erkennung von Fehlern möglich, sodass Folgefehler vermieden werden können und schlussendlich weniger Korrektur nötig ist.

4 Zusammenfassung und Ausblick

Es folgen eine Zusammenfassung dieser Arbeit und ein Ausblick auf mögliche Weiterentwicklungen.

4.1 Zusammenfassung

Der Zweck des *UCP-Frameworks* ist eine automatische Generierung eines *User Interfaces* aus einem Kommunikationsmodell. Dies geschieht durch einen zweistufigen Prozess, wobei ein *Structural UI*-Modell als Zwischenergebnis dient, aus welchem *User Interfaces* generiert werden können. Die Transformation eines Kommunikationsmodells in ein *Structural UI*-Modell wird anhand von Transformationsregeln durchgeführt.

Damit ein schnelleres Auffinden von Fehlern in diesen Modellen möglich wird, wurden die Metamodelle von Diskursmodellen, *Structural UI*-Modellen und Transformationsregeln um Modellintegritätskriterien erweitert, welche Einschränkungen darstellen, was in diesen Modellen modelliert werden kann.

Mit Hilfe des *Validation Frameworks* von EMF können diese Modelle auf die Einhaltung ihrer Modellintegritätskriterien hin geprüft und das Resultat dem Benutzer angezeigt werden. Das *EMF Validation Framework* wurde zur Implementierung gewählt, da dieses die Infrastruktur für eine Modellprüfung bietet und mit dem *UCP-Framework*, welches auf Basis des EMF implementiert wurde, gut zusammenarbeitet.

Die Implementierung der Modellintegritätskriterien erfolgte als *Constraints* im *Validation Framework*. Dabei wurden zur Implementierung die Sprachen OCL und Java gewählt. Zum Einen wurde OCL benutzt, da sich damit schnell einfache Objektbeziehungen und Wertebereiche von Objekteigenschaften überprüfen lassen. Zum Anderen wurde Java verwendet, um komplexe *Constraints* zu implementieren, welche außerhalb des Sprachumfangs von OCL liegen. Durch das *Validation Framework* werden Modelle auch hinsichtlich der Erfüllung der Modelldefinitionen, welche in den Metamodellen beschrieben werden, geprüft. Dadurch konnten auch durch die Anpassung der Metamodelle zu überprüfende Modellintegritätskriterien implementiert werden.

Diese *Constraints* sind Teil eines *Validation*-Adapters, welcher für die Modellintegritätsprüfung zuständig ist. Es wurde für Diskursmodelle, *Structural UI*-Modelle und Transformationsregeln ein eigener *Validation*-Adapter erstellt, in welchen die die benötigten *Constraints* implementiert wurden. Diese Adapter wurden mit dem *UCP-Framework* getestet und in dieses integriert.

Dadurch besteht nun im *UCP-Framework* die Möglichkeit, Diskursmodelle, *Structural UI*-Modelle und Transformationsregeln auf Modellintegritätskonformheit zu prüfen.

4.2 Ausblick

Diese Überprüfung soll nun dazu führen, dass korrekte Modelle rascher erstellt werden können und damit ein schnelleres Erstellen eines *User Interfaces* bzw. einer Applikation möglich wird.

Je mehr man modelliert, desto mehr Punkte entdeckt man, an denen noch weitere Verfeinerungen der Metamodelle vorgenommen werden könnten. Auch wenn von der Implementierung von zahlreichen möglichen weiteren Warnungen Abstand genommen wurde, da diese in ihrer Masse kaum mehr wahrgenommen würden, haben sich einige Punkte gezeigt, an denen eine sinnvolle Weiterentwicklung möglich wäre.

Eine Modellintegritätsprüfung im graphischen Editor und ein Markieren fehlerhafter Elemente darin, würde gemeinsam mit der Umwandlung geeigneter *Constraints* in *Live-Constraints* die Modellierung sinnvoll unterstützen.

Eine Überprüfung der *Conditions* und *RepeatConditions* von *Links*, ob diese korrekt geparst werden konnten und eine Überprüfung des Kommunikationsablaufs, ob alle Variablen, welche für diese *Conditions* benötigt werden, einen Wert zugewiesen bekommen haben, stellt eine weitere mögliche Weiterentwicklung dar.

Gewisse Modellelemente benötigen gewisse Kombinationen von Modellelementen. So müssen zum Beispiel in den Unterzweigen eines *Joint* alle *Opening Communicative Acts* denselben Agenten besitzen, damit dieser Kommunikationsablauf in einem Fenster dargestellt werden kann. Dazu muss allerdings noch evaluiert werden, welche Relationen derartiges verlangen, wie die erlaubten Muster aussehen und welche Modellelemente hier sinnvoll in Verbindung gebracht werden müssen. Generell könnte man alle möglichen Kombinationen von Modellelementen untersuchen, ob diese erlaubt bzw. sinnvoll sind. Dies hätte allerdings den Rahmen dieser Arbeit bei Weitem überschritten.

Bei der Entwicklung der Regel-*Constraints* hat sich gezeigt, dass diese sowohl vereinfacht als auch enger definiert werden könnten. In der jetzigen Form sind diese an grobe Muster gebunden. Diese gehen von heuristischen Annahmen über erwartete und sinnvolle Muster an *Adjacency Pairs* aus. Es könnten hier weitere Einschränkungen definiert werden, welche Elemente was erzeugen. Da das *UCP-Framework* allerdings weiterentwickelt wird, ist zum jetzigen Zeitpunkt noch nicht ersichtlich, wie dies genau aussehen werden. Die in der vorliegenden Arbeit entwickelten *Constraints* bieten einen Rahmen, um die häufigsten Fehler zu entdecken.

A Constraints

Im Folgenden eine Auflistung der implementierten *Constraints*.

Die Error Nummer identifiziert jedes *Constraint* eindeutig und ist gleich dem Status Code. Wobei die ersten beiden Ziffern die Gruppe angeben, zu welcher dieses *Constraint* gehört und die letzten beiden Ziffern die *Constraints* dieser Gruppe durchnummerieren.

Dann folgen der *Constraint*-Name und darunter die durch dieses *Constraint* ausgegebene Fehlermeldung. In der Fehlermeldung bedeutete {0}, dass an dieser Stelle das fehlerhafte Modellobjekt ausgegeben wird.

In der Spalte lang findet sich die Art des *Constraints*. Dies kann entweder OCL, Java oder ECore sein.

Unter *Target* sind die Modellobjekte, auf denen dieses *Constraint* ausgeführt wird, aufgelistet.

Wenn es sich um ein OCL-*Constraint* handelt, findet sich in der letzten Spalte der OCL-Code. Im Fall eines Java-*Constraints* ist die Klasse angegeben, in welcher dieses implementiert wurde. Sollte es sich um ein ECore-Constraint handeln, bleibt diese Spalte frei.

errornr	name and error message	lang	target	oclcode or java class
category	Discourse Constraints			
01xx	warnings			
0101	name not empty	ocl	SenderAgent	name.size() > 0
	{0} has no name		Node CommunicativeAct Discourse	
0102	goal not empty	ocl	Discourse	self.goal -> size() > 0
	{0} has no goal			
0103	tree has empty repeat condition	ocl	IfUntil	self.children -> select(type = LinkType::TREE) ->
	{0} can be an endless loop			forAll(l:Link| if (l.repeatCondition.size()=0).oclIsInvalid() then false else l.repeatCondition.size()>0 endif)
0104	else needs a then	ocl	IfUntil	self.children -> select(type = LinkType::ELSE) -> size() > 0 implies
	{0} has else without then			self.children -> select(type = LinkType::THEN) -> size() > 0
0105	result 1 n	ocl	Result	(self.children -> forAll(l:Link|l.type = LinkType::NUCLEUS or l.type = LinkType::SATELLITE))
	{0} has no satellite			and (self.children -> select(l:Link|l.type = LinkType::NUCLEUS) -> size() = 1) and (self.children -> select(l:Link|l.type = LinkType::SATELLITE) -> size() = 0)
02xx	Adjacency Pairs			
0201	AP must have an opening act	ocl	AdjacencyPair	self.openingCommunicativeAct -> size() = 1
	{0} has no opening communicative Act			
0202	check closing CA	ocl	AdjacencyPair	(self.closingCommunicativeActs -> size() = 0) implies
	{0} has no closing CA or opening CA is not Informing			(self.openingCommunicativeAct.oclIsTypeOf(Informing))
0203	diffrent opening and closing CA - Agent	ocl	AdjacencyPair	if (self.openingCommunicativeAct.oclIsTypeOf(Informing).not())
	{0} same opening and closing agent			forAll(belongsTo <> self.openingCommunicativeAct.belongsTo) then self.closingCommunicativeActs ->
else true endif
03xx	Communicative Acts			
0301	CA must have an agent	ocl	CommunicativeAct	self.belongsTo -> size() = 1
	{0} has no agent			
0302	check content not empty	ocl	CommunicativeAct	if (self.content.specification.size() > 0).oclIsInvalid()
	{0} has no valid content			then (self.oclIsTypeOf(Accept) or self.oclIsTypeOf(Answer) or self.oclIsTypeOf(Ok) or self.oclIsTypeOf(Reject)) else self.content.specification.size() > 0 endif
0303	opening xor closing	ocl	CommunicativeAct	(self.closingCommunicativeActParent -> size() = 1)
	{0} must be openingCA xor closingCA			xor (self.openingCommunicativeActParent -> size() = 1)

04xx	Links			
0401	then must hava a condition	ocl	Link	self.type = LinkType::THEN implies
	then must hava a condition			if (self.condition.size() > 0).oclIsInvalid()
				then false else self.condition.size() > 0 endif
0402	then, else parent ifuntil or condition	ocl	Link	(type = LinkType::THEN or type = LinkType::ELSE) implies
	{0} has no parent of type ifuntil or condition			(parent.oclIsTypeOf(IfUntil) or parent.oclIsTypeOf(Condition))
0403	tree parent ifuntil	ocl	link	(type = LinkType::TREE) implies (parent.oclIsTypeOf(IfUntil))
	{0} has no parent of type ifuntil			
0404	parent child set	ocl	Link	(self.parent -> size() = 1) and (self.child -> size() = 1)
	{0} hasnt parent and child			
0405	in tree condition must not be set	ocl	Link	if (type = LinkType::TREE)
	{0} tree must have repeat conditon but no condition			then
				if (condition.oclIsInvalid())
				then true
				else condition.size()=0
				endif
				else true
				endif
05xx	adjacent Communicative Acts			
0501	openingCA of type cQ oQ Req Off Inf	ocl	AdjacencyPair	self.openingCommunicativeAct.oclIsKindOf(Question) or
	{0} - openingCA not of type cQ oQ Req Off Inf			self.openingCommunicativeAct.oclIsTypeOf(Request) or
				self.openingCommunicativeAct.oclIsTypeOf(Offer) or
				self.openingCommunicativeAct.oclIsTypeOf(Informing)
0502	closingCA of type Answ Acc Rej ok	ocl	AdjacencyPair	closingCommunicativeActs ->
	{0} - closingCA not of type Answ Acc Rej ok			forAll(oclIsTypeOf(Answer) or oclIsTypeOf(Accept)
				or oclIsTypeOf(Reject) or oclIsTypeOf(Ok))
0503	Q adjacent to Answ	ocl	AdjacencyPair	self.openingCommunicativeAct.oclIsKindOf(Question) implies
	{0} - Question not adjacent to Answer			self.closingCommunicativeActs -> forAll(oclIsTypeOf(Answer))
0504	req adjacent to inf acc rej	ocl	AdjacencyPair	self.openingCommunicativeAct.oclIsTypeOf(Request) implies
	{0} - request not adjacent to acc rej Ok			(self.closingCommunicativeActs -> forAll(oclIsTypeOf(Ok)
				or oclIsTypeOf(Accept) or oclIsTypeOf(Reject)))
0505	Off adjacent to acc rej Ok	ocl	AdjacencyPair	self.openingCommunicativeAct.oclIsTypeOf(Offer) implies
	{0} - Offer not adjacent to Accept or Reject or Ok			(self.closingCommunicativeActs ->
				forAll(oclIsTypeOf(Accept) or oclIsTypeOf(Reject) or oclIsTypeOf(Ok)))
06xx	Relations			
0601	sequence ordered	ocl	Sequence	self.children -> forAll(I1, I2|I1 <> I2 implies I1.condition <> I2.condition)
	{0} is not properly ordered			
0602	ifuntil 1 tree 0..1 then 0..1 else	ocl	IfUntil	(self.children -> forAll(I|I.type = LinkType::TREE
	{0} hasnt 1 tree and 0..1 then 0..1 else			or I.type = LinkType::THEN or I.type = LinkType::ELSE))

				and (self.children -> select(type = LinkType::TREE) -> size() = 1) and (self.children -> select(type = LinkType::THEN) -> size() < 2) and (self.children -> select(type = LinkType::ELSE) -> size() < 2)
0603	Condition 1 then 1 else	ocl	Condition	(self.children -> forAll(l|l.type = LinkType::THEN or l.type = LinkType::ELSE))
	{0} hasnt 1 then and 1 else			and (self.children -> select(type = LinkType::THEN) -> size() = 1) and (self.children -> select(type = LinkType::ELSE) -> size() = 1)
0604	RSTSingleNucleusRelation 1 n 1 s	ocl	Background	(self.children -> forAll(l:Link|l.type = LinkType::NUCLEUS or l.type = LinkType::SATELLITE))
	{0} hasnt 1 nucleus and 1 satellite		Elaboration Annotation Title	and (self.children -> select(l:Link|l.type = LinkType::NUCLEUS) -> size() = 1) and (self.children -> select(l:Link|l.type = LinkType::SATELLITE) -> size() = 1)
0605	MultiNucleusRelation and Sequence 2..* n	ocl	Joint	(self.children -> forAll(l|l.type = LinkType::NUCLEUS))
	{0} hasnt 2..* N		Contrast Alternative Sequence	and (self.children -> size() > 1)
0606	some relations need an agent	ocl	Elaboration	agent.oclIsUndefined().not() or self.oclIsTypeOf(Annotation)
	{0} has no agent defined		IfUntil Alternative Condition	or self.oclIsTypeOf(Background)
0607	result 1 n s=2..*	ocl	Result	(self.children -> forAll(l:Link|l.type = LinkType::NUCLEUS or l.type = LinkType::SATELLITE))
	{0} has more than 1 Satellite			and (self.children -> select(l:Link|l.type = LinkType::NUCLEUS) -> size() = 1) and (self.children -> select(l:Link|l.type = LinkType::SATELLITE) -> size() > 1)
07xx	Discourse			
0701	verify content objects versus the domain and action r	java		org.ontoucp.discourse.validation.adapter.constraints.ContentCheck
0703	check AM, DM, Agent, Roots	java		org.ontoucp.discourse.validation.adapter.constraints.CheckDiscourse
0704	check for loops	java		org.ontoucp.discourse.validation.adapter.constraints.ModelStructure
08xx	Ids	ids		
0802	id not empty and A oder B	ocl	SenderAgent	id = 'A' or id = 'B'
	{0} id must be A or B			
0801	id not empty and not A and not B	ocl	CA	id.size() > 0 and id <> 'A' and id <> 'B'
	{0} id may not be empty and not "A" or "B"			
0803	id unique	ecore	SenderAgent CA	
	Structural UI - Constraints			
2001	check structural UI elements	ocl	InputWidget	(
	{0} does not apply to a valid pattern		OutputWidget	content -> size() = 0 and contentReference -> size() = 0 and if contentSpecification -> size() = 0

				then true else contentSpecification -> forAll(size() = 0) endif and if text = null then false else text.size() > 0 endif and tracesTo -> size() = 1) or (content -> size() = 1 and contentReference -> size() = 0 and if contentSpecification -> size() = 0 then false else contentSpecification -> forAll(size() > 0) endif and if text = null then true else text.size() = 0 endif and tracesTo -> size() = 1) or (content -> size() = 1 and contentReference -> size() = 1 and if text = null then true else text.size() = 0 endif and tracesTo -> size() = 1) or (content -> size() = 0 and contentReference -> size() = 0 and if contentSpecification -> size() = 0 then false else contentSpecification -> forAll(size() > 0) endif and if text = null then true else text.size() = 0 endif and tracesTo -> size() = 1) or (content -> size() = 0 and contentReference -> size() = 0 and if contentSpecification -> size() = 0 then false else contentSpecification -> forAll(size() > 0) endif and if text = null then false else text.size() > 0 endif and tracesTo -> size() = 1)
2002	first level must be choice first level object isnt Choice	ocl	Widget	self.parent = null implies self.oclIsTypeOf(Choice)
2003	second level must be frame second level object isnt Frame	ocl	Choice	self.parent = null implies self.widgets -> forAll(oclIsTypeOf(Frame))
2004	input widget must have event {0} has no event	ocl	Button ComboBox DateTimePicker TextBox ImageMap	event -> size() > 0

			TextField	
2005	colNumber = 1 except for renderingType Panel	ocl	ListWidget	colNumber = 1 or renderingType = ListRenderingType::PANEL
	{0}: colNumber != 1 or renderingType not panel			
2006	list must have event except panel	ocl	ListWidget	(self.event -> size() > 0) or renderingType = ListRenderingType::PANEL
	{0}: has no event or renderingType not panel			
2007	list panel must have input widget	ocl	ListWidget	renderingType = ListRenderingType::PANEL
	{0} has no input widget			implies (widgets -> select(oclIsKindOf(InputWidget)) -> size() > 0)
2008	if contentReference set implies contetn set	ocl	Widget	(contentReference -> size() > 0) implies (content -> size() > 0)
	{0} has no content			
2009	no input / output widgets	ocl	InputWidget	self.oclIsTypeOf(InputWidget).not() and
	{0}: just spezialiciations of input or output widgets allowed he		OutputWidget	self.oclIsTypeOf(OutputWidget).not()
2010	tracesTo ca or node	Java	Widget	org.ontoucp.structuralui.validation.adapter.constraints.checkTracesToCA.java
	{0} doesnt trace to element of disourse model			
	Rules Constraints			
3001	mapping soure and target not empty	ecore	Mapping	
3002	traces to ca/rel in rule	Java	Rule	
	doesnt trace to ca/rel in rule			org.ontoucp.discourse.model2ui.rendering.validation.adapter.constraints.checkTracesTo.java
3003	mapping only on elements of rule	Java	Rule	
	{0} maps on a element not in discourse or structuralUI			org.ontoucp.discourse.model2ui.rendering.validation.adapter.constraints.checkMapping.java
3004	source and target not empty	ecore	Rule	
3101	check openquestion answer	Java	Rule	org.ontoucp.discourse.model2ui.rendering.validation.adapter.constraints.check_OQ_A.java
3102	check closedquestion answer	Java	Rule	org.ontoucp.discourse.model2ui.rendering.validation.adapter.constraints.check_CQ_A.java
3103	check offer request - accept reject	Java	Rule	org.ontoucp.discourse.model2ui.rendering.validation.adapter.constraints.check_OR_AR.java
3104	check informing	Java	Rule	org.ontoucp.discourse.model2ui.rendering.validation.adapter.constraints.check_INF.java
3105	check question answer	Java	Rule	org.ontoucp.discourse.model2ui.rendering.validation.adapter.constraints.check_Q_A.java

B Diskursmodell-Online Shop

Es folgt das Diskursmodell des Online Shop Beispiels aus Kapitel 2.2.3.

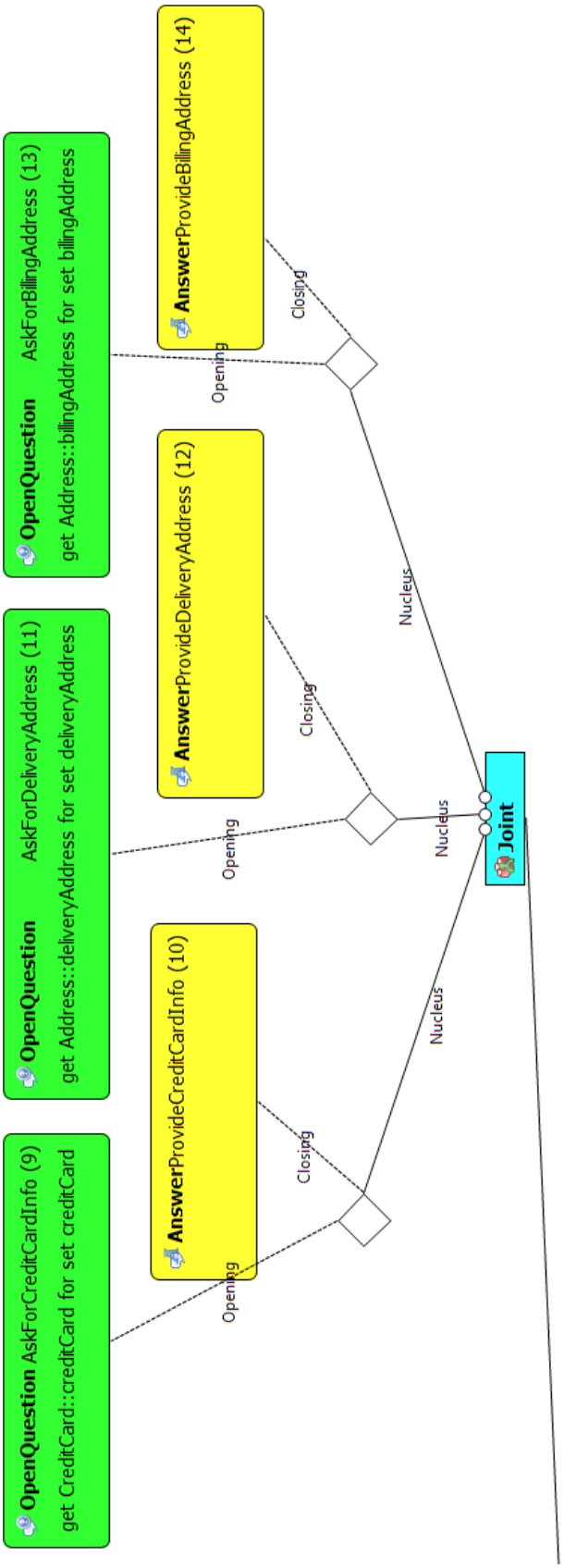

Abbildung B.1: Diskursmodell Online Shop - Teil 1

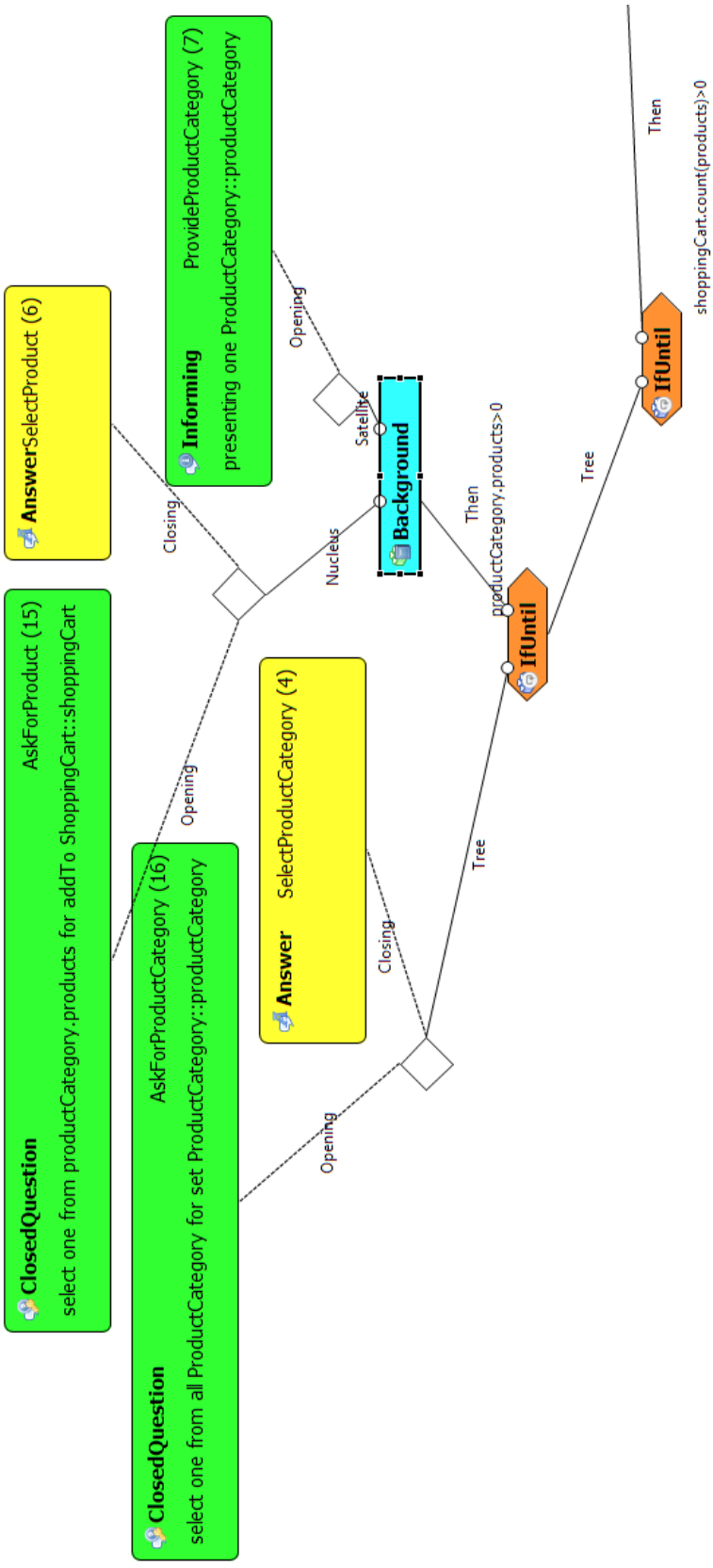

Abbildung B.2: Diskursmodell Online Shop - Teil 2

Abbildungsverzeichnis

Tabellenverzeichnis

Internet Referenzen

[1] Matteo Risoldi *Recipe: How To use the Eclipse Validation Framework with OCL constraints defined in a separate filer* http://wiki.eclipse.org/EMF/Validation/Recipes.

[2] *Enabling OCL property check in EMF's generated editor* http://smv.unige.ch/members/risoldi/otherdocs/ocl-emf.

[3] *Validation Rule Implementation* https://teambruegge.informatik.tu-muenchen.de/groups/unicase/wiki/10947/Validation_Rule_Implementation.html.

[4] *EMF - Eclipse Modelling Framework* http://www.eclipse.org/modeling/emf/.

[5] *Eclipse documentation - EMF Validation Framework Overview* EMF Validation Framework Developer Guide >Programmer's Guide >Validation Framework Overview http://help.eclipse.org/galileo/index.jsp.

[6] Lars Vogel *Eclipse Modeling Framework (EMF) - Tutorial* http://www.vogella.de/articles/EclipseEMF/article.html. 2010

[7] Assembla K-Made *Documentation Emf* http://www.assembla.com/wiki/show/Kmade/Documentation_emf

[8] Wikipedia *Design by Contract* http://de.wikipedia.org/wiki/Design_by_contract.

Wissenschaftliche Literatur

[BEF+10] C. Bogdan, D. Ertl, J. Falb, A. Green, S. Kavaldjian, D. Raneburger, and A. Szép. Report on development of dialogue design support features. *Report*, page 26, 2010.

[BFK+08] Cristian Bogdan, Jürgen Falb, Hermann Kaindl, Sevan Kavaldjian, Roman Popp, Helmut Horacek, Edin Arnautovic, and Alexander Szep. Generating an abstract user interface from a discourse model inspired by human communication. In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS-41)*, Piscataway, NJ, USA, January 2008. IEEE Computer Society Press.

[FGOG07] Lorenz Froihofer, Gerhard Glos, Johannes Osrael, and Karl M. Goeschka. 29th international conference on software engineering (icse'07). In *Overview and Evaluation of Constraint Validation Approaches in Java*, 2007.

[HS08] Manfred Hennig and Heiko Seeberger. Einführung in den Extension Point - Mechanismus von Eclipse. *Javaspektrum*, 1, 2008.

[KFK09] Sevan Kavaldjian, Jürgen Falb, and Hermann Kaindl. Generating content presentation according to purpose. In *Proceedings of the 2009 IEEE International Conference on Systems, Man and Cybernetics (SMC2009)*, San Antonio, TX, USA, Oct. 2009.

[KKHH04] Hermann Kaindl, Stefan Kramer, Mario Hailing, and Vahan Harput. Interactive metamodel-compliance checking of requirements in a semiformal representation. In *Managing Complexity and Change! - INCOSE 2004 - 14th Annual International Symposium Proceedings*, 2004.

[KRR+10] S. Kavaldjian, D. Raneburger, R.Popp, M. Leitner, J. Falb, and H. Kaindl. Automated optimization of user interfaces for screens with limited resolution. In *Proceedings of the MDDAUI'10 Workshop on Model Driven Development of Advanced User Interfaces*, 2010.

[Lei10] Michael Leitner. Space-saving placement using a structural user interface model. Master's thesis, TU Vienna, 2010. Master Thesis, TU Vienna.

[LFG90] Paul Luff, David Frohlich, and Nigel Gilbert. *Computers and Conversation*. Academic Press, London, UK, January 1990.

[MT88] W. C. Mann and S.A. Thompson. Rhetorical Structure Theory: Toward a functional theory of text organization. *Text*, 8(3):243–281, 1988.

[OMG06] Object Management Group. Object constraint language omg available specification version 2.0, 2006.

[Ran08] David Raneburger. Automated graphical user interface generation based on an abstract user interface specification. Master's thesis, TU Vienna, 2008.

[Sea69] J. R. Searle. *Speech Acts: An Essay in the Philosophy of Language.* Cambridge University Press, Cambridge, England, 1969.

[Son03] Runqiu Song. Einführung in die Object-Contraint-Language OCL, 2003.