



JOHANNES KEPLER  
UNIVERSITÄT LINZ

Netzwerk für Forschung, Lehre und Praxis

# Realisierung der semantischen Annotierung für pModeler, einem System zur semantischen Prozessmodellanalyse

## Diplomarbeit

zur Erlangung des akademischen Grades eines  
Magisters der Sozial- und Wirtschaftswissenschaften  
(Mag.rer.soc.oec.)

Angefertigt am Institut für Wirtschaftsinformatik -  
Data & Knowledge Engineering

Eingereicht von  
**Martin Bartsch**

Begutachter  
o. Univ.-Prof. Dr. Michael Schreffl

Mitbetreuer  
Mag. Andreas Bögl

Linz, Juni 2010



# Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, Juni 2010

Bartsch Martin

# Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich während der Entstehung dieser Diplomarbeit und während meines gesamten Studiums unterstützt haben.

Bei meinem Betreuer Mag. Andreas Bögl möchte ich mich für die tatkräftige Unterstützung und die vielen Diskussionen, welche zur Erstellung dieser Arbeit einen wesentlichen Beitrag geleistet haben, bedanken. Bei Herrn o. Univ.-Prof. Dr. Michael Schrefl möchte ich mich für die Anmerkungen in der finalen Phase der Fertigstellung der Diplomarbeit und für die rasche Begutachtung bedanken.

Weiters möchte ich mich bei meinen Freunden und Studienkollegen bedanken, welche mir immer zur Seite gestanden sind und mich aufgerichtet haben, wenn es mal nicht nach Wunsch gelaufen ist.

Schließlich möchte ich mich vor allem bei meinen Eltern, Ernestine und Herbert Bartsch, herzlichst bedanken, welche mir das Studium ermöglicht haben und mich während der gesamten Studiendauer tatkräftig im Bereich ihrer Möglichkeiten unterstützt und nie das Vertrauen in mich verloren haben.

# Abstract

Modeling business processes is a key component of business process management, which can be supported by different reuse approaches. A frequently discussed approach both in research and practice suggest to exploit a knowledge base which provides process solutions in terms of process patterns. Process patterns indicate proven process solutions in the sense of "Best Practices" for recurring problems or process goals.

Available approaches for discovering process patterns (so-called Pattern Mining) are primarily human-driven, e.g. pattern mining workshops wherein process patterns are discovered and documented by collaborative work between process experts and process owners. A human-driven discovery of pattern solutions in a huge set of process models available in enterprises and the associated maintenance of the knowledge base constitutes a tedious, hardly feasible task.

To approach this problem, an automated pattern discovery and integration into a knowledge base is required. However, this automatization requires well-defined semantics of the modeling constructs of existing process models, which is partially contained implicitly in natural language expressions of the model element labels. Based on an extension of semi-formal process models by means of formal terms of an ontology, an machinable processing of natural language expressions is facilitated.

The work at hand deals with the design and realization of an ontology, to enable a linguistic comparison of process activities and process states, which are described by natural language. Based on the realized ontology an automated semantic annotation of process activities and process states has been implemented prototypically. The purpose of this annotation is the assignment of formalized terms to process model elements. Semantically annotated process models contribute to process pattern discovery by facilitating a linguistic comparison for the detection of equal or similar activities and states in process models.

# Zusammenfassung

Die Modellierung von Geschäftsprozessen ist zentraler Bestandteil des Geschäftsprozessmanagements, welche durch unterschiedliche Ansätze der Wiederverwendung unterstützt werden kann. Ein in Forschung und Praxis vielfach diskutierter Ansatz empfiehlt, die Nutzung einer Wissensbasis, welche Prozesslösungen in Form von Prozessmustern bereitstellt. Prozessmuster bezeichnen bewährte Prozesslösungen im Sinne von "Best Practices" für wiederkehrende Probleme oder Prozessziele.

Verfügbare Ansätze für die Auffindung von Prozessmustern (sogenanntes Pattern Mining) sind primär durch eine manuelle Vorgehensweise gekennzeichnet, wie bspw. Pattern Mining Workshops, in welchen Prozessmuster durch Zusammenarbeit zwischen Prozessexperten und Prozessverantwortlichen aufgefunden und dokumentiert werden. Eine manuelle Auffindung von Prozesslösungen in einer sehr großen Menge von in Unternehmen verfügbaren Prozessmodellen und eine zugehörige Pflege der Wissensbasis stellt eine mühsame und kaum durchführbare Aufgabe dar.

Um dieses Problem anzugehen, wird eine automatisierte Musterauffindung und -integration in eine Wissensbasis benötigt. Allerdings setzt diese Automatisierung eine wohldefinierte Semantik der Modellierungskonstrukte von bestehenden Prozessmodellen voraus, welche zum Teil implizit in den natürlich-sprachlichen Ausdrücken der Modellelementbezeichnungen enthalten ist. Basierend auf einer Erweiterung von semi-formalen Prozessmodellen mit Hilfe von formalen Begriffen einer Ontologie, wird eine maschinelle Verarbeitung von natürlich-sprachlichen Ausdrücken ermöglicht.

Die vorliegende Arbeit beschäftigt sich mit dem Entwurf und der Realisierung einer Ontologie, um einen linguistischen Vergleich von Prozessaktivitäten und Prozesszuständen zu ermöglichen, welche mittels natürlicher Sprache beschrieben sind. Basierend auf der realisierten Ontologie wurde eine automatisierte, semantische Annotierung von Prozessaktivitäten und Prozesszuständen prototypisch implementiert. Die Zielsetzung dieser Annotierung ist die Zuordnung von formalisierten Begriffen zu Prozessmodellelementen. Semantisch annotierte Prozessmodelle unterstützen die Auffindung von Prozessmustern, indem ein linguistischer Vergleich für die Erkennung von gleichen oder ähnlichen Aktivitäten und Zuständen in Prozessmodellen ermöglicht wird.

# Arbeitsumgebung

Der nicht veröffentlichte Arbeitsbericht "*pModeler: Ein System zur Semantischen Prozessmodellanalyse*" [Bög07] am Institut für Data & Knowledge Engineering, verfasst von Herrn Mag. Andreas Bögl, beinhaltet die grundlegenden Konzepte für die prototypische Implementierung von pModeler. Grund für die derzeitige Nichtveröffentlichung bilden laufende Patentierungsverfahren seitens der Siemens CT SE 3.

Bei pModeler handelt es sich um einen Forschungsprototypen, der gemeinsam mit der Siemens CT SE 3 München am Institut Data & Knowledge Engineering im Rahmen der Dissertation von Herrn Mag. Andreas Bögl entwickelt wird. Die prototypische Implementierung von pModeler umfasst neben dieser Diplomarbeit derzeit vier weitere Diplomarbeiten:

Titel: Analyse von Prozessmodellen mit Hilfe von Data Mining Methoden

Diplomand<sup>in</sup>: Mag.<sup>a</sup> Alexandra Grömer

Titel: Realisierung von Prozessmodellmetriken für pModeler, einem System zur semantischen Prozessmodellanalyse

Diplomand: Mag. Peter Zierl

Arbeitstitel<sup>1</sup>: Realisierung einer Nativen Prozessmodellverwaltung und einer Importschnittstelle für pModeler - einem System zur semantischen Prozessmodellanalyse

Diplomand: Thomas Hostnik

Titel: Entwurf und Implementierung eines Process Knowledge Warehouses für pModeler, einem System zur semantischen Prozessmodellanalyse

Diplomand<sup>in</sup>: Mag.<sup>a</sup> Christina Lachner

---

<sup>1</sup>Hinweis: diese Diplomarbeit ist zum gegenwärtigen Zeitpunkt noch nicht abgeschlossen.

Kapitel 1.1, welches einen Überblick über pModeler gibt und für das Verständnis dieser Arbeit maßgeblich ist, wurde auszugsweise wörtlich aus dem Arbeitsbericht "*pModeler: Ein System zur semantischen Prozessmodellanalyse*" [Bög07] und aus der Diplomarbeit "*Entwurf und Implementierung eines Process Knowledge Warehouses für pModeler, einem System zur semantischen Prozessmodellanalyse*" [Lac08, S. 13ff] übernommen.



# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b> . . . . .	<b>11</b>
<b>Tabellenverzeichnis</b> . . . . .	<b>15</b>
<b>1 Einleitung</b> . . . . .	<b>16</b>
1.1 pModeler: ein Überblick . . . . .	17
1.2 Zielsetzung und Lösungsansatz . . . . .	26
1.3 Aufbau der Arbeit . . . . .	28
<b>2 Grundlagen</b> . . . . .	<b>30</b>
2.1 Geschäftsprozessmanagement . . . . .	31
2.1.1 Geschäftsprozesse . . . . .	33
2.1.2 Beschreibung von Geschäftsprozessen . . . . .	34
2.2 Architektur Integrierter Informationssysteme (ARIS) . . . . .	40
2.3 Die Ereignisgesteuerte Prozesskette (EPK) . . . . .	45
2.3.1 Modellierungskonstrukte . . . . .	45
2.3.2 Namenskonventionen . . . . .	49
2.3.3 Erweiterung um Prozessobjekte . . . . .	50
2.4 Semantische Prozessmodellierung . . . . .	53
2.4.1 Ontologien für Informationssysteme . . . . .	55
2.4.2 Ontologien für das Geschäftsprozessmanagement . . . . .	60
<b>3 Ontologie zur Annotierung von EPK-Modellen</b> . . . . .	<b>74</b>
3.1 Anforderungen an die Ontologie . . . . .	74
3.2 Abgrenzung zu verwandten Arbeiten . . . . .	77
3.3 Domänenspezifisches Wörterbuch . . . . .	79
3.4 Konzepte und Beziehungen der Ontologie . . . . .	80
3.5 Verwaltung der Ontologie in pModeler . . . . .	90
3.5.1 Process Object Viewer . . . . .	91
3.5.2 Process Object Editor . . . . .	93
3.5.3 Process Activity Viewer . . . . .	99
3.5.4 Function-, Condition- und Parameter-Editor . . . . .	103
3.5.5 Process Activity Editor . . . . .	108

<b>4</b>	<b>Semantische Annotierung von EPK-Modellen . . . . .</b>	<b>111</b>
4.1	Unklarheiten in EPK-Modellen . . . . .	111
4.2	Semantische Annotierung von EPK-Funktionen und -Ereignissen . . .	113
4.3	Semantische Aufbereitung von Prozessaktivitäten . . . . .	123
4.4	Darstellung der semantischen Annotierung in pModeler . . . . .	129
4.4.1	pModeler Analysekomponente . . . . .	130
4.4.2	Darstellung der annotierten Process Objects . . . . .	131
4.4.3	Darstellung der annotierten Process Activities . . . . .	133
<b>5</b>	<b>Implementierung . . . . .</b>	<b>135</b>
5.1	pModeler Architekturüberblick . . . . .	135
5.2	Datenmodell . . . . .	142
5.2.1	Teildatenmodell Process Object . . . . .	142
5.2.2	Teildatenmodell semantisch aufbereitete Modellelemente . . .	144
5.2.3	Teildatenmodell ProcessActivity . . . . .	146
5.3	Objektmodell . . . . .	147
5.3.1	Datenobjekte der Ontologie . . . . .	149
5.3.2	Managerklassen der Ontologie . . . . .	154
5.3.3	Analyseklassen . . . . .	160
<b>6</b>	<b>Schlussbetrachtung . . . . .</b>	<b>171</b>
	<b>Literaturverzeichnis . . . . .</b>	<b>174</b>
<b>A</b>	<b>Anhang . . . . .</b>	<b>183</b>
A.1	Annotierte Process Objects . . . . .	184
A.2	Annotierte Functions & Conditions . . . . .	187
A.3	Annotierte Activities . . . . .	190
<b>B</b>	<b>Anhang . . . . .</b>	<b>193</b>
B.1	Objektmodelle und Schnittstellenbeschreibungen . . . . .	194

# Abbildungsverzeichnis

Abb. 1.1:	Pattern-driven Process Design [Bög07]	18
Abb. 1.2:	pModeler Architektur [Bög07]	19
Abb. 1.3:	Beispiel für Ontologieinstanzen	20
Abb. 1.4:	Beispiel eines Zielbaums	22
Abb. 1.5:	Beispiel eines komponierten bzw. von elementaren Prozessmustern	22
Abb. 1.6:	Beispiel eines generischen Prozessmustern	23
Abb. 1.7:	Beispiel eines generischen komponierten Prozessmusters	23
Abb. 1.8:	Visuelle Notation der Frequency eines Prozessmusters	24
Abb. 1.9:	Beispiel für den Distance Parameter	24
Abb. 1.10:	Beispiel für ein Prozessmustersystem	25
Abb. 1.11:	Schematische Servicearchitektur [Bög07]	25
Abb. 1.12:	Knowledge Acquisition Service	26
Abb. 2.1:	BPM-Lifecycle [Aal04]	32
Abb. 2.2:	Sichtenkonzepte der Geschäftsprozessmodellierung [Gad05, S. 64ff]	35
Abb. 2.3:	Einsatzzwecke von Prozessmodellen [BK05, S. 51ff]	38
Abb. 2.4:	ARIS-Metageschäftsprozessmodell (vgl. [Sch98a, S. 31 bzw. S. 34f])	41
Abb. 2.5:	Beschreibungsebenen eines Informationssystems [Sch97, S. 14ff]	42
Abb. 2.6:	ARIS-Haus	43
Abb. 2.7:	Grundelemente der EPK-Notation	45
Abb. 2.8:	Verknüpfungsarten (vgl. [KNS92])	47
Abb. 2.9:	Erweiterung der EPK-Notation	48
Abb. 2.10:	Schematische Darstellung der eEPK	49
Abb. 2.11:	Ableitung eines Prozessobjektmodells (vgl. [Ros96, S. 81])	51
Abb. 2.12:	Beziehungen zwischen Prozessobjekten (vgl. [Ros96, S. 76ff])	52
Abb. 2.13:	Gegenüberstellung "Kritische IT/Prozess Trennung" & "Semantic Business Process Management" [HLD <sup>+</sup> 05]	53
Abb. 2.14:	Semantic Business Process Management Life Cycle [FSM <sup>+</sup> 07]	54
Abb. 2.15:	Ontologie Skala nach [LM01]	58
Abb. 2.16:	Arten von Ontologien (vgl. [Gua97] und [Gua98])	59
Abb. 2.17:	Klassen und Beziehungen von PIF [LGJ <sup>+</sup> 98]	63
Abb. 2.18:	Prozessdimension des POP* Meta-Modells [ATH05, S. 12]	64
Abb. 2.19:	sEPC Konzepthierarchie	66
Abb. 2.20:	General Process Ontology (GPO) [Lin08, S. 72]	68
Abb. 2.21:	Beispiel eines semantisch annotierten EPK-Modells [TF06]	70
Abb. 2.22:	Architektur der benutzerfreundlichen Annotierung [BDW07]	71

Abb. 2.23: Beispiel Definition eines Datenobjekts nach Born et al. [BDW07]	72
Abb. 3.1: Ausgangssituation: lexikalisch annotierte Modellelemente	79
Abb. 3.2: Ontologie zur Annotierung von EPK-Funktionen und -Ereignissen	80
Abb. 3.3: Konzept: <code>Process Object</code>	81
Abb. 3.4: Konzept: <code>Parameter</code>	82
Abb. 3.5: Konzept: <code>Function</code> und <code>Condition</code>	83
Abb. 3.6: Zusammenhang <code>Process Dictionary</code> und Ontologie	85
Abb. 3.7: Konzept: <code>ProcessActivity</code>	86
Abb. 3.8: Beispiel: verschachtelter, zusammengesetzter Kontext	88
Abb. 3.9: Beispiel: semantische Annotierung	89
Abb. 3.10: Anwendungsfalldiagramm <code>Process Object Viewer</code>	90
Abb. 3.11: <code>Process Object Viewer</code>	91
Abb. 3.12: Anwendungsfalldiagramm <code>Process Object Editor</code>	94
Abb. 3.13: <code>Process Object Editor</code> : "Process Object bearbeiten / anlegen"	95
Abb. 3.14: Abfrage "Stammwort setzen?"	95
Abb. 3.15: <code>Process Object Editor</code> : "Bearbeiten semantischer Beziehungen"	97
Abb. 3.16: Abfrage "Bestehende semantische Beziehung"	97
Abb. 3.17: <code>Process Object Editor</code> : "Änderungen bestätigen und speichern"	98
Abb. 3.18: Anwendungsfalldiagramm <code>Process Activity Viewer</code>	99
Abb. 3.19: <code>Process Activity Viewer</code>	100
Abb. 3.20: Explorersichten im <code>Process Activity Viewer</code>	101
Abb. 3.21: Anzeige einer <code>ProcessActivity</code>	102
Abb. 3.22: Beispiel referenzierte Modellelemente	102
Abb. 3.23: Element Editoren - schematischer Aufbau	103
Abb. 3.24: Element Editoren - Aufbau der Elemente	104
Abb. 3.25: Element Editoren - verfügbare <code>Process Objects</code>	104
Abb. 3.26: <code>Parameter Editor</code>	105
Abb. 3.27: <code>Function Editor</code>	106
Abb. 3.28: <code>Function Editor</code> - Checkbox lokales Element	107
Abb. 3.29: <code>Function Editor</code> - fehlendes passives Verb	107
Abb. 3.30: <code>Process Activity Editor</code>	108
Abb. 3.31: <code>Process Activity Editor</code> - Listen verfügbarer Elemente	109
Abb. 3.32: <code>Process Activity Editor</code> - Kontextmenü logische Operatoren	110
Abb. 4.1: Unklarheiten in EPK-Modellen	112
Abb. 4.2: Beispiel: Aufbereitung EPK-Funktion und -Ereignis	113
Abb. 4.3: Vorgehensweise: Element-Identifikation	114
Abb. 4.4: Rekursion für die Ermittlung der Generalisierungshierarchien	117
Abb. 4.5: <code>Process Objects</code> , Generalisierungen und Migrationen	119
Abb. 4.6: Beispiel: automatisiert generiertes Trivialereignis	120
Abb. 4.7: Beispiel einer zusammengesetzten EPK-Funktion	121
Abb. 4.8: Beispiel semantisch aufbereitete <code>ProcessActivity</code>	124
Abb. 4.9: Vorgehensmodell: Ermittlung der Kontextinformation	125

Abb. 4.10:	Aufbereitung der Post-Condition . . . . .	127
Abb. 4.11:	Beispiel: zusammengesetzte EPK-Funktion mit semantisch anno- tierten <code>ProcessActivities</code> . . . . .	129
Abb. 4.12:	Analysekomponente . . . . .	130
Abb. 4.13:	Darstellung annotierter <code>Process Objects</code> in pModeler . . . . .	131
Abb. 4.14:	Detailinformationen zu annotierten <code>Process Objects</code> . . . . .	132
Abb. 4.15:	Darstellung annotierter <code>Process Activities</code> in pModeler . . . . .	133
Abb. 4.16:	Detailansicht zu annotierten <code>Process Activities</code> . . . . .	134
Abb. 5.1:	pModeler Architekturüberblick . . . . .	136
Abb. 5.2:	Klassen - <code>Normalization Service</code> . . . . .	138
Abb. 5.3:	Klassen - <code>Knowledge Extraction Service</code> . . . . .	139
Abb. 5.4:	Klassen - <code>Pattern Mining Service</code> . . . . .	140
Abb. 5.5:	Teildatenmodell <code>Process Object</code> und semantische Beziehungen .	143
Abb. 5.6:	Teildatenmodell semantisch aufbereitete Modellelemente . . . . .	145
Abb. 5.7:	Teildatenmodell <code>ProcessActivity</code> . . . . .	147
Abb. 5.8:	Schematisches Objektmodell für die Verwaltung der Ontologie . .	148
Abb. 5.9:	Objektmodellüberblick - Datenobjekte der Ontologie . . . . .	149
Abb. 5.10:	Klassen - <code>OntologyBaseObject &amp; ProcessObject</code> . . . . .	150
Abb. 5.11:	Klasse - <code>OntologyElement_NativeModelItem&lt;OntologyElement&gt;</code>	150
Abb. 5.12:	Verwendung eines Typparameter anhand eines Beispiels . . . . .	151
Abb. 5.13:	Klasse - <code>ProcessObject_DictionaryEntry</code> . . . . .	151
Abb. 5.14:	Klassen - Semantische Beziehungen des <code>ProcessObject</code> . . . . .	152
Abb. 5.15:	Klasse - <code>Function_Condition</code> . . . . .	152
Abb. 5.16:	Klassen - <code>Context</code> . . . . .	153
Abb. 5.17:	Objektmodellüberblick - Managerklassen der Ontologie . . . . .	154
Abb. 5.18:	Schnittstelle - <code>IElementManager</code> . . . . .	155
Abb. 5.19:	Instanziierung - <code>IElementManager&lt;ProcessObject&gt;</code> . . . . .	155
Abb. 5.20:	Methode - <code>FunctionManager Save(...)</code> . . . . .	157
Abb. 5.21:	Methode - <code>FunctionManager SaveUncommitted(...)</code> . . . . .	157
Abb. 5.22:	Methode - <code>ConditionManager Create(...)</code> . . . . .	158
Abb. 5.23:	Methode - <code>ConditionManager CreateLocalCondition(...)</code> . . . . .	159
Abb. 5.24:	Programmausschnitt - Aufbereitung der Modellelement-Syntax . .	161
Abb. 5.25:	Methode - <code>ProcessObjectIdentification processProcessObject(...)</code>	162
Abb. 5.26:	Methode - <code>ProcessObjectIdentification buildProcessObject(...)</code>	163
Abb. 5.27:	Methode - <code>ProcessActivityAnalysis processProcessActivity(...)</code>	164
Abb. 5.28:	Methode - <code>ProcessActivityAnalysis processInitialContext(...)</code> .	165
Abb. 5.29:	Methode - <code>ProcessActivityAnalysis setInitialContext(...)</code> . . . .	166
Abb. 5.30:	Beispiel - Ebenen für Generierung des <code>Context</code> . . . . .	167
Abb. A.1:	Beispielmodell Hardware - annotierte <code>Process Objects</code> . . . . .	185
Abb. A.2:	Beispielmodell Software - annotierte <code>Process Objects</code> . . . . .	186
Abb. A.3:	Beispielmodell Hardware - annotierte <code>Functions &amp; Conditions</code> .	188
Abb. A.4:	Beispielmodell Software - annotierte <code>Functions &amp; Conditions</code> . .	189

Abb. A.5: Beispielmodell Hardware - annotierte <b>Activities</b> . . . . .	191
Abb. A.6: Beispielmodell Software - annotierte <b>Activities</b> . . . . .	192
Abb. B.1: Objektmodell - Datenobjekte der Ontologie . . . . .	194
Abb. B.2: Objektmodell - Datenobjekte der Ontologie . . . . .	195
Abb. B.3: Objektmodell - Managerklassen <b>IElementManager &amp; ProcessObject</b> . . . . .	196
Abb. B.4: Objektmodell - Managerklassen <b>Activity &amp; Parameter</b> . . . . .	201
Abb. B.5: Objektmodell - Managerklassen <b>Function &amp; Event</b> . . . . .	204
Abb. B.6: Objektmodell - Managerklassen <b>Method &amp; State</b> . . . . .	207
Abb. B.7: Objektmodell - Klassen <b>OntologyElement_NativeModelItem</b> . . . . .	210
Abb. B.8: Objektmodell - Klassen <b>ProcessObject_DictionaryEntry</b> . . . . .	212
Abb. B.9: Objektmodell - Klassen <b>SemanticRelation</b> . . . . .	214
Abb. B.10: Objektmodell - Klassen <b>ProcessObjectMigration</b> . . . . .	216
Abb. B.11: Objektmodell - Klassen <b>Function_Event</b> . . . . .	218
Abb. B.12: Objektmodell - Klassen <b>Context</b> . . . . .	221
Abb. B.13: Objektmodell - Klassen <b>Context_Context</b> . . . . .	223
Abb. B.14: Objektmodell - Klassen <b>Context_Event</b> . . . . .	225

# Tabellenverzeichnis

Tab. 2.1:	Namenskonvention für Funktionen und Ereignisse . . . . .	50
Tab. 3.1:	Konzepte der Ontologie . . . . .	76
Tab. 4.1:	Zuordnungsvorschrift von Worttypen . . . . .	115
Tab. 4.2:	Zerlegung zusammengesetzter Funktionen . . . . .	122
Tab. 5.1:	Beispiel - Generierung des <code>InitialContext</code> . . . . .	169
Tab. B.1:	Schnittstelle - <code>IElementManager</code> . . . . .	198
Tab. B.2:	Schnittstelle - <code>IProcessObjectManager</code> . . . . .	200
Tab. B.3:	Schnittstelle - <code>IActivityManager</code> . . . . .	202
Tab. B.4:	Schnittstelle - <code>IParameterManager</code> . . . . .	203
Tab. B.5:	Schnittstelle - <code>IFunctionManager</code> . . . . .	205
Tab. B.6:	Schnittstelle - <code>IEventManager</code> . . . . .	206
Tab. B.7:	Schnittstelle - <code>IMethodManager</code> . . . . .	208
Tab. B.8:	Schnittstelle - <code>IStateManager</code> . . . . .	209
Tab. B.9:	Schnittstelle - <code>IOntologyElement_NativeModelItemManager</code> . . .	211
Tab. B.10:	Schnittstelle - <code>IProcessObject_DictionaryEntryManager</code> . . . .	213
Tab. B.11:	Schnittstelle - <code>ISemanticRelationManager</code> . . . . .	215
Tab. B.12:	Schnittstelle - <code>IProcessObjectMigrationManager</code> . . . . .	217
Tab. B.13:	Schnittstelle - <code>IFunction_EventManager</code> . . . . .	220
Tab. B.14:	Schnittstelle - <code>IContextManager</code> . . . . .	222
Tab. B.15:	Schnittstelle - <code>IContext_ContextManager</code> . . . . .	224
Tab. B.16:	Schnittstelle - <code>IContext_EventManager</code> . . . . .	226

# 1. Einleitung

Flexibles Geschäftsprozessmanagement ist von zentraler Bedeutung, um auf sich rasch ändernde Umwelteinflüsse in allen Unternehmensbereichen reagieren zu können. Hierfür ist die Kenntnis und die ständige Pflege sämtlicher Geschäftsprozesse eines Unternehmens notwendig. Von grundlegender Bedeutung sind in diesem Zusammenhang Modelle dieser Prozesse, welche für eine Vielzahl an Aufgaben, wie bspw. zur Dokumentation, zur Kommunikation oder für Simulationen, vorliegen. Daher ist die Modellierung dieser Prozess als wesentlicher Bestandteil des Geschäftsprozessmanagements zu betrachten.

Zur Unterstützung der Geschäftsprozessmodellierung wurden daher über die Jahre hinweg eine Reihe unterschiedlicher Methoden und Werkzeuge entwickelt. Für die Darstellung von Prozessmodellen werden in Forschung und Praxis eine Vielzahl unterschiedlicher Modellierungssprachen eingesetzt, wobei sich semiformale, grafische Darstellungen, wie bspw. die Ereignisgesteuerte Prozesskette (EPK) oder die Business Process Management Notation (BPMN), aufgrund ihrer einfachen und intuitiv verständlichen Form durchgesetzt haben [TF09].

Modellierungssprachen stellen für die Darstellung von Prozessen unterschiedliche Modellierungskonstrukte zur Verfügung. EPKs verwenden bspw. Funktionen und Ereignisse zur Beschreibung von Tätigkeiten bzw. von Zuständen, welche mittels eines Kontrollflusses verbunden und mittels logischer Konnektoren verknüpft werden können (vgl. u.a. [KNS92]). Die Bezeichnungen dieser Modellelemente werden in natürlicher Sprache modelliert, welche aufgrund ihrer Mehrdeutigkeit unterschiedliche Interpretationen ermöglicht. Probleme, welche sich hierdurch ergeben, treten einerseits bei der Kombination, der Suche und der Transformation von Prozessmodellen unterschiedlicher Modellierer und andererseits bei der automatisierten Verarbeitung von Prozessmodellen auf.

Diese Problemstellungen sind zum Teil zentraler Betrachtungsgegenstand des Semantic Business Process Management (SBPM). Zielsetzung dieses Ansatzes ist es, die Kluft zwischen der fachlichen und technischen Betrachtungsweise von Prozessen durch die Verwendung semantischer Technologien, wie bspw. Ontologien, zu überbrücken, um einerseits die Kommunikation zwischen den Beteiligten der beiden Bereiche zu erleichtern und andererseits den Automatisierungsgrad in den einzelnen Phasen des Geschäftsprozessmanagement zu erhöhen [HLD<sup>+</sup>05].

Die semantische Prozessmodellierung erweitert Prozessmodelle um eine Annotie-



rung mit Hilfe von Ontologien, um deren Bedeutung explizit, in maschinell interpretierbarer Form zu spezifizieren. In diesem Zusammenhang können zwei grundsätzliche Zielsetzungen unterschieden werden. Zum einen die Annotierung von Modellierungskonstrukten auf Metamodellebene, um zwischen unterschiedlichen Modellierungssprachen übersetzen zu können und zum anderen die Annotierung der Modellelemente auf Modellebene, um die Semantik der Inhalte, d.h. der natürlichsprachlichen Bezeichnungen, explizit zu spezifizieren. Die semantische Annotierung von Prozessmodellen ermöglicht somit eine automatisierte Verarbeitung bestehenden Prozesswissens, mit welcher eine Reihe von Vorteilen, wie bspw. erweiterte Suchmöglichkeiten in Prozessmodellen, eine bessere Wiederverwendung von Prozessfragmenten oder die Vereinfachung der Integration von Prozessmodellen unterschiedlicher Sprachen, einhergehen (vgl. u.a. [WMF<sup>+</sup>07] oder [LBS08]).

## 1.1. pModeler: ein Überblick

Der Entwurf, die Anpassung und Verbesserung von Prozessmodellen ist eine aufwändige und komplexe Aufgabe. Bereits mit dem Konzept der Referenzmodellierung wurden die Vorteile der Verwendung vorgefertigter Modelle bzw. Modellbausteine, welche für unterschiedliche Unternehmen als Grundlage zur Modellierung verwendet und an spezifische Anforderungen angepasst werden können, erkannt [FL02]. Auch die semantische Prozessmodellierung versucht den manuellen Aufwand in diesem Bereich zu reduzieren, indem auf Basis einer semantischen Annotierung von Prozessmodellen mittels Ontologien u.a. erweiterte Suchmöglichkeiten in Prozessmodellen oder verbesserte Wiederverwendung von Prozessfragmenten ermöglicht werden sollen (vgl. bspw. [LBS08] oder [WMF<sup>+</sup>07]).

Bei pModeler handelt es sich um ein System zur semantischen Prozessmodellanalyse mit der Zielsetzung aus bestehenden, semi-formalen Prozessmodellen Prozessmuster zu identifizieren bzw. zu konstruieren [BKS08]. Dieser Vorgang, der auch als **Process Pattern Mining** bezeichnet wird, unterstützt im Lifecycle der semantischen Prozessmodellierung somit die Wiederverwendung von Prozessfragmenten. **Process Pattern Mining** zielt darauf ab, Prozesswissen aus Prozessmodellen zu extrahieren und dieses Wissen in Form von semantisch aufbereiteten Prozessmustern zu speichern. Dieser Rekonstruktionsansatz dient der Analyse von Prozessen, um Muster und Eigenschaften zu erkennen, die bereits in einem Geschäftsprozess modelliert wurden [FE03].

In Anlehnung an Hagen [Hag05, S. 45] definiert ein Prozessmuster *”einen bewährten Prozess, um ein Problem, das in einem bestimmten Kontext wiederholt aufgetreten ist, zu lösen”*. Ein Prozessmuster repräsentiert eine formale, ontologiebasierte Beschreibung einer Prozesslösung zur Erreichung eines konkreten Prozessziels. Das Konzept der Prozessmuster ermöglicht es, dokumentiertes Prozesswissen in einer strukturierten und eindeutig definierten Weise auf Basis eines Schemas zu erfassen

und für eine Wiederverwendung zur Verfügung zu stellen.

Die Wiederverwendung von Prozesswissen im Lifecycle der Prozessmodellierung zielt darauf ab, bei der Erstellung von neuen Modellen nicht immer von Grund auf neu zu modellieren, sondern auf bewährtes Wissen in Form von Prozessmuster zurückzugreifen. Abbildung 1.1 verdeutlicht den Unterschied zwischen einer Modellierung "von Grund auf neu" (**Design from Scratch**) und der Verwendung von Prozessmustern, was als musterbasierte Prozessmodellierung (**Pattern-Driven Process Design**) bezeichnet wird.

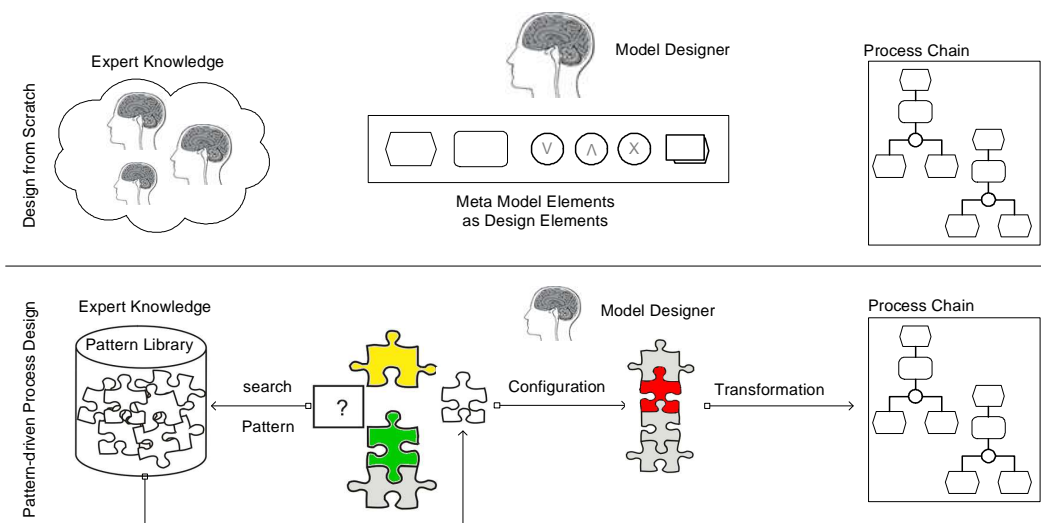


Abbildung 1.1.: Pattern-driven Process Design [Bög07]

Das für **Design from Scratch** erforderliche Prozesswissen muss von Experten der jeweiligen Anwendungsdomäne zur Verfügung gestellt werden. Prozessmodelle werden erstellt, indem die Metakonstrukte (z.B. Funktionen und Ereignisse in EPKs) der zugrunde liegenden Modellierungssprache instanziiert werden. Im Gegensatz dazu wird mit der musterbasierten Modellierung das erforderliche, anwendungsspezifische Expertenwissen in einer Musterbibliothek (**Process Pattern Library**) zur Verfügung gestellt. Die in dieser Musterbibliothek enthaltenen Prozessmuster stellen problemspezifische "Common Practice Solutions" dar, die anwendungsspezifisch angepasst werden können.

Das von pModeler verfolgte Konzept zur Wiederverwendung von Prozesswissen stützt sich auf den Ansatz des Case Based Reasoning (CBR). Case Based Reasoning basiert dabei auf einer Problemlösung durch Analogieschluss, d.h. zur Lösung eines gegebenen Problems wird die Lösung eines ähnlichen und früher bereits gelösten Problems herangezogen. In pModeler repräsentiert die Musterbibliothek die sogenannte Falldatenbank eines CBR-Systems. Liegt ein neues Problem vor, so kann nach einem geeigneten Prozessmuster zur Erreichung eines Prozessziels gesucht werden.

Datengrundlage für das **Process Pattern Mining** bilden semi-formale, ereignis-

gesteuerte Prozessketten (EPKs). Aufbauend auf einer semantischen Annotierung von EPK Modellelementen werden Prozessmuster extrahiert und im sogenannten **Process Knowledge Warehouse** zur Wiederverwendung bzw. für weitere Analysen zur Verfügung gestellt.

Abbildung 1.2 gibt einen Überblick über das gesamte pModeler Architekturkonzept, welches sich aus den Komponenten **Process Warehouse**, **Semantic Process Model Analysis** und dem **Process Knowledge Warehouse** zusammensetzt.

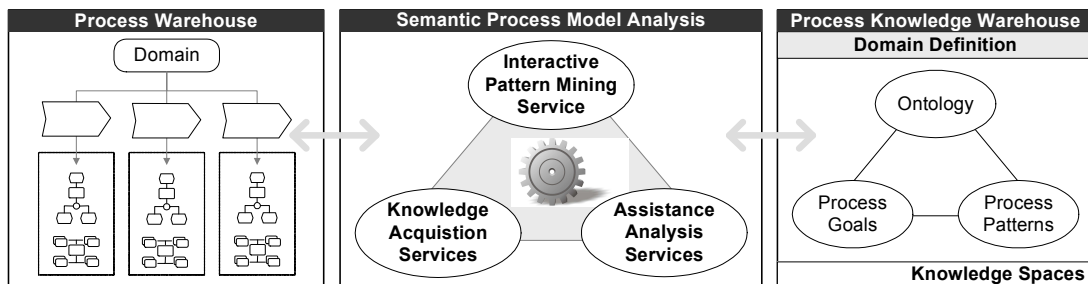


Abbildung 1.2.: pModeler Architektur [Bög07]

Das **Process Warehouse** dient der zentralen Speicherung und Organisation (**Native Model Management**) von nativen, semi-formalen Prozessmodellen, die im ARIS Toolset modelliert wurden. Native Prozessmodelle bilden die Menge an Prozessmodellen, für die in weiterer Folge die in pModeler spezifizierten Analysis Services durchgeführt werden sollen.

Schnittstelle zum ARIS Toolset bildet das proprietäre Austauschformat AML (ARIS Markup Language), d.h. die Prozessmodelle werden entsprechend der AML Schemadefinition als XML-Dokumente aus ARIS exportiert und in pModeler importiert. Das **Process Warehouse** unterstützt den Import von ereignisgesteuerten Prozessketten und Funktionszuordnungsdiagrammen.

Neben der zentralen Speicherung und Organisation beinhaltet das **Native Model Management** Komponenten zur Modellvisualisierung, Modellversionierung und zur Berechnung der **Native Model Metrics** der im **Process Warehouse** gespeicherten Prozessmodelle.

Das im **Process Knowledge Warehouse** dokumentierte, wiederverwendbare Prozesswissen ist in die drei **Knowledge Spaces** **Ontologie** (**Ontology**), **Prozessziele** (**Process Goals**) und **Prozessmuster** (**Process Patterns**) unterteilt.

Mit Hilfe der Ontologie wird die in den natürlichsprachlichen Elementen enthaltene Semantik auf explizite Weise formalisiert. Ontologieinstanzen repräsentieren somit ein kontrolliertes Vokabular für modellierte Geschäftsprozesse. Die Semantik wird durch Annotierung von EPK-Funktionen und -Ereignissen zu Ontologieinstanzen spezifiziert, wobei die in der Ontologie beschriebenen Konzepte auf die Anforderungen des **Process Pattern Mining** reduziert sind.

Die Ontologie stellt Konzepte zur semantischen Beschreibung der Modellelementbezeichnungen bereit, d.h. sie dient der expliziten Ausweisung der einzelnen Elemente, aus welchen EPK-Funktionen und -Ereignisse zusammengesetzt werden können. Während die Bezeichnung einer EPK-Funktion aus einem **Task** (z.B. Define, Specify) und einem **Process Object** (z.B. Software Architecture) gebildet wird, wird die Bezeichnung eines EPK-Ereignisses durch ein **Process Object** gefolgt von einem **State** (z.B. Defined, Specified) beschrieben.

Des Weiteren werden eine Reihe von Beziehungen zwischen diesen Konzepten bereitgestellt, welche die automatisierte Identifikation von Prozessmustern unterstützen. So umfassen bspw. die semantischen Beziehungen zwischen den **Process Objects** die Generalisierung (Is-A-Beziehung), Komposition (isPartOf-Beziehung) und Prozessobjektmigrationen (Migrates-to-Beziehung).

Neben der semantischen Aufbereitung der einzelnen Modellelemente ermöglicht die Ontologie die Bildung elementarer Modellierungsbausteine, den sogenannten **ProcessActivities**, welche die Grundlage zur Beschreibung eines Prozessmusters bilden. Eine **ProcessActivity** verknüpft eine semantisch aufbereitete Funktion mit ihren Eingangs- und Ausgangszuständen, welche als **Pre- und Postconditions** beschrieben sind und durch semantisch aufbereitete Ereignisse repräsentiert werden. Abbildung 1.3 zeigt ein Beispiel für die semantische Annotierung der EPK-Modellelemente mit Instanzen der Ontologie.

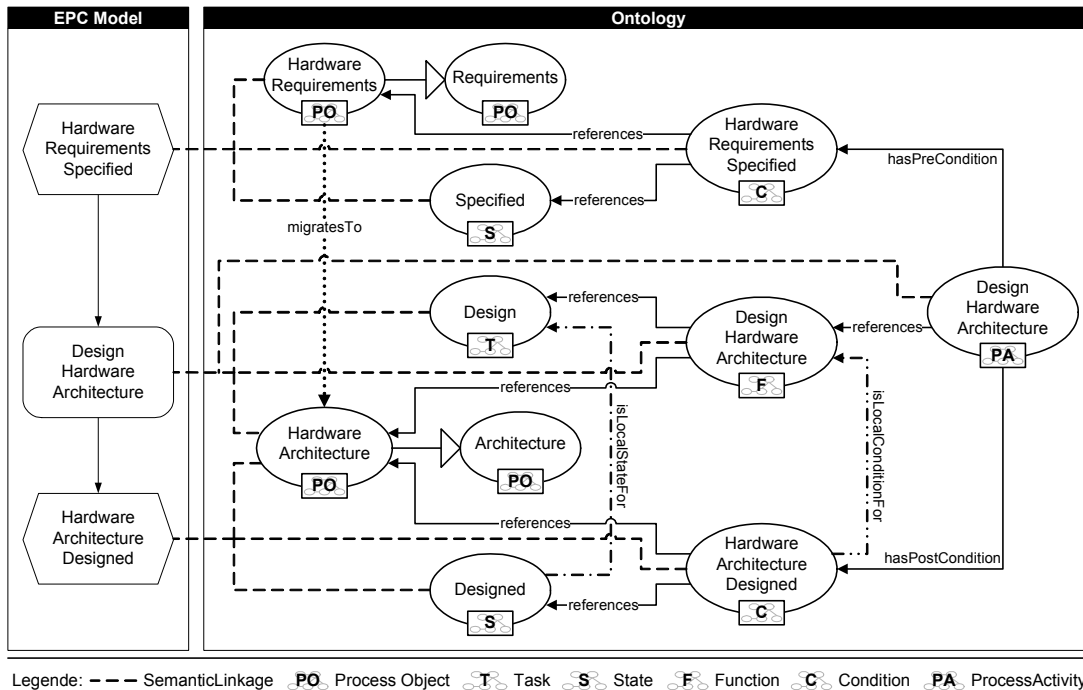


Abbildung 1.3.: Beispiel für Ontologieinstanzen

**Beispiel:**

Die in ARIS modellierte EPK-Funktion "Design Hardware Architecture" setzt sich aus dem Task "Design" und dem Process Object "Hardware Architecture" zusammen, wobei letzteres in einer <Is-A> Beziehung mit der allgemeineren Instanz "Architecture" steht. Die ProcessActivity "Design Hardware Architecture" wird aus dieser semantisch aufbereiteten EPK-Funktion und den entsprechenden Pre- und Postconditions, d.h. den semantisch aufbereiteten EPK-Ereignissen "Hardware Requirements Specified" bzw. "Hardware Architecture Designed", gebildet.

Für die Konstruktion von Prozessmustern ist die Häufigkeit des Auftretens zur Erreichung eines konkreten Prozessziels eine wesentliche Eigenschaft. Ein Prozessziel kennzeichnet somit ein wesentliches Beschreibungsmerkmal zur Identifikation eines Prozessmusters.

Prozessziele werden in Anlehnung an Mendes et al. [MVC<sup>+</sup>01] wie folgt definiert: "Goals represent the purpose or the outcome that the business as a whole is trying to achieve. Goals control the behaviour of the business and show the desired states of some resources in the business".

Das Konzept der Prozessziele unterscheidet zwischen funktionalen und nicht funktionalen Prozesszielen [KK97]. Funktionale Prozessziele werden verwendet um den Output, sogenannte Delivery Objects, eines Geschäftsprozesses oder Teilprozesses zu spezifizieren. Nicht-funktionale Prozessziele hingegen beziehen sich auf die Qualität des Prozesses bzw. auf dessen Ausführung, wie z.B. Performance, Usability, Customizations etc. Prozessziele repräsentieren keine elementaren Bausteine, Beziehungen zwischen Prozesszielen werden in Form von semantischen Beziehungstypen ausgedrückt, die als Zielbaum abgebildet werden.

In pModeler werden in der ersten Ausbaustufe lediglich funktionale Prozessziele beschrieben bzw. berücksichtigt. Grund dafür ist, dass funktionale Prozessziele aus bestehenden Prozessmodellen automatisch aus den semantisch annotierten EPK-Ereignissen abgeleitet werden können. Die semantische Abbildung wird durch eine hierarchische Baumstruktur repräsentiert, die miteinander in Beziehung stehende, funktionale Prozessziele abbildet (siehe Abb. 1.4).

**Beispiel:**

Um das funktionale Prozessziel B (z.B. Project Planned) in einem Software Development Process zu erreichen, müssen vorher der Milestone Goal SW.M1 (z.B. Project Acquired), anschließend das Prozessziel A (z.B. Project Manager Nominated) in chronologischer Reihenfolge erreicht werden. Diese Zielerreichung wird durch die sequentielle Ausführung der Prozessmuster P(A) und P(B) sowie die dem Milestone SW.M1 zugeordneten Prozessmuster erreicht.

Ein Prozessmuster repräsentiert eine *Common Practice Solution* um ein bestimmtes funktionales Prozessziel zu erreichen. Charakteristisch für ein Prozessmuster ist

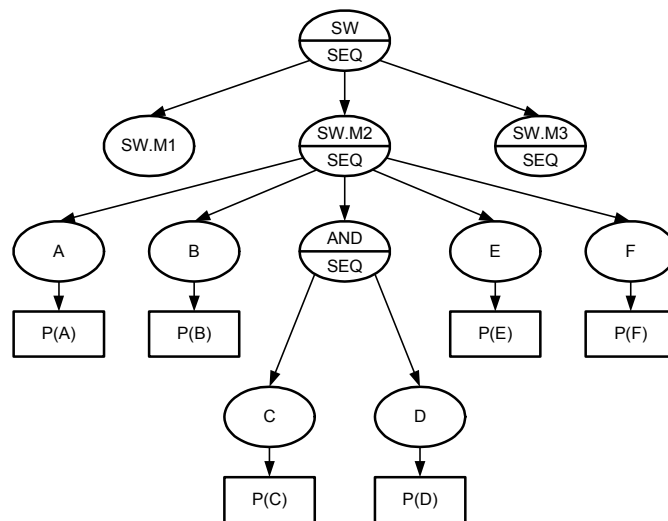


Abbildung 1.4.: Beispiel eines Zielbaums

daher, dass es ein bestimmtes funktionales Prozessziel in einer bestimmten Anwendungsdomäne oder -kontext erreicht.

Ein elementares Prozessmuster (**Elementary Process Pattern**) beschreibt den kleinsten, semantischen Baustein zur Erreichung eines funktionalen Prozessziels. Komponierte Prozessmuster (**Composite Process Pattern**) sind Bestandteil eines umfangreichen Prozessmustersystems, d.h. ein System von miteinander in Beziehung stehenden Prozessmustern. Beziehungen zwischen Prozessmustern werden durch unterschiedliche Beziehungstypen (z.B. *Sequence*, *Uses*, etc.) ausgedrückt.

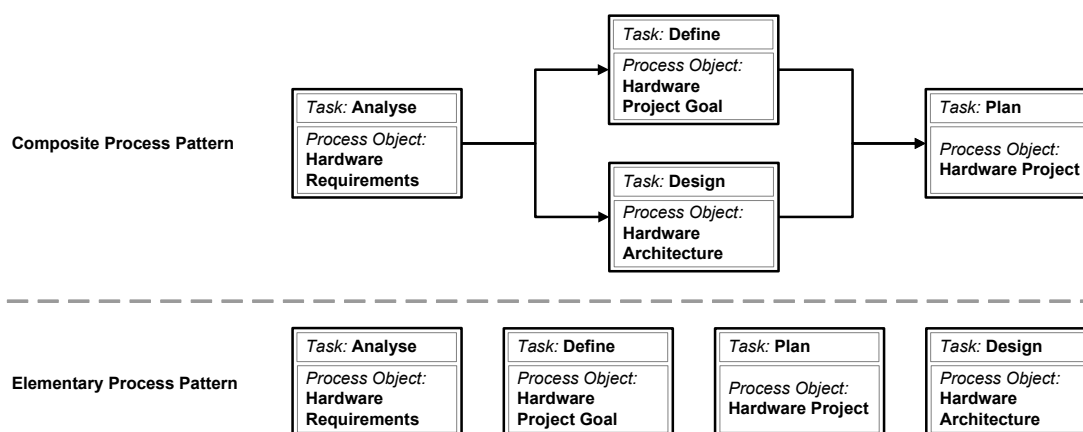


Abbildung 1.5.: Beispiel eines komponierten bzw. von elementaren Prozessmustern

Elementare bzw. komponierte Prozessmuster können entweder *instanziiert* oder *generisch* sein und die Eigenschaft *Frequent* besitzen, d.h. sie treten in einer bestimmten Anwendungsdomäne oder einem bestimmten Anwendungskontext häufig auf. Generische Prozessmuster stellen Klassen ähnlicher Prozessmuster, welchen Pa-

parameter in Form von **Pattern Variablen (PV)** zugewiesen werden. Grundlage für die Abstraktion von elementaren Prozessmustern bilden einerseits die spezifizierten Anwendungsdomänen und andererseits die automatisiert ermittelten Generalisierungshierarchien der **Process Objects**. Abbildung 1.6 zeigt ein Beispiel für ein generisches Prozessmuster mit beiden Arten von Patternvariablen.

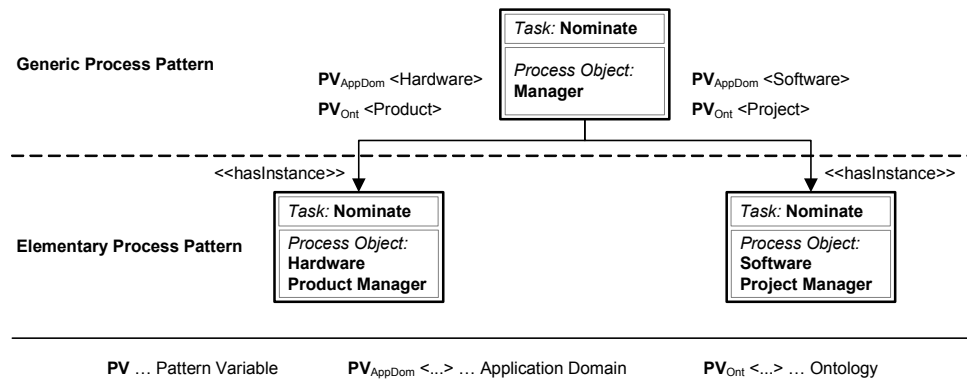


Abbildung 1.6.: Beispiel eines generischen Prozessmusters

### Beispiel:

Die Patterns "Nominate Hardware Product Manager" und "Nominate Software Project Manager" können unter Berücksichtigung der definierten Anwendungsdomänen (Hardware, Software) und der Generalisierungshierarchien der **Process Objects** zu "Nominate Manager" abstrahiert werden. Dieses **Generic Process Pattern** besitzt somit die Patternvariablen für die Anwendungsdomäne ( $PV_{AppDom}$ ) mit den zulässigen Ausprägungen {Hardware, Software} und für die Abstraktion auf Basis der Generalisierungshierarchien der Ontologie ( $PV_{Ont}$ ) mit den zulässigen Ausprägungen {Product, Project}.

Auch generische Prozessmuster können zu komponierten Prozessmustern zusammengefasst werden. Abbildung 1.7 zeigt ein Beispiel eines solchen Prozessmusters, welches aus generischen, elementaren Prozessmustern besteht.

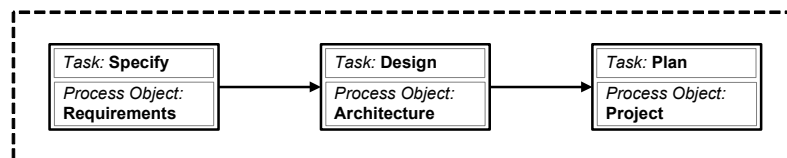


Abbildung 1.7.: Beispiel eines generischen komponierten Prozessmusters

Ein elementares Prozessmuster besitzt die Eigenschaft *Frequent*, wenn es entweder innerhalb eines Prozessmodells oder prozessmodellübergreifend ein funktionales Prozessziel häufig erreicht bzw. als solches in Prozessen modelliert wurde. Für die Bestimmung der Häufigkeit des Auftretens wird der Parameterwert  $Frequency_{min}$

im Rahmen des **Process Pattern Mining Service** definiert. Wird bspw. der Parameterwert  $Frequency_{min}$  mit dem Wert zwei belegt, so ist ein Prozessmuster dann *Frequent*, wenn es mindestens zweimal innerhalb eines Prozessmodells oder prozessmodellübergreifend modelliert wurde. Die Bestimmung dieser Eigenschaft wird als *Frequency Check* bezeichnet.

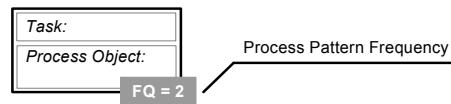


Abbildung 1.8.: Visuelle Notation der Frequency eines Prozessmusters

Innerhalb eines Systems von Prozessmustern können neben elementaren Prozessmustern auch komponierte Prozessmuster häufig auftreten. Damit ein komponiertes Prozessmuster die Eigenschaft *Frequent* erfüllt, müssen folgende Voraussetzungen erfüllt sein:

1. Die Bausteine eines komponierten Prozessmusters (elementare Prozessmuster) müssen häufig auftreten, d.h. die Eigenschaft *Frequent* erfüllen.
2. Die Beziehungen (z.B. *Sequence* Beziehung) zwischen diesen Bausteinen müssen ebenso die Eigenschaft *Frequent* erfüllen, d.h. größer gleich dem Parameterwert  $Frequency_{min}$ .
3. Die Distanz zwischen den Bausteinen muss kleiner gleich dem Parameterwert  $Pattern Distance_{max}$  liegen. Der Parameterwert  $Pattern Distance_{max}$  legt fest, wie viele häufig auftretende Prozessmuster zwischen zwei häufig auftretenden Prozessmustern modelliert sein dürfen.

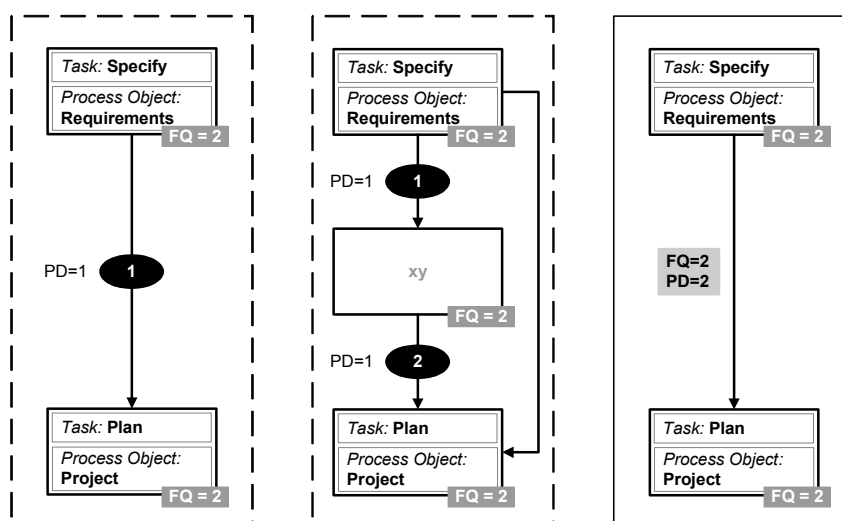


Abbildung 1.9.: Beispiel für den Distance Parameter



Abbildung 1.10 zeigt ein Beispiel, indem ein Auszug aus einem Prozessmustersystem dargestellt ist. Das Prozessmustersystem besteht aus den *häufigen*, elementaren Prozessmustern P(A), P(C), P(D) und P(E). Nachdem P(A) die Eigenschaft *Frequent* besitzt, ist es Bestandteil des Prozessmustersystems. Da jedoch die *Sequence* Beziehung zu P(Z) nicht häufig ist (FQ=1), ist P(A) nicht Bestandteil des komponierten Prozessmusters P(Z).

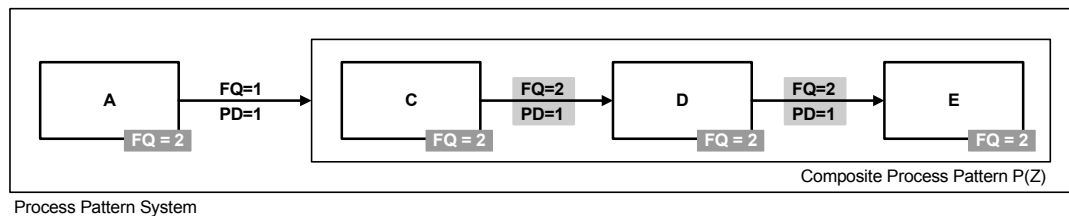


Abbildung 1.10.: Beispiel für ein Prozessmustersystem

Nachdem in den vorangegangenen Absätzen die grundlegende Idee hinter dem pModeler Ansatz dargestellt und die zugrunde liegenden Konzepte skizziert wurden, werden in weiterer Folge kurz die für die Identifikation von Prozessmustern bereitgestellten Services vorgestellt. pModeler spezifiziert drei Klassen von **Analysis Services**, die den Aufbau der spezifizierten **Knowledge Spaces** im **Process Knowledge Warehouse** semi-automatisch unterstützen. Abbildung 1.11 zeigt schematisch die allgemeine Servicearchitektur in pModeler.

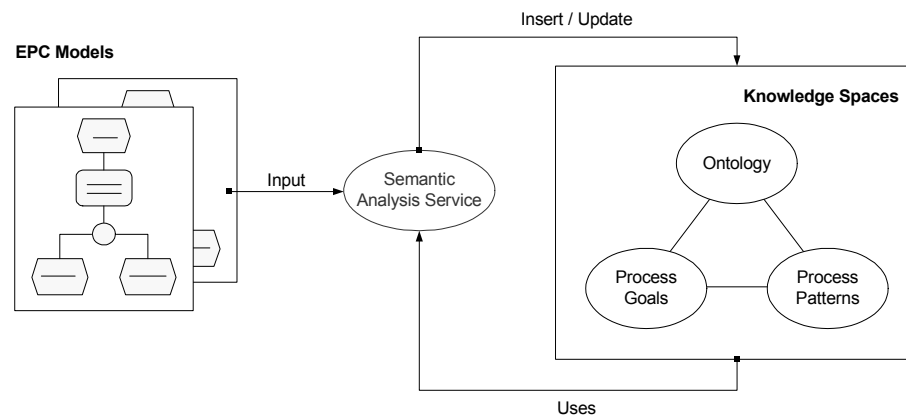


Abbildung 1.11.: Schematische Servicearchitektur [Bög07]

Jeder **Analysis Service** benötigt als Input ein oder mehrere semi-formale Prozessmodelle. Jede Ausführung eines Services greift auf die bestehende Wissensbasis zu bzw. führt eine Aktualisierung mit Analyseergebnissen durch.

Zielsetzung des **Knowledge Acquisition Services** sind die Extraktion von implizitem Prozesswissen aus bestehenden, semi-formalen Prozessmodellen sowie die Integration dieses Wissens in das **Process Knowledge Warehouse**. Diese Services

unterstützen den Aufbau und die Wartung des semantischen Prozesswissens und bilden die Grundlage für die weiteren **Analysis Services**.

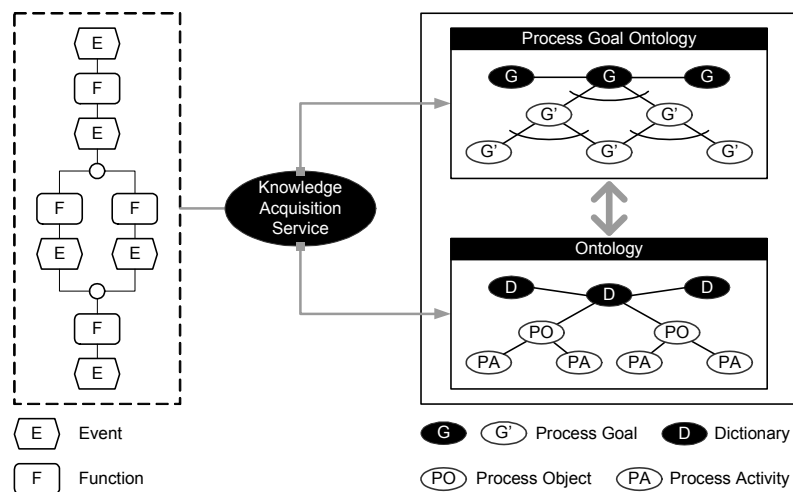


Abbildung 1.12.: Knowledge Acquisition Service

Derzeit besteht der **Assistance Analysis Service** aus zwei Sub-Services, um eine Prozessevaluierung durchzuführen. Mit dem **Metrics Service** ist es möglich, die Qualität und die Komplexität von semi-formalen Prozessmodellen zu berechnen. Anhand dieser Metriken können Prozessmodelle verglichen bzw. bewertet werden. Eine ausführliche Beschreibung des **Metrics Service** befindet sich in [Zie07].

Der **Model Clustering Service** ermöglicht die Bildung von Modellclustern mit Hilfe von Data Mining Methoden. Ein Cluster beinhaltet Prozessmodelle, die ähnliche Zielsetzungen verfolgen bzw. sich auf unterschiedlichen Niveaus der Abstraktion ähnlich sind. Nähere Ausführungen dazu befinden sich in [Grö06].

Der **Interactive Pattern Mining Service** bildet das Kernstück des Analyseframeworks. Mit Hilfe dieses Services werden Prozessmuster in semi-formalen Prozessmodellen identifiziert, weiter abstrahiert und im **Process Knowledge Warehouse** gespeichert. Input für diesen Service bilden das semantische Prozesswissen (Output des **Knowledge Acquisition Services**) der zu analysierenden semi-formalen Prozessmodelle.

## 1.2. Zielsetzung und Lösungsansatz

Die Zielsetzung dieser Arbeit liegt im Entwurf und der Realisierung einer Ontologie zur semantischen Annotierung von EPK-Funktionen und -Ereignissen, um die automatisierte Extraktion von Prozessmustern zu ermöglichen. Des Weiteren umfasst die Arbeit einen Ansatz zur Automatisierung der semantischen Annotierung

bestehender EPK-Modelle mit Instanzen der Ontologie, welche automatisiert aus den Modellelementbezeichnungen extrahiert werden.

Grundlegende Zielsetzung der Ontologie ist es, die maschinelle Vergleichbarkeit von bestehenden EPK-Modellen zu erhöhen, d.h. einen Vergleich von Modellelementen zu ermöglichen, welcher über eine rein textuelle Betrachtung der natürlichsprachlichen Bezeichnungen hinausgeht. Hierfür werden die Bezeichnungen der Modellelemente semantisch beschrieben, indem die einzelnen Elemente, aus welchen sich diese zusammensetzen, explizit ausgewiesen werden. In der vorliegenden Version werden lediglich EPK-Funktionen und -Ereignisse untersucht, wobei erstere *Tätigkeiten*, welche auf *Prozessobjekte* ausgeführt werden, und letztere *Zustände*, in welchen sich die *Prozessobjekte* befinden, beschreiben. Die explizite Annotierung der Modellelemente mit entsprechenden Konzepten der Ontologie, d.h. für die *Prozessobjekte*, *Tätigkeiten* und *Zustände*, ermöglicht einen gezielten Vergleich auf Basis der Semantik der einzelnen Elemente.

Die semantische Annotierung der Modellelemente unterstützt außerdem die Auflösung von Sprachdefekten. Hierfür wird bei der Instanzierung der Konzepte der Ontologie ein domänenspezifisches Wörterbuch verwendet. Aufgabe dieses Wörterbuchs ist die Verwaltung der lexikalischen Repräsentation der Modellelemente, d.h. die Bereitstellung der in den Bezeichnungen enthaltenen Wörter, welche anhand ihres Worttyps klassifiziert werden. Zusätzlich stellt es eine Reihe von Beziehungen zur Auflösung von Sprachdefekten zur Verfügung, welche die Grundlage für die Standardisierung der Instanzen der Ontologie bilden. Der Vorteil dieser Standardisierung ist, dass bei einem Vergleich der Modellelemente nicht jedesmal die einzelnen Wörter und Beziehungen des zugrunde liegenden Wörterbuchs untersucht werden müssen, sondern gezielt die Instanzen der Ontologie gegenübergestellt werden können.

Neben der semantischen Aufbereitung der Modellelementbezeichnungen ermöglicht die Ontologie die Bildung von Prozessaktivitäten. Eine Prozessaktivität stellt den kleinst-möglichen Modellierungsbaustein zur Beschreibung eines elementaren Prozessmusters dar und besteht aus der durchzuführenden Aktivität, der die Durchführung auslösenden Vorbedingung, sowie dem aus der Durchführung resultierenden Ergebnis. Während der erste Teil durch eine semantisch aufbereitete Funktion repräsentiert wird, werden letztere durch semantisch aufbereitete Ereignisse dargestellt. Außerdem wird mit einer Prozessaktivität sichergestellt, dass jeder Funktion ein Ergebnis, d.h. zumindest ein nachfolgendes Ereignis zugewiesen wird. Dies ist vor allem bei fehlenden Trivialereignissen, welche der Übersichtlichkeit wegen häufig nicht modelliert werden, von Bedeutung, da auf Basis der Ereignisse die Prozessziele abgeleitet werden. Mit diesen Prozesszielen wird in weiterer Folge ein Zielbaum für das EPK-Modell konstruiert, welcher die Grundlage für die Identifikation der Prozessmuster bildet. Die Konstruktion dieses Zielbaums ist allerdings nicht mehr Gegenstand dieser Arbeit.

Da die semantische Annotierung bei einer Vielzahl an Modellen ein komplexes

und zeitaufwändiges Vorhaben darstellt, wird zusätzlich ein Ansatz zur Automatisierung dieser Aufgabe vorgestellt. Die Zielsetzung hierbei liegt in der automatisierten Aufbereitung der EPK-Funktionen und -Ereignisse entsprechend der Konzepte der Ontologie sowie der semantischen Annotierung der Modellelemente mit diesen Instanzen. Basierend auf der lexikalischen Darstellung der Modellelemente aus dem Wörterbuch und einer Reihe von Ableitungsregeln (z.B. wird ein *aktives Verb* zur *Tätigkeit* der Ontologie), werden die unterschiedlichen Konzepte der Ontologie instanziiert und zu den entsprechenden Modellelementen annotiert. Des Weiteren umfasst der Ansatz die automatisierte Generierung von Generalisierungshierarchien von Prozessobjekten sowie von nicht modellierten Trivialereignissen, welche wie bereits erwähnt für die Ableitung der Zielbäume von wesentlicher Bedeutung sind.

### 1.3. Aufbau der Arbeit

Nachdem in den vorangegangenen Unterkapiteln Motivation und Gegenstand sowie Zielsetzung und Lösungsansatz beschrieben wurden, folgt nun ein kurzer Überblick über die weitere Gliederung dieser Arbeit.

In Kapitel 2 werden die theoretischen Grundlagen für diese Arbeit aufgearbeitet. Unterkapitel 2.1 beschäftigt sich mit dem allgemeinen Anwendungsbereich, dem Geschäftsprozessmanagement, wobei das Hauptaugenmerk auf der Definition und der Beschreibung von Geschäftsprozessen liegt. Im Anschluss daran wird in Unterkapitel 2.2 ein allgemeiner Überblick über die Architektur integrierter Informationssysteme (ARIS) gegeben, bevor in Unterkapitel 2.3 eine Beschreibung der ereignisgesteuerten Prozessketten erfolgt, indem deren Modellierungskonstrukte ausführlich beschrieben werden. Außerdem werden der Zweck der Verwendung von Namenskonventionen bei der Modellierung aufgezeigt und die Erweiterung von EPKs um Prozessobjekte vorgestellt. Zum Abschluss liefert Unterkapitel 2.4 einen Überblick über die semantische Prozessmodellierung. Ausgehend von einer Beschreibung von Ontologien für Informationssysteme, werden eine Reihe von Ontologien aus dem Bereich des Geschäftsprozessmanagements vorgestellt.

In Kapitel 3 wird die im Rahmen dieser Arbeit realisierte Ontologie zur semantischen Beschreibung von EPK-Funktionen und -Ereignissen vorgestellt. Ausgehend von einer Darstellung der Anforderungen an die Ontologie wird eine Abgrenzung zu verwandten Ansätzen vorgenommen. Im Anschluss daran werden die einzelnen Konzepte und Beziehungen zur semantischen Annotierung der EPK-Modellelemente beschrieben. Neben der konzeptionellen Beschreibung der Ontologie werden die Komponenten des pModeler Prototypen vorgestellt, welche für die Verwaltung und Bearbeitung der Ontologie verwendet werden.

Den zweiten Kernbereich dieser Arbeit bildet die automatisierte semantische Annotierung der EPK-Modelle, welche in Kapitel 4 beschrieben wird. Ausgehend von

einer Darstellung der Problemstellung bei der Aufbereitung der Modellelemente wird die Vorgehensweise bei der automatisierten Instanzierung der Ontologie auf Basis bestehender EPK-Modelle beschrieben und anhand von Beispielen verdeutlicht. Den Abschluss des Kapitels bildet eine Darstellung der im pModeler-Prototypen implementierten Analysekomponente sowie der Benutzeroberflächen für die Darstellung der den EPK-Modellen annotierten Instanzen der Ontologie.

Kapitel 5 dient der Beschreibung der prototypischen Implementierung der Kernbereiche dieser Arbeit und wird von einem Überblick über die gesamte Architektur des pModeler Prototypen eingeleitet. Im Anschluss daran wird das Datenmodell, mit welchem die Ontologie umgesetzt wurde, beschreiben. Des Weiteren wird das Objektmodell für die Instanzierung und die Verwaltung der Ontologie anhand von ausgewählten Beispielen näher beschrieben.

Zum Abschluss werden in Kapitel 6 kurz die Ergebnisse dieser Arbeit zusammengefasst und einige mögliche Erweiterungen der bestehenden Ansätze diskutiert.

## 2. Grundlagen

Dieses Kapitel dient der Aufarbeitung der theoretischen Grundlagen für diese Arbeit und ist in die folgenden Unterkapitel untergliedert.

Unterkapitel 2.1 liefert einleitend einen allgemeinen Überblick über das Geschäftsprozessmanagement und die Geschäftsprozessmodellierung, welche den grundlegenden Anwendungsbereich dieser Arbeit bildet. In Weiterer Folge wird der Begriff Geschäftsprozess definiert und es werden die unterschiedlichen Arten von Prozessen vorgestellt. Im Anschluss daran wird auf die Beschreibung von Geschäftsprozessen eingegangen, indem unterschiedliche Sichten der Geschäftsprozessmodellierung vorgestellt, der Modellbegriff erläutert, Kategorien von Modellierungssprachen gegenübergestellt und die Zielsetzungen der Prozessmodellierung dargestellt werden.

Da die zur Analyse verwendeten EPK-Modelle mittels des ARIS (Architektur Integrierter Informationssysteme) Toolsets modelliert wurden, wird in Unterkapitel 2.2 kurz auf das allgemeine ARIS Konzept eingegangen. Hierbei werden das zugrunde liegende Phasenmodell, das Sichtenkonzept sowie die Integration dieser beiden Dimensionen in das sogenannte ARIS Haus vorgestellt. Außerdem erfolgt eine Einordnung der im Rahmen dieser Arbeit verwendeten Ereignisgesteuerten Prozessketten in das Gesamtkonzept.

Grundlage für die Analysen dieser Arbeit bilden Ereignisgesteuerte Prozessketten (EPKs), welche in Unterkapitel 2.3 beschrieben werden. Der Fokus liegt hierbei auf der Darstellung der Modellierungskonstrukte, mit welchen EPKs modelliert werden können. Außerdem werden die im Rahmen dieser Arbeit verwendeten Namenskonventionen für die Bezeichnungen von EPK-Funktionen und -Ereignissen vorgestellt und die Erweiterung von EPKs um Prozessobjekte beschrieben.

Zielsetzung dieser Arbeit ist die semantische Annotierung von EPK-Modellen mittels Instanzen einer Ontologie. Die Annotierung von Prozessmodellen ist Teil der semantischen Prozessmodellierung, welche in Unterkapitel 2.4 vorgestellt wird. Nach einer kurzen Einleitung über das semantische Geschäftsprozessmanagement und der semantischen Prozessmodellierung werden die Vorteile der semantischen Annotierung von Prozessmodellen dargestellt. Im Anschluss daran wird auf Ontologien eingegangen, indem der Begriff definiert, ihre Komponenten vorgestellt und verschiedene Arten von Ontologien unterschieden werden. Zum Abschluss wird eine Auswahl an Ontologien aus dem Bereich des Geschäftsprozessmanagement vorgestellt.

## 2.1. Geschäftsprozessmanagement

Schon immer wurden Tätigkeiten, Aufgaben und Arbeitsabläufe analysiert, um Optimierungsmöglichkeiten für die Verbesserung der Marktsituation eines Unternehmens zu erschließen. Waren es früher jedoch noch einzelne Teilaufgaben und lediglich kurze Arbeitsabläufe, traten mit dem Konzept der Geschäftsprozesse zunehmend längere, zusammenhängende Tätigkeitsabfolgen in den Mittelpunkt der Betrachtung [Sta06, S. 5].

Geschäftsprozessmanagement bezieht seine Berechtigung daraus, dass die Geschäftsprozesse eines Unternehmens einem ständigen Wandel unterliegen. Die Notwendigkeit zur Neugestaltung können durch die Veränderungen des Unternehmensumfelds, neue Entwicklungen auf dem Technologiemarkt, die Erschließung neuer Märkte oder durch die Identifikation von Schwachstellen hervorgerufen werden [NPW05, S. 299ff]. Um schnell auf neue Entwicklungen reagieren zu können, ist daher ein unternehmensweites Geschäftsprozessmanagement notwendig.

Eine einheitliche oder verbindliche Definition des Begriffs Geschäftsprozessmanagement (GPM) bzw. Business Process Management (BPM) liegt bis dato nicht vor. Van der Aalst et al. definieren BPM bspw. wie folgt: *”Supporting business processes using methods, techniques, and software to design, enact, control, and analyze operational processes involving humans, organizations, applications, documents and other sources of information”* [AHW03]. Diese Definition hebt die unterstützende Funktion des BPM durch die Bereitstellung geeigneter Methoden, Techniken und Software in allen Phasen des BPM-Lifecycles hervor. Die Zielsetzung des BPM liegt in der Beherrschung komplexer Geschäftsprozesse. Es umfasst daher eine ganzheitliche Betrachtung der in einem Unternehmen vorhandenen Geschäftsprozesse, von der Erfassung und Beschreibung, über deren Durchführung, bis zur Überwachung und Optimierung derselben (vgl. u.a. [SS07, S. 4ff] oder [Wes07, S. 4ff]). Auch Allweyer teilt diese Auffassung und betrachtet *”die systematische Gestaltung, Steuerung, Überwachung und Weiterentwicklung der Geschäftsprozesse eines Unternehmens”* als Aufgabenbereiche dieser Disziplin [All05, S. 12].

Für eine systematische Vorgehensweise wurden in der Literatur verschiedene zyklische Phasenmodelle formuliert, welche sich lediglich durch die Detaillierung der einzelnen Phasen oder durch unterschiedliche Begrifflichkeiten unterscheiden (vgl. u.a. [All05, S. 89ff], [Wes07, S. 11ff] oder die Phasen des kontinuierlichen Prozessmanagements [NPW05, S. 309ff]). Da diese Phasen den Lebenszyklus des Geschäftsprozessmanagements beschreiben, wird dieses Vorgehensmodell häufig unter dem Begriff BPM-Lifecycle zusammengefasst.

Van der Aalst [Aal04] unterscheidet die Phasen *process design* (Prozessdesign), *system configuration* (Systemkonfiguration), *process enactment* (Prozessinkraftsetzung) und *diagnosis* (Diagnose), welche in Abbildung 2.1 zusammengefasst sind. Sehr ähnlich beschreiben Wetzstein et. al. [WMF<sup>+</sup>07] den BPM-Lifecycle, welche

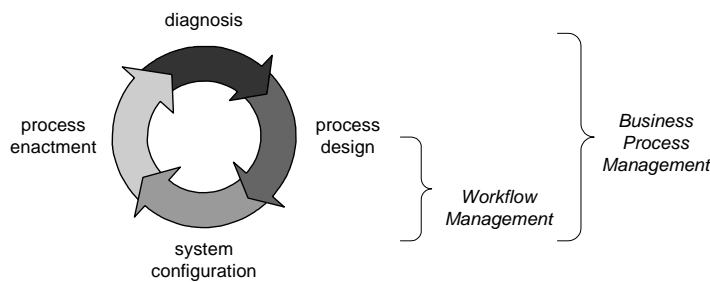


Abbildung 2.1.: BPM-Lifecycle [Aal04]

die Phasen *Process Modeling* (Prozessmodellierung), *Process Implementation* (Prozessimplementierung), *Process Execution* (Prozessausführung) und *Process Analysis* (Prozessanalyse) unterscheiden. Die grundlegenden Inhalte der einzelnen Phasen werden in den folgenden Absätzen kurz zusammengefasst:

- **Modellierungsphase:** in diesem Bereich werden die einzelnen Tätigkeiten und die Prozessabläufe beschrieben. Hierzu zählt sowohl eine Bestandsaufnahme der bestehenden Prozesse als auch die Beschreibung wie diese Abläufe sein sollten. Neben der Darstellung der Prozessabläufe sind weitere Einflüsse, wie beispielsweise organisatorische Aspekte oder Datenflüsse, zu berücksichtigen.
- **Implementierungsphase:** bezeichnet die Phase, in welcher die entworfenen Prozesse umgesetzt werden. Neben organisatorischen Maßnahmen werden auch Informationssysteme zur Unterstützung oder der Automatisierung von Prozessabläufen entwickelt.
- **Ausführungsphase:** die Ausführung stellt einen kontinuierlichen Prozess dar. Die entwickelten Geschäftsprozesse und Informationssysteme werden produktiv eingesetzt, um die verfolgten Unternehmensziele zu verwirklichen. Diese Phase wird durch die Erfassung von Prozessdaten begleitet, welche in der Analysephase ausgewertet werden.
- **Analysephase:** diese kann in die Bereiche *Process Monitoring* und *Process Mining* unterteilt werden. Das Process Monitoring liefert Informationen über laufende Prozesse und protokolliert diese. Diese kontinuierliche Überwachung unterstützt eine möglichst zeitnahe Identifikation von Störfällen. Das Process Mining analysiert aufgezeichnete Protokolldaten und liefert dadurch Informationen zur Bewertung der Prozesseffizienz oder zur Identifikation von Optimierungsmöglichkeiten.

Die Kernbereiche dieser Arbeit können in die Modellierungsphase eingeordnet werden, da die semantische Annotierung der Prozessmodelle neue Modellierungsprojekte unterstützen kann. Wetzstein et. al. [WMF<sup>+</sup>07] haben in diesem Zusammenhang die Phasen des klassischen BPM-Lifecycles um semantische Konzepte erweitert, um das Management der Geschäftsprozesse zu unterstützen (vgl. Kapitel



2.4). Die Zielsetzung dieser Arbeit ist allerdings keine Automatisierung der Prozesse im Sinne des Workflow Managements. Vielmehr soll durch die semi-automatische Aufbereitung bestehender Prozessmodelle das Prozesswissen erweitert und durch die Persistierung in einer Ontologie zugänglich gemacht werden.

### 2.1.1. Geschäftsprozesse

Die hohe Bedeutung von Geschäftsprozessen ist auch durch die Definitionsvielfalt in der Literatur ersichtlich. Ziel dieses Kapitels ist keine ausführliche Beschreibung des Begriffes, wie es beispielsweise in [Sta06, S. 5ff] oder [Gad05, S. 34ff] der Fall ist. Vielmehr soll auf Basis ausgewählter Beispiele ein grundsätzliches Verständnis für den Geschäftsprozessbegriff geschaffen werden.

- Hammer und Champy definieren *”einen Unternehmensprozess als Bündel von Aktivitäten, für das ein oder mehrere unterschiedliche Inputs benötigt werden und das für den Kunden ein Ergebnis von Wert erzeugt”* (zitiert aus [Sta06, S. 9]).
- In [HHR04, S. 282] wird ein Geschäftsprozess als *”eine Menge integrierter, logisch miteinander verbundener und messbarer Tätigkeiten [...], die für die Schaffung eines spezifischen Ergebnisses für einen Kunden oder Markt durchgeführt werden und für das Unternehmen von wesentlicher Bedeutung sind”* definiert.
- Sehr gründlich erörtert Staud den Begriff und definiert ihn selbst sehr ausführlich: *”Ein Geschäftsprozess besteht aus einer zusammenhängenden abgeschlossenen Folge von Tätigkeiten, die zur Erfüllung einer betrieblichen Aufgabe notwendig sind. Die Tätigkeiten werden von Aufgabenträgern in organisatorischen Einheiten unter Nutzung der benötigten Produktionsfaktoren geleistet. Unterstützt wird die Abwicklung der Geschäftsprozesse durch das Informations- und Kommunikationssystem des Unternehmens”* [Sta06, S. 9].

Geschäftsprozesse setzen sich also im Wesentlichen aus Aktivitäten zusammen, welche zeitlich und logisch nachvollziehbar durchgeführt werden. Während ihrer Ausführung konsumieren sie Produktionsfaktoren, also Mittel und Leistungen materieller oder immaterieller Art, um ein gewünschtes Resultat zu erzielen. Als Ergebnis liefern sie materielle Produkte oder Dienstleistungen, welche für den Kunden (interne oder externe) einen definierten Mehrwert darstellen [Ros02, S. 1ff].

Aus Sicht der Organisationslehre werden die Geschäftsprozesse in die Ablauforganisation eingeordnet. Während die Aufbauorganisation die eher statische Betrachtungsweise der Strukturen eines Unternehmens und die Zuteilung von Aufgabenträgern zu Aufgaben zum Inhalt hat, beschreibt die Ablauforganisation die Ausführungsabfolge von Aufgaben und deren räumliche und zeitliche Koordinierung [BK05, S. 6ff].

Zusammenfassend kann festgehalten werden, dass grundsätzlich Einigkeit darüber besteht, was ein Geschäftsprozess ist. Rump fasst die Gemeinsamkeiten wie folgt zusammen [Rum99, S. 19]:

- Ein Geschäftsprozess besteht aus einer Menge von Aktivitäten in einem zeitlichen, sachlogischen Zusammenhang.
- Das Ergebnis stellt einen Nutzen für den Kunden dar.
- Die Durchführung betrifft häufig mehrere Organisationseinheiten.
- Mit seiner Durchführung wird ein bestimmtes Ziel verfolgt.
- Für seine Ausführung werden Ressourcen, wie Personen oder Maschinen benötigt.

Geschäftsprozesse stellen den Untersuchungsgegenstand dieser Arbeit dar, wodurch eine Abgrenzung zu dem nahe verwandten Begriff des Workflows naheliegend ist. Während Geschäftsprozesse eine überwiegend strategisch orientierte Betrachtungsweise von Prozessen darstellen, liegt der Fokus im Bereich der Workflows mehr auf technisch-operativer Ebene. Die Workflow Management Coalition (WfMC) definiert einen Workflow daher als *"the computerised facilitation or automation of a business process, in whole or part"* [Hol95]. Kernbereich des Workflow Management ist also die Unterstützung und/oder Automatisierung von Prozessen oder Teilen davon durch IT Systeme.

Es gibt unterschiedliche Arten von Geschäftsprozessen, welche nach ihrer Nähe zum Kerngeschäft des Unternehmens unterteilt werden können. In diesem Zusammenhang können Steuerungsprozesse, Kerngeschäftsprozesse und Unterstützungsprozesse unterschieden werden [Gad05, S. 39f]:

- **Steuerungsprozesse** regeln die reibungslose Durchführung aller Geschäftsprozesse eines Unternehmens und koordinieren die leistungserstellenden und unterstützenden Prozesse.
- **Kernprozesse** sind die zentralen, wettbewerbskritischen Prozesse eines Unternehmens. Sie bilden den Leistungserbringungsprozess ab und erzielen somit die eigentliche Wertschöpfung für das Unternehmen.
- **Unterstützungsprozesse** liefern keinen direkten Beitrag zur Wertschöpfung. Jedoch wäre ohne diese die Durchführung der Kernprozesse nicht möglich.

### 2.1.2. Beschreibung von Geschäftsprozessen

Geschäftsprozesse setzen sich aus einer Menge unterschiedlicher Komponenten, wie bspw. Tätigkeiten oder Aufgabenträger, zusammen, welche durch verschiedenste interne und externe Beziehungen in Verbindung gesetzt werden und deren Abläufe

durch unterschiedliche Informationsflüsse geprägt sind. Die Geschäftsprozessmodellierung verfolgt daher die Zielsetzung die Komplexität dieses Systems zu reduzieren. Aus diesem Grund werden Modelle entworfen, welche sich lediglich auf untersuchungsrelevante Teilbereiche und deren Beziehungen konzentrieren, um Transparenz für die beteiligten Personen zu schaffen [KNS92].

Zur Verbesserung der Anschaulichkeit der entworfenen Modelle wird in der Literatur die Aufteilung des Untersuchungsbereichs in verschiedene Teilmodelle vorgeschlagen, welche unterschiedliche Sichtweisen auf dasselbe Problem darstellen. Abbildung 2.2 stellt in diesem Zusammenhang verschieden Ansätze gegenüber, von welchen jener von Scheer wohl der bekannteste ist. Dieser im Rahmen des ARIS-Konzepts entwickelte Ansatz unterscheidet einerseits die vorwiegend statischen Bereiche Organisations-, Funktions-, Daten- und Leistungssicht, andererseits die dynamische Steuerungs- oder Prozesssicht, welche den Prozessablauf abbildet und die übrigen Teilbereiche integriert (für eine detaillierte Darstellung vgl. Kapitel 2.2).

Sichtenkonzepte der Geschäftsprozessmodellierung				
Scheer	Österle	Ferstl/Sinz	Gehring	Gadatsch
Organisations-Sicht	Organisation	Leistungssicht	Organisations-Sicht	Prozess-Sicht
Funktionssicht	Funktionen	Lenkungsicht	Funktionssicht	Organisations-struktursicht
Datensicht	Daten	Ablaufssicht	Datensicht	Aktivitäts-struktursicht
Steuerungssicht	[Personal]			Applikations-Struktursicht
Leistungssicht	[...]			Informations-struktursicht

Abbildung 2.2.: Sichtenkonzepte der Geschäftsprozessmodellierung [Gad05, S. 64ff]

Einen weiteren Ordnungsrahmen für die Beschreibung von Prozessen führt Jablonski an [Jab95, S. 17ff]. Obwohl dieser Ansatz vor allem im Bereich der Workflows anzusiedeln ist, lassen sich parallelen zu den Sichtenkonzepten der Geschäftsprozessmodellierung erkennen. Eine grobe Untergliederung erfolgt in die Bereiche sachliche Aspekte, welche den Inhalt der Prozesse beschreiben, und technische Aspekte, welche im Kontext von Workflows deren Ausführung aus softwaretechnischer Betrachtungsweise definieren. Da die technischen Aspekte lediglich im Zusammenhang mit der Workflowmodellierung von Bedeutung sind, werden diese lediglich der Vollständigkeit halber kurz dargestellt. Folgende Aspekte können unterschieden werden (vgl. auch [HZJ04]):

- **sachliche Aspekte:**

- Der **funktionale Aspekt** beschreibt die Tätigkeiten, welche im Rahmen eines Prozesses durchgeführt werden. Neben der Spezifikation dieser Tätigkeiten können auch Konsistenzbedingungen definiert werden. Zu diesen

zählen beispielsweise Angaben zur Ausführungsdauer einer Tätigkeit oder kontextspezifische Restriktionen, wie Vor- und Nachbedingungen.

- Der **operationale Aspekt** spezifiziert die Art der Prozessdurchführung bzw. -unterstützung. In diesem Teilbereich werden sämtliche Hilfsmittel (Applikationen, Ressourcen, etc.) zusammengefasst, welche den Ablauf der Prozesse unterstützen.
- Die Beschreibung des Kontrollflusses erfolgt im Rahmen des **verhaltensbezogenen Aspekts**. In diesem Bereich wird geregelt, wann die einzelnen Prozessschritte durchgeführt werden, wobei grundlegende Kontrollflusskonstrukte, wie Sequenz, Parallelisierung und alternative Abläufe Verwendung finden.
- Im Bereich des **informationsbezogenen Aspekts** werden die Daten beschrieben, welche von den Prozessen benötigt, erzeugt oder manipuliert werden.
- Mit den bislang vorgestellten Teilbereichen kann die Ablauforganisation beschrieben werden. Die aufbauorganisatorischen Konzepte werden nun im Rahmen des **organisatorischen Aspekts** aufbereitet. Hauptaufgabe dieses Bereichs ist die Regelung der Verantwortlichkeiten für die verschiedenen Prozessschritte. Hierfür werden Rollen definiert, welche den unterschiedlichen Tätigkeiten zugewiesen werden können.
- Den letzten sachlichen Aspekt bildet der **kausale Aspekt**. Dieser beschreibt, *„warum ein Workflow überhaupt so spezifiziert wird, wie er vorliegt, und warum ein Workflow überhaupt ausgeführt wird“* [Jab95, S. 58]. Diese Beschreibung wird im Wesentlichen von spezifischen Randbedingungen, wie rechtlichen oder unternehmensinternen Vorschriften, geprägt.

#### - **technische Aspekte:**

- Der **historische Aspekt** umfasst sämtliche Beschreibungen, welche für die Protokollierung der Prozessabläufe benötigt werden.
- Um die konsistente Durchführung und Wiederherstellung von einzelnen Prozessschritten zu gewährleisten, werden im Rahmen der Beschreibungen des **transaktionalen Aspekts** Transaktionsmodelle definiert, welche u.a. spezifizieren, was als konsistenter Zustand betrachtet werden kann.

Ein Kernbereich der Beschreibung von Geschäftsprozessen ist deren Darstellung in Form von Modellen. Ein Modell stellt die vereinfachte Darstellung der Wirklichkeit oder eines Ausschnitts daraus dar, indem die für den Untersuchungszweck relevanten Aspekte betont, die übrigen jedoch vernachlässigt werden. So definieren beispielsweise Heinrich et. al. ein Modell als *„jede vereinfachende Abbildung eines*

*Ausschnitts der Wirklichkeit oder eines Vorbilds für die Wirklichkeit (Beschreibungsmodell), wobei trotz aller Vereinfachung Strukturgleichheit oder zumindest Strukturähnlichkeit zwischen Wirklichkeit und Abbildung bzw. Vorbild und Wirklichkeit gefordert wird” [HHR04].*

Aufbauend auf der allgemeinen Definition des Modellbegriffs definiert Wyssusek den Begriff Geschäftsprozessmodell als eine *”zweckorientierte, vereinfachte Abbildungen von Geschäftsprozessen [...] Ihre Struktur spiegelt die zeitlich-sachlogische Abfolge der betrachteten Funktionen wider”* (vgl. [MBB<sup>+</sup>01, S. 210]). Dass mit dem Modellbegriff nicht notwendigerweise eine grafische Darstellung einhergehen muss, deuten Frank und van Laak [FL03] in ihrer Definition an: *”Ein Geschäftsprozessmodell ist eine zweckgerichtete Abstraktion eines Geschäftsprozesstyps, die häufig - aber nicht notwendig - mit einer grafischen Darstellung einhergeht”*. Als Geschäftsprozess-typen betrachten die Autoren eine Klasse gleichartiger Geschäftsprozesse.

Für die grafische Darstellung von Geschäftsprozessen werden Modellierungssprachen verwendet. Aufgrund der großen Bedeutung der Geschäftsprozessmodellierung kann mittlerweile eine große Anzahl an Sprachen unterschieden werden. Diese setzen sich grundsätzlich aus einer Reihe von Symbolen, einer Syntax und einer Semantik zusammen. Die Syntax beschreibt sowohl die bereitgestellte Symbolmenge und deren Anordnungsvorschriften (abstrakte Syntax) als auch die konkrete Darstellungsform der verschiedenen Symbole (konkrete Syntax oder Notation). Die Semantik hingegen legt einerseits den Sinn und die Bedeutung der einzelnen Symbole fest, andererseits schränkt sie die syntaktischen Anordnungsvorschriften ein, indem sie diese um semantische Regeln erweitert.

Die Formalisierung legt den Grad der Detaillierung der Syntax und Semantik fest. Je nachdem wie präzise die einzelnen Teile formuliert sind, können unterschiedliche Kategorien von Modellierungssprachen unterschieden werden [FL03]:

- **formale Modellierungssprachen:** die Menge der zulässigen Symbole, sowie deren Semantik ist eindeutig festgelegt. Weiters zeichnet sich diese durch eine eindeutige Syntax und eindeutige semantische Integritätsbedingungen aus (z.B.: Petri-Netze).
- **semi-formale Modellierungssprachen:** stellt eine eindeutig definierte Symbolmenge bereit, deren Syntax zumindest teilweise definiert ist (z.B.: EPK, UML).
- **informale Modellierungssprachen:** umfasst lediglich eine Menge von Symbolen, ohne konkret die Syntax oder Semantik zu spezifizieren.

Die Zielsetzung, welche mit der Modellierung von Geschäftsprozessen verfolgt wird, ist vielfältig. Diese Heterogenität der Einsatzbereiche stellt unterschiedliche Anforderungen an die Prozessmodelle. In diesem Zusammenhang unterteilen Rosemann et. al. [RSD05, S. 51ff] die Einsatzzwecke von Prozessmodellen in die beiden Hauptgruppen Organisationsgestaltung und Anwendungssystemgestaltung (vgl.

Abb. 2.3).

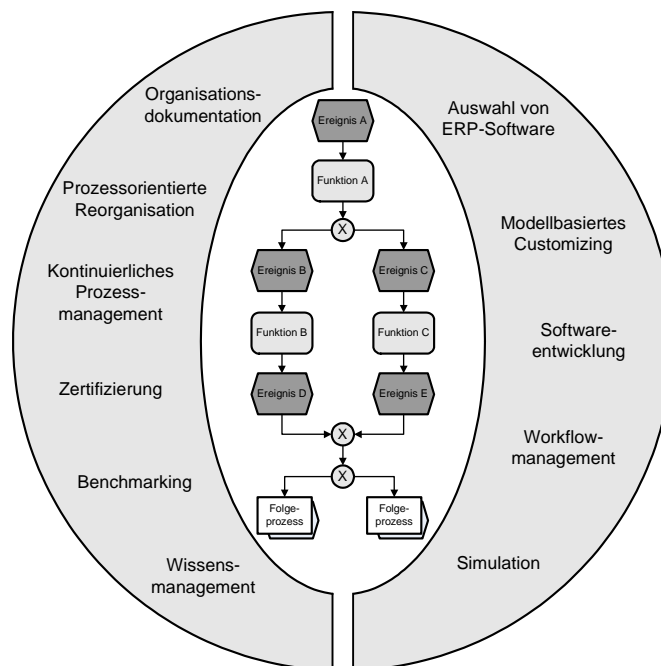


Abbildung 2.3.: Einsatzzwecke von Prozessmodellen [BK05, S. 51ff]

Im Bereich der Organisationsgestaltung liegt der Schwerpunkt auf einer hohen Anschaulichkeit der Prozessmodelle. Dies soll durch die Bereitstellung von weitestgehend selbsterklärenden Symbolen unterstützt werden, um einen intuitiven Zugang zum Untersuchungsbereich für alle Beteiligten zu ermöglichen.

Im Rahmen der Anwendungssystemgestaltung hingegen liegt der Fokus auf der Umsetzung von fachlich-konzeptionellen Beschreibungen in DV- bzw. Implementierungskonzepten. Aufgrund der Nähe zur Implementierung ergeben sich hierbei formale Anforderungen an die Prozessmodelle. Des Weiteren bestehen zumeist enge Beziehungen zu anderen Informationsmodellen, wie beispielsweise Daten- oder Objektmodellen.

Die folgenden Aufzählungen enthalten einige ausgewählte Beispiele für Einsatzzwecke von Prozessmodellen, um die Bedeutung der Prozessmodellierung zu unterstreichen (für eine ausführliche Beschreibung der einzelnen Punkte und weitere Beispiele vgl. u.a. [RSD05, S. 51], [Ros96, S. 42ff] oder [Rum99, S. 20ff]):

### Organisationsgestaltung:

- Organisationsdokumentation
- Zertifizierungsmaßnahmen nach ISO 9000
- Geschäftsprozessoptimierung

- Benchmarking
- etc.

### **Anwendungssystemgestaltung**

- Spezifikation / Auswahl und Konfiguration von Softwaresystemen
- Workflowmanagement / Spezifikation von Workflows
- Simulationsstudie
- etc.

## 2.2. Architektur Integrierter Informationssysteme (ARIS)

Die *Architektur integrierter Informationssysteme (ARIS)* bezeichnet sowohl ein Konzept als auch ein Softwareprodukt zur ganzheitlichen Beschreibung von Geschäftsprozessen. Das ARIS-Konzept wurde in den neunziger Jahren am Institut für Wirtschaftsinformatik (IWi) der Universität des Saarlandes unter der Leitung von Prof. August-Wilhelm Scheer entwickelt und wird seit 1992 als kommerzielles Softwareprodukt ARIS Toolset entwickelt und erfolgreich vertrieben. Das ARIS-Konzept bildet ein methodisches Rahmenwerk für eine Vielzahl an Methoden, welche für die Modellierung von Geschäftsprozessen verwendet werden können. Zur Reduzierung der Komplexität bei der Beschreibung von Geschäftsprozessen werden unterschiedliche Sichten in ein Life-Cycle-Modell eingeordnet, welchen entsprechende Modellierungsmethoden zugewiesen werden. Die konzeptionelle Durchgängigkeit und die breite Akzeptanz des ARIS Toolsets in Wirtschaft und Forschung machen ARIS zum de facto Standard im Bereich der Geschäftsprozessmodellierung (vgl. u.a. [Sch98a], [Sch98b], [Sei06, S. 11]).

Wie bereits erwähnt setzten sich Geschäftsprozesse aus unterschiedlichen Elementen zusammen. Dieser Tatsache entsprechend stellt auch das ARIS-Metageschäftsprozessmodell eine Vielzahl verschiedener Klassen, wie bspw. Ereignisse, Funktionen, Organisationseinheiten oder Anwendungssoftware, zur Verfügung, welche durch unterschiedliche semantische Beziehungen miteinander verknüpft werden können. Zur Reduzierung der Komplexität des Gesamtsystems wird eine Untergliederung in ein Sichtenkonzept vorgenommen, indem Klassen mit ähnlichen semantischen Zusammenhängen zusammengefasst werden [Sch98a, S. 33ff]. Die Unterteilung erfolgt hierbei einerseits anhand der vier statischen Sichten Funktions-, Organisations-, Daten- und Leistungssicht, andererseits anhand der statisch-dynamischen Prozess- oder Steuerungssicht, welche die Zusammenhänge zwischen den einzelnen statischen Sichten herstellt [Mül05, S. 219ff].

Abbildung 2.4 zeigt das ARIS-Metageschäftsprozessmodell und die Einordnung der einzelnen Elemente in die statischen Sichten des ARIS-Konzepts. Der wesentliche Vorteil dieser Untergliederung liegt in der Reduktion der Komplexität des Gesamtsystems. Die einzelnen Teilbereiche können mehr oder weniger unabhängig voneinander bearbeitet werden und es können gezielt geeignete Methoden zu deren Modellierung eingesetzt werden. Weiters werden durch die Integration der Elemente der vier statischen Sichten in der Prozesssicht Redundanzen vermieden [Mül05, S. 219ff].

### Das ARIS-Phasenmodell

Das ARIS-Phasenmodell bildet einen Rahmen für eine schrittweise Umsetzung der betriebswirtschaftlichen Fachbeschreibungen der Geschäftsprozesse in konkrete Konzepte der Informationstechnik (vgl. Abb. 2.5). Es unterstellt allerdings keine sequen-



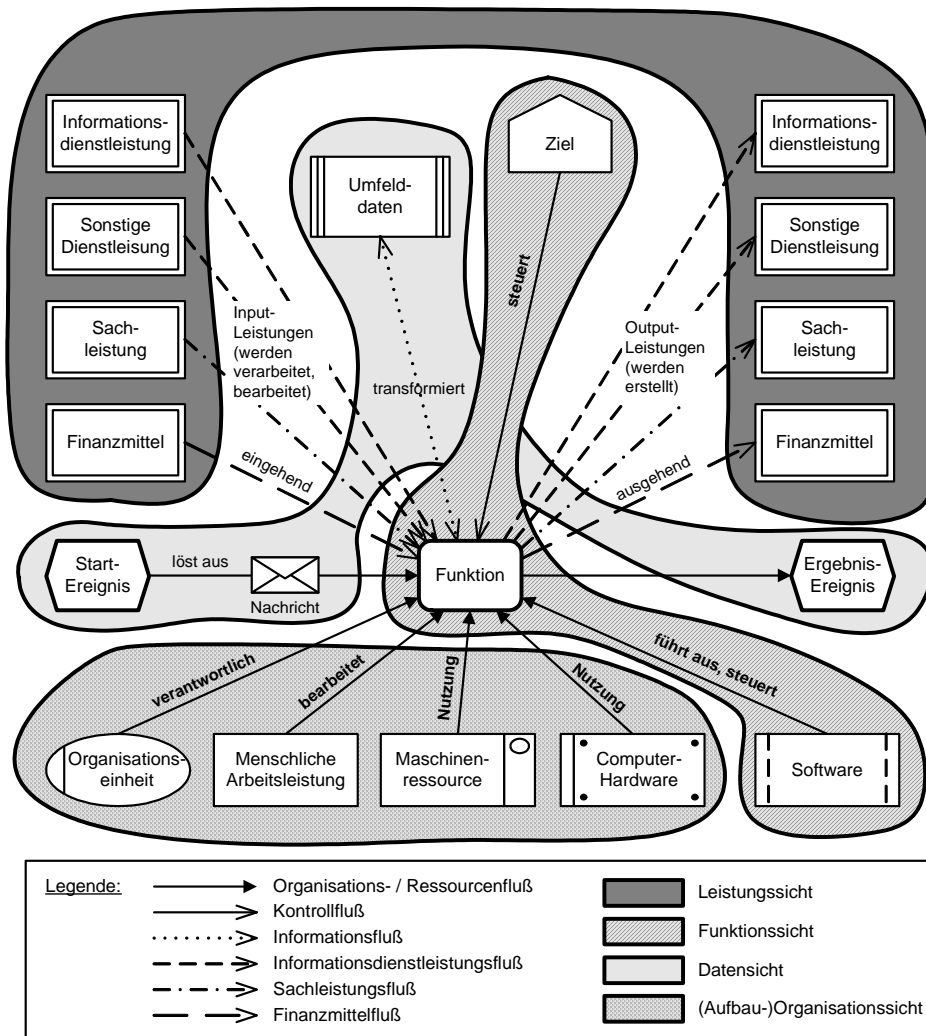


Abbildung 2.4.: ARIS-Metageschäftsprozessmodell (vgl. [Sch98a, S. 31 bzw. S. 34f])

zielle Abfolge, sondern legt eine prototypische Entwicklung nahe, in welcher Ergebnisse eines Schrittes weitere Verfeinerungen in der vorangegangenen Phase bewirken können [Sch98a, S. 38ff].

- **Fachkonzept:** Ausgangspunkt bilden die im Rahmen der Unternehmensplanung festgelegten, langfristig gültigen Unternehmensziele. Die Modellierung der einzelnen Sichten ist primär durch betriebswirtschaftlich-organisatorische Inhalte gekennzeichnet. Jedoch werden diese Beschreibungen soweit formalisiert, dass eine konsistente Überführung in die DV-technischen Konzepte möglich ist.
- **DV-Konzept:** die überwiegend betriebswirtschaftlichen Beschreibungen der Fachkonzeptebene werden mit den Anforderungen und Einschränkungen der DV-technischen Konzepte abgestimmt. Ohne direkt Bezug auf konkrete In-

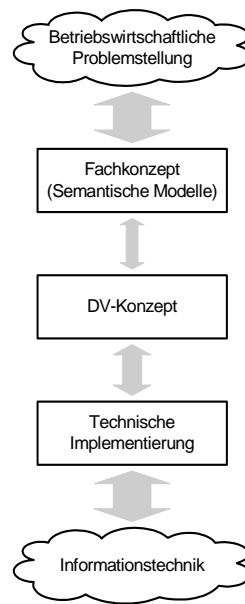


Abbildung 2.5.: Beschreibungsebenen eines Informationssystems [Sch97, S. 14ff]

formationstechnologien zu nehmen, werden Anpassungen an die Schnittstellen der Implementierungswerkzeuge, wie bspw. Datenbanksysteme oder Programmiersprachen, vorgenommen.

- **Implementierung:** es erfolgt die Realisierung der definierten Anforderungen in physische Komponenten der Informationstechnik. Dazu zählen u.A. physische Datenstrukturen, Hardware-Komponenten und Softwaresysteme.

### Die ARIS-Sichten

Das im vorangegangenen Abschnitt dargestellte Phasenmodell wird mit den bereits einleitend angesprochenen Sichten kombiniert und zum sogenannten ARIS-Haus zusammengefasst (vgl. Abb. 2.6). Dieses ermöglicht eine vollständige und zweckmäßige Abbildung sämtlicher Aspekte der Geschäftsprozesse, indem für jede Komponente des ARIS-Hauses geeignete Beschreibungsmethoden bzw. die entsprechenden Modellierungsmodelle bereitgestellt werden [Sei06, S. 24f].

Das ARIS-Haus und die verwendeten Methoden wurden bereits vielfach in der Literatur beschrieben (vgl. u.A. [Sch98b], [Sch98a], [Gad05, S. 113ff], [Sei06, S. 11ff]). Ziel der folgenden Ausführungen ist daher, lediglich einen Überblick über die Inhalte der einzelnen Sichten zu geben. Des weiteren erfolgt eine Einschränkung auf die Ebene des Fachkonzepts, da die Kernbereiche dieser Arbeit in dieser Phase anzusiedeln sind.

- **Funktionssicht:**  
Die Hauptaufgabe der Funktionssicht liegt in der Beschreibung der fachlichen

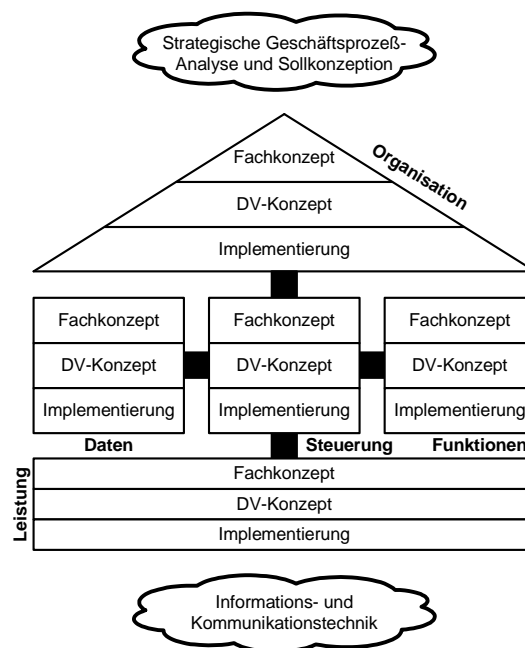


Abbildung 2.6.: ARIS-Haus

Aufgaben und Tätigkeiten zur Erreichung der Unternehmensziele. Neben den Funktionen werden auch die Anwendungssysteme und deren Module beschrieben, welche diese unterstützen bzw. umsetzen.

In der Phase des **Fachkonzepts** werden hierbei die Funktionen und deren Beziehungen erfasst. Diese werden auf unterschiedlichen Abstraktionsniveaus beschrieben und mittels *Funktionsbäumen* dargestellt. Aufgrund des engen Bezugs zu den Zielen des Unternehmens, werden auch diese in diesem Bereich definiert. Die Darstellung erfolgt mittels *Zieldiagrammen* und es werden auch Beziehungen zu den Funktionen, welche diese realisieren, hergestellt.

- **Organisationssicht:**

Die Darstellung der Aufbauorganisation des Unternehmens ist Aufgabe der Organisationssicht. Neben der fachlichen und disziplinarischen Hierarchisierung der Organisation umfasst dieser Bereich auch die Erhebung der zur Unterstützung bereitgestellten Betriebsmittel und EDV-Systeme.

Das **Fachkonzept** ist hierbei durch die Definition von Organisationseinheiten geprägt, welche durch die Zusammenfassung ähnlicher Aufgaben gebildet werden. Hauptaufgabe ist also die Strukturierung der Aufgaben, der Aufgabenträger und deren Beziehungen. Diese erfolgt mittels *Organigrammen*, welche eine hierarchische Darstellung auf unterschiedlichen Abstraktionsebenen ermöglichen.

- **Datensicht:**

Die Datensicht dient der Beschreibung sämtlicher Informationsobjekte, inklusive derer Attribute und Beziehungen. Weiters werden die Ereignisse, welche Prozesszustände darstellen, in diesem Bereich definiert.

Die **Fachkonzept**-Ebene beschäftigt sich in diesem Bereich mit der Ermittlung der Datenstrukturen, welche mittels des *erweiterten Entity-Relationship-Modells (eERM)* beschrieben werden. Neben dieser sehr detaillierten Darstellungsmethode, können auch sogenannte *Fachbegriffsmodelle* erstellt werden, welche die Begrifflichkeiten eines Unternehmens und deren Zusammenhänge darstellen.

- **Leistungssicht:**

In der Leistungssicht werden die materiellen und immateriellen Input- und Output-Leistungen, sowie die Geldflüsse der Prozesse beschrieben. Sie enthält keine spezifischen Verfahren für die Modellierung der Ebenen des DV-Konzepts und der Implementierung. In der Phase des **Fachkonzepts** werden u.A. *Produktmodelle* und *Leistungsbäume* für die Darstellung verwendet.

- **Steuerungssicht:**

Im Rahmen der Steuerungs- oder Prozesssicht werden die Zusammenhänge zwischen den Elementen der einzelnen Sichten hergestellt. Im Unterschied zu den vorwiegend statischen Beschreibungen der übrigen Sichten, beschreibt sie den ablaufbezogenen, zeitlich-logischen Zusammenhang zwischen den Funktionen.

Für die Modellierung des **Fachkonzepts** werden u.a. die erweiterte ereignisgesteuerte Prozesskette (eEPK), Wertschöpfungsketten- und Funktionszuordnungsdiagramme verwendet. Mit *Wertschöpfungskettendiagrammen* werden jene Aktivitäten dargestellt, welche direkt an der Wertschöpfung des Unternehmens beteiligt sind. Sie stellen also die Prozesse der oberen Unternehmensebene dar und eignen sich daher als Überblicksmodelle. Die *erweiterte ereignisgesteuerte Prozesskette (eEPK)* bildet den Ablauf von Prozessen ab und setzt die Funktionen mit ihren Aufgabenträgern, Input- und Output-Objekten und anderen Elementen in Beziehung (für eine detaillierte Darstellung vgl. Kapitel 2.3). Um die Übersichtlichkeit zu gewährleisten, ist es möglich die, mit den Funktionen in Beziehung stehenden Elemente in einem eigenen Modelltyp, dem *Funktionszuordnungsdiagramm*, zu hinterlegen.

Die Kernbereiche dieser Arbeit setzen auf den Ergebnissen des Fachkonzepts der Steuerungssicht auf. Die Grundlage für die Analysen dieser Arbeiten bilden hierbei die Modelle der ereignisgesteuerten Prozesskette, deren Funktionen und Ereignisse für die Ontologie aufbereitet werden. Aufgrund der besonderen Relevanz wird daher die EPK-Methodik im folgenden Unterkapitel näher betrachtet.

## 2.3. Die Ereignisgesteuerte Prozesskette (EPK)

Ereignisgesteuerte Prozessketten (EPK) sind Teil des Fachkonzepts der Steuerungssicht. Im Unterschied zu den übrigen Sichten des ARIS-Konzepts bildet sie die dynamische Sicht, indem sie den ablaufbezogenen Zusammenhang zwischen den Funktionen abbildet [KNS92]. Grundlage für die EPK-Methodik bildet die Petri-Netz-Theorie, welche um logische Verknüpfungsoperatoren erweitert wurde [SNZ95]. EPKs ermöglichen die Beschreibung von Geschäftsprozessen in grafischer und intuitiv verständlicher Form. Der Schwerpunkt liegt hierbei primär auf der Darstellung betriebswirtschaftlicher Sachverhalte und weniger auf Ebene einer formalen Spezifikation [Aal99].

EPKs können den semi-formalen Modellierungssprachen zugeordnet werden. Die Menge der Symbole zur Darstellung von Prozessen ist zwar definiert, jedoch fehlt eine formale Beschreibung der Syntax und Semantik (für Ansätze zur Formalisierung der EPK vgl. u.A. Nüttgens und Keller [NR02], van der Aalst [Aal99] und Kindler [Kin03]).

### 2.3.1. Modellierungskonstrukte

Eine EPK ist ein gerichteter, bipartiter Graph, welcher sich in seiner grundlegenden Form aus Funktionen, Ereignissen und logischen Operatoren zusammensetzt, welche mittels einer gerichteten Kante, dem sogenannten Kontrollfluss verbunden werden. Bipartit bedeutet in diesem Zusammenhang, dass abwechselnd Funktionen und Ereignisse, gegebenenfalls verknüpft durch logische Operatoren, modelliert werden müssen [RSD05, S. 65ff]. Da diese grundlegenden Elemente im Rahmen dieser Arbeit verarbeitet werden, werden sie in den folgenden Absätzen kurz näher beschrieben:

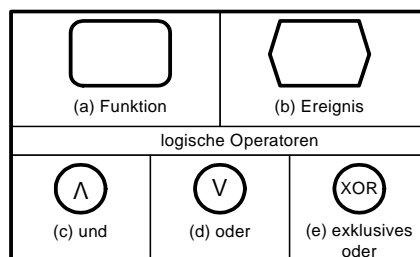


Abbildung 2.7.: Grundelemente der EPK-Notation

#### Funktion

Die Funktion bildet den Kern eines EPK-Modells. Mit ihr werden die im Rahmen eines Prozesses durchzuführenden Tätigkeiten abgebildet, welche Input- in Outputdaten transformieren [RSD05, S. 65]. Keller et al. [KNS92] beschreiben eine Funktion

als betriebswirtschaftlichen Vorgang und bezeichnen sie als aktive Komponente des Prozessmodells, welche Entscheidungskompetenz über nachfolgende Funktionen besitzt.

Funktionen werden im Rahmen der Funktionssicht erfasst und können auf unterschiedlichen Abstraktionsniveaus modelliert werden. Scheer [Sch98b, S. 25] unterscheidet in diesem Zusammenhang die Ebenen Funktionsbündel, Funktion, Teilfunktion und Elementarfunktion, wobei eine Elementarfunktion als nicht weiter sinnvoll zerlegbare Tätigkeit betrachtet wird [Gad05, S. 38].

Funktionen werden in der gängigsten EPK-Notation als Rechteck mit abgerundeten Kanten dargestellt (vgl. Abb. 2.7.a) und besitzen genau eine eingehende und eine ausgehende Kante [NR02].

### Ereignis

Ein Ereignis beschreibt eine ablaufrelevante Zustandsausprägung eines Informationsobjekts, welches weder Zeit noch Ressourcen verbraucht [RSD05, S. 65ff]. Ereignisse bilden die Bedingung für die Durchführung und sind das Ergebnis von Funktionen [Sta06, S. 62f]. Sie werden von Keller et al. [KNS92] als passive Komponenten betrachtet, welchen die Entscheidungskompetenz über nachfolgende Funktionen fehlt.

Ereignisse werden aufgrund des engen Bezugs zu den Informationsobjekten im Rahmen der Datensicht erhoben. Sie können ebenfalls auf unterschiedlichen Abstraktionsniveaus modelliert werden, wobei sich der Grad der Abstraktion an jener der Funktionen orientieren muss [Sta06, S. 63].

Ereignisse werden als Sechseck dargestellt (vgl. Abb. 2.7.b) und besitzen genau eine eingehende und/oder eine ausgehende Kante. Besitzt es lediglich eine Kante handelt es sich um ein Start- bzw. ein Endereignis [NR02]. Um sicherzustellen, dass die Anfangs- und Endbedingungen eines Prozesses klar definiert sind, startet und endet ein jedes EPK-Modell mit einem oder mehreren Ereignissen [Ros96, S. 64ff].

Wie bereits dargestellt sind EPK-Modelle streng genommen bipartite Graphen. Aus Gründen der Übersichtlichkeit werden allerdings häufig die sogenannten Trivialereignisse weggelassen. Becker et al. definieren ein Trivialereignis als "*... rein transitorische Ereignisse in EPKs, die für das Verständnis des Prozesses nicht notwendig sind*" [BDKK02, S. 68]. Ähnlich Hüselmann [Hüs03, S. 159], welcher Trivialereignisse als Ereignisse beschreibt, welche keinen informativen Mehrwert enthalten. Als Beispiel kann in diesem Zusammenhang das Trivialereignis "*Project Planned*" als Ergebnis der Funktion "*Plan Project*" betrachtet werden.

### Logische Operatoren

Funktionen und Ereignisse können mittels logischer Operatoren in Beziehung gesetzt werden, um parallel durchzuführende oder alternative Prozesspfade abbilden zu können. Für die Darstellung solcher nicht-linearen Prozessverläufe werden der

AND- (konjunktive Verknüpfung), der OR- (adjunktive Verknüpfung) und der XOR-Operator (disjunktive Verknüpfung) unterschieden. Diese Operatoren beschreiben, wie die einzelnen Elemente miteinander verknüpft werden [KNS92]:

- **konjunktive Verknüpfung:** die Gesamtaussage zweier Aussagen ist wahr, wenn beide gleichzeitig wahr sind.
- **adjunktive Verknüpfung:** die Gesamtaussage zweier Aussagen ist wahr, wenn mindestens eine der beiden wahr ist.
- **disjunktive Verknüpfung:** die Gesamtaussage zweier Aussagen ist wahr, wenn genau eine der beiden wahr ist.

Dargestellt werden die logischen Operatoren durch einen Kreis mit einem entsprechenden Symbol (vgl. Abb. 2.7.c - e). Sie haben entweder eine eingehende und mehrere ausgehende Kanten (Ausgangsverknüpfung oder Split-Operator) oder mehrere eingehende und eine ausgehende Kante (Eingangsverknüpfung oder Join-Operator) [NR02]. In diesem Zusammenhang ist zu beachten, dass lediglich Funktionen mit Funktionen und Ereignisse mit Ereignissen verknüpft werden dürfen [Sta06, S. 67]. Während die Verknüpfung mehrerer Ereignisse mit einer Funktion *Ereignisverknüpfung* genannt wird, wird die Verknüpfung mehrere Funktionen mit einem Ereignis als *Funktionsverknüpfung* bezeichnet [RSD05, S. 64ff].

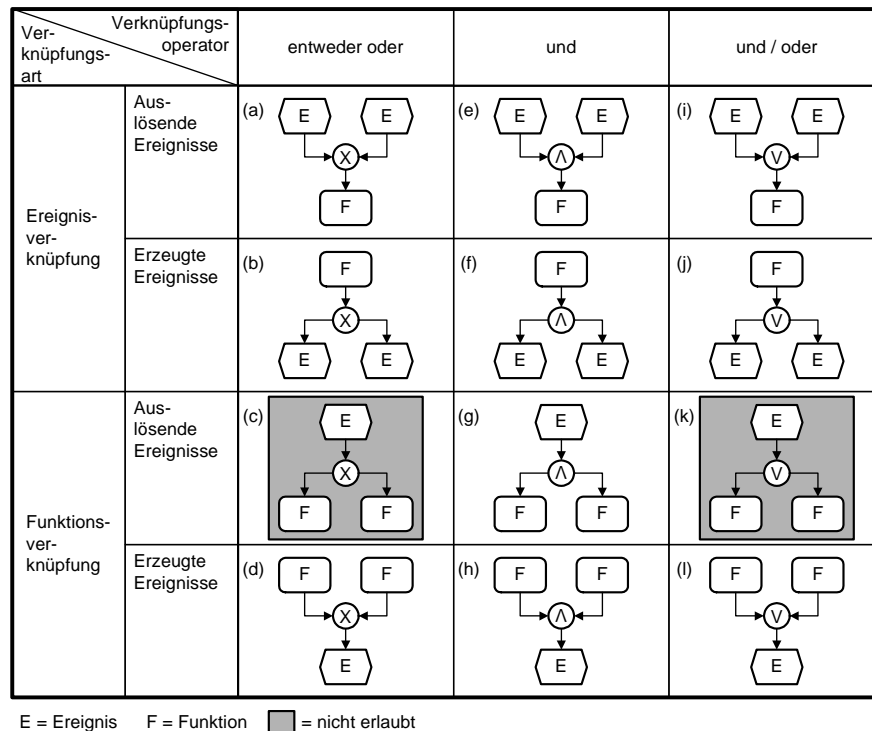


Abbildung 2.8.: Verknüpfungsarten (vgl. [KNS92])

In Abbildung 2.8 werden die möglichen Kombinationen zusammengefasst. Da diese weitestgehend selbsterklärend sind, wird auf eine Beschreibung der einzelnen Fälle an dieser Stelle verzichtet. Es sei lediglich angemerkt, dass die beiden grau hinterlegten Fälle nicht erlaubt sind, da einem Ereignis die nötige Entscheidungskompetenz für den weiteren Verlauf des Kontrollflusses fehlt [RSD05, S. 64ff].

### EPK vs. eEPK

Werden lediglich die in den vorangegangenen Abschnitten vorgestellten, grundlegenden Modellelemente verwendet, spricht man von einer einfachen oder schlanken EPK. Für eine vollständige Darstellung eines Prozesses reichen diese Konstrukte allerdings nicht aus. Daher kann die Notation um Elemente erweitert werden, um bspw. Organisationseinheiten oder Informationsobjekte abbilden zu können [Sei06, S. 21]. In diesem Fall wird von einer erweiterten EPK (eEPK) gesprochen. Da diese Elemente im Rahmen dieser Arbeit noch nicht aufbereitet werden, werden sie an dieser Stelle lediglich kurz vorgestellt:

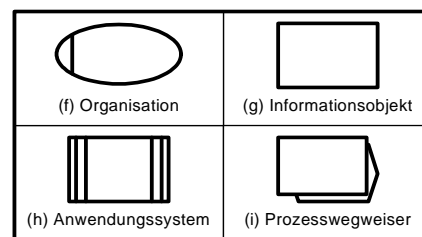


Abbildung 2.9.: Erweiterung der EPK-Notation

- **Organisationseinheit** (vgl. Abb. 2.9.a):  
Die im Rahmen der Organisationssicht erhobenen Organisationseinheiten werden mit den Funktionen in Beziehung gesetzt, um die für die Durchführung verantwortliche Stelle darzustellen. Die Zuordnung erfolgt hierbei mittels einer ungerichteten Kante [Sta06, S. 63f].
- **Informationsobjekt** (vgl. Abb. 2.9.b):  
Funktionen benötigen und erzeugen bei ihrer Durchführung Daten. Diese werden durch Informationsobjekte dargestellt, welche im Rahmen der Datensicht erhoben werden. Sie werden mittels einer gerichteten Kante mit der Funktion verknüpft, um Input- und Output-Beziehungen darzustellen [Sta06, S. 64f].
- **Anwendungssystem** (vgl. Abb. 2.9.c):  
Wird eine Funktion automatisiert durchgeführt, ist es möglich, das für die Bearbeitung zuständige Anwendungssystem anzugeben [RSD05, S. 68]. Die Anwendungssysteme werden in der Funktionssicht spezifiziert und mittels einer ungerichteten Kante mit der Funktion verbunden.
- **Prozesswegweiser** (vgl. Abb. 2.9.d):  
Mittels eines Prozesswegweisers ist es möglich verschiedene (Teil-)Prozessketten



zu verknüpfen. Er stellt eine explizite Schnittstelle zu vor- bzw. nachgelagerten Prozessen dar, welcher vor allem bei umfangreichen Prozessen der Navigation und der Übersichtlichkeit dient. Ein Prozesswegweiser besitzt genau eine eingehende oder eine ausgehende Kante und kann lediglich mit einem Ereignis verknüpft werden [NR02].

Abbildung 2.10 skizziert die Verknüpfung der erweiterten Modellelemente mit einer Funktion. Um die Übersichtlichkeit bei umfangreichen Prozessmodellen zu gewährleisten, ist es möglich, diese Beziehungen in einem eigenen Modell, dem sogenannten *Funktionszuordnungsdiagramm* zu hinterlegen. Dadurch kann der eigentliche Prozess mittels einer "schlanken" EPK modelliert werden [Sei06, S. 21f].

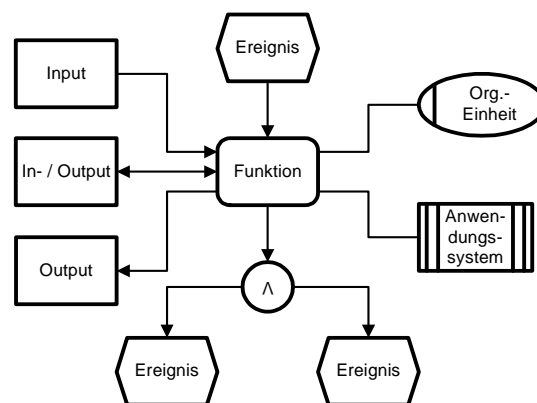


Abbildung 2.10.: Schematische Darstellung der eEPK

### 2.3.2. Namenskonventionen

Namenskonventionen verfolgen die Zielsetzung, Bezeichnungen von Modellelementen zu standardisieren, um die Heterogenität bei der Modellierung zu vermeiden. Sie bilden die Grundlage für eine einheitliche Benennung innerhalb einer Modellwelt und schränken somit den Gestaltungs- und Interpretationsspielraum bei der Modellierung ein (vgl. [Ros96, S. 187ff]).

Schütte [Sch98c, S. 189ff] identifiziert die beiden Informationsobjekttypen Objekt und Aktivität als Kern sämtlicher Informationsmodelle. Seine Empfehlung für die Modellierung von Funktionen und Ereignissen basiert hierbei auf einer durchgängigen Verwendung von Substantiven und Verben. Objekte beschreiben reale Gebilde der Wirklichkeit und werden aus einem oder mehreren Substantiv(en) gebildet (z.B. Projektleiter, Kundenauftrag, Geschäftspartner, Lieferant, etc.). Aktivitäten bzw. Funktionen zeichnen sich durch ihren aktiven Charakter aus und beschreiben Verrichtungen an Objekten. Aus diesem Grund setzt sich die Bezeichnung einer Funktion aus einem Objekt und einem aktiven Verb zusammen (z.B. *Projektleiter ernennen*). Ereignisse hingegen beschreiben den Zustand, in welchem sich das Objekt zu

einem bestimmten Zeitpunkt befindet (z.B. *Projektleiter ernannt*). Ihre Bezeichnung setzt sich daher aus einem Objekt und einem Verb im Partizip Perfekt zusammen. Tabelle 2.1 fasst die eben vorgestellten Konventionen nochmals zusammen.

	Funktion	Ereignis
<b>Deutsch</b>	Substantiv(e) + Verb Bsp.: <i>Projektleiter ernennen</i>	Substantiv(e) + Verb im Partizip Perfect Bsp.: <i>Projektleiter ernannt</i>
<b>Englisch</b>	Verb + Substantiv(e) Bsp.: <i>Nominate Project Manager</i>	Substantiv(e) + Verb in Past Participle Bsp.: <i>Project Manager nominated</i>

Tabelle 2.1.: Namenskonvention für Funktionen und Ereignisse

Da die im Rahmen dieser Arbeit verwendeten Testmodelle in englischer Sprache vorliegen, mussten die Namenskonventionen der englischen Grammatik entsprechend angepasst werden (vgl. Tabelle 2.1). Im Englischen setzen sich daher Funktionen aus einem aktiven Verb und einem Objekt (z.B.: *Nominate Project Manager*) und Ereignisse aus einem Objekt und einem Verb im Past Participle (z.B.: *Project Manager Nominated*) zusammen.

### 2.3.3. Erweiterung um Prozessobjekte

Bereits bei der Darstellung der Namenskonventionen wurde die Beziehung zwischen Prozessen und Objekten angedeutet. Dieser Bezug wird in einer Reihe von Prozessdefinitionen konkret aufgegriffen. So beschreibt bspw. Nippa einen Prozess als "Abfolge von Bearbeitungsschritten [...], die an einem oder mehreren Objekten vollzogen werden" [Nip95, S. 50f]. Auch Rosemann zentriert den Objektbegriff und definiert einen Prozess als "die inhaltlich abgeschlossene, zeitliche und sachlogische Abfolge von Funktionen [...], die zur Bearbeitung eines betriebswirtschaftlich relevanten Objekts ausgeführt werden" [Ros96, S. 9]. Dieses Objekt bezeichnet Rosemann als prozessprägendes Prozessobjekt, da es durch seine Zustandsausprägung die folgenden Bearbeitungsschritte bestimmt. Aufgrund der besonderen Relevanz fordert Rosemann eine explizite Modellierung des Prozessobjekts und leitet daraus eine Prozessobjektmodell ab, welches in den folgenden Absätzen nun kurz beschrieben wird [Ros96, S. 9f bzw. S. 76ff].

Die Modellierung des Prozessobjekts erfolgt als eigenständiges Informationsobjekt, welches als gestrichelte Ellipse dargestellt und bei seinem ersten Auftreten im Modell mit einem Ereignis in Beziehung gesetzt wird (vgl. Abb. 2.11). Neben der expliziten Annotation der Prozessobjekte in den Modellen, beschreibt Rosemann ein sogenanntes Prozessobjektmodell, in welchem die verschiedenen Prozessobjekte mittels Generalisierungs- und Kompositionshierarchien, sowie mittels Migrationsbeziehungen miteinander verknüpft werden. Die beiden Hierarchisierungen ergeben

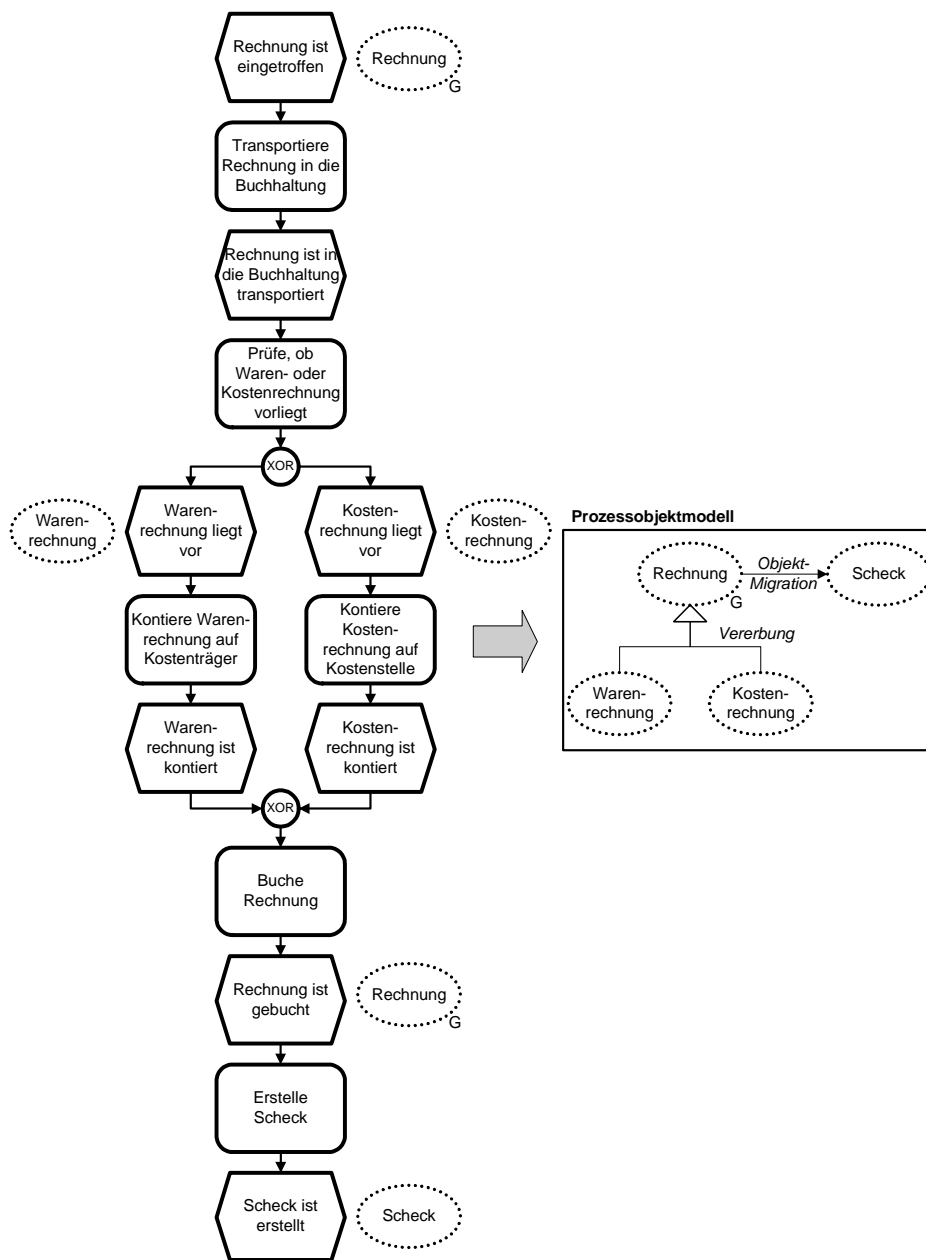


Abbildung 2.11.: Ableitung eines Prozessobjektmodells (vgl. [Ros96, S. 81])

sich im Regelfall, wenn sich der Prozessverlauf in mehrere Teilprozessketten aufspaltet (siehe bspw. die Generalisierungshierarchie für *Rechnung* in Abb. 2.11).

Abbildung 2.12 fasst die verschiedenen Beziehungstypen nochmals zusammen, welche an dieser Stelle kurz einzeln betrachtet werden:

- Die **Generalisierung** (vgl. Abb. 2.12.a) ist durch die Is-a Beziehung der Vererbung gekennzeichnet. Die in der Abbildung dargestellte Beziehung zwischen *Rechnung* und *inländischer Rechnung* besagt, dass sämtliche Prozessreferen-

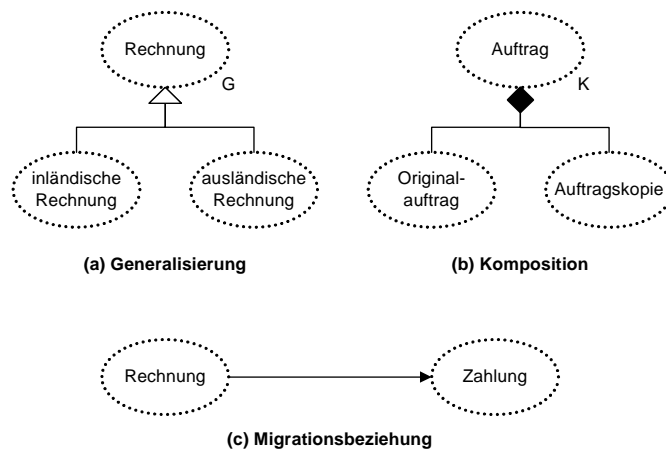


Abbildung 2.12.: Beziehungen zwischen Prozessobjekten (vgl. [Ros96, S. 76ff])

zen der *Rechnung* an die *inländische Rechnung* vererbt werden, da letztere dieselben Prozessschritte durchlaufen muss, wie jene der übergeordneten Instanz. Weiters kann die *inländische Rechnung* um spezifische Prozessreferenzen erweitert werden.

- Der **Komposition** (vgl. Abb. 2.12.b) liegt eine Part-of Beziehung zugrunde und beschreibt somit die Beziehung zwischen Objekten und deren Teilobjekten. Als häufiges Beispiel nennt Rosemann verschiedene Versionen eines Objekts.
- Die **Prozessobjektmigration** (vgl. Abb. 2.12.c) veranschaulicht den Wechsel von Prozessobjekten entlang eines Prozessverlaufs. So folgt bspw. auf das Prozessobjekt *Rechnung* in der Regel die fällige *Zahlung*.

## 2.4. Semantische Prozessmodellierung

Geschäftsprozesse können aus geschäftlicher und technischer Perspektive betrachtet werden. Daraus leitet sich eine wesentliche Aufgabe von *Business Process Management (BPM)* ab, nämlich zwischen diesen beiden Ebenen zu vermitteln, d.h. als Übersetzer zwischen diesen zu fungieren. In Unternehmen ist eine Menge an Prozesswissen, bspw. in Form von Prozessmodellen oder in Codefragmenten, gespeichert, welches allerdings nicht automatisiert verarbeitet werden kann. Daraus folgt die Problematik, dass sowohl bei der Analyse bestimmter Prozesse als auch bei der Implementierung neuer bzw. der Anpassung bestehender ein nicht unerheblicher manueller Aufwand investiert werden muss (vgl. Abb. 2.13.a) [HLD<sup>+</sup>05] [HR07].

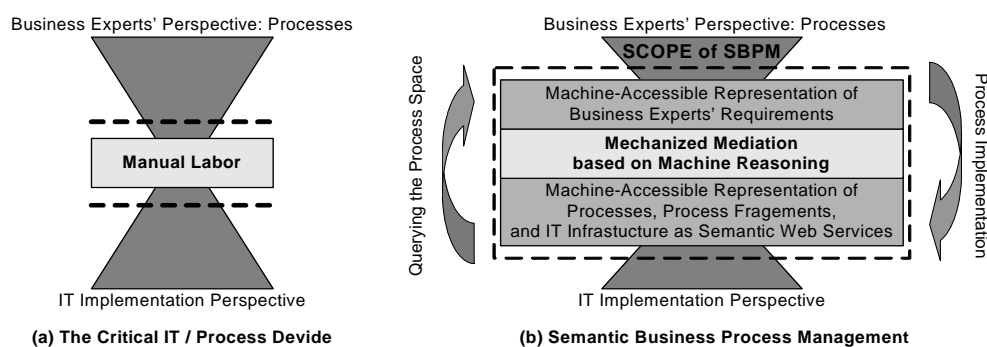


Abbildung 2.13.: Gegenüberstellung "Kritische IT/Prozess Trennung" & "Semantic Business Process Management" [HLD<sup>+</sup>05]

Um den Grad der Automatisierung bei der Übersetzung zwischen den beiden Perspektiven zu erhöhen, schlagen Hepp et al. [HLD<sup>+</sup>05] vor, *BPM* mit Technologien aus dem Bereich des *Semantic Web*, wie bspw. Ontologien, zu kombinieren, da diese geeignete Methoden und Werkzeuge für die maschinen-interpretierbare Repräsentation und Manipulation von Wissen bereitstellen. Die grundlegende Idee hinter diesem, als *Semantic Business Process Management (SBPM)* bezeichneten Ansatz liegt darin, sowohl die geschäftliche als auch die technische Betrachtungsweise von Geschäftsprozessen mittels Ontologien zu beschreiben, während Techniken des maschinellen Reasonings die automatische bzw. semi-automatische Vermittlung zwischen diesen beiden übernehmen (vgl. Abb. 2.13.b).

Die Zielsetzung von *SBPM* liegt also darin, durch den Einsatz semantischer Technologien die Automatisierung in allen Phasen des *BPM Life Cycles* zu erhöhen. In diesem Zusammenhang beschreiben Fantini et al. [FSM<sup>+</sup>07] und Wetzstein et al. [WMF<sup>+</sup>07] den sogenannten *SBPM Life Cycle* und heben zusätzliche Funktionalitäten und deren Nutzen hervor. Abbildung 2.14 zeigt den *SBPM Life Cycle* mit den Phasen *SBP (Semantic Business Process) Modeling*, *SBP Implementation*, *SBP Execution* und *SBP Analysis*.

Die Phase *SBP Modeling* erweitert die herkömmliche Prozessmodellierung um

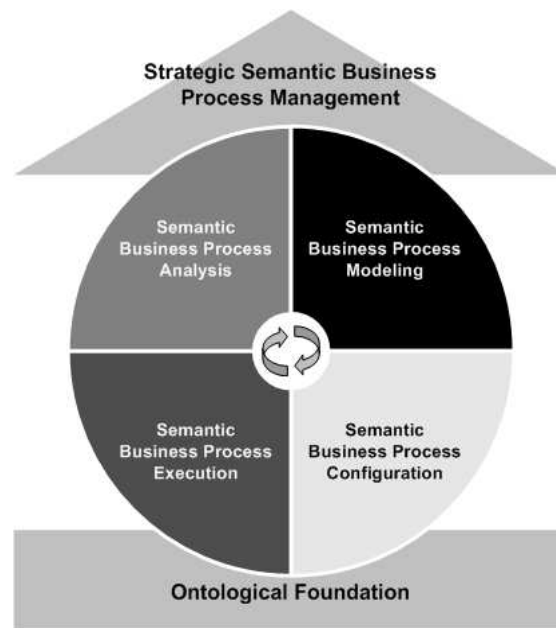


Abbildung 2.14.: Semantic Business Process Management Life Cycle [FSM<sup>+</sup>07]

die semantische Annotierung von Prozessmodellen mittels Ontologiekonzepten. Das Ziel dieser Annotierung ist die explizite Spezifikation der Semantik von Prozesselementen und Entscheidungen im Kontrollfluss in maschinen-interpretierbarer Form, welche die Grundlage für sämtliche semantik-basierten Funktionalitäten des *SBPM Life Cycle* bildet [WMF<sup>+</sup>07]. Im Rahmen der semantischen Annotierung von Prozessmodellen können zwei Ebenen der Annotierung unterschieden werden [LBS08]:

- **metamodel-level annotation** dient der semantischen Beschreibung der Elemente von Metamodellen. Mit der zugrunde liegenden Ontologie wird hierbei versucht, die Modellierungskonstrukte unterschiedlicher Modellierungssprachen auf einen gemeinsamen Nenner zu bringen, um diese vergleichbar zu machen und den Austausch von Prozessmodellen zwischen diesen zu ermöglichen.
- **model-level annotation** dient der semantischen Annotierung von Modellelementen konkreter Prozessmodelle. Die Zielsetzung in diesem Bereich liegt darin, die Bedeutung der natürlich-sprachlichen Bezeichnungen zu präzisieren und durch ein standardisiertes Vokabular zu vereinheitlichen.

Als grundlegende Zielsetzung der semantischen Annotierung von Prozessmodellen kann die Schaffung eines gemeinsamen Verständnisses aller Beteiligten über die Geschäftsprozesse betrachtet werden. Darüber hinaus bietet die maschinen-interpretierbare Formalisierung dieser Annotierungen eine Reihe weiterer Vorteile [LBS08]:

- erweitertes Suchen in Prozessmodellen,
- verbesserte Validierung von Prozessmodellen,

- automatisierte Prozessausführung,
- bessere Wiederverwendung von Prozessfragmenten,
- Austausch von Prozessfragmenten,
- Unterstützung der Integration verschiedener Abteilungen und Unternehmen,
- automatisierte Erstellung, Anpassung und Vervollständigung von Prozessmodellen,
- Modellierung von B2B Szenarien.

### 2.4.1. Ontologien für Informationssysteme

Der Begriff Ontologie stammt ursprünglich aus der Philosophie und bedeutet die "Lehre vom Sein" [Hes02]. Sie bildet einen Zweig der Metaphysik und beschäftigt sich mit der systematischen Erforschung alles Existierenden, d.h. mit der Natur und der Organisation der Realität [GG95]. Ontologie setzt sich mit den grundlegenden Fragen "*What is being?*" und "*What kinds of things are there?*" auseinander. Basierend auf diesen Fragestellungen haben Philosophen, wie bspw. Aristoteles oder Emmanuel Kant (vgl. [GPCFL04, S. 3ff]), versucht die Dinge, welche in der realen Welt existieren, mittels genereller, ontologischer Kategorien zu klassifizieren und deren allgemeinen Eigenschaften zu identifizieren [GHA07, S. 68f].

Im Bereich der Computerwissenschaften wurde der Begriff Ontologie übernommen, um ein technisches Artefakt zur Beschreibung eines Ausschnitts aus der Realität mittels Konzepten und Beziehungen zwischen diesen zu bezeichnen. Diese Definitionen werden um Annahmen ergänzt, welche die beabsichtigte Bedeutung dieses speziellen Vokabulars explizit definieren [Gua98]. Die Aufgabe einer Ontologie ist die Formalisierung eines bestimmten Wissensbereichs in Form einer Terminologie, welche einerseits von Maschinen interpretiert werden kann und andererseits von einer Gruppe von Individuen als allgemein gültig anerkannt wird, um ein gemeinsames Verständnis über eine bestimmte Domäne zu erlangen. Eine Ontologie definiert also ein gemeinsames Vokabular, welches die grundlegenden Konzepte und deren Beziehungen, sowie domänenspezifische Axiome festlegt, wobei letztere der Spezifikation der beabsichtigten Bedeutung und der Gewährleistung einer klar definierten Verwendung der bereitgestellten Begriffe dienen. Eine Ontologie wird daher nicht für die Beschreibung von Wissen an sich verwendet, sondern spezifiziert die Struktur und die Semantik von Wissen.

Für Ontologien im informationstechnischen Sinne sind in der Literatur eine Reihe von Definitionen zu finden. Eine der gängigsten Definitionen (vgl. u.a. [Dit07, S. 29], [GDD06, S. 46], [LM01], [HSW97, S. 10]) stellt jene von Gruber dar, welcher Ontologie auf Basis des, von Genesereth und Nilsson beschriebenen Konzepts der Konzeptualisierung (vgl. [GN87, S. 9ff]), wie folgt definiert:

”An ontology is an explicit specification of a conceptualization” [Gru93a][Gru93b].

Basierend auf den Definitionen von Gruber und Borst, welcher Ontologie als ”... formal specification of a shared conceptualization” [Bor97, S. 12] definiert, definieren Studer et al. den Begriff sehr ausführlich, indem sie die wesentlichen Charakteristika einer Ontologie in einer Definition zusammenfassen:

”An ontology is a formal, explicit specification of a shared conceptualization” [SBF98, S. 25].

Bei genauerer Betrachtung dieser Definition können die folgenden wesentlichen Charakteristika von Ontologien identifiziert werden (vgl. u.A. auch [Fen04, S. 3] [GHA07, S. 69f], [SBF98, S. 25]):

- **conceptualization:** Darunter wird eine vereinfachte, abstrakte Darstellung eines Ausschnitts aus der realen Welt verstanden, indem dieser mittels Konzepten, Objekten und Beziehungen beschrieben wird (siehe auch [Gru93a], [Gru93b] und [GDD06, S. 46]). Die Beschreibung der Konzepte und Beziehungen erfolgt hierbei auf möglichst generelle Weise, damit möglichst viele Situationen, d.h. reale Ausprägungen, des zu untersuchenden Bereiches erfasst werden können.
- **specification:** Gasevic et al. [GDD06, S. 46] betrachten eine Spezifikation im Kontext dieser Definition als eine formale, deklarative Beschreibung. Aufgrund der Tatsache, dass Ontologien von Maschinen interpretierbar sein sollen, erfolgt die Darstellung mittels einer formalen Beschreibungssprache.
- **explicit:** Darunter wird die explizite Definition der von einer Ontologie zur Verfügung gestellten Begriffe verstanden. Ausdrücke hingegen, welche nicht explizit spezifiziert werden, sind nicht Teil der Konzeptualisierung und somit dem System nicht zugänglich.
- **formal:** Darunter wird einerseits die Maschineninterpretierbarkeit der von ihr bereitgestellten Begriffe und andererseits die eindeutige Interpretierbarkeit ihrer Semantik verstanden.
- **shared:** Eigenschaft einer Ontologie, allgemein akzeptiertes Wissen bereitzustellen, d.h. das in einer Ontologie spezifizierte Wissen wird nicht von einer einzelnen Person verwendet, sondern wird von einer Gruppe von Individuen anerkannt.

Die in Grimm et al. [GHA07, S. 69f] angeführte Definition geht sogar einen Schritt weiter, indem sie diese Definition um die Eigenschaft ”... of a domain of interest” erweitern. Diese Erweiterung hebt hervor, dass sich die Modelle, welche mittels Ontologien beschrieben werden, in der Regel auf einen bestimmten Bereich der realen Welt, d.h. eine bestimmte Domäne, beschränken.



Aus technischer Sicht bestehen Ontologien aus *Konzepten*, *Beziehungen*, *Axiomen* und *Instanzen* (vgl. u.A. [CFLGP06, S. 5f], [GHA07, S. 70f], [Gru93a]), deren Bezeichnungen sich allerdings in Abhängigkeit vom verwendeten Formalismus zur Darstellung der Ontologie unterscheiden können:

- **Konzepte** werden häufig auch als Klassen bezeichnet und bilden eine abstrakte Sichtweise auf Phänomene der realen Welt. Sie werden typischerweise mittels Beziehungen verknüpft und durch Attribute beschrieben.
- **Beziehungen** oder Relationen werden für die Spezifikation von Zusammenhängen zwischen Konzepten verwendet, wobei in Ontologien üblicherweise binäre Beziehungen eingesetzt werden. In manchen Formalismen zur Darstellung von Ontologien werden Beziehungen auch für die Definition der Eigenschaften von Konzepten verwendet.
- **Axiome** dienen der Spezifikation von Einschränkungen für die korrekte Verwendung und die Interpretation von Konzepten und Relationen. Mit ihnen werden allgemeingültige Sätze, d.h. Aussagen welche in der Domäne immer wahr sind, formuliert.
- **Instanzen** beschreiben konkrete Ausprägungen von Konzepten und Relationen.

Ontologien finden in unterschiedlichen Bereichen Anwendung, wobei in der Literatur verschiedene Arten von Ontologien beschrieben werden, um den spezifischen Anforderungen gerecht zu werden. In diesem Zusammenhang werden in den folgenden Absätzen drei gängige Kategorisierungen vorgestellt, welche Ontologien anhand der Verwendeten Konstrukte, ihrer formalen Ausdrucksstärke bzw. ihrer Allgemeingültigkeit klassifizieren.

Ein grobe Unterscheidung von Ontologien kann auf Grundlage der zur Modellierung verwendeten Konstrukte vorgenommen werden. In diesem Zusammenhang werden leichtgewichtige (*lightweight ontologies*) und schwergewichtige Ontologien (*heavyweight ontologies*) unterschieden (vgl. bspw. [GPCFL04, S. 8]):

- **lightweight ontologies:** Ontologien, mit welchen lediglich Konzepte, Taxonomien, Beziehungen zwischen Konzepten und Eigenschaften, welche Konzepte beschreiben, erfasst werden.
- **heavyweight ontologies:** Erweiterung von leichtgewichtigen Ontologien um Axiome und Einschränkungen, welche die beabsichtigte Bedeutung der Definitionen verdeutlichen sollen. Hierdurch wird der Schritt von einer syntaktisch korrekten Anwendung der Konzepte zu einer semantisch korrekten Verwendung ermöglicht [Dit07, S. 75f].

Lassila und McGuinness nehmen eine Klassifikation von Ontologien vor, welche auf der Mächtigkeit ihrer internen Struktur zur Darstellung der Informationen

basiert. Sie bilden diese in Form einer linearen Skala ab (vgl. Abb. 2.15) und unterscheiden die folgenden Kategorien [LM01] (vgl. auch [GPCFL04, S. 26ff]):

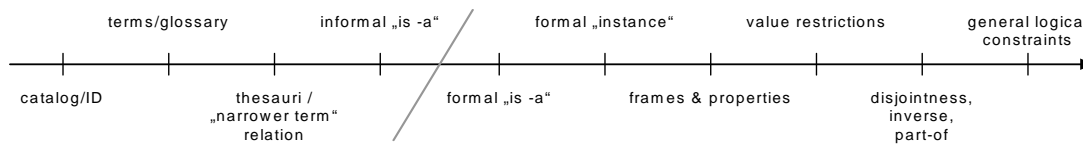


Abbildung 2.15.: Ontologie Skala nach [LM01]

- **catalog** für die Aufbereitung eines kontrollierten Vokabulars, d.h. einer abgeschlossenen Menge von Begriffen, deren eindeutige Verwendung durch IDs sichergestellt wird.
- **glossary** verwaltet Listen von Begriffen, welchen Bedeutungen in natürlicher Sprache zugewiesen werden.
- **thesauri** stellen semantische Beziehungen, wie bspw. Synonym-Beziehungen, zwischen Begriffen her, wobei allerdings die Möglichkeit einer expliziten Definition von Hierarchien fehlt.
- **informal "is-a"** stellt Generalisierungen und Spezialisierungen dar, welche keiner strikten "is-a" Hierarchisierung unterworfen sind. So kann bspw. durch die Zuweisung der Begriffe "Autoverleih" und "Hotel" zum Überbegriff "Reise" ausgedrückt werden, dass beide Teil einer "Reise" sein können (vgl. [GPCFL04, S. 28]).
- **formal "is-a"** zeichnet sich durch strikte Unterklassen Beziehungen aus, indem sie die Möglichkeiten der Vererbung ausnutzen.
- **formal "instance"** erweitert die Hierarchisierungen der vorhergehenden Kategorie um grundlegende Instanzen.
- **frames & properties** ermöglichen die Zuweisung von Eigenschaften zu Klassen. Diese können innerhalb von Hierarchien vererbt werden.
- **value restrictions** Erweiterung um eine Einschränkung der Wertebereiche der Eigenschaften von Klassen.
- **general logical constraints** die letzte Kategorie bildet die ausdrucksstärkste Art von Ontologien. Mit Hilfe von logischen Ausdrücken können weitere Einschränkungen für die Verwendung von Begriffen definiert werden.

Eine gängige Art der Klassifikation ist es, Ontologien anhand ihrer Generalität (Allgemeingültigkeit) zu unterscheiden. In diesem Zusammenhang unterscheiden van Heijst et al. [HSW97] unter Berücksichtigung des Gegenstands der Konzeptualisierung die Kategorien *Representation Ontologies*, *Generic Ontologies*, *Domain Ontologies* und *Application Ontologies*.

Eine ähnliche Klassifizierung nimmt Guarino [Gua97] vor, welcher die Kategorien *Top-Level Ontologies*, *Domain Ontologies*, *Task Ontologies* und *Application Ontologies* unterscheidet und diese, wie in Abbildung 2.16 dargestellt, ihrem Abstraktionsniveau entsprechend, anordnet. Betrachtet man die Abbildung top-down werden die Ontologien immer spezifischer, wobei eine Ontologie die Konzepte und Beziehungen der darüberliegenden Ebene verwenden und verfeinern kann.

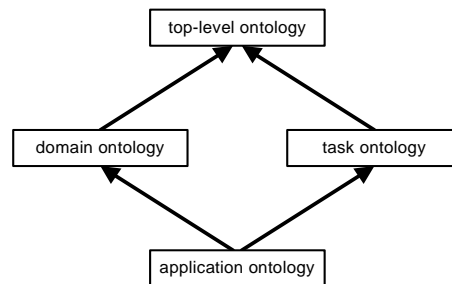


Abbildung 2.16.: Arten von Ontologien (vgl. [Gua97] und [Gua98])

Unter Berücksichtigung dieser beiden Kategorisierungen können zusammenfassend die folgenden Arten von Ontologien identifiziert werden (vgl. u.A. auch [Fen04, S. 5f], [GHA07, S. 61ff], [SBF98, S. 27]):

- **Representation Ontologies:**

Diese Kategorie von Ontologien stellt ein Rahmenwerk für die Darstellung von Wissen bereit. Sie machen hierbei keine Aussagen darüber, was in einer Ontologie spezifiziert wird, sondern wie die Repräsentation des Wissens zu erfolgen hat [HSW97]. Guarino [Gua97] bezeichnet diese Arten von Ontologien als Meta-Ontologien. Mit ihnen werden Modellierungsprimitiven beschrieben, welche von Wissensrepräsentationssprachen verwendet werden.

- **Top-Level Ontologies:**

Mit *Top-Level Ontologies* wird grundlegendes Wissen, welches unabhängig von einem bestimmten Anwendungsbereich ist, abgebildet. Die Konzepte und Beziehungen werden hierbei sehr abstrakt definiert und bilden die Grundlage für sämtliche darauf aufbauende Ontologien. Diese Kategorie entspricht den *Generic Ontologies* der Kategorisierung von van Heijst et al. [HSW97] und spezifiziert Konzepte wie bspw. Zustand, Ereignis oder Prozess.

- **Domain- & Task Ontologies:**

Ziel dieser Ontologien ist die Bereitstellung eines grundlegenden Vokabulars, mit welchem domänenspezifisches Wissen (*Domain Ontologies*) bzw. Wissen über Aufgaben und deren Problemlösung (*Task Ontologies*) erfasst werden kann. Da sie die Konzepte der Top-Level Ontologien verfeinern, wird ihr Geltungsbereich entsprechend eingeschränkt.

- **Application Ontologies:**

Mittels *Application Ontologies* wird jenes Vokabular spezifiziert, welches für

eine bestimmte Anwendung benötigt wird. Üblicherweise verfeinern diese, unter Berücksichtigung eines konkreten Anwendungskontexts, sowohl *Domain-* als auch *Task Ontologies*.

## 2.4.2. Ontologien für das Geschäftsprozessmanagement

Wie bereits in der Einleitung dieses Kapitel erwähnt, haben Ontologien in den letzten Jahren zunehmend Einzug in das Geschäftsprozessmanagement gehalten, um die Automatisierung in allen Phasen des Lebenszyklus von Geschäftsprozessen zu erhöhen.

Im Bereich der semantischen Prozessmodellierung werden in der Literatur eine Reihe von Ontologien mit unterschiedlichen Zielsetzungen beschrieben. Eine Auswahl an Ontologien aus diesem Bereich wird nun in den folgenden Absätzen beschrieben, wobei folgende Zielsetzungen unterschieden werden können:

- zur Organisation von Prozesswissen (*MIT Process Handbook*).
- zur Erfassung und Analyse zentraler Aspekte von Unternehmen (*TOVE Ontologies* und *Enterprise Ontology*).
- zur Überbrückung von Heterogenitäten von Modellierungskonstrukten unterschiedlicher Modellierungssprachen (*Process Specification Language (PSL)*, *Process Interchange Format (PIF)*, *Process, Organisation, Product and others (POP\*)*, Teile des *SUPER Projekts* und die *General Process Ontology GPO* des umfassenden Frameworks von Lin).
- zur expliziten Spezifikation der Semantik der Inhalte von Modellelementen (*Model Annotation*) des Ansatzes von Lin, der Ansatz von Thomas und Fellmann und der Ansatz von Born et al.).

### MIT Process Handbook

Das *MIT Process Handbook* stellt eine online Wissensbasis dar, welche der systematischen Organisation von Prozesswissen dient. Die Zielsetzung dieses Handbuchs liegt darin, Benutzern bei der Umgestaltung von Prozessen und der Entwicklung neuer Prozesse zu unterstützen sowie allgemein Wissen über organisatorische Praktiken zu vermitteln [MCL<sup>+</sup>99].

Prozesse können mittels verschiedener Einträge über Aktivitäten beschrieben werden. Ein solcher Eintrag besteht aus einer allgemeinen Beschreibung der Aktivität und deren Eigenschaften, wie bspw. das Datum der letzten Modifikation. Des Weiteren können Verweise zu Teilaktivitäten, zu zugehörigen Prozessen, Spezialisierungen, Generalisierungen und zu Prozessen, in welchen die Aktivität verwendet wird, definiert werden [HM03, S. 223ff].

### Toronto Virtual Enterprise (TOVE) Ontologies

Im Rahmen des *TOVE Projects* des *Enterprise Integration Laboratory (EIL)* der Universität von Toronto werden Ontologien für die Modellierung von Unternehmen entwickelt, welche als *TOVE Ontology* bezeichnet werden und deren Zielsetzung wie folgt zusammengefasst werden kann [FCF93]:

- Bereitstellung einer gemeinsamen, maschinen-interpretierbaren Terminologie,
- präzise und möglichst eindeutige Spezifikation der Semantik aller Begriffe,
- Definition von Axiomen zur automatisierten Beantwortung grundlegender Fragestellungen das Unternehmen betreffend,
- Darstellung von Begriffen und daraus aufgebauten Konzepten in graphischer Form

Die *TOVE Ontologies* umfassen eine Reihe von generischen Ontologien zur Beschreibung von Prozessen, Ressourcen, organisatorischen Strukturen und dergleichen. Zur Zeit stehen folgende Ontologien zur Verfügung:

- *Activity Ontology* dient der Darstellung von Prozessen und Operationen durch Konzepte wie Aktivitäten, Zustände und dergleichen, sowie Beziehungen zwischen diesen zur Darstellung von bspw. Transformationen (vgl. [GF94]).
- *Resource Ontology* dient der Beschreibung der Eigenschaften von Ressourcen, wie bspw. Maschinen, Rohstoffen, für die Durchführung von Aktivitäten benötigtes Kapital oder menschliche Fähigkeiten, etc. (vgl. [FFG94]).
- *Organization Ontology*: fokussiert die Beschreibung von organisatorischen Strukturen, Rollen, Befugnissen und Bevollmächtigungen (vgl. [FBGL98]).
- *Product & Requirements Ontology* ermöglicht die Beschreibung von Produkten sowie zur Spezifikation von Eigenschaften der zu erstellenden Produkte (vgl. [LFB96]).
- *Quality Ontology* spezifiziert Konzepte, welche für die Erstellung von Unternehmensmodellen für das Qualitätsmanagement benötigt werden (vgl. [KFG99]).
- *Cost Ontologie* dient der Bereitstellung einer gemeinsame Terminologie für relevante Kostendaten (vgl. [TFG94]).

### Enterprise Ontology

Die *Enterprise Ontology* wurde im Rahmen des *Enterprise Projects* an der Universität von Edinburgh entwickelt. Die Zielsetzung des Projekts lag darin, Methoden und Werkzeuge zur Erfassung und Analyse zentraler Aspekte von Unternehmen in einem Framework zur Unternehmensmodellierung zu integrieren. Die *Enterprise Ontology*, welche die Grundlage des Frameworks bildet, liegt sowohl in informaler, d.h. einer

natürlich sprachlichen Beschreibung der Begriffe der Ontologie, als auch in einer mittels Ontolingua formalisierten Form vor [UKMZ98].

Zur Beschreibung der verschiedenen Aspekte von Unternehmen wird die *Enterprise Ontology* in die folgenden Teilbereiche untergliedert [UKMZ98]:

- **Meta-Ontology and Time:** definiert grundlegende Begriffe der Ontologie, wie bspw. *Entity*, *Relationship* oder *Role*, sowie zeitliche Zusammenhänge, wie bspw. *Time-Interval*.
- **Activity, Plan, Capability and Resource:** Begriffe bezogen auf Prozesse und Planung, wie bspw. *Activity*, *Planning* oder *Resource Allocation*.
- **Organisation:** Begriffe zur Beschreibung von organisatorischen Strukturen, wie bspw. *Person*, *Legal Entity* oder *Organisational Unit*.
- **Strategy:** Begriffe für die hochrangige Planung, wie bspw. *Purpose*, *Mission* oder *Critical Success Factor*.
- **Marketing:** Begriffe bezogen auf die Vermarktung und den Verkauf von Waren oder Dienstleistungen, wie bspw. *Sale*, *Customer*, *Price*, *Brand* oder *Promotion*.

### Process Specification Language (PSL)

Die *Process Specification Language (PSL)* wurde vom *National Institute of Standards and Technology (NIST)* als neutrale, standardisierte Sprache zur Prozessspezifikation für den Herstellungsbereich entwickelt. Die Zielsetzung liegt darin den Austausch von Prozessinformationen zwischen verschiedenen Anwendungen mittels eines Mappings zwischen den systemeigenen Prozessbeschreibungen und einem gemeinsamen Austauschformat zu ermöglichen [SGCL99].

Die *PSL Ontologie* besteht aus einem Netzwerk aufeinander aufbauender Module, deren Grundlage im *PSL Core* definiert ist. Im *PSL Core* werden sehr allgemeine Primitiven axiomatisiert, mit welchen die grundlegenden Konzepte von Prozessen beschrieben werden können. Er umfasst die Konzepte *Activity*, *Activity Occurrence*, *Object* und *Timepoint*, die Prädikate *BeginOf* und *EndOf* zur Spezifikation von Start- und Endzeitpunkten von Aktivitäten, sowie einer Reihe weiterer Prädikate zur Beschreibung unterschiedlicher Zusammenhänge zwischen den Elementen der Ontologie. Um die Beschreibung komplexer Prozesse zu ermöglichen, wird er *PSL Core* um eine Reihe von Modulen erweitert, wobei eine Unterscheidung in Kerntheorien und definatorische Erweiterungen vorgenommen wird. Erstere ergänzen die Terminologie des *PSL Core* um weitere Primitiven zur Beschreibung von bspw. Teilaktivitäten oder zusammengesetzten Aktivitäten. Letztere erweitern schließlich die Kerntheorien und werden in die Gruppen *Activity*-, *Temporal and State*-, *Activity Ordering and Duration*- und *Resource Role Extensions* unterteilt, welche die Beschreibung verschiedener Teilaspekte von Prozessen ermöglichen [GM03].

## Process Interchange Format (PIF)

Das *Process Interchange Format (PIF)* ist eine formale Prozessmodellierungssprache zur Unterstützung des automatisierten Austauschs von Prozessbeschreibungen zwischen verschiedenen Systemen aus den Bereichen Prozessmodellierung und -unterstützung. *PIF* wurde ursprünglich als Spin-Off Projekt des *MIT Process Handbook* entwickelt und ist mittlerweile in den *PSL-Core* und dessen Erweiterungen integriert worden [LYPWG94] [Lin08].

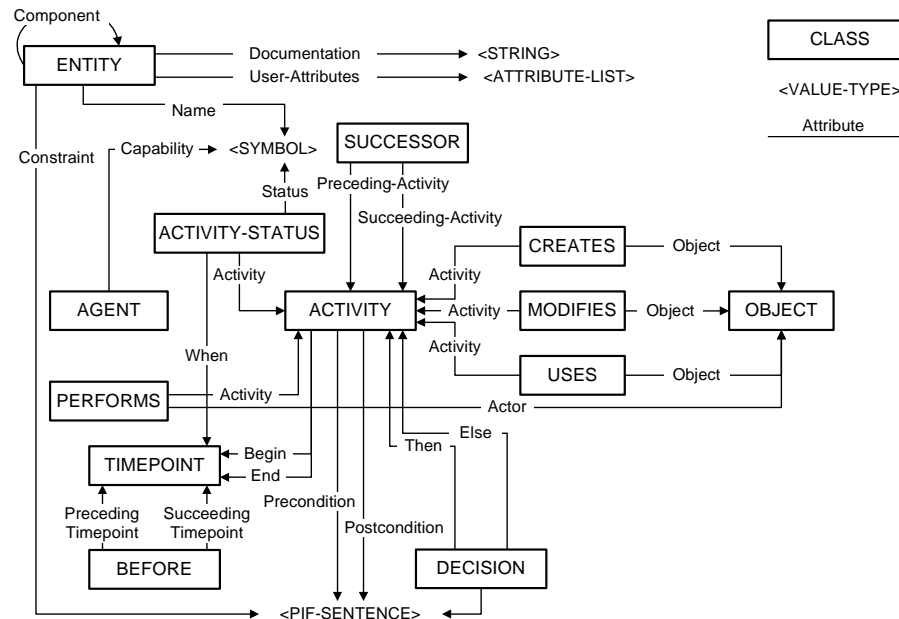


Abbildung 2.17.: Klassen und Beziehungen von PIF [LGJ<sup>+</sup>98]

Abbildung 2.17 zeigt die Klassen und Beziehungen des PIF Formats (Version 1.2). Den Klassen liegt ein hierarchisches Modell zugrunde, in welchem die Klasse *ENTITY* die Basisattribute sämtlicher Klassen kapselt und an diese vererbt. Die grundlegenden Klassen zur Modellierung von Prozessen bilden die Klasse *ACTIVITY*, zur Beschreibung einzelner Tätigkeiten, die Klasse *OBJECT*, mit welcher Akteure und Ressourcen beschrieben werden, die Klasse *TIMEPOINT*, für die Spezifikation von bestimmten Zeitpunkten und der Klasse *RELATION* und deren Unterklassen, welche für die Definition unterschiedlicher Beziehungen, wie bspw. zwischen Aktivitäten (vgl. *SUCCESSOR*), zwischen Aktivitäten und Akteuren (vgl. *PERFORMS*), zwischen Aktivitäten und Ressource (vgl. bspw. *CREATES*), etc. [LGJ<sup>+</sup>98].

## Process, Organisation, Product and others (POP\*)

Die *Process, Organisation, Product and others (POP\*)* Methodik war ein Forschungsbeitrag aus dem Projekt *ATHENA*<sup>1</sup>, welches die Unterstützung der Interope-

<sup>1</sup>Advanced technologies for interoperability of Heterogeneous Enterprise Networks and their Applications

rabilität zwischen Unternehmen zum Ziel hatte. Die Zielsetzung der *POP\** Methodik liegt hierbei in der Bereitstellung eines Meta-Modells zur Beschreibung grundsätzlicher Modellierungskonstrukte und deren Beziehungen, um den Austausch von Prozessmodellen unterschiedlicher Modellierungssprachen zu ermöglichen. Des Weiteren umfasst die *POP\** Methodik Richtlinien für die Verwendung dieses Meta-Modells, d.h. wie Prozessmodelle mittels des *POP\** Meta-Modells abgebildet werden können [GCSP06].

Die durch das *POP\** Meta-Modell definierten Basiskonstrukte zur Modellierung von Prozessen werden in die folgenden fünf Dimensionen eingeordnet, welche unterschiedliche Betrachtungsweisen auf die Komponenten eines Prozesses bilden (vgl. [ATH05] und [GCSP06]):

- **Process Dimension** für die Darstellung von Aktivitäten und Aufgaben, sowie von Prozessabläufen
- **Organisation Dimension** für die Darstellung organisatorischer Strukturen
- **Product Dimension** für die Darstellung von Produkten bzw. Dienstleistungen von Unternehmen
- **Decision Dimension** für die Darstellung von Entscheidungsprozessen, sowie dafür benötigte Strukturen
- **Infrastructure Dimension** zur Darstellung der Infrastruktur von Informations- und Kommunikationstechnologien

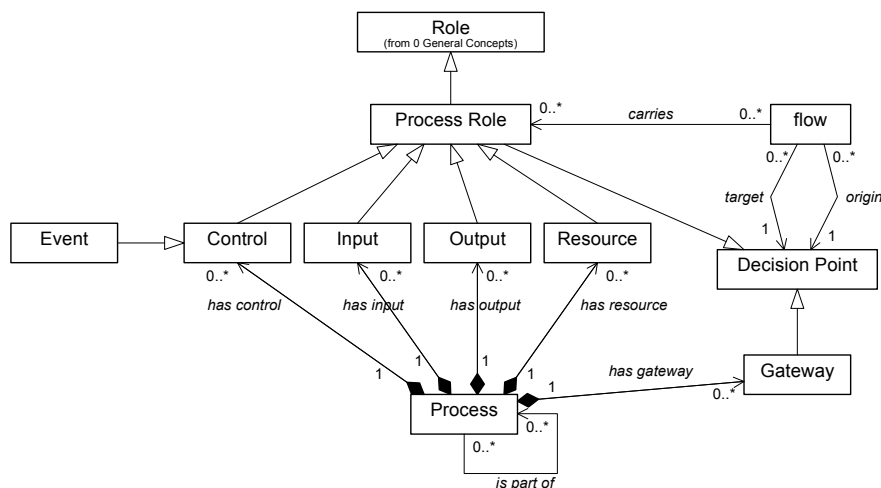


Abbildung 2.18.: Prozessdimension des *POP\** Meta-Modells [ATH05, S. 12]

Im Rahmen der *Process Dimension* (vgl. Abb. 2.18) werden die einzelnen Aktivitäten, Aufgaben, Abläufe und mit diesen in Beziehung stehende Objekte beschrieben. Das zentrale Konzept bildet *Process* zur Beschreibung von Tätigkeiten, welches durch die *is part of* Beziehung mit zugehörigen Teilaktivitäten verknüpft



werden kann. Während die Darstellung des Kontrollflusses mittels der Konzepte *flow* und *Decision Point* erfolgt, wird die Beteiligung unterschiedlicher Unternehmensobjekte aus den anderen Dimensionen durch das Konzept *Process Role* und dessen Unterklassen beschrieben [ATH05, S. 12ff] [GCSP06].

### Das SUPER Projekt

Das SUPER Projekt versucht die Zielsetzung des SBPM, d.h. den Fokus des Geschäftsprozessmanagements von der IT-Ebene auf die betriebliche Ebene zu verlagern, umzusetzen. Hierfür werden eine Reihe von Methoden und Techniken entwickelt, welche alle Phasen des SBP Life Cycle unterstützen sollen [FSM<sup>+</sup>07].

Grundlage für die SUPER Methodik bildet ein ontologisches Framework, mit welchem grundlegende Aspekte von Geschäftsprozessen beschrieben werden können. Folgende Ontologien werden durch das Framework bereitgestellt [BCD<sup>+</sup>07]:

- **Organisational Related Ontologies** umfassen Konzepte zur Darstellung organisatorischer Aspekte, wie bspw. Abteilungen, Angestellte, Rollen sowie unterschiedlicher Beziehungen zwischen diesen.
- **Upper Process Ontology (UPO)** spezifiziert Konzepte, mit welchen eine sehr abstrakte Beschreibung von Prozessen möglich ist. Gemeinsam mit den übrigen Ontologien dient sie vor allem der Beantwortung von Abfragen auf betrieblicher Ebene, wie bspw. "Existiert ein Prozess, an welchen Unternehmen X beteiligt ist?"
- **Semantic EPC (sEPC)** und **Semantic BPMN (sBPMN)** stellen ontologische Versionen von EPK bzw. BPMN dar. Sie stellen Konzepte für deren Modellierungskonstrukte bereit und erweitern diese. Da beiden Modellierungssprachen eine klare Definition der Verhaltenssemantik fehlen, werden in den beiden Ontologien lediglich strukturelle Aspekte dargestellt.
- **Business Process Modelling Ontology (BPMO)** generalisiert die Konzepte der beiden zuvor genannten Ontologien, indem sie allgemeine Konzepte unter einer gemeinsamen Terminologie für die Modellierungskonstrukte zusammenfasst. Außerdem wird sie als Bindeglied zwischen den Prozessen und den Organisationseinheiten der *Organisational Related Ontologies* verwendet.
- **Semantic BPEL (sBPEL)** stellt die ontologische Version von BPEL4WS dar.
- **Events Ontology (EVO)** dient der Darstellung von Ereignissen, welche während der Prozessausführung auftreten und unterstützt Monitoring- und Managementaufgaben. Sie bildet die Grundlage für die Generierung von Event Logs und ermöglicht die Analyse auf unterschiedlichen Abstraktionsniveaus.
- **Process Mining Ontology (PMO)** integriert und verwendet sämtliche in

SUPER definierten Ontologien, wie bspw. *EVO* oder *Organisational Related Ontologies*, mit der Zielsetzung das Mining von Prozessen semantisch auf unterschiedlichen Abstraktionsebenen zu unterstützen. Die gegenwärtig vorliegende *PMO* Version dient der Beschreibung der durch Anwendung von Process Mining Algorithmen auf Event Logs analysierten Prozessmodelle. Im Zuge der Weiterentwicklung sollen weitere durch Process Mining erzielte Analyseergebnisse, wie bspw. Organisationsmodelle, in die Beschreibung mit aufgenommen werden [PD07].

Zur Illustration der semantischen Annotierung im Rahmen des SUPER Projekts wird in den folgenden Absätzen kurz die *sEPC* vorgestellt.



Abbildung 2.19.: *sEPC* Konzepthierarchie

Die *Semantic EPC Ontology* wird für die Formalisierung der wichtigsten Konzepte und Beziehungen von EPKs verwendet, und soll die Annotierung von EPK-Modellen unterstützen. Sie ist im Wesentlichen mit der *EPC Markup Language (EPML)* kompatibel, vernachlässigt allerdings die in *EPML* enthaltenen graphischen Informationen zu den Modellen. Abbildung 2.19 zeigt die in der *sEPC Ontology* spezifizierten Konzepte, welche im Wesentlichen den Modellelementen von EPK-Modellen

entsprechen. Wird ein EPK-Modell semantisch annotiert, werden für die entsprechenden Konzepte Instanzen angelegt, d.h. das Prozessmodell wird in Form einer *sEPC*-Instanz gespeichert.

In der Ontologie werden für die Modellelemente eines EPK-Modells entsprechende Konzepte, wie bspw. *Function*, *EPCEvent* oder *OrganisationalUnit*, spezifiziert, wobei einige dieser Konzepte durch Spezialisierungen die Semantik der Modellelemente erweitern. So werden bspw. durch die Definition der Konzepte *StartEvent*, *EndEvent* und *Event*, welche den *EPCEvent* verfeinern, die verschiedenen Ereignisarten explizit unterschieden. Für die logischen Konnektoren werden ebenfalls entsprechende Konzepte bereitgestellt, wobei grundsätzlich Split- (*Branch*) und Join-Konnektoren (*Merge*) unterschieden werden, welche jeweils durch Konzepte zur Darstellung der möglichen Verknüpfungsarten verfeinert werden. Der Kontrollfluss kann entweder mittels entsprechender Eigenschaften in den Konzepten (*hasIncommingFlow* bzw. *hasOutgoingFlow*) oder mittels des Konzepts *Arc* mit den Eigenschaften *hasSource* und *hasTarget* dargestellt werden.

### Framework zur semantischen Prozessmodellannotierung nach Lin

Lin stellt in ihrer Arbeit ein Framework zur semantischen Annotierung von Prozessmodellen vor, welches die Heterogenität von unterschiedlichem Prozessmodellierungssprachen überbrücken soll [Lin08, S. 66ff]. Das Framework setzt sich aus den Teilbereichen *Profile Annotation*, *Meta-Model Annotation* und *Model Annotation* zusammen, welche in den folgenden Absätzen kurz beschreiben werden.

Ziel der *Profile Annotation* [Lin08, S. 69] ist die Erfassung grundlegender Informationen zu den Prozessmodellen, wie bspw. Versionsnummer oder Verwendungszweck (für eine ausführliche Auflistung der annotierbaren Informationen vgl. [Lin08, S. 70]). Diese Informationen werden den Prozessmodellen in Form von Metadaten hinterlegt, wobei zur Vermeidung von Heterogenitäten auf vordefinierte Daten zugegriffen werden kann. Die *Profile Annotation* umfasst Metadaten zur Verwaltung (*Administrative*), zur Beschreibung (*Descriptive*) und zur Dokumentation von Prozessmodellen (*Preservation*), sowie zur Annotierung technischer Informationen (*Technical*) und zur Wahrung von Informationen betreffend der Verwendung der Prozessmodelle (*Use*).

Die *Meta-Model Annotation* [Lin08, S. 69ff] bildet den Kern des Frameworks und dient der Überbrückung semantischer Heterogenitäten zwischen unterschiedlichen Prozessmodellierungssprachen. Hierfür wird eine Prozessontologie verwendet, welche als Mediator zwischen den Modellierungskonstrukten verschiedener Modellierungssprachen fungiert. Zu diesem Zweck werden in einer als *General Process Ontology* (*GPO*) bezeichneten Ontologie allgemeine Konzepte definiert, welche den Modellierungskonstrukten der Metamodelle der unterschiedlichen Modellierungssprachen annotiert werden können, um diesen eine gemeinsame semantische Bedeutung zu geben. Abbildung 2.20 zeigt die *GPO*, in welcher die folgenden Konzepte unterschieden

werden:

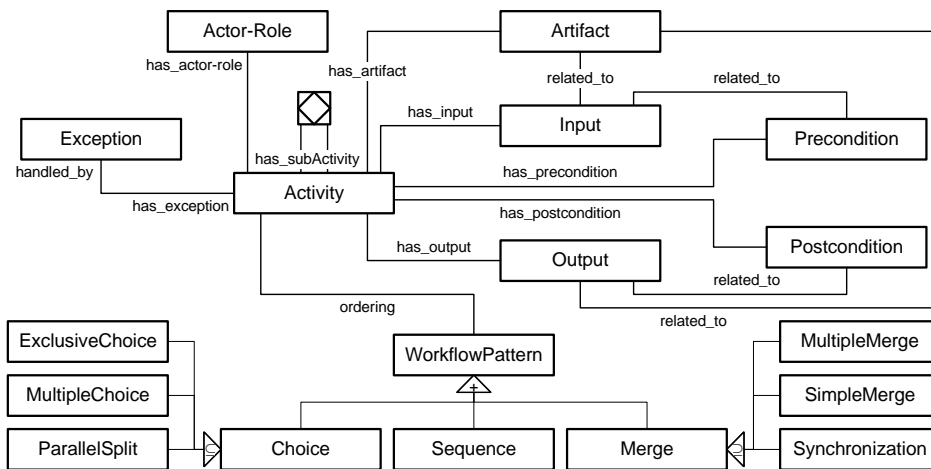


Abbildung 2.20.: General Process Ontology (GPO) [Lin08, S. 72]

- **Activity** bildet das Zentrale Konzept der Prozessontologie. Es werden sowohl elementare als auch zusammengesetzte Tätigkeiten unterschieden.
- **Artifact** spezifiziert Objekte, welche an der Durchführung einer *Activity* beteiligt sind, wie bspw. Werkzeuge.
- **Input & Output** definieren Objekte, welche von einer *Activity* verbraucht bzw. erzeugt werden.
- **Actor-role** spezifiziert jene Objekte, welche mit der *Activity* interagieren, wie bspw. Personen oder Organisationen.
- **Pre- & Postcondition** definieren Einschränkungen, welche zu Beginn bzw. am Ende einer *Activity* gegeben sein müssen.
- **Exception** dient der Erfassung von Störungen und Sonderfällen von Prozessen, welche durch *Activities* behandelt werden können.
- **WorkflowPattern** basiert auf den von van der Aalst et al. beschriebenen WorkflowPatterns [ABHK00] und werden für die Darstellung des Kontrollflusses von Prozessen verwendet.

Die Konzepte der *GPO* werden mittels Mapping-Regeln, welche für jede Modellierungssprache definiert werden müssen, mit den Konstrukten der entsprechenden Metamodelle verknüpft. Durch die semantische Beziehung zwischen den Konzepten der Metamodelle und den Konzepten der *GPO* ist es möglich, Geschäftsprozesse, welche mit unterschiedlichen Modellierungssprachen modelliert wurden, zu vergleichen, da ihnen eine gemeinsame Semantik der Modellierungskonstrukte zugrunde liegt. Ein Prozessmodell, für welches eine *Meta-Model Annotation* durchgeführt wurde,

wird als *GPO*-annotiertes Prozessmodell bezeichnet und in Form eines sogenannten *Process Semantic Annotation Model (PSAM)* formalisiert.

Im Unterschied zur *Meta-Model Annotation*, welche die den Modellelementen zugrunde liegende Semantik vereinheitlicht, wird mit der *Model Annotation* [Lin08, S. 74ff] die Zielsetzung verfolgt, die Inhalte in den Modellelemente, d.h. die Semantik derer Bezeichnungen in Einklang zu bringen. Hierfür werden Domain Ontologien verwendet, welche eine gemeinsame Terminologie und Konzeptualisierung für die natürlich-sprachlichen Bezeichnungen von Modellelementen bereitstellt.

Für die semantische Annotierung der Modellelemente mit Konzepten einer Domain Ontologie werden eine Reihe vordefinierter Beziehungen verwendet, um einerseits sprachliche Ungenauigkeiten auszuräumen und andererseits Begriffe auf unterschiedliche Weise miteinander zu vernetzen. Neben Beziehungen zur Annotierung von Synonymen (*alternative\_name*(terminology Level) bzw. *same\_as*()) sowie von Ober- (*kind\_of*) und Unterbegriffen (*superConcept\_of*) werden unterschiedliche Teil-von Beziehungen für Artefakte (*part\_of*(Artifact)), Akteure (*member\_of*(Actor-Role)), Aktivitäten (*phase\_of*(Activity)) und Ausnahmen (*partialEffect\_of*(Exception)) sowie einer hierzu inversen Beziehung (*compositionConcept\_of*) definiert.

Die semantische Annotierung erfolgt nicht im ursprünglichen Prozessmodell, sondern direkt im *GPO*-annotierten Prozessmodell, d.h. dem Ergebnis der *Meta-Model Annotation*.

### Semantische Annotierung von EPKs nach Thomas und Fellmann

Thomas und Fellmann [TF06] stellen einen Ansatz zur semantischen Annotierung von EPKs vor, mit welchem die Semantik der natürlich-sprachlichen Bezeichnungen der Modellelemente explizit gemacht werden soll. Hierfür werden die Modellelemente mit Konzepten einer Ontologie verknüpft, welche zur terminologischen und konzeptionellen Vereinheitlichung verwendet wird und erweiterte Anfragen über ein EPK-Modell ermöglicht.

Der Ansatz umfasst ein drei-schichtiges Ebenenmodell zur semantischen Annotierung der Modellelemente von EPK-Modellen, welches folgende Ebenen unterscheidet [TF06]:

- **Ontologieebene:** Auf dieser Ebene werden die für ein Unternehmen relevanten Konzepte, einschließlich Spezialisierungs- und sonstigen Beziehungen zwischen diesen definiert. Des weiteren werden auf der Ontologieebene konkrete Instanzen zu diesen Konzepten definiert, welche für die semantische Annotierung der EPK-Modelle verwendet werden.
- **Metadatenebene:** Diese Ebene fungiert als Bindeglied zwischen den beiden anderen Ebenen und wird für die Verknüpfung der Instanzen der Ontologie mit den Modellelementen des EPK-Modells verwendet. Hierfür liegt dieser eine On-

tologie zugrunde, welche das EPK-Modell und dessen Modellierungskonstrukte mittels Konzepten und Beziehungen beschreibt.

- **Modellebene:** Diese Ebene bezieht sich auf das für die semantische Annotierung vorliegende EPK-Modell.

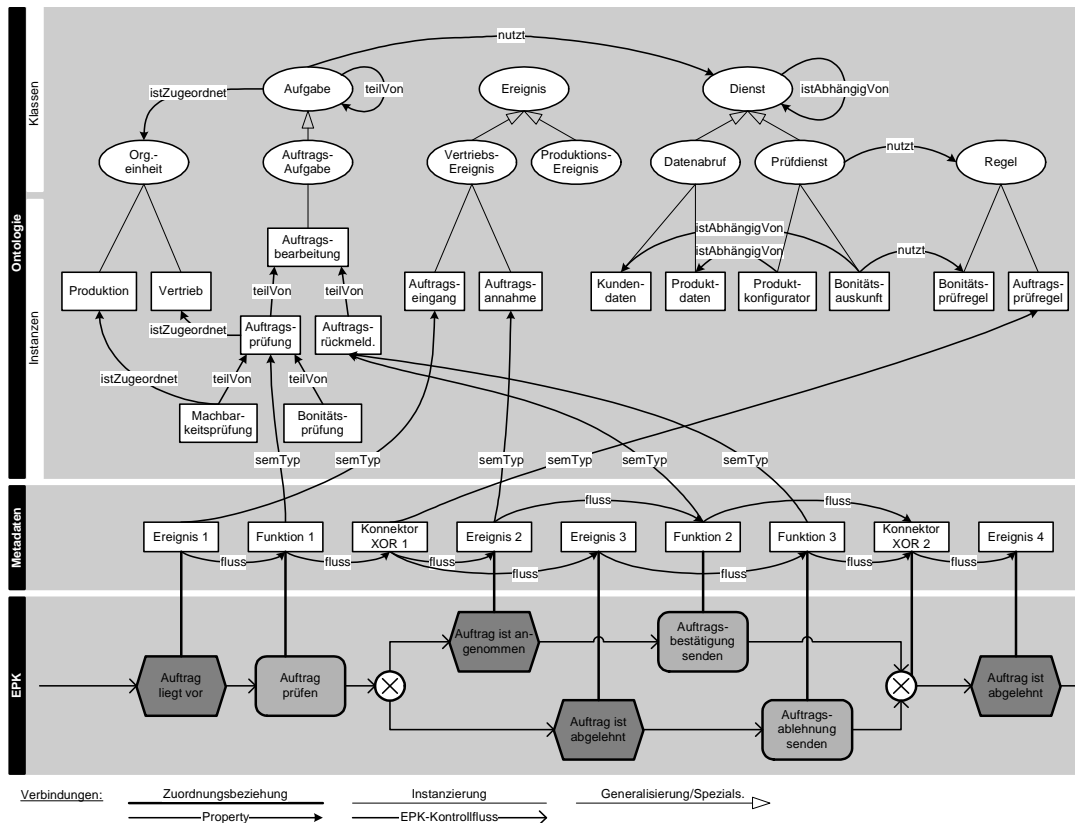


Abbildung 2.21.: Beispiel eines semantisch annotierten EPK-Modells [TF06]

Abbildung 2.21 zeigt ein Beispiel für ein semantisch annotiertes EPK-Modell mit den drei zuvor vorgestellten Ebenen. Auf der Ontologieebene werden grundlegende Konzepte wie *Aufgabe* oder *Ereignis*, Spezialisierungen zu einigen Konzepten, wie bspw. *Vertriebsereignis* und *Produktionsereignis* zum Konzept *Ereignis*, sowie weiterer Beziehungen zwischen diesen Konzepten, wie bspw. die Beziehung *nutzt* zwischen *Aufgabe* und *Dienst* definiert. Des Weiteren werden konkrete Instanzen zu diesen Konzepten, wie bspw. *Auftragseingang* als Instanz des Konzepts *Vertriebsereignis*, spezifiziert. Auf der Metadatenebene werden Instanzen jener Ontologie dargestellt, welche die Modellelemente von EPKs semantisch beschreiben und den konkreten Elementen des zu annotierenden Modells zugeordnet werden. Über den Beziehungstyp *semTyp* werden die Elemente der Metadatenebene schließlich mit den Instanzen der Ontologie der obersten Ebene verknüpft. Durch diese Verknüpfung der EPK-Modellelemente mit den Instanzen einer Ontologie, welche gemäß ihrer Definition ein gemeinsames Vokabular einer Gruppe darstellt, wurde die Semantik der natürlich-

sprachlichen Bezeichnungen explizit gemacht.

### Benutzerfreundliche semantische Annotierung nach Born et al.

Born et al. [BDW07] stellen einen Ansatz zur Erweiterung von Modellierungswerkzeugen sowie eine prototypische Implementierung für "Maestro for BPMN" vor, welcher die semantische Annotierung von Prozessmodellen mit Elementen einer Domain Ontologie bei der Modellierung unterstützt und vereinfacht. Des weiteren ermöglicht die im Rahmen des Ansatzes definierte Struktur der Domain Ontologie die Ableitung von Modellierungsvorschlägen.

Abbildung 2.22 zeigt die zugrunde liegende Architektur des Ansatzes, welche zwei Ontologien zur Darstellung des Prozessmodells (*Extended BPMN Ontology*) bzw. der Domainwelt (*Domain Ontology*) sowie einer Reihe von Funktionalitäten zur Unterstützung der Verknüpfung der Elemente der beiden Ontologien (*Matchmaking Functionalities*) umfasst:

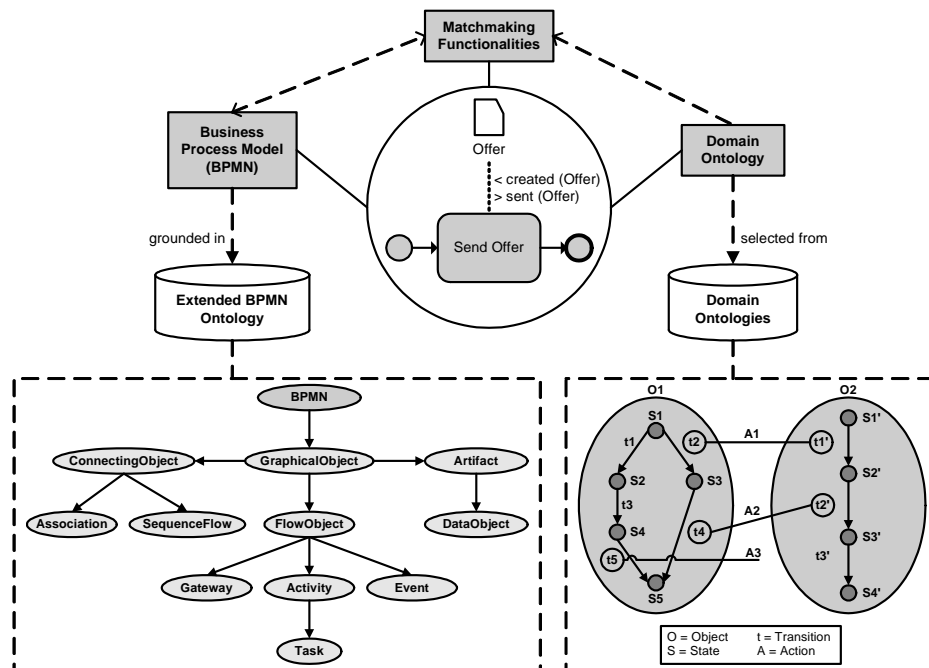


Abbildung 2.22.: Architektur der benutzerfreundlichen Annotierung [BDW07]

- **Extended BPMN Ontology:** Der Ansatz erweitert die sBPMN Ontologie des SUPER Projekts, um die Definition von einer Aktivität vor- und nachgelagerten Zuständen und deren Annotierung mit entsprechenden Konzepten der Domain Ontologie, sowie die natürlich-sprachliche Beschreibung von Vor- und Nachbedingungen zu ermöglichen.
- **Domain Ontology:** Im rechten unteren Bereich von Abbildung 2.22 wird die im Rahmen des Ansatzes entwickelte Struktur für die Domain Ontologie darge-

stellt. Grundlage bilden die verschiedenen Objekte der Domäne (*Object*). Um den Lebenszyklus dieser Objekte zu spezifizieren, werden zu diesen Zustände (*State*) sowie Zustandsübergänge definiert (*Transitions*). Zur Repräsentation von Aktivitäten in der Ontologie werden Aktionen (*Actions*) definiert, welche unterschiedlichen Zustandsübergängen auf verschiedenen Objekten zugewiesen werden können.

- **Matchmaking Functionalities:** Diese unterstützen die Verknüpfung der Elemente der grafischen Prozessmodelle mit jenen der Domain Ontologie, indem sie basierend auf einem ausgewählten Modellelement Vorschläge für die semantische Annotierung oder für den weiteren Modellierungsverlauf ableiten. Hierfür werden sowohl text-basierte Methoden zur Untersuchung der Bezeichnungen als auch Kontextinformationen aus dem Prozessmodell verwendet.

Abbildung 2.23 zeigt ein Beispiel für die Unterstützung der semantischen Annotierung. In der Domain Ontologie werden die Zustände und Zustandsübergänge für die beiden Objekte *Offer* und *CustomerOrder* sowie die zugehörigen Aktionen definiert. Um darzustellen, dass die Zustandsübergänge *t1* des Objekts *Offer* und *t1'* des Objekts *CustomerOrder* semantisch gleichbedeutend sind, werden diese durch die Aktion *send Offer* miteinander verknüpft.

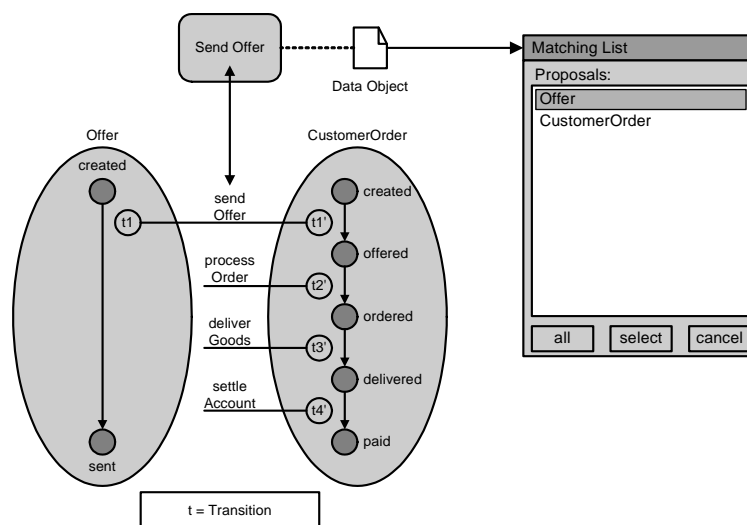


Abbildung 2.23.: Beispiel Definition eines Datenobjekts nach Born et al. [BDW07]

Bei der Unterstützung während der Modellierung können im wesentlichen zwei Arten unterschieden werden. Zum einen die Ableitung von Vorschlägen für die semantische Annotierung von Modellelementen und zum anderen für den weiteren Modellierungsverlauf. Ausgehend von einem ausgewählten Element im Prozessmodell, wie bspw. der Aktivität in Abbildung 2.23, können diese wie folgt skizziert werden:

- Vorschläge für die semantische Annotierung von Modellelementen basieren auf



Objekten der Domain Ontologie. Wurde die Aktivität noch nicht mit einer Aktion der Domain Ontologie verknüpft, wird nach Übereinstimmungen auf basis einer textuellen Analyse der Bezeichnung gesucht und als Ergebnis lediglich das Objekt *Offer* angezeigt. Die Hinterlegung der Aktion *send Offer* hat zur Folge, dass durch die Verknüpfung der Aktion mit Zustandsübergang *t1'* auch das Objekt *CustomerOrder* abgeleitet werden kann.

- Vorschläge für den weiteren Modellierungsverlauf sind kontext-basiert und leiten aus den Zustandsübergängen der Domain Ontologie mögliche nachfolgende Aktionen ab. So kann bspw. mit Hilfe der Domain Ontologie in Abbildung 2.23 die Aktion *process Order* auf Basis von Zustandsübergang *t2'* als nächste Aktivität vorgeschlagen werden.

## 3. Ontologie zur Annotierung von EPK-Modellen

Zielsetzung des pModeler-Ansatzes ist die automatisierte Extraktion von Prozessmustern aus bestehenden Prozessmodellen, welche mittels EPKs modelliert wurden. Grundlage hierfür bildet ein Vergleich der Inhalte, d.h. der Bezeichnungen der einzelnen Modellelemente, welche mittels natürlich-sprachlicher Ausdrücke beschrieben werden. Um deren Vergleichbarkeit zu erhöhen, werden diesen Modellelementen Instanzen einer Ontologie annotiert, welche einerseits die einzelnen Elemente zur Bildung dieser Bezeichnungen explizit ausweisen und andererseits die Beseitigung von Sprachdefekten unterstützen. In diesem Kapitel wird die im Rahmen dieser Arbeit entwickelte Ontologie zur semantischen Annotierung der EPK-Funktionen und -Ereignisse sowie zur Aufbereitung elementarer Modellierungsbausteine, den sogenannten Prozessaktivitäten vorgestellt, welche die automatisierte Aufbereitung der Prozessmuster unterstützen bzw. ermöglichen.

Beginnend mit den Anforderungen an die Ontologie wird eine Abgrenzung zu verwandten Arbeiten aus dem Bereich der semantischen Prozessmodellierung vorgenommen. Im Anschluss daran wird das Wörterbuch vorgestellt, welches der Verwaltung der lexikalischen Aufbereitung der Modellelemente dient (vgl. [Grö06]) und somit die Grundlage für die Instanzen der Ontologie bildet. Daraufhin werden die Konzepte und Beziehungen der Ontologie vorgestellt, mit welchen sowohl EPK-Funktionen und -Ereignisse als auch Prozessaktivitäten, welche Funktionen mit ihren auslösenden und resultierenden Ereignissen in Beziehung setzen, semantisch beschrieben werden. Abschließend werden die im Rahmen der prototypischen Implementierung entwickelten Komponenten zur Verwaltung und Bearbeitung der Ontologie vorgestellt.

### 3.1. Anforderungen an die Ontologie

Zielsetzung der Ontologie dieser Arbeit ist die semantische Annotierung bestehender EPK-Modelle, um einen Vergleich von EPK-Funktionen und -Ereignissen auf einer linguistischen Ebene zu ermöglichen. Hierfür werden Konzepte zur semantischen Beschreibung der Inhalte von EPK-Funktionen und -Ereignissen, d.h. ihren Bezeichnungen sowie zur Darstellung elementarer Modellierungsbausteine, welche

die Grundlage für die Beschreibung von Prozessmustern bilden, bereitgestellt.

Grundlage für die automatisierte Identifikation von Prozessmustern bildet der Vergleich der Inhalte der Modellelemente von EPK-Modellen, deren Bezeichnungen mittels natürlich-sprachlicher Ausdrücke modelliert werden. Um den Vergleich dieser Modellelemente nicht lediglich auf Basis dieser natürlich-sprachlichen Bezeichnungen, d.h. der schlichten Aneinanderreihung von Wörtern, durchzuführen, werden diese mittels Konzepten einer Ontologie semantisch beschrieben. Unter semantischer Beschreibung wird in diesem Zusammenhang die explizite Darstellung der Elemente, aus welchen sich eine Bezeichnung zusammensetzt, verstanden. Bei der Konzeption dieser Elemente wird der an die englische Sprache angepassten Modellierungsempfehlung von Schütte (vgl. Kap. 2.3.2) gefolgt:

- Eine Funktion wird aus einem *aktiven Verb* gefolgt von einem *Objekt* gebildet und beschreibt eine Tätigkeit (in der Ontologie als **Task** bezeichnet), welche auf dem Objekt (**Process Object**) ausgeführt wird.
- Ein Ereignis wird aus einem *Objekt* gefolgt von einem *passiven Verb* gebildet und beschreibt einen Zustand (**State**), in welchem sich das Objekt befindet.

Die explizite Annotierung der Modellelemente mit diesen Konzepten ermöglichte einen semantischen Vergleich der Bezeichnungen, indem nicht mehr lediglich textuelle Ausdrücke untersucht, sondern gezielt **Tasks**, **States** und **Process Objects** gegenübergestellt werden können.

Neben der semantischen Aufbereitung der Bezeichnungen der Modellelemente ermöglicht die Ontologie die Beschreibung elementarer Modellierungsbausteine, den sogenannten Prozessaktivitäten (**ProcessActivity**). Eine Prozessaktivität verknüpft eine Funktion mit ihren auslösenden und resultierenden Ereignissen, d.h. ihrer Vorbedingung bzw. ihrem Ergebnis. Da in EPK-Modellen häufig auf die Modellierung von Trivialereignissen, d.h. Ereignisse, welche direkt aus der Funktion ableitbar sind, verzichtet wird, wird auf diese Weise sichergestellt, dass jeder Funktion ein Ergebnis zugewiesen wird, welches für die Ableitung der Prozessziele bei der Transformation des Prozessmodells in einen Zielbaum benötigt wird (vgl. Kap. 1.1).

Tabelle 3.1 fasst die grundlegenden Konzepte zur semantischen Beschreibung der Bezeichnungen von EPK-Funktionen und -Ereignissen sowie zur Aufbereitung von Prozessaktivitäten zusammen. Die Konzepte der Ontologie sind durch eine Reihe von Beziehungen zur Unterstützung der Aufbereitung der Prozessmodelle und der automatisierten Extraktion von Prozessmustern verknüpft, welche in Kapitel 3.4 beschrieben werden.

Somit ergeben sich eine Reihe von Anforderungen, welche in Form sogenannter *Competency Questions* [UG96], d.h. Fragestellungen, welche von der Ontologie beantwortet werden können sollen, formuliert sind. Diese können wie folgt zusammengefasst werden:

Konzept	Semantik
Process Object	wird aus einer Abfolge von Nomen, gegebenenfalls eingeleitet durch ein Adjektiv, gebildet und repräsentiert ein Objekt
Task	wird aus einem aktiven Verb gebildet und repräsentiert eine Tätigkeit, welche auf ein Process Object ausgeführt werden kann
State	wird aus einem passiven Verb gebildet und repräsentiert einen Zustand, in welchen sich ein Process Object befinden kann
Parameter	wird durch eine Präposition eingeleitet und aus einem Process Object gebildet und beschreibt zusätzliche Informationen in EPK-Funktionen bzw. -Ereignissen
Function	wird aus einem Task, einem Process Object und gegebenenfalls einem Parameter gebildet und repräsentiert die semantische Darstellung einer EPK-Funktion
Condition	wird aus einem Process Object, gegebenenfalls einem Parameter und einem State gebildet und repräsentiert die semantische Darstellung eines EPK-Ereignisses
ProcessActivity	wird aus einer Function und deren Pre- und Postconditions, dargestellt durch Conditions, gebildet und repräsentiert den kleinst-möglichen Modellierungsbaustein zur Beschreibung eines Prozessmusters

Tabelle 3.1.: Konzepte der Ontologie

- Was ist das Objekt (**ProcessObject**) einer Funktion / eines Ereignisses?
- Existieren Generalisierungen / Spezialisierungen zu einem **ProcessObject** und welche sind das?
- Steht **ProcessObject**  $PO_1$  in irgendeiner semantische Beziehung mit **ProcessObject**  $PO_2$  (Generalisierungs- oder Kompositionshierarchie)?
- Migriert, d.h. verändert sich das **Process Object** zwischen zwei aufeinander folgenden Funktionen?
- Welches aktive Verb repräsentiert die Tätigkeit (**Task**), welche auf ein **Process Object** einer Funktion ausgeführt wird.

- Welches passive Verb beschreibt den Zustand (**State**) eines **Process Object** eines Ereignisses.
- Was ist das entsprechende Trivialereignis zu einer Funktion bzw. welche Funktion erzeugt ein entsprechendes Trivialereignis?
- Was sind die Start- bzw. Endereignisse einer Funktion (Beschrieben durch eine **ProcessActivity**)?
- Welche Funktion kann bzw. welche Funktionen können von einem Ereignis ausgelöst werden.
- Welches Ereignis kann bzw. welche Ereignisse können von einer Funktion ausgelöst werden?

## 3.2. Abgrenzung zu verwandten Arbeiten

Auf Grundlage der im vorangegangenen Kapitel dargestellten Anforderungen an die Ontologie dieser Arbeit wird in den folgenden Absätzen eine Abgrenzung zu bestehenden Arbeiten zur semantischen Beschreibung bzw. Annotierung von Prozessmodellen vorgenommen, welche bereits in Kapitel 2.4.2 vorgestellt wurden.

Grundsätzlich legt die explizite Ausweisung der Elemente zur Bildung von Modellelementbezeichnungen, wie bspw. **Tasks** oder **Process Objects**, die Verwendung einer lexikalischen Datenbank, wie WordNet nahe. Auch die Ontologie dieser Arbeit verwendet für die Instanzierung der Konzepte zur semantischen Annotierung der Modellelemente ein solches Wörterbuch (vgl. Kap. 3.3) und erweitert dieses um Konzepte und Beziehungen zur Unterstützung der automatisierten Extraktion von Prozessmustern. Dieses Wörterbuch wurde von Frau Mag.<sup>a</sup> Alexandra Grömer im Rahmen ihrer Diplomarbeit entwickelt, welche auch die Gründe für die Verwendung eines eigenen Wörterbuchs anstelle von WordNet im Kontext der Prozessmodellierung diskutiert (vgl. [Grö06, S. 70]).

Das *MIT Process Handbook* ermöglicht die systematische Organisation von Prozesswissen. Das zugrunde liegende Modell ist nicht für die Annotierung von Prozessen ausgelegt, sondern für eine allgemeine Beschreibung von Aktivitäten, Prozessen und anderen prozess-relevanten Elementen in natürlicher Sprache, sowie der Definition von unterschiedlichen Beziehungen zwischen diesen Elementen.

Die *TOVE Ontologies* und die *Enterprise Ontologies* unterstützen die Erfassung und Analyse zentraler Aspekte von Unternehmen. Die umfangreichen Axiomatisierungen der Ontologien ermöglichen die Beantwortung grundlegender Fragestellungen, wie bspw. "Welche Abfolge von Aktivitäten muss abgeschlossen sein, um ein bestimmtes Ziel zu erreichen? Zu welchen Zeitpunkten müssen diese Aktivitäten gestartet und beendet werden?". Eine solche Auswertung auf Instanzebene (vgl. die

zweite Fragestellung) liegt außerhalb des Anwendungsbereichs der Ontologie dieser Arbeit, welche die semantische Annotierung von Modellelementen auf Typebene zum Ziel hat. Grundlegende Parallelen bestehen zur *Activity Ontology* der *TO-VE Ontologies*, welche Konzepte zur Beschreibung von Aktivitäten und Zuständen bereitstellt. Ähnlich wie die Prozessaktivität dieser Arbeit ermöglicht der *Activity Cluster* die Verknüpfung von Aktivitäten mit ihren auslösenden und resultierenden Zuständen. Ähnlich die *Activity* der *Enterprise Ontology*, welche die Spezifikation von *Pre-Conditions* und *Effects* zur Aktivität ermöglicht.

Viele Arbeiten beschäftigen sich mit der Überbrückung von Heterogenitäten zwischen den Modellierungskonstrukten unterschiedlicher Sprachen (*PSL*, *PIF*, *POP\**, die *General Process Ontology* des Ansatzes von Lin oder das SUPER Projekt). Die Übersetzung wird hierbei durch die Bereitstellung eines gemeinsamen Austauschformats mit allgemeinen Konstrukten und der Definition von Mapping-Regeln zwischen den Prozessmodellen unterschiedlicher Sprachen und dem gemeinsamen Format ermöglicht. Eine Unterstützung mehrerer Modellierungssprachen liegt allerdings außerhalb des Anwendungsbereichs dieser Arbeit. Grundlage für diese Ansätze bilden die Modellierungskonstrukte und nicht die konkrete Bezeichnungen von Modellelementen, wie in dieser Arbeit. Dennoch stellen auch diese Konzepte zur Beschreibung von Aktivitäten und zur Zuweisung von Vorbedingungen und Ergebnissen bereit und könnten als Metaebene zwischen dem Prozessmodell und der Ontologie verwendet werden. Die Zielsetzung dieser Arbeit liegt allerdings nicht in der Transformation in eine neue Darstellung, sondern in der direkten Annotierung der Modellelemente. Explizite Konzepte oder Beziehungen zur Darstellung des Prozessflusses werden nicht benötigt, da für die Extraktion der Prozessmuster die Ableitung eines Zielbaums erfolgt.

Im Unterschied zur zuvor vorgestellten Vereinheitlichung der Modellierungskonstrukte versuchen Ansätze zur semantischen Annotierung auf Modellebene die Interpretationsspielräume von Prozessmodellen einzuschränken (vgl. den Ansatz von Fellmann und Thomas oder die *Model Annotation* des umfangreichen Frameworks von Lin). Durch die Annotierung von Begriffen aus Domänenontologien sollen Sprachdefekte, welche sich aus den natürlich-sprachlichen Bezeichnungen der Modellelemente ergeben können, aufgelöst werden. Domänenontologien sind von Domänenexperten entwickelte, explizite Konzeptualisierungen einer Domäne, welche zusätzlich erweiterte Informationen zu Begriffen sowie verschiedene Beziehungen zwischen den Begriffen bereitstellen können und somit die Ableitung neuer, nicht explizit modellierter Fakten ermöglichen. Eine ähnliche Zielsetzung wird mit der Ontologie dieser Arbeit verfolgt, wobei allerdings für die Vereinheitlichung der Modellelementbezeichnungen keine zuvor entwickelte Domänenontologie verwendet wird. Die Auflösung von Sprachdefekten bei der Instanzierung der zur Annotierung der Modellelemente verwendeten Konzepte erfolgt auf Grundlage eines standardisierten Vokabulars, welches durch ein Wörterbuch bereitgestellt wird.

### 3.3. Domänenspezifisches Wörterbuch

Grundlage für die semantische Aufbereitung der Funktionen und Ereignisse von EPK-Modellen bildet eine lexikalische Aufbereitung der natürlich-sprachlichen Bezeichnungen der Modellelemente, welche von Frau Mag.<sup>a</sup> Alexandra Grömer im Rahmen ihrer Diplomarbeit realisiert wurde (vgl. [Grö06, S. 70-87]).

Zielsetzung der lexikalische Aufbereitung ist es, die einzelnen Wörter der natürlich-sprachlichen Bezeichnungen der EPK-Funktionen und -Ereignisse zu extrahieren und in einem Wörterbuch, dem **Process Dictionary**, zu verwalten. Die Wörter werden anhand ihres Worttyps kategorisiert, wobei *Adjektive*, *Artikel*, *Konjunktionen*, *Nomen*, *Präpositionen* und *Verben* unterschieden werden.

Neben der grundlegenden Klassifikation der Wörter ermöglicht das **Process Dictionary** die Definition semantischer Beziehungen zwischen einzelnen Wörtern. Zu diesen Beziehungen zählen sowohl Synonyme (*Synonym*) und Abkürzungen (*Abbreviation*), als auch Beziehungen zwischen aktiven und passiven Verben (*PassiveVerb*). Synonyme und Abkürzungen tragen zur Standardisierung der Instanzen der Ontologie bei, indem eines der beiden Wörter als Stammwort definiert wird, welches beim Bearbeiten (vgl. Kap. 3.5.2.1) und bei der Extraktion (vgl. Kap. 4.2) verwendet wird. Beziehungen zwischen aktiven und passiven Verben ermöglichen schließlich die Ableitung nicht modellierter Trivialereignisse aus EPK-Funktionen (vgl. Kap. 4.2).

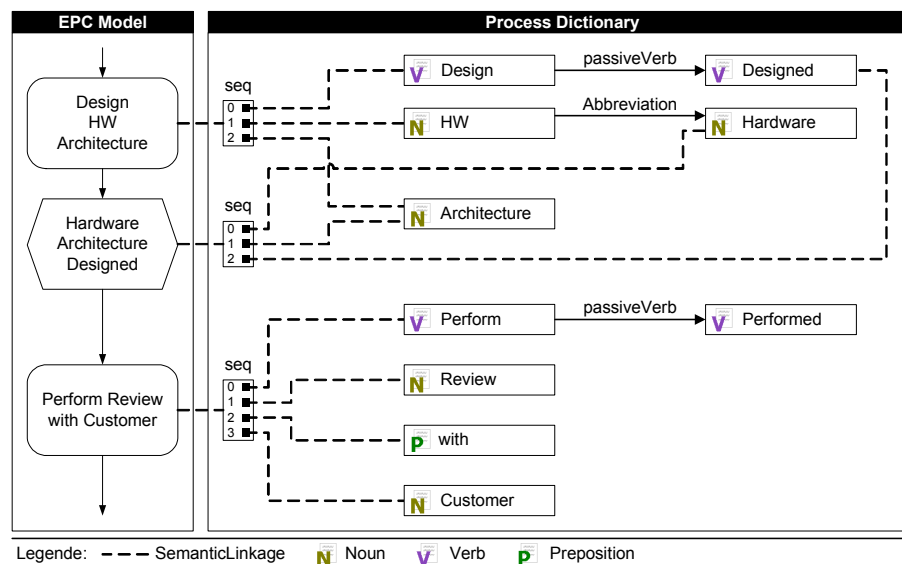


Abbildung 3.1.: Ausgangssituation: lexikalisch annotierte Modellelemente

Abbildung 3.1 zeigt im linken Bereich einen Ausschnitt aus einem EPK-Modell und im rechten Bereich den entsprechenden Auszug aus dem **Process Dictionary**. Die aus den Bezeichnungen der EPK-Funktionen bzw. -Ereignisse extrahierten

Wörter werden diesen unter Berücksichtigung der Reihenfolge (vgl. *seq*) hinterlegt. Neben der Annotierung der einzelnen Wörter werden in der Abbildung auch Beziehungen zwischen aktiven und passiven Verben (vgl. bspw. "Design" *PassiveVerb* "Designed") und eine Abkürzung (vgl. "HW" *Abbreviation* "Hardware") definiert, welche für die semantische Aufbereitung der EPK-Funktionen und -Ereignisse von wesentlicher Bedeutung sind.

### 3.4. Konzepte und Beziehungen der Ontologie

In diesem Abschnitt werden sämtliche Konzepte und Beziehungen der im Rahmen dieser Arbeit entwickelten Ontologie beschrieben, welche in Abbildung 3.2 dargestellt werden. Neben der Ontologie zeigt die Abbildung die beiden Schnittstellen zu den Prozessmodellen (*Process Warehouse*) und dem zugrunde liegenden Wörterbuch (*Process Dictionary*).

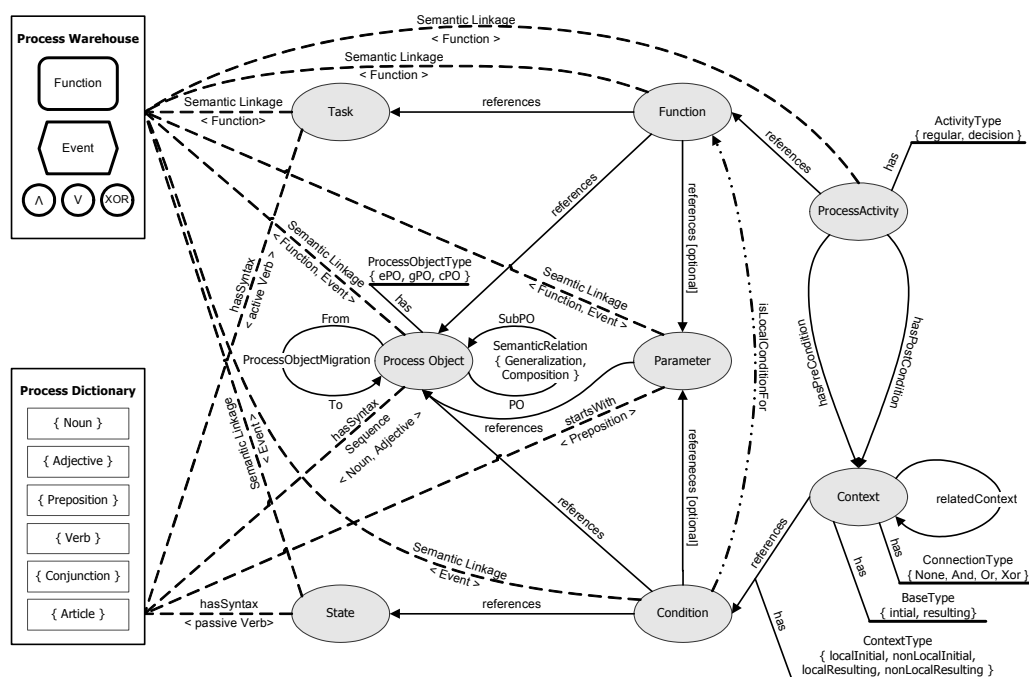


Abbildung 3.2.: Ontologie zur Annotierung von EPK-Funktionen und -Ereignissen

EPK-Funktionen und -Ereignisse werden durch die Konzepte **Function** bzw. **Condition** dargestellt, wobei erstere aus einem **Task** und einem **Process Object** und letztere aus einem **Process Object** und einem **State** gebildet werden. Des Weiteren können beide Konzepte um einen **Parameter** erweitert werden. Prozessaktivitäten, d.h. elementare Tätigkeiten, werden durch eine **ProcessActivity** beschrieben, welche eine EPK-Funktion, d.h. eine **Function**, mit ihren Start- bzw. Endereignissen, d.h. den entsprechenden **Conditions**, verknüpft.



In den folgenden Abschnitten werden nun die einzelnen Konzepte und Beziehungen zwischen diesen näher betrachtet. Des weiteren werden die Beziehungen zu den Elementen der EPK-Modelle sowie zum bereits einleitend vorgestellten **Process Dictionary** beschrieben.

Zentrales Konzept der Ontologie bildet das **Process Object** (vgl. Abb. 3.3), mit welchem materielle und immaterielle Dinge der Wirklichkeit beschrieben werden. Ein **Process Object** wird aus einer Abfolge von Adjektiven und/oder Nomen aus dem **Process Dictionary** gebildet (vgl. *hasSyntax*) und einer EPK-Funktion bzw. einem EPK-Ereignis semantisch annotiert (vgl. *SemanticLinkage*). Bei der Bildung von **Process Objects** kann durch die Verwendung der im **Process Dictionary** definierten Stammwörter (vgl. Kap. 3.3) deren Bezeichnung standardisiert werden. Der Vorteil dieser Standardisierung liegt darin, dass bei einem Vergleich der Modellelemente lediglich die Instanzen der Ontologie und nicht jedes mal die Wörter und deren Beziehungen im zugrunde liegenden Wörterbuch untersucht werden müssen.

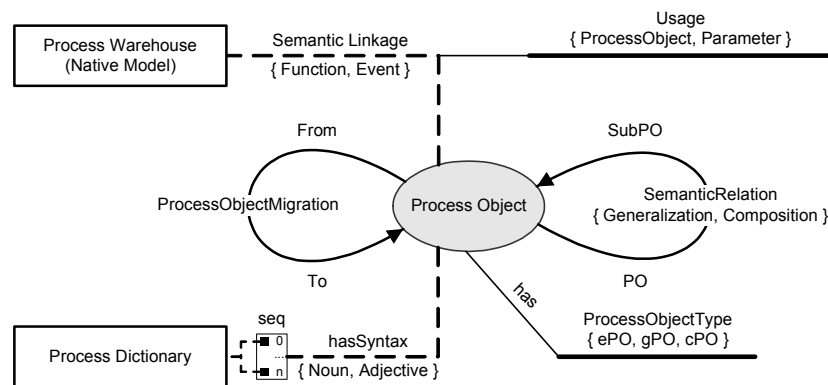


Abbildung 3.3.: Konzept: **Process Object**

### Beispiel: Standardisierung von **Process Objects**

Für die Abkürzung "HW" wurde im **Process Dictionary** der Begriff "Hardware" als Stammwort definiert. Wird diese Beziehung bei der Aufbereitung der EPK-Funktion "Design HW Architecture" berücksichtigt, kann das **Process Object** "Hardware Architecture" gebildet werden. Ein durchgängige Verwendung dieser Stammwörter trägt zur Standardisierung der Bezeichnungen der Instanzen der Ontologie bei.

Die Konzeption der semantischen Beziehungen zwischen den **Process Objects** orientiert sich an den Ansätzen von Rosemann (vgl. Kap. 2.3.3), welcher die Beziehungstypen Prozessobjektmigration (**Process Object Migration**), Generalisierung (**Generalization**) und Komposition (**Composition**) unterscheidet. Eine Prozessobjektmigration beschreibt den Übergang eines **Process Objects** in ein anderes entlang des Prozessverlaufs. Es ist anzumerken, dass Generalisierungen keine Vererbungshierarchien darstellen, sondern lediglich als Generalisierungen im linguistischen Sinne zu betrachten sind.

Generalisierungen und Kompositionen werden im Beziehungstyp *SemanticRelation* zusammengefasst und durch entsprechende Subtypen unterschieden. Diese beiden Typen bestimmen den **Process Object Type**, welcher für eine grundlegende Klassifizierung der **Process Objects** eingeführt wurde. Die folgenden Subtypen können hierbei unterschieden werden:

- **Elementary Process Object (ePO):** **Process Objects**, welchen weder Spezialisierungen noch Dekompositionen zugewiesen wurden.
- **Generalized Process Object (gPO):** Als generalisiert werden jene **Process Objects** gekennzeichnet, welche über den Beziehungstyp **Generalization** mit Spezialisierungen in Verbindung gesetzt werden.
- **Composed Process Object (cPO):** kategorisiert zusammengesetzte **Process Objects**, welche aus der Menge der untergeordneten **Process Objects** bestehen (semantischer Beziehungstyp **Composition**).

#### Beispiel: Process Object und Process Object Type

Für das EPK-Ereignis "Hardware Architecture Designed" konnte das **Process Object** "Hardware Architecture" gebildet werden. Dieses kann zum **Process Object** "Architecture" generalisiert werden, welches ersterem zugewiesen und durch den Typ **Generalized Process Object** beschrieben werden kann. Werden dem **Process Object** "Hardware Architecture" keine weiteren Spezialisierungen oder Dekompositionen zugewiesen, wird dieses mit dem Typ **Elementary Process Object** gekennzeichnet.

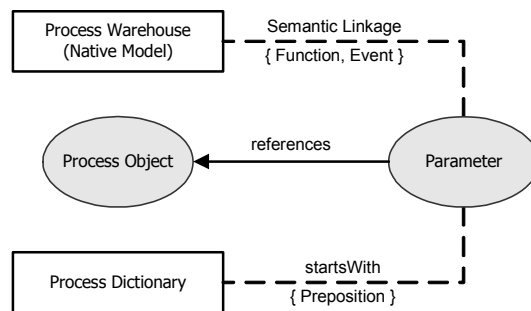


Abbildung 3.4.: Konzept: Parameter

Bei der Analyse bestehender EPK-Modelle konnte festgestellt werden, dass EPK-Funktionen und -Ereignisse häufig mit zusätzlichen Informationen modelliert werden, welche durch eine Präposition eingeleitet und durch ein weiteres Objekt beschrieben werden. Um diesen Sachverhalt in der Ontologie darstellen zu können wurden die Konzepte zur Bildung von Modellelementbezeichnungen um den sogenannten **Parameter** (vgl. Abb. 3.4) erweitert. Diesem werden mittels der Beziehung *startsWith* die einleitende Präposition und mittels der Beziehung *references* das **Process Object**, welches die zusätzliche Information beschreibt, zugewiesen. Auch der

**Parameter** wird zu den bestehenden EPK-Funktionen und -Ereignissen annotiert (vgl. *SemanticLinkage*).

### Beispiel: Parameter

Für die EPK-Funktion "Perform Review with Customer" kann der **Parameter** "with Customer" gebildet werden. Dieser wird durch die Präposition "with" eingeleitet und durch das **Process Object** "Customer" beschrieben.

Die letzten beiden Teilkonzepte zur semantischen Beschreibung von EPK-Funktionen und -Ereignissen bilden der **Task** und der **State** (vgl. Abb. 3.5). Ein **Task** beschreibt die Tätigkeit, welche auf ein **Process Object** ausgeführt werden kann und wird durch ein aktives Verb aus dem **Process Dictionary** dargestellt. Im Gegensatz dazu stellt der **State** einen Zustand, in welchem sich ein **Process Object** befinden kann, dar und wird daher aus einem passiven Verb gebildet. Der Semantik der beiden Teilkonzepte entsprechend wird der **Task** zu einer EPK-Funktion und der **State** zu einem EPK-Ereignis annotiert.

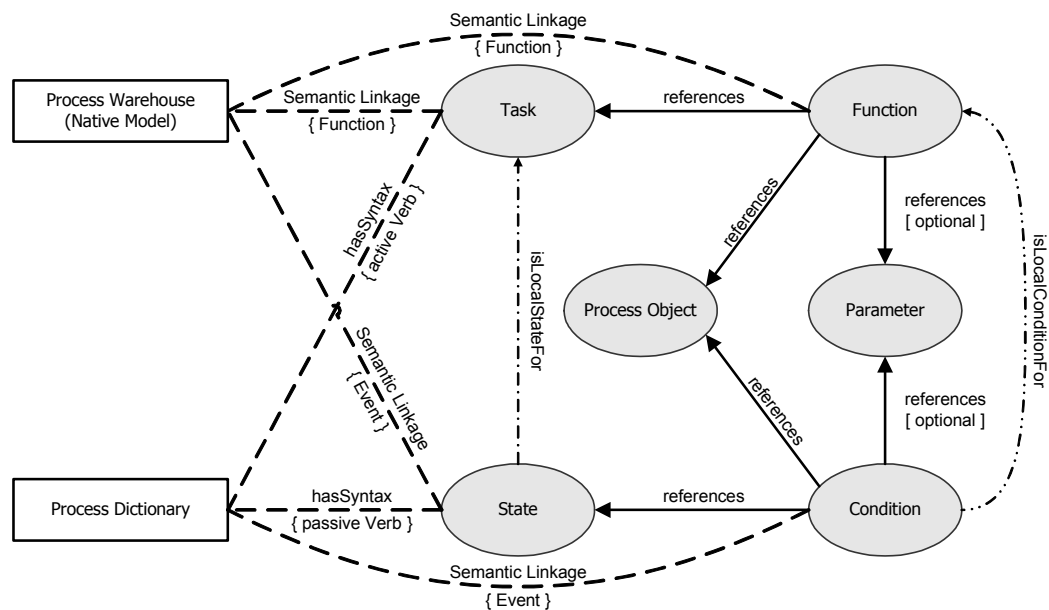


Abbildung 3.5.: Konzept: Function und Condition

Grundsätzlich wäre es möglich die Konzepte **Task** und **State** wegzulassen und die entsprechenden aktiven bzw. passiven Verben aus dem **Process Dictionary** direkt mit den **Functions** bzw. **Conditions** in Beziehung zu setzen. Der Grund für die Einführung dieser beiden Konzepte liegt darin, dass eine klare Trennung zwischen der vorliegenden Ontologie und dem **Process Dictionary** verfolgt wurde, da diese beiden Teilbereiche in unterschiedlichen Arbeiten entwickelt wurde. Außerdem werden die Konzepte der Ontologie mit Häufigkeiten belegt, um ermitteln zu können, wie oft diese in den analysierten Modellen auftreten (vgl. Kap. 4.2).

Des Weiteren zeigt Abbildung 3.5 den Zusammenhang zwischen den einzelnen Teilkonzepten. Diese werden in den Konzepten **Function** und **Condition** zusammengefasst, welche für die semantische Beschreibung der EPK-Funktionen bzw. -Ereignisse verwendet werden. Beide Konzepte referenzieren ein **Process Object** und gegebenenfalls einen **Parameter**. Während eine **Function** zusätzlich einen **Task** referenziert, wird der **Condition** ein **State** zugewiesen. Auch die **Functions** und **Conditions** können den entsprechenden EPK-Funktionen bzw. -Ereignissen annotiert werden.

#### Beispiel: Function

*Bei der semantischen Aufbereitung der EPK-Funktion "Perform Review with Customer" können der Task "Perform", das Process Object "Review" und der Parameter "with Customer" gebildet werden, welche zur entsprechenden Function der Ontologie zusammengefasst werden.*

#### Beispiel: Condition

*Für die Aufbereitung des EPK-Ereignisses "Hardware Architecture Designed" werden das Process Object "Hardware Architecture" und der State "Designed" gebildet und zur entsprechenden Condition zusammengesetzt.*

Neben den einzelnen Teilkonzepten und deren Beziehungen zueinander werden in Abbildung 3.5 zwei weitere Beziehungstypen dargestellt. Mit dem semantischen Beziehungstyp *isLocalStateFor* wird eine Beziehung zwischen einem **Task** und seinem lokalen **State** hergestellt. Als lokaler **State** wird jenes passive Verb bezeichnet, welches direkt aus dem aktiven Verb, welches den **Task** repräsentiert, abgeleitet werden kann. So stellt bspw. "Designed" den lokalen **State** für den **Task** "Design" dar (entspricht dem Beziehungstyp *PassiveVerb* des **Process Dictionary**). Dieser Systematik entsprechend verknüpft der Beziehungstyp *isLocalConditionFor* eine **Function** mit ihrer lokalen **Condition**, d.h. ihrem **Trivialereignis**. Diese Beziehungstypen sind vor allem für die automatische Generierung der **Trivialereignisse**, welche bei der Modellierung der Übersichtlichkeit wegen häufig weggelassen werden, erforderlich (vgl. Kap. 4.2). Im Rahmen dieser Arbeit setzt sich ein **Trivialereignis** zu einer Funktion wie folgt zusammen:

#### Process Object der Function

- + **Parameter** der **Function** (falls vorhanden)
- + **State** = passives Verb des **Tasks** der **Function**

#### Beispiel: Trivialereignis

*Gegeben ist die EPK-Funktion "Design Hardware Architecture". Das entsprechende Trivialereignis zu dieser setzt sich aus dem Process Object "Hardware Architec-*

ture" und dem State "Designed", welcher die passive Verbform zum Task "Design" darstellt, zusammen.

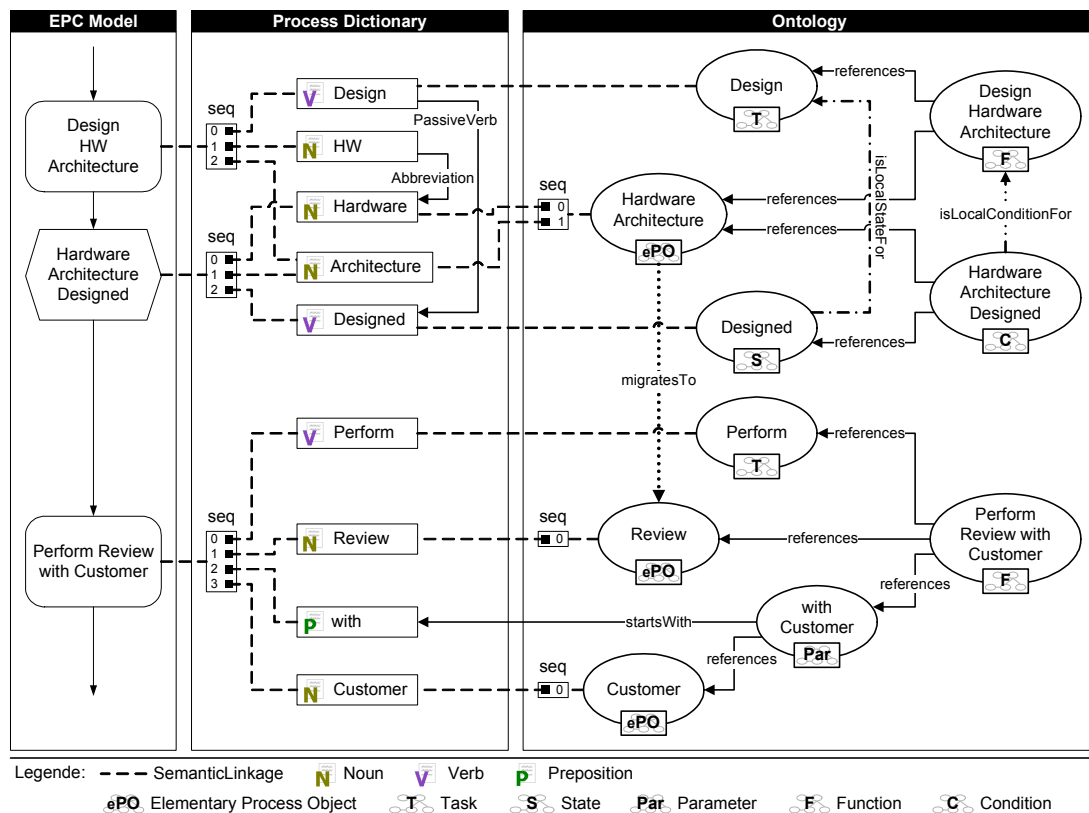


Abbildung 3.6.: Zusammenhang Process Dictionary und Ontologie

Abbildung 3.6 veranschaulicht den Zusammenhang zwischen **Process Dictionary** und **Ontologie** anhand eines Beispiels. Während im linken Bereich ein Ausschnitt aus einem EPK-Modell dargestellt wird, zeigt der mittlere Bereich den entsprechenden Auszug aus dem **Process Dictionary**, in welchem die einzelnen Wörter und die Beziehungen zwischen diesen dargestellt werden. Der rechte Bereich zeigt die Instanzen der **Ontologie**, welche für diesen Ausschnitt aufbereitet wurden, und deren Beziehungen zu den einzelnen Wörtern des **Process Dictionary**. Aus Übersichtlichkeitsgründen wurden die semantischen Beziehungen zwischen den Instanzen und den Modellelementen des EPK-Modells sowie die Generalisierungshierarchien der **Process Objects** weggelassen.

Neben der semantischen Aufbereitung der EPK-Funktionen und -Ereignisse ermöglicht die **Ontologie** die Beschreibung von Prozessaktivitäten. Eine Prozessaktivität beschreibt eine elementare Tätigkeit, indem sie eine EPK-Funktion mit ihren Start- bzw. Endereignissen in Beziehung setzt. Eine Prozessaktivität bildet einerseits den minimalen Modellierungsbaustein zur Beschreibung eines Prozessmusters und andererseits wird mit ihr sichergestellt, dass jeder Funktion ein Ergebnis in Form eines Ereignisses, welches für die Ableitung des Prozessziels benötigt wird, zugewiesen

wird.

Zentrales Element einer **ProcessActivity** (vgl. Abb. 3.7) bildet eine semantisch aufbereitete **Function** aus der Ontologie, welcher eine Menge von Start- bzw. Endereignisse zugewiesen werden können. Für die Bildung dieser Pre- bzw. Post-Condition wird jeweils ein Konzept **Context** instanziiert, mit welchem die eigentlichen **Conditions** zusammengefasst werden. Diese beiden Kontextinformationen werden mittels der Beziehungstypen *hasPreCondition* bzw. *hasPostCondition* mit der **ProcessActivity** verknüpft. Die semantische Annotierung einer **ProcessActivity** erfolgt durch eine Beziehung zur entsprechenden EPK-Funktion im Prozessmodell.

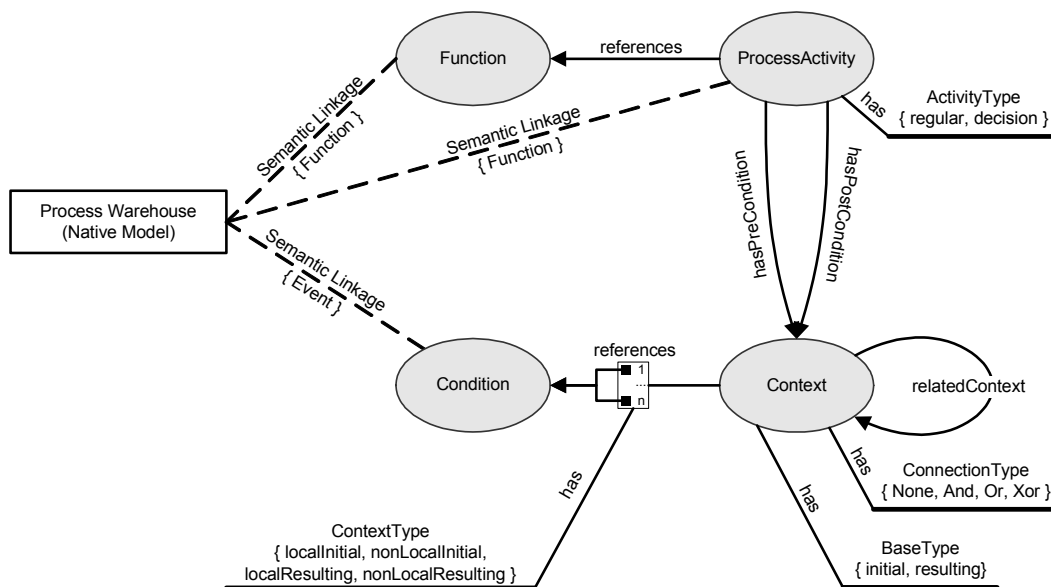


Abbildung 3.7.: Konzept: ProcessActivity

Wie in Abbildung 3.7 dargestellt, wird einer **ProcessActivity** ein Typ zugewiesen. Dieser **ActivityType** wird dazu verwendet, um die **Function** als normale (**regular**) oder als Entscheidungsfunktion (**decision**) zu kennzeichnen. Dieser Typ ergibt sich aus der Post-Condition. Enthält diese einen OR- oder XOR-Konnektor, wird die **ProcessActivity** als Entscheidungsfunktion gekennzeichnet.

Wie bereits einleitend dargestellt werden die **Conditions** mittels des Konzepts **Context** mit der **ProcessActivity** in Beziehung gesetzt. Dieses Konstrukt wird vor allem für die Bildung zusammengesetzter Kontextinformationen benötigt, wenn beispielsweise aus der Durchführung einer EPK-Funktion mehrere EPK-Ereignisse resultieren. Daher ist es möglich, dass ein **Context** mehrere **Conditions** referenziert. Ein **Context** wird mittels der semantischen Beziehungen *hasPreCondition* und *hasPostCondition* mit der **ProcessActivity** verknüpft, um die Start- bzw. Endereignisse der **Function** darzustellen.

Dem **Context** können zwei unterschiedliche Typen zugewiesen werden. Für eine grundsätzliche Unterscheidung wird der **BaseType** verwendet, welcher den **Context**

als Pre- (**initial**) oder Post-Condition (**resulting**) kennzeichnet. Der **ConnectionType** hingegen, legt fest, wie die referenzierten **Conditions** miteinander in Beziehung stehen. Bei zusammengesetzten Kontextinformationen wird mit diesem die Art der Verknüpfung festgelegt (**And**, **Or** oder **Xor**). Wird der **Context** allerdings lediglich auf Basis einer einzelnen **Condition** gebildet, wird dieser mit der Ausprägung **None** belegt.

Die Beziehung zwischen **Context** und **Condition** wird durch den **ContextType** näher spezifiziert. Bei einer Pre-Condition legt dieser Typ lediglich fest, ob das **Process Object** zwischen der **Condition** und der **Function** migriert. Bei der Typisierung einer Post-Condition wird zusätzlich geprüft, ob zwischen dem **State** und dem **Task** eine **isLocalStateFor**-Beziehung besteht, d.h. ob die **Condition** das Charakteristikum eines Trivialereignisses erfüllt. Folgende Typen können somit unterschieden werden:

- **localInitialContext**: die **Condition** und die darauf folgende **Function** referenzieren dasselbe **Process Object**.
- **nonLocalInitialContext**: die **Condition** und die darauf folgende **Function** referenzieren unterschiedliche **Process Objects**.
- **localResultingContext**: die **Condition** erfüllt das Charakteristikum eines Trivialereignisses, d.h. es referenziert dasselbe **Process Object** wie die vorgelegte **Function** und der **State** bildet die passive Verbform des **Tasks**.
- **nonLocalResultingContext**: die **Condition** erfüllt nicht das Charakteristikum eines Trivialereignisses, d.h. es werden entweder unterschiedliche **Process Objects** referenziert oder der **State** bildet nicht die passive Verbform des **Tasks** oder beides.

EPKs ermöglichen die Modellierung aufeinander folgender Konnektoren. Um diese Verschachtelung von Konnektoren darstellen zu können, ermöglicht die Ontologie die Bildung zusammengesetzter Kontextinformationen. Hierfür können einzelne **Contexts** mittels der semantischen Beziehung *hasRelatedContext* miteinander verknüpft werden (vgl. Abb. 3.8).

### Beispiel: Verschachtelter, zusammengesetzter Kontext

Abbildung 3.8 zeigt den Aufbau eines verschachtelten, zusammengesetzten **Context** anhand eines allgemeinen Beispiels. Die Abbildung zeigt einen Ausschnitt aus einem EPK-Modell im linken und die daraus resultierende Post-Condition der Ontologie im rechten Bereich. Zuerst wird **Context** "Con<sub>2</sub>" mit dem **ConnectionType** "And" gebildet, welchem die beiden **Conditions** "C<sub>2</sub>" und "C<sub>3</sub>" zugewiesen werden. Im Anschluss daran wird **Context** "Con<sub>1</sub>" mit dem **ConnectionType** "Xor" gebildet, welcher aus der **Condition** "C<sub>1</sub>" und dem zuvor gebildeten **Context** "Con<sub>2</sub>" zusammengesetzt wird. Aufgrund des "Xor"-Konnektors in "Con<sub>1</sub>" wird die **Function** "F<sub>1</sub>" der **ProcessActivity** "PA<sub>1</sub>" als Entscheidungsfunktion gekennzeichnet.

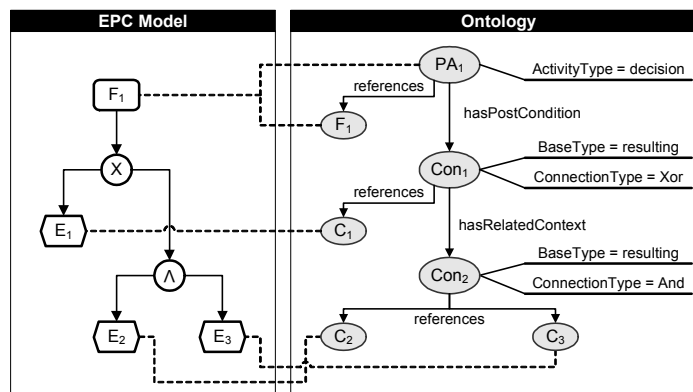


Abbildung 3.8.: Beispiel: verschachtelter, zusammengesetzter Kontext

Abbildung 3.9 zeigt die semantische Annotierung eines Prozessmodells mit den entsprechenden Instanzen der Ontologie anhand eines Beispiels. Während im linken Bereich ein Ausschnitt aus einem EPK-Modell dargestellt wird, zeigt der rechte Bereich den entsprechenden Auszug aus der Ontologie. Die Abbildung zeigt die den EPK-Funktionen und -Ereignissen annotierten Teilkonzepte sowie die für die Modellelemente aufbereiteten **Functions** und **Conditions**. Neben den abgeleiteten Generalisierungshierarchien und Migrationsbeziehungen der **Process Objects** werden auch die lokalen Beziehungen zwischen den **Tasks** und **States** und zwischen den **Functions** und **Conditions** dargestellt. Des Weiteren werden in der Abbildung die für diesen Prozessausschnitt aufbereiteten **ProcessActivities** mit ihren Pre- und Post-Conditions dargestellt.



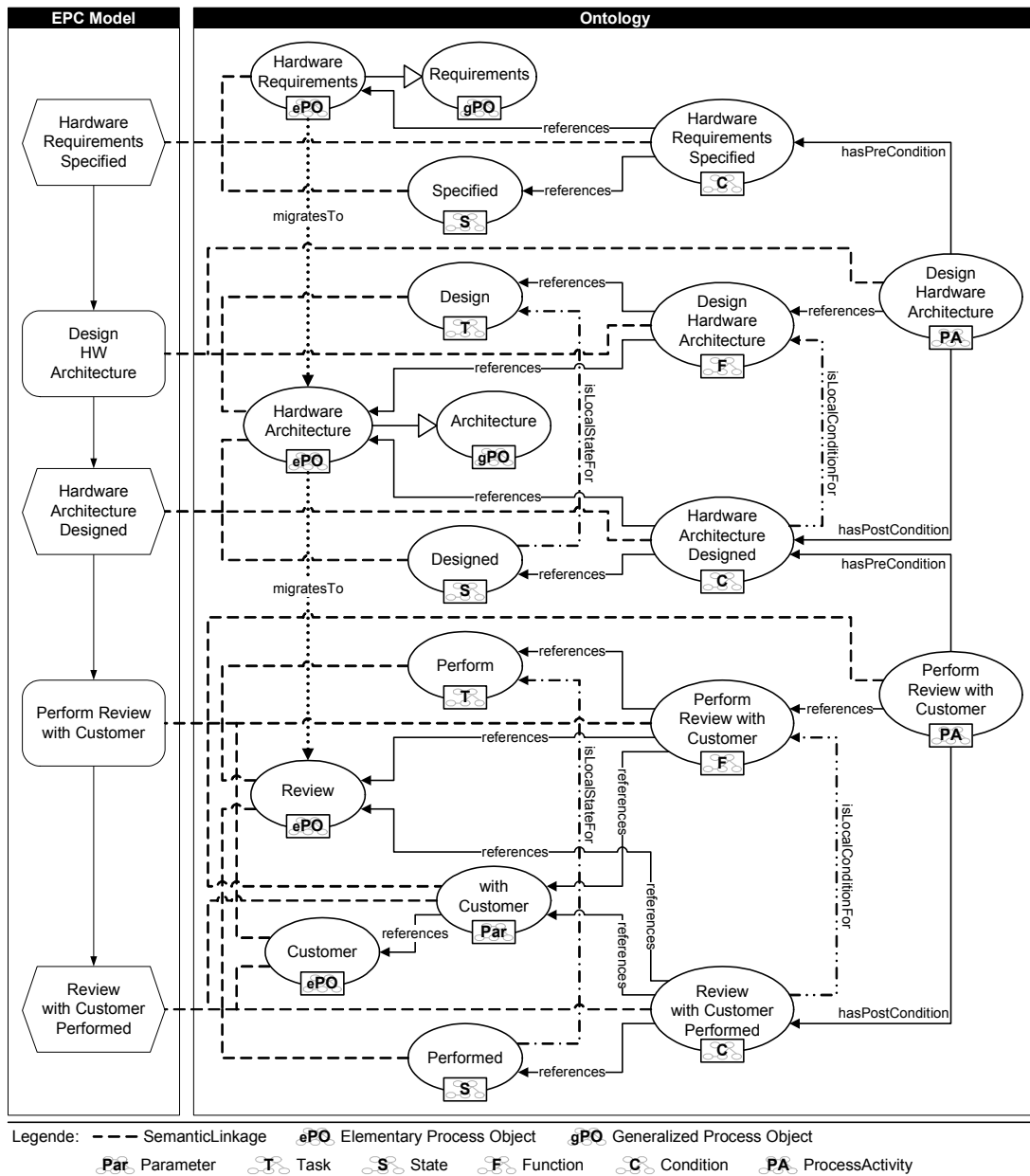


Abbildung 3.9.: Beispiel: semantische Annotierung

## 3.5. Verwaltung der Ontologie in pModeler

Nachdem im vorangegangenen Kapitel die unterschiedlichen Konzepte und Beziehungen der Ontologie vorgestellt wurden, werden in diesem Kapitel die Komponenten zur Verwaltung und Bearbeitung der Wissensbasis, welche im Rahmen der prototypischen Implementierung vom pModeler entwickelt wurden, beschrieben.

Zuerst wird der **Process Object Viewer** beschrieben, welcher für die Verwaltung der **Process Objects** verwendet wird. Im Anschluss daran wird der Assistent zur Bearbeitung und Instanzierung von **Process Objects** vorgestellt, welcher die Bearbeitung von Bezeichnungen und die Definition von Beziehungen zwischen **Process Objects** ermöglicht.

In weiterer Folge wird der **Process Activity Viewer** beschrieben, welcher neben der Verwaltung der **ProcessActivities** auch der Darstellung der **Functions**, **Conditions**, **Tasks** und **States** dient. Zum Abschluss dieses Kapitels werden die einzelnen Editoren zur Bearbeitung und Instanzierung von **ProcessActivities**, **Functions**, **Conditions** und **Parameters** beschrieben.

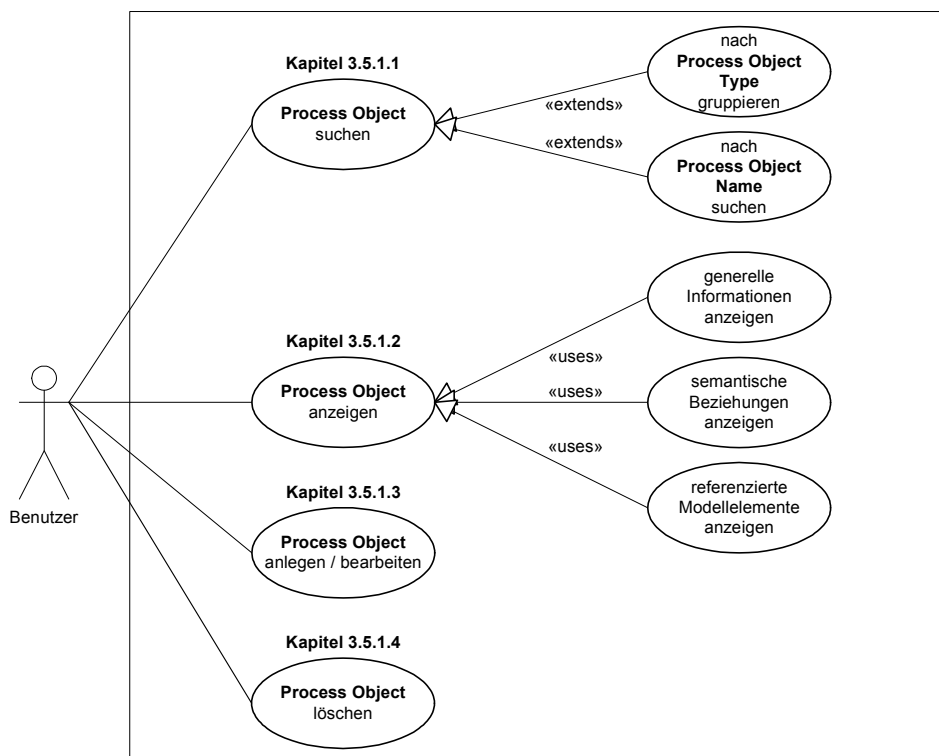


Abbildung 3.10.: Anwendungsfalldiagramm Process Object Viewer

### 3.5.1. Process Object Viewer

Der **Process Object Viewer** wird für die Verwaltung bestehender **Process Objects** verwendet. Die Komponente ermöglicht die gezielte Suche nach **Process Objects**, die Anzeige von detaillierten Informationen zu diesen, sowie das Anlegen, Bearbeiten und Löschen von Instanzen. Das Use-Case-Diagramm in Abbildung 3.10 fasst die Funktionalität des **Process Object Viewer** zusammen, wobei auf die entsprechenden Unterkapitel verwiesen wird, in welchen diese näher beschrieben werden.

Abbildung 3.11 zeigt den **Process Object Viewer**, welcher im pModeler Prototypen über den Menüeintrag **3 Knowledge Management » Process Objects** aufgerufen werden kann. Der linke Bereich zeigt drei unterschiedliche Exploreranzeigen, mit welchen die **Process Objects** anhand ihres Typs gruppiert werden. Im Hauptbereich werden detaillierte Informationen zu einem ausgewählten **Process Object** angezeigt.

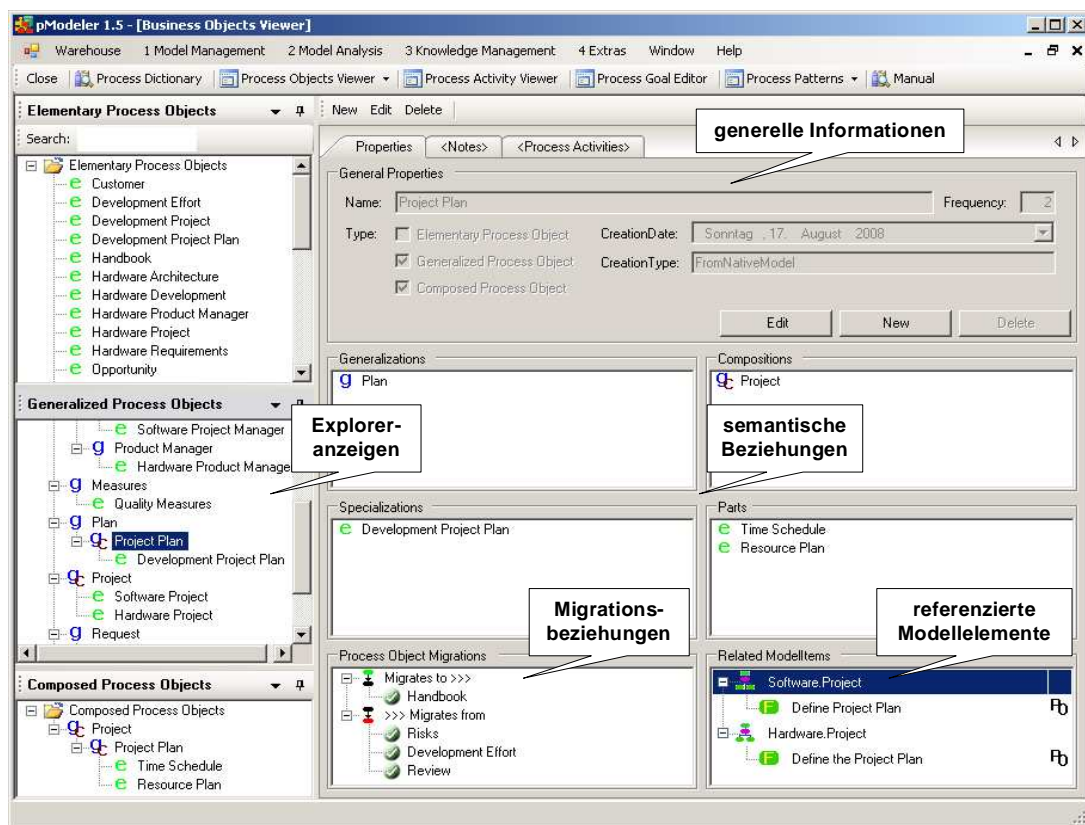


Abbildung 3.11.: Process Object Viewer

In den drei Exploreranzeigen werden sämtliche bereits in der Wissensbasis enthaltene **Process Objects** anhand ihres **Process Object Types** gruppiert und in entsprechenden Baumstrukturen aufgelistet. Die unterschiedlichen Typen werden

hierbei durch die folgenden Icons dargestellt:

- e elementares Process Object
- g generalisiertes Process Object
- c komponiertes Process Object
- gc generalisiertes & komponiertes Process Object

### 3.5.1.1. Process Object suchen

Für die Suche nach Process Objects werden zwei Mechanismen angeboten. Der erste bietet die Möglichkeit einzelne Explorer ein- und auszublenden, um gezielt nach dem gewünschten Typ suchen zu können. Die zweite Möglichkeit bietet eine Suche nach dem Namen eines Process Objects. Hierbei werden nur jene Instanzen angezeigt, welche mit dem eingegebenen Suchbegriff beginnen. In den Exploreranzeigen für die generalisierten bzw. für die komponierten Process Objects, werden dabei die kompletten Hierarchien angezeigt, welche das gesuchte Process Object enthalten.

### 3.5.1.2. Process Object anzeigen

Im Hauptbereich des Process Object Viewer werden generelle Informationen, direkte semantische Beziehungen und referenzierte Modellelemente für das gewählte Process Object angezeigt. Zu den allgemeinen Informationen, welche für alle Konzepte der Ontologie gelten, zählen der Name, die Häufigkeit des Auftretens in bestehenden Modellen, das Erstellungsdatum und die Erstellungsart. Bei den Process Objects werden diese Informationen um den Process Object Type erweitert. Die Erstellungsart beschreibt, wie das Process Object instanziiert wurde, wobei die folgenden Arten unterschieden werden können:

- **fromUser:** das Element, wurde von einem Benutzer manuell angelegt.
- **fromNativeModel:** Elemente dieses Typs wurden durch die Aufbereitung bestehender Modelle gebildet.
- **fromSystem:** alle Elemente, welche vom System automatisch generiert wurden, wie beispielsweise Process Objects in automatisch gebildeten Generalisierungen oder generierte Trivialereignisse.

Unterhalb der generellen Informationen werden in vier Listen die direkten semantischen Beziehungen, d.h. zu den Generalisierungen (**Generalizations**) und Spezialisierungen (**Specializations**) sowie den Kompositionen (**Compositions**) und Teil-Beziehungen (**Parts**), aufgelistet. Links unterhalb befindet sich der Bereich für

die Migrationen, welcher eine Auflistung sämtlicher `Process Objects` enthält, die in den analysierten Modellen vor (`Migrates from`) bzw. nach (`Migrates to`) dem gewählten `Process Object` auftreten. In der letzten Liste werden, gruppiert nach den Modellen, jene Modellelemente angezeigt, welchen das `Process Object` annotiert wurde. Während das Icon vor dem `Process Object` den Typ des nativen Modellelements anzeigt (EPK-Funktion bzw. -Ereignis), kennzeichnet das Icon danach die Art der Verwendung der Instanz in diesem Modellelement (`Process Object` oder Teil des `Parameter`).

### 3.5.1.3. Process Object anlegen oder bearbeiten

Mit den Schaltflächen *Edit* bzw. *New* können bestehende `Process Objects` bearbeitet oder neue angelegt werden. Hierfür wird der `Process Object Editor` aufgerufen (siehe Kap. 3.5.2). Wird der Editor geschlossen, werden die Informationen zum gerade bearbeiteten bzw. angelegten `Process Object` angezeigt.

### 3.5.1.4. Process Object löschen

Mit der Schaltfläche *Delete* können bestehende `Process Objects` gelöscht werden. Um die Konsistenz der Wissensbasis zu gewährleisten, können lediglich jene `Process Objects` gelöscht werden, welche weder von einem EPK-Modell noch von einem anderen Element der Ontologie, wie bspw. einer `Function`, referenziert werden. Sind andere `Process Objects` mit der gelöschten Instanz semantisch verknüpft, werden deren `Process Object Types` gegebenenfalls angepasst.

## 3.5.2. Process Object Editor

Der `Process Object Editor` wird für das Anlegen neuer und das Bearbeiten bestehender `Process Objects` sowie für die Definition oder Entfernung semantischer Beziehungen verwendet. Der `Process Object Editor` ist wie ein Assistent aufgebaut, welcher den Benutzer bei der Bearbeitung der `Process Objects` unterstützt. In Abbildung 3.12 werden die einzelnen Schritte anhand eines Use-Case-Diagramms zusammengefasst, wobei auf die einzelnen Unterkapitel, in welchen diese ausführlich beschrieben werden, verwiesen wird. Überblicksmäßig können die folgenden Arbeitsschritte unterschieden werden:

1. Schritt "Process Object bearbeiten": dieser Schritt ermöglicht das Anlegen neuer bzw. die Bearbeitung bestehender `Process Objects` mit den Einträgen aus dem `Process Dictionary`.
2. Schritt "Semantische Beziehungen bearbeiten": dieser Schritt ermöglicht die Definition neuer bzw. das Entfernen bestehender semantischer Beziehungen.

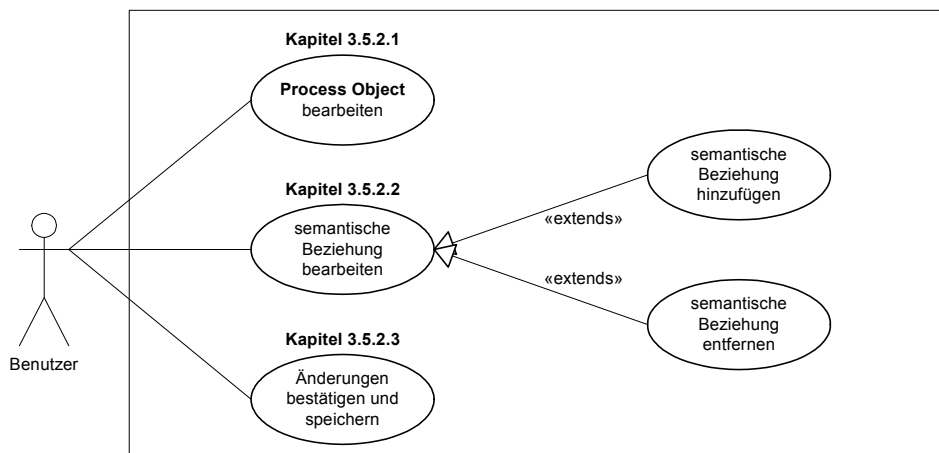


Abbildung 3.12.: Anwendungsfalldiagramm Process Object Editor

- Schritt "Änderungen bestätigen und speichern": abschließend werden die durchgeführten Änderungen zusammengefasst und es ist möglich Änderungen, welche nicht übernommen werden sollen, zurückzunehmen.

### 3.5.2.1. Process Object bearbeiten oder anlegen

Abbildung 3.13 zeigt den ersten Schritt des Assistenten, welcher für die Bearbeitung bestehender bzw. die Instanzierung neuer **Process Objects** verwendet wird. Diese Komponente wird auch allein stehend von anderen Editoren, wie bspw. dem **Function Editor** (vgl. Kap. 3.5.4.2), aufgerufen, wenn eine neue Instanz angelegt werden soll, aber keine Notwendigkeit für die Definition semantischer Beziehungen besteht.

Die Ansicht unterteilt sich in eine Liste zur Anzeige der syntaktischen Repräsentation des **Process Objects** auf der linken Seite und eine Liste mit den zur Verfügung stehenden Wörtern aus dem **Process Dictionary** auf der rechten Seite. Im unteren Teil des Fensters liefert eine Informationstextbox Hinweise, ob das gerade bearbeitete **Process Object** bereits existiert oder nicht.

Für die Bearbeitung wird auf bereits bestehendes Wissen zugegriffen, indem die Einträge aus dem **Process Dictionary** verwendet werden. Die Bezeichnung eines **Process Objects** setzt sich aus Adjektiven und/oder Nomen zusammen, welche alphabetisch sortiert aufgelistet werden. Es ist möglich gezielt nach Begriffen zu suchen, wobei das Ergebnis der Suche mittels der beiden Filter *Noun* und *Adjective* eingeschränkt werden kann.

Ein Wort kann mittels Doppelklick oder durch Betätigung der Schaltfläche *Set* der syntaktischen Repräsentation des **Process Objects** hinzugefügt werden. Das Wort wird dabei nach dem ausgewählten Eintrag in der Liste eingefügt.

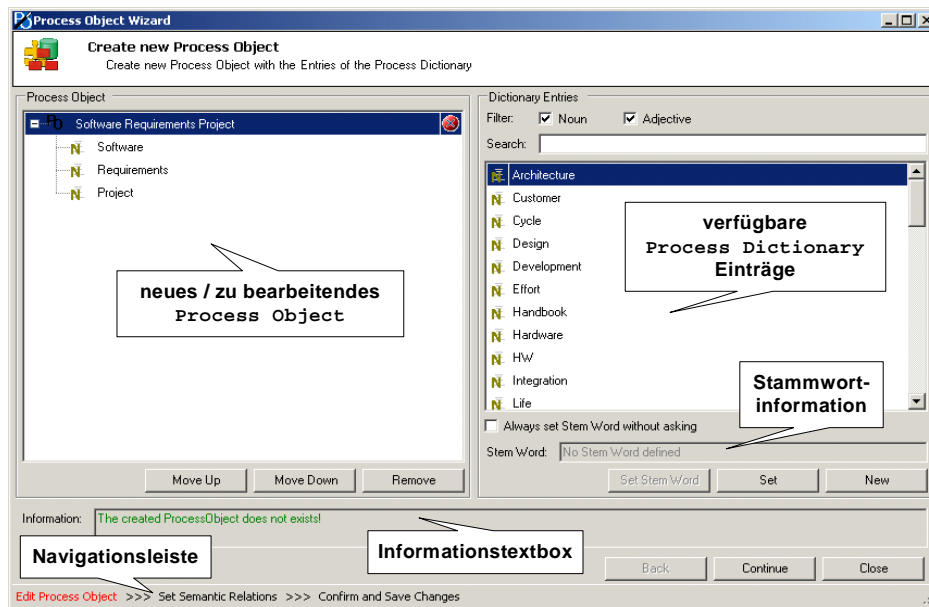


Abbildung 3.13.: Process Object Editor: "Process Object bearbeiten / anlegen"

Eine Zielsetzung der Ontologie liegt in der Standardisierung der bereitgestellten Elemente. Wurde für ein ausgewähltes Wort ein Stammwort definiert, wird dieses in der Textbox "Stem Word" unter der Liste angezeigt. Dieses Stammwort kann mittels der Schaltfläche *Set Stem Word* hinzugefügt werden. Wird jedoch versucht das ausgewählte Wort auf herkömmliche Weise mittels *Set* oder Doppelklick zu setzen, erscheint ein Dialog, welcher auf ein bestehendes Stammwort hinweist (vgl. Abb. 3.14). Daraufhin kann das Stammwort (Auswahl *Ja*) oder das gewählte Wort (Auswahl *Nein*) zugewiesen werden. Für eine durchgängige Nutzung der Stammwörter kann die Checkbox "Always set Stem Word without asking" ausgewählt werden.

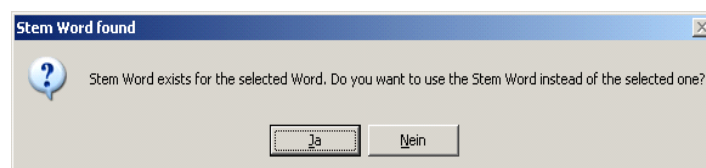


Abbildung 3.14.: Abfrage "Stammwort setzen?"

Konnte ein Begriff nicht gefunden werden, kann mittels der *New* Schaltfläche die Komponente zur Verwaltung des *Process Dictionary* geöffnet werden, um einen neuen Eintrag anzulegen. Nachdem die Komponente geschlossen wird, ist das neu hinzugefügte Wort in der Liste verfügbar (wenn ein Nomen oder ein ein Adjektiv angelegt wurde) und kann dem *Process Object* hinzugefügt werden.

In der linken Liste der Komponente wird die syntaktische Aufbereitung des *Process Objects* angezeigt. Es werden der komplette Name des *Process Objects* und

darunter die einzelnen Wörter mit ihren Typen aufgelistet. Das Icon rechts neben dem `Process Object` zeigt an, ob die vorliegende Instanz bereits in der Wissensbasis enthalten ist (☒ `Process Object` existiert noch nicht, ☑ `Process Object` existiert bereits in der Wissensbasis).

Die Schaltflächen unter dieser Liste werden für die Bearbeitung bereits zugewiesener Einträge verwendet. Mittels der *Remove* Schaltfläche kann der ausgewählte Eintrag wieder entfernt werden (ist der Name des `Process Objects` ausgewählt, werden sämtliche Einträge entfernt). Zur Bearbeitung der Reihenfolge der Einträge werden die beiden Schaltflächen *Move Up* und *Move Down* verwendet, mit welchen der ausgewählte Eintrag nach oben bzw. nach unten verschoben werden kann. Sämtliche Änderungen werden direkt im Namen des `Process Objects` übernommen.

Die vorliegende Version unterliegt hinsichtlich der Bearbeitung bestehender `Process Objects` einigen Einschränkungen. Derzeit ist es nicht möglich die Bezeichnung von `Process Objects` zu bearbeiten, welche den bestehenden Modellen annotiert wurden oder welche von anderen Elementen der Ontologie, wie bspw. von `Functions`, referenziert werden. Die Informationstextbox liefert dafür einen entsprechenden Hinweis und sämtliche Komponenten zum Bearbeiten der syntaktischen Repräsentation werden deaktiviert (es ist allerdings möglich semantische Beziehungen zu bearbeiten).

Neben diesen Einschränkungen ist es nicht möglich ein `Process Object` mit derselben Bezeichnung ein zweites mal anzulegen. Existiert die erstellte Instanz bereits in der Wissensbasis, liefert die Informationstextbox wiederum einen entsprechenden Hinweis und die Auswahl der folgenden Arbeitsschritte ist nicht möglich.

Sind alle Eingaben korrekt, wird die *Continue* Schaltfläche aktiv, mit welcher der nächste Schritt zur Bearbeitung der semantischen Beziehungen angesteuert werden kann. Alternativ ist es möglich mittels der Navigationsleiste in der Fußzeile direkt zum letzten Schritt des Assistenten zu navigieren, um die Änderungen zu bestätigen und zu speichern, ohne semantische Beziehungen zu definieren. Außerdem kann der Assistent mittels *Close*, ohne zu speichern, beendet werden.

### 3.5.2.2. Bearbeiten der semantischen Beziehungen

Im zweiten Schritt des `Process Object Editors` können semantische Beziehungen zum `Process Object` bearbeitet werden. Dieser Schritt ermöglicht die Definition bzw. das Entfernen von Generalisierungen und Spezialisierungen, von Kompositionen und Dekompositionen sowie von Migrationszielen.

Abbildung 3.15 zeigt die Komponente zur Bearbeitung der semantischen Beziehungen. Während im oberen Bereich die allgemeinen Informationen zum `Process Object` (vgl. Kap. 3.5.1.2) angezeigt werden, befinden sich im unteren Bereich die einzelnen Reiter zur Bearbeitung der semantischen Beziehungen.



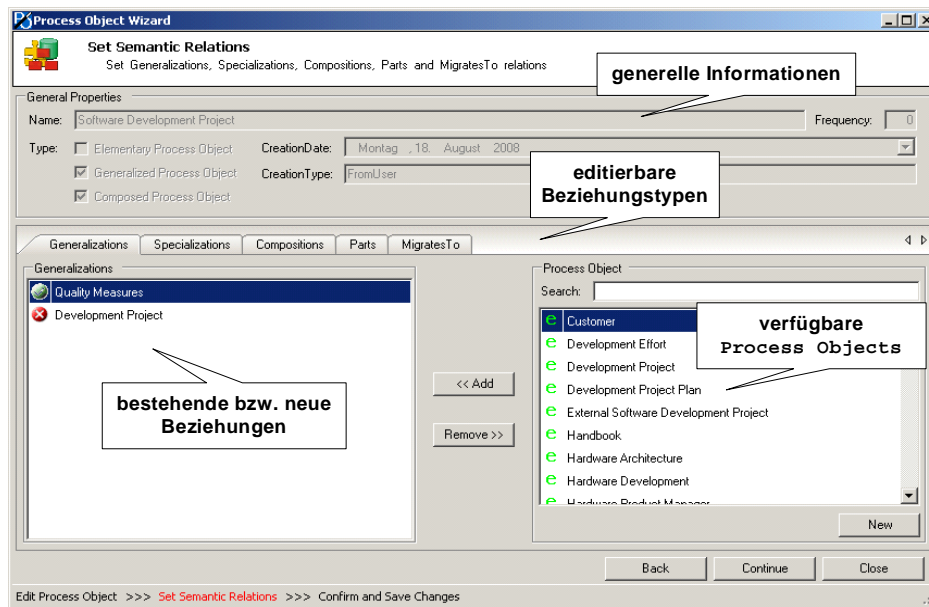




Abbildung 3.15.: Process Object Editor: "Bearbeiten semantischer Beziehungen"

In der linken Liste werden jene **Process Objects** angezeigt, welche bereits mit der zu bearbeitenden Instanz in Beziehung stehen, wobei der ausgewählte Reiter den Beziehungstyp festlegt. Während das Icon  bereits in der Wissensbasis enthaltene Beziehungen kennzeichnet, steht das Icon  für jene Beziehungen, welche neu hinzugefügt wurden.

Im rechten Bereich der Komponente werden die verfügbaren **Process Objects** aus der Ontologie aufgelistet. In dieser Liste kann gezielt nach der gewünschten Instanz gesucht werden. Konnte das benötigte **Process Object** nicht gefunden werden, so kann mittels der Schaltfläche *New* der vereinfachte **Process Object Editor** aufgerufen werden, mit welchem eine neue Instanz angelegt werden kann (vgl. Kap. 3.5.1.3). Nachdem das neue **Process Object** angelegt wurde, erscheint dieses in der Liste und steht zur Auswahl bereit.

Eine semantische Beziehung kann entweder durch Doppelklick auf das **Process Object** oder durch Betätigung der Schaltfläche *Add* hinzugefügt werden. Steht das gewählte **Process Object** bereits in einer anderen semantischen Beziehung mit dem gerade bearbeiteten **Process Object**, erscheint eine entsprechende Information, um

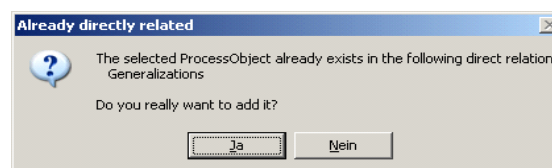


Abbildung 3.16.: Abfrage "Bestehende semantische Beziehung"

sicherzustellen, dass die Beziehung trotzdem gesetzt werden soll (siehe Abb. 3.16). Mittels der *Remove* Schaltfläche können bestehende Beziehungen entfernt werden. Die Anpassungen der *Process Object Types* der beiden *Process Objects* werden vom System automatisch vorgenommen.

Nach Bearbeitung der Beziehungen kann durch Betätigung der *Continue* Schaltfläche zum letzten Schritt, mit welchem die Änderungen bestätigt und gespeichert werden können, navigiert werden. Außerdem ist es möglich mittels *Back* zur Bearbeitung der Bezeichnung des *Process Objects* zurückzukehren oder mittels *Close* die Bearbeitung, ohne zu speichern, abzubrechen.

### 3.5.2.3. Änderungen bestätigen und speichern

Den letzten Schritt des *Process Object Editors* bildet ein Bestätigungsschirm, welcher alle vorgenommenen Änderungen zusammenfasst und die Möglichkeit bietet, Änderungen, welche nicht in der Ontologie übernommen werden sollen, zurückzunehmen (siehe Abb. 3.17).

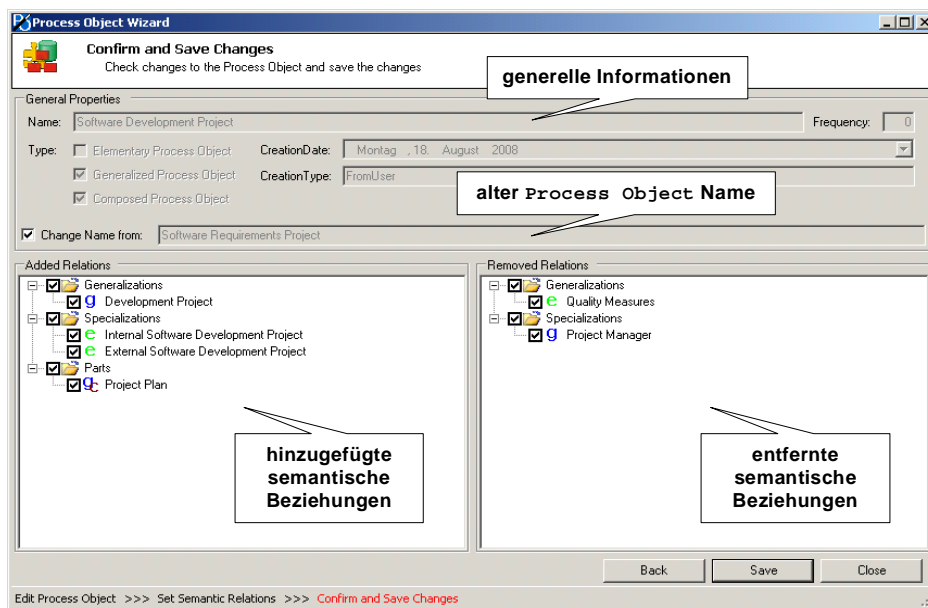


Abbildung 3.17.: *Process Object Editor*: "Änderungen bestätigen und speichern"

Im oberen Bereich werden wieder die generellen Informationen zum *Process Object* angezeigt, wobei im Textfeld für den Namen die neue Bezeichnung steht, wenn der Name geändert wurde. Das Textfeld "Change Name from" enthält bei Bearbeitung der Bezeichnung eines bestehenden *Process Objects* den alten Namen. Mit der Checkbox vor diesem Feld kann nun bestimmt werden, ob die Änderung wirklich in der Wissensbasis übernommen werden soll.

In den beiden darunter liegenden Listen werden die hinzugefügten ("Added Relations") bzw. die entfernten semantischen Beziehungen ("Removed Relations") angezeigt. Mittels der vorangestellten Checkbox kann wieder festgelegt werden, ob die Änderung übernommen werden soll, oder nicht. Um alle Änderungen für einen Beziehungstypen rückgängig zu machen, kann die Checkbox vor dessen Bezeichnung ausgewählt werden.

Die Bearbeitung eines **Process Object** kann mittels der *Save* Schaltfläche abgeschlossen werden, woraufhin sämtliche noch aktiven Änderungen in der Wissensbasis gespeichert werden. Des Weiteren ist es möglich mittels *Back* zum Bearbeiten der semantischen Beziehungen oder mittels der Navigationsleiste in der Fußzeile zur Bearbeitung der Bezeichnung des **Process Objects** zurückzukehren. Mittels *Close* kann der Assistent, ohne zu speichern, geschlossen werden.

### 3.5.3. Process Activity Viewer

Der **Process Activity Viewer** wird für die Verwaltung der bereits in der Wissensbasis enthaltenen **ProcessActivities**, **Functions** und **Conditions** verwendet. Die Komponente ermöglicht die gezielte Suche nach bestehenden Elementen, die Anzeige dieser Elemente und derer Teilkonzepte sowie das Anlegen, Bearbeiten und Löschen bestehender Elemente. Die bereitgestellte Funktionalität wird im Use-Case-Diagramm in Abbildung 3.18 zusammengefasst, wobei auf die entsprechenden Unterkapitel, in welchen diese näher beschrieben werden, verwiesen wird.

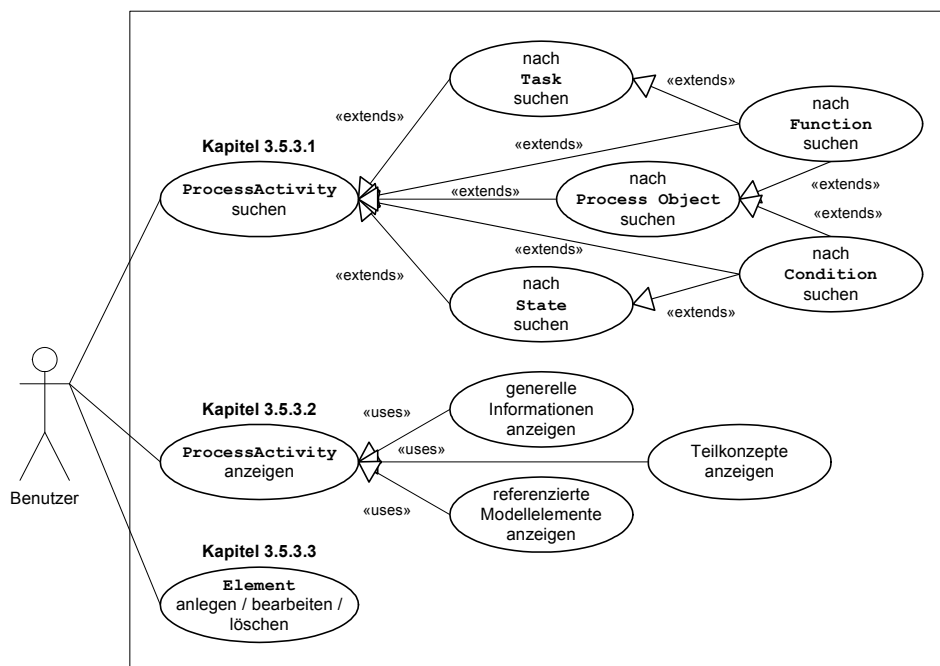


Abbildung 3.18.: Anwendungsfalldiagramm Process Activity Viewer

Abbildung 3.19 zeigt den **Process Activity Viewer**, welcher im pModeler Prototypen über den Menüeintrag **3 Knowledge Management » Process Activities** geöffnet werden kann. Den linken Bereich bilden drei unterschiedliche Explorersanzeigen für die Darstellung der **Tasks**, **States** und **Process Objects**, mit welchen eine Gruppierung der **Functions** und **Conditions** durchgeführt wird. Die Auswahl eines der Elemente in einem dieser Explorer schränkt die Menge der aufgelisteten **ProcessActivities** im Hauptbereich ein. Unter dieser Auflistung werden auf zwei Reitern die erweiterten Informationen zum ausgewählten Element angezeigt, zu welchen die generellen Informationen und die referenzierten Modellelemente zählen.

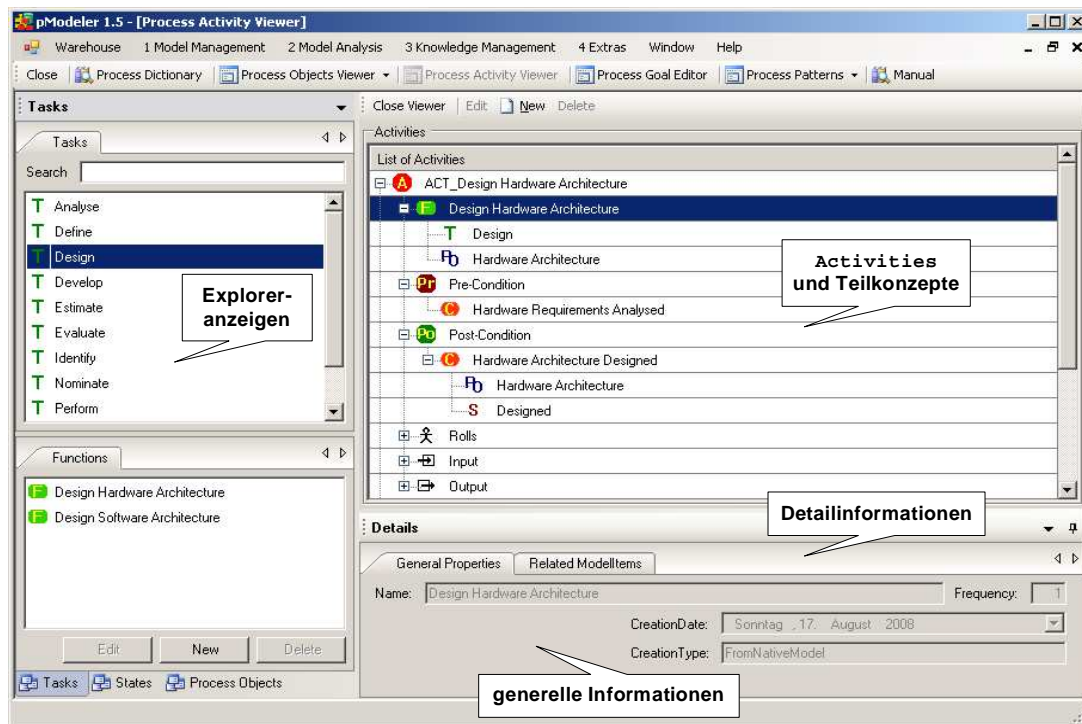


Abbildung 3.19.: Process Activity Viewer

### 3.5.3.1. ProcessActivity suchen

Für die gezielte Suche nach **ProcessActivities** können die bereits angesprochenen Explorer verwendet werden, welche in Abbildung 3.20 gegenüber gestellt werden. Jeder Explorer lässt sich in zwei Bereiche unterteilen, um eine gezielte Suche nach den Teilelementen der **Functions** und **Conditions** zu ermöglichen. Grundsätzlich erfolgt eine erste Gruppierung der Elemente nach den **Tasks** (Abbildung 3.20.a), den **States** (Abbildung 3.20.b) und den **Process Objects** (Abbildung 3.20.c). Neben dieser Gruppierung ist es möglich gezielt nach diesen Elementen zu suchen, wobei sich der eingegebene Text auf den Wortbeginn bezieht. Im unteren Teilbereich eines jeden Explorer, werden die, durch die Auswahl der oberen Liste eingeschränkten

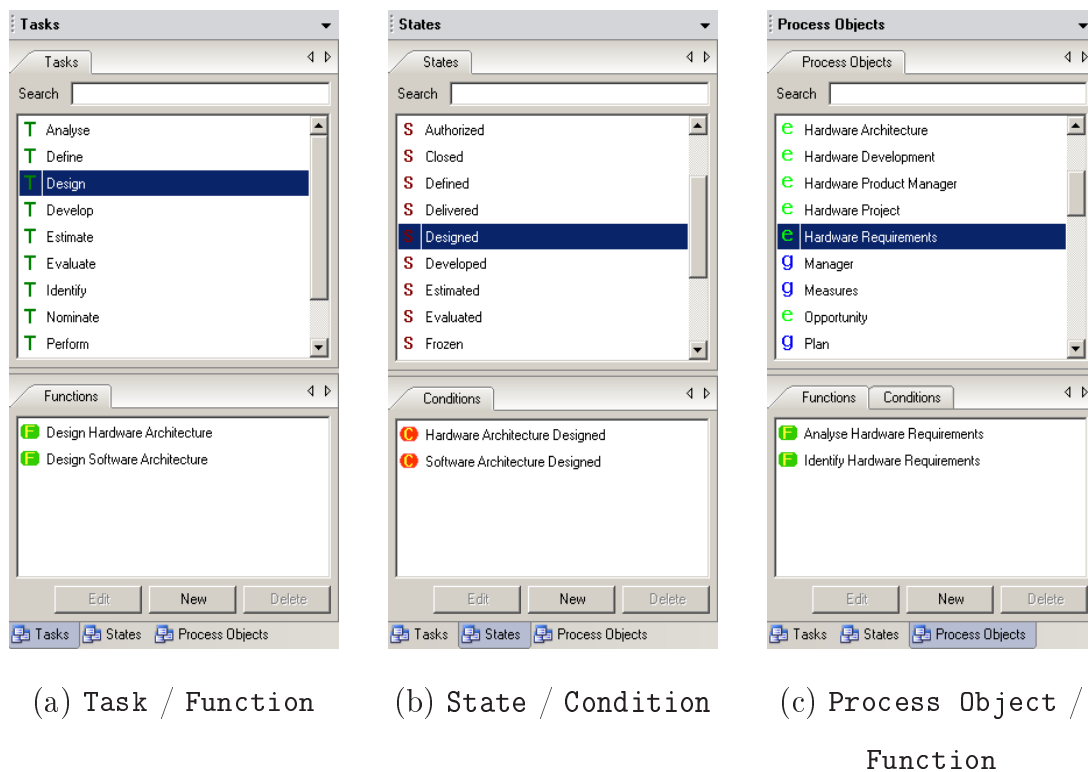


Abbildung 3.20.: Exploreransichten im Process Activity Viewer

Functions bzw. Conditions angezeigt. Wird beispielsweise der Task "Design" ausgewählt, werden lediglich jene Functions aufgelistet, welche diesen Task enthalten. Bei der Gruppierung nach Process Objects wird der untere Abschnitt zweigeteilt, da diese in Functions und Conditions vorkommen können. Die Auswahl bezieht sich in diesem Zusammenhang lediglich auf das eigentliche Process Object der Function bzw. der Condition und berücksichtigt dabei nicht das Vorkommen als Teil eines Parameter. Im Hauptbereich werden lediglich jene ProcessActivities aufgelistet, welche die ausgewählte Function bzw. Condition enthalten (je nach dem welche Anzeige gerade aktiv ist), wobei für letztere sowohl die Pre- als auch die Post-Condition durchsucht werden.

### 3.5.3.2. ProcessActivity anzeigen

Zu Beginn werden zu einer ProcessActivity zur Verkürzung der Ladezeit lediglich die Function und die Ordner für Pre- und Post-Condition, sowie für Rollen, Inputs und Outputs angezeigt, wobei letztere drei in der vorliegenden Version noch nicht aufbereitet werden (vgl. Abb. 3.21.a). Erst nach Auswahl eines Elementes werden die entsprechenden Teilelemente geladen und angezeigt (vgl. Abb. 3.21.b). Neben den Functions und Conditions, welche die grundlegenden Bestandteile einer ProcessActivity bilden, können auch deren Teilkonzepte angezeigt werden.

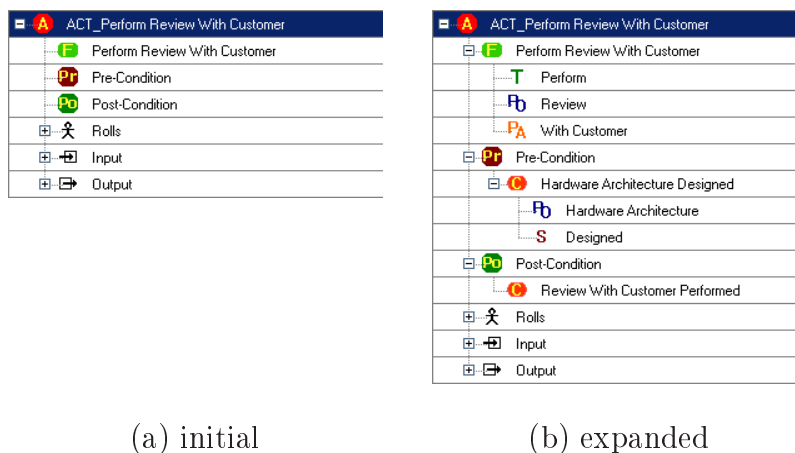


Abbildung 3.21.: Anzeige einer ProcessActivity

Unterhalb dieser Auflistung werden auf zwei Reitern erweiterte Informationen zum ausgewählten Element angezeigt. Während auf dem ersten die generellen Informationen zu diesem Element angezeigt werden (vgl. Kap. 3.5.1.2), werden auf dem zweiten jene Modellelemente aufgelistet, welchen dieses Element annotiert wurde (Abb. 3.22 zeigt im unteren Bereich die Modelle und Modellelemente, in welchen der ausgewählte Task "Design" enthalten ist).

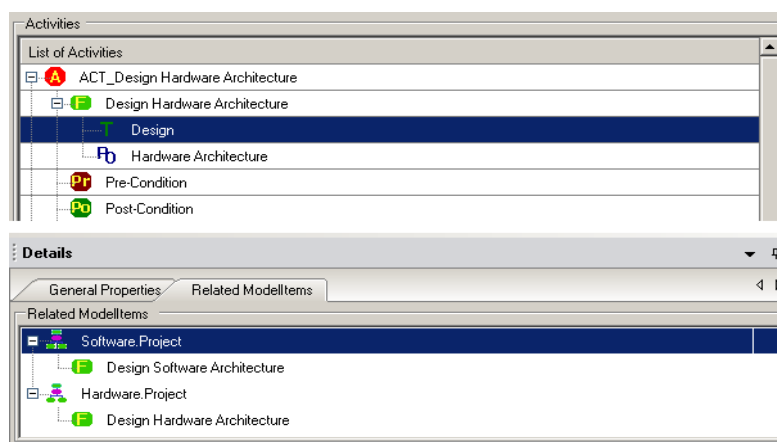


Abbildung 3.22.: Beispiel referenzierte Modellelemente

### 3.5.3.3. ProcessActivity, Function, Condition anlegen, bearbeiten und löschen

Für die Instanzierung, die Bearbeitung und das Löschen von **Functions** und **Conditions** werden unter den Explorersansichten entsprechende Schaltflächen zur Verfügung gestellt. In der vorliegenden Version können lediglich jene Elemente bearbeitet oder gelöscht werden, welche keinem Modellelement annotiert wurden und nicht für

die Bildung einer **ProcessActivity** verwendet werden. Durch Betätigung der *New* bzw. der *Edit* Schaltfläche wird ein entsprechender Editor geöffnet, mit welchem eine neue Instanz angelegt bzw. eine bestehende bearbeitet werden kann. Entsprechend der gerade aktiven Exploreransicht, wird hierfür entweder der **Function**- oder der **Condition Editor** geöffnet (vgl. Kap. 3.5.4). Zum Löschen von manuell erstellten **Functions** oder **Conditions** wird die *Delete* Schaltfläche verwendet.

Die Schaltflächen zum Bearbeiten, Instanzieren oder Löschen von **ProcessActivities** befinden sich im oberen Bereich der Komponente. Wie bei den übrigen Elementen der Ontologie können lediglich jene **ProcessActivities** bearbeitet oder gelöscht werden, welche keinem Modellelement annotiert wurden. Mittels der *New* bzw. *Edit* Schaltfläche wird der **Process Activity Editor** (vgl. Kap. 3.5.5) geöffnet, mit welchem neue **ProcessActivities** angelegt bzw. bestehende bearbeitet werden können. Mittels der *Delete* Schaltfläche können schließlich manuell erstellte **ProcessActivities** gelöscht werden.

### 3.5.4. Function-, Condition- und Parameter-Editor

In diesem Kapitel werden die einzelnen Editoren beschrieben, mit welchen **Parameter**, **Functions** und **Conditions** erstellt oder bearbeitet werden können. Grundsätzlich unterscheiden sich diese Editoren nur in wenigen Einzelheiten. Daher werden in diesem Abschnitt kurz die Gemeinsamkeiten, welche den Aufbau und die Funktionalität betreffen, allgemein beschrieben. Danach werden die einzelnen Editoren gezeigt, wie sie im Rahmen des Prototyps realisiert wurden.

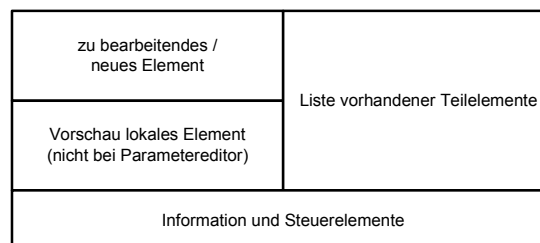


Abbildung 3.23.: Element Editoren - schematischer Aufbau

Abbildung 3.23 zeigt den schematischen Aufbau der Element-Editoren. Im linken oberen Bereich wird das gerade bearbeitete bzw. neu angelegte Element angezeigt. Darunter wird beim **Functions**- und **Conditions Editor** die Vorschau für das entsprechende "lokale" Element angezeigt. Für eine **Function** ist das "lokale" Element das **Trivialereignis**, für eine **Condition** die **Function**, welche diese herbeiführt. Die in diesem Bereich verwendete Liste ist nicht auswählbar, da sie nur zur Anzeige bestimmt ist. Im rechten Bereich werden die zur Zuweisung verfügbaren Teilelemente aufgelistet, welche bereits in der Ontologie bzw. im **Process Dictionary** enthalten sind. Im unteren Bereich werden Informationen zum Bearbeitungsfortschritt des

entsprechenden Elements angezeigt. Des Weiteren befinden sich die Steuerelemente zum Speichern des Elements und zum Schließen des Editors in diesem Bereich.

Beim Instanzieren bzw. Bearbeiten eines Elements erfolgt eine Einschränkung, indem bereits zu Beginn der grundsätzliche Aufbau vorgegeben wird (siehe Abb. 3.24). Bei den **Functions** und **Conditions** folgt dieses Template der, dieser Arbeit zugrunde liegenden, englischen Grammatik der Modellelemente (vgl. Kap. 2.3.2), wobei der **Parameter** nicht zugewiesen werden muss. Ein **Parameter** wird, wie bereits dargestellt, aus einer einleitenden Präposition und einem **Process Object** gebildet. Die Icons in der rechten Spalte geben Auskunft darüber, ob das entsprechende Element bereits in der Wissensbasis enthalten ist (✓) oder nicht (✗).

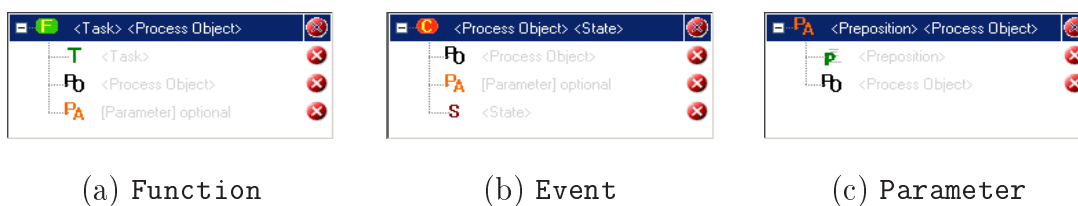


Abbildung 3.24.: Element Editoren - Aufbau der Elemente

Beim Erstellen eines Elements wird, wie beim **Process Object Editor** (vgl. Kap. 3.5.2), auf bereits bestehendes Wissen zurückgegriffen. Entsprechend der Auswahl in der Vorlage wird die Liste mit den verfügbaren Elementen geladen. Wird beispielsweise "ProcessObject" ausgewählt, werden in dieser Liste sämtliche in der Ontologie verfügbaren **Process Objects** aufgelistet (siehe Abb. 3.25). In dieser Liste ist es nun möglich gezielt nach Elementen zu suchen. Konnte das benötigte Element nicht gefunden werden, ist es möglich mittels der **New** Schaltfläche den entsprechenden Editor zu starten, um dieses anzulegen (d.h. wenn gerade die **Process Objects** angezeigt werden, wird der **Process Object Editor** geöffnet).

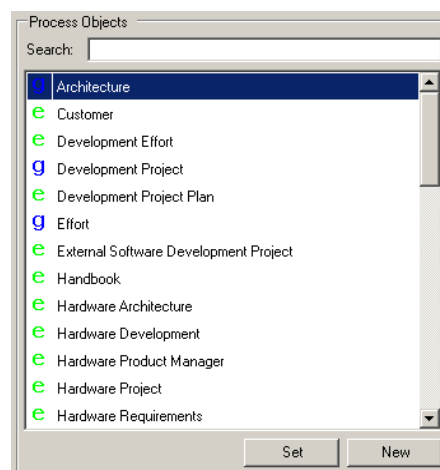


Abbildung 3.25.: Element Editoren - verfügbare Process Objects



Die Zuweisung eines Elements kann entweder durch Doppelklick oder mittels der Schaltfläche *Set* vorgenommen werden. Bei der Bearbeitung von **Functions** und **Conditions** wird das entsprechende Element in beiden Vorlagen, d.h. auch in jener für das automatisch generierte lokale Gegenstück, eingefügt (für eine detaillierte Erklärung dazu vgl. Kap. 3.5.4.2). Um Einträge aus dem bearbeiteten Element wieder entfernen zu können, wird unter der Vorlage eine *Remove* Schaltfläche angeboten.

Sind alle benötigten Teilelemente zugewiesen, wird geprüft, ob dieses Element bereits in der Wissensbasis existiert. Konnte es gefunden werden, wird in der Informationstextbox ein entsprechender Hinweis angezeigt und ein neuerliches Speichern des Elementes ist nicht möglich. Wurde allerdings ein neues Element angelegt, wird die *Save* Schaltfläche aktiviert und es kann gespeichert werden.

### 3.5.4.1. Parameter Editor

Mit dem **Parameter Editor**, können die, in den **Functions** und **Conditions** optional enthaltenen **Parameter** angelegt werden. Dieser Editor kann in der vorliegenden Version des Prototyps lediglich vom **Function**- bzw. **Condition Editor** aus aufgerufen werden. Außerdem ist es derzeit nicht möglich bestehende **Parameter** zu bearbeiten oder zu löschen. Abbildung 3.26 zeigt den **Parameter Editor**, welcher dem bereits vorgestellten grundlegenden Aufbau folgt. Das Anlegen eines **Parameter** erfolgt indem eine Präposition aus dem **Process Dictionary** und ein **Process Object** aus der Ontologie zusammengefasst werden. Wurden beide Elemente zugewiesen und der erstellte **Parameter** existiert noch nicht in der Wissensbasis, wird die *Save* Schaltfläche aktiviert und die neue Instanz kann gespeichert werden.

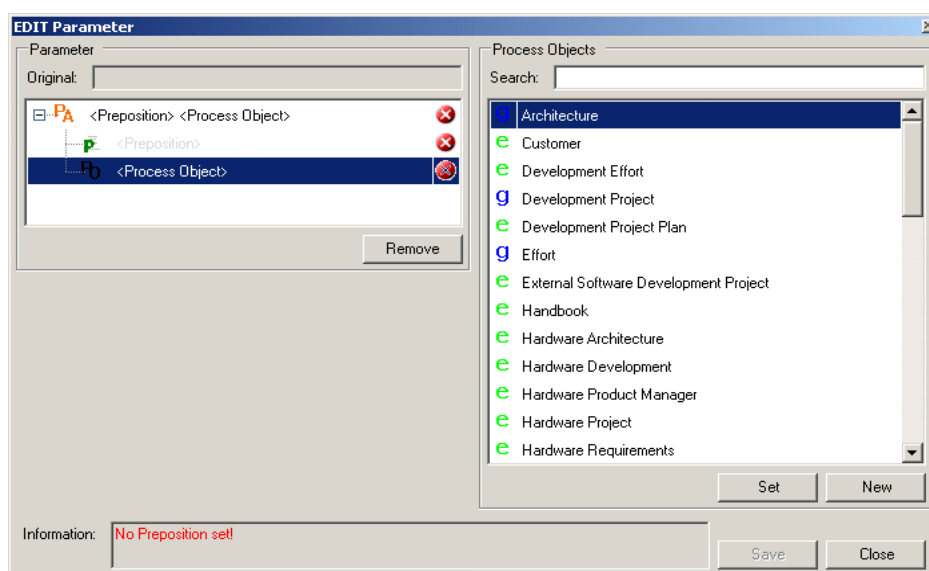


Abbildung 3.26.: Parameter Editor

### 3.5.4.2. Function- und Condition Editor

Mit diesen beiden Editoren können **Functions** bzw. **Conditions** angelegt bzw. bearbeitet werden. Da sich deren Funktionalität lediglich in wenigen Details unterscheidet, werden diese beiden Editoren in einem Kapitel beschrieben. In den nächsten Abschnitten wird der **Function Editor** (siehe Abb. 3.27) ausführlich erklärt, wobei auf die Unterschiede zum **Condition Editor** an den entsprechenden Stellen hingewiesen wird.

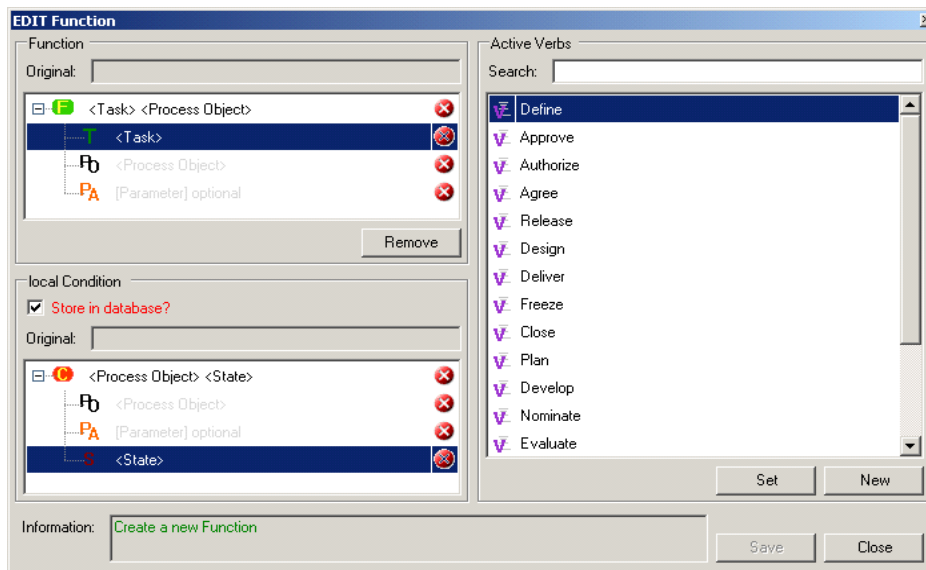


Abbildung 3.27.: Function Editor

Der Editor gliedert sich in die einleitend erwähnten Teilbereiche, mit der Vorlage für die zu bearbeitende **Function**, der Vorlage für das automatisch generierte Trivialereignis und der Liste mit den zu setzenden Teilelementen aus der Ontologie. Da die grundlegende Funktionalität der Editoren bereits einleitend beschrieben wurde, wird in diesem Kapitel lediglich auf die Besonderheiten beim Erstellen von **Functions** eingegangen.

Wie bereits bei der Beschreibung der Konzepte **Function** und **Condition** erwähnt (vgl. Kap. 3.4), wurde bei der Implementierung eine strikte Trennung zwischen Ontologie und **Process Dictionary** verfolgt. Da ein **Task** allerdings lediglich durch aktives Verb dargestellt wird, werden in der Liste der verfügbaren Elemente sämtliche aktiven Verben aus dem **Process Dictionary** aufgelistet. Der Grund für den Zugriff auf das **Process Dictionary** liegt darin, dass nicht für alle aktiven Verben entsprechende **Tasks** in der Ontologie enthalten sein müssen, da diese erst bei Bedarf angelegt werden. Die Aufbereitung eines **Tasks** im Rahmen des Editors erfolgt automatisch und bleibt nach außen hin verborgen. Dies entspricht der Vorgehensweise beim **Condition Editor**, bei welchem die passiven Verben aus dem **Process Dictionary** aufgelistet werden und die Aufbereitung des **States** automatisch erfolgt.

Der Editor ermöglicht die automatische Generierung von Trivialereignissen (bzw. von entsprechenden **Functions** beim Instanzieren bzw. Bearbeiten von **Conditions**). Die entsprechende **Condition** wird hierbei im Bereich "local Condition" angezeigt. Bevor diese generiert wird, wird geprüft, ob diese bereits in der Wissensbasis enthalten ist. Konnte die **Condition** gefunden werden, wird diese mit der **Function** verknüpft, anderenfalls wird eine neue Instanz angelegt.

Die Checkbox im Bereich "local Condition" liefert eine Reihe von Informationen, welche im Zusammenhang mit dem Trivialereignis stehen. Diese Informationen werden in Abbildung 3.28 zusammengefasst und im Rahmen der Ausführungen der folgenden Absätze beschrieben.



(a) kein lokaler Zustand    (b) Beziehung speichern    (c) bestehende Beziehung

Abbildung 3.28.: Function Editor - Checkbox lokales Element

Die automatisierte Generierung eines Trivialereignisses zu einer **Function** setzt voraus, dass zum **Task** ein entsprechendes passives Verb im **Process Dictionary** definiert ist, um den **State** ableiten zu können. Ist dieses passive Verb nicht verfügbar, wird bei der Zuweisung des **Task** ein entsprechender Hinweis angezeigt (siehe Abb. 3.29). Durch die Bestätigung dieses Dialogs wird die Komponente zur Verwaltung des **Process Dictionary** geöffnet und das benötigte passive Verb kann angelegt werden. Im Anschluss daran erfolgt eine Prüfung, ob ein passives Verb zum **Task** definiert wurde. Ist das der Fall, wird es der **Condition** als **State** zugewiesen, anderenfalls erfolgt eine neuerliche Benachrichtigung. Wird kein passives Verb angelegt, kann das Trivialereignis nicht generiert werden und die Checkbox wird dementsprechend angepasst (siehe Abb. 3.28.a). Analog dazu erfolgt die automatische Bildung der generierenden **Function** zu einer **Condition**.

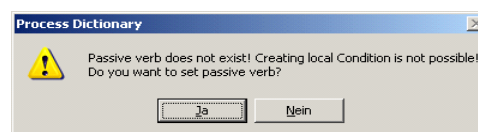


Abbildung 3.29.: Function Editor - fehlendes passives Verb

Kann das Trivialereignis automatisch generiert werden, d.h. es konnte ein entsprechendes passives Verb gefunden werden, ist es möglich auf die Aufbereitung zu verzichten (vgl. Abb. 3.28.b). Ausschlaggebend für diese Auswahlmöglichkeit ist, dass für **Conditions** nicht immer sinnvolle **Functions** generiert werden können (vgl. bspw. Endereignisse von Prozessmodellen, wie "Project Milestone reached"). Außerdem ermöglicht diese Checkbox das nachträgliche generieren von Trivialereignissen zu bestehenden **Functions**, wenn für diese noch keine entsprechende Beziehung definiert ist.

Die Checkbox in Abbildung 3.28.c wird schließlich angezeigt, wenn zu einer bestehenden **Function** eine Beziehung zu ihrem Trivialereignis definiert wurde. Da das Löschen dieser Beziehung als nicht sinnvoll erachtet wird, wird die Checkbox deaktiviert.

Nachdem eine **Function** vollständig gebildet wurde, d.h. es wurden zumindest ein **Task** und ein **Process Object** zugewiesen, kann diese mittels der *Save* Schaltfläche gespeichert werden.

### 3.5.5. Process Activity Editor

Der **Process Activity Editor** (siehe Abb. 3.30) ermöglicht die Instanzierung bzw. Bearbeitung von **ProcessActivities**. Auch dieser folgt dem Ansatz, bereits bestehendes Wissen zu nutzen und durch die Bereitstellung einer grundlegenden Vorlage fehlerhafte Eingaben zu vermeiden.

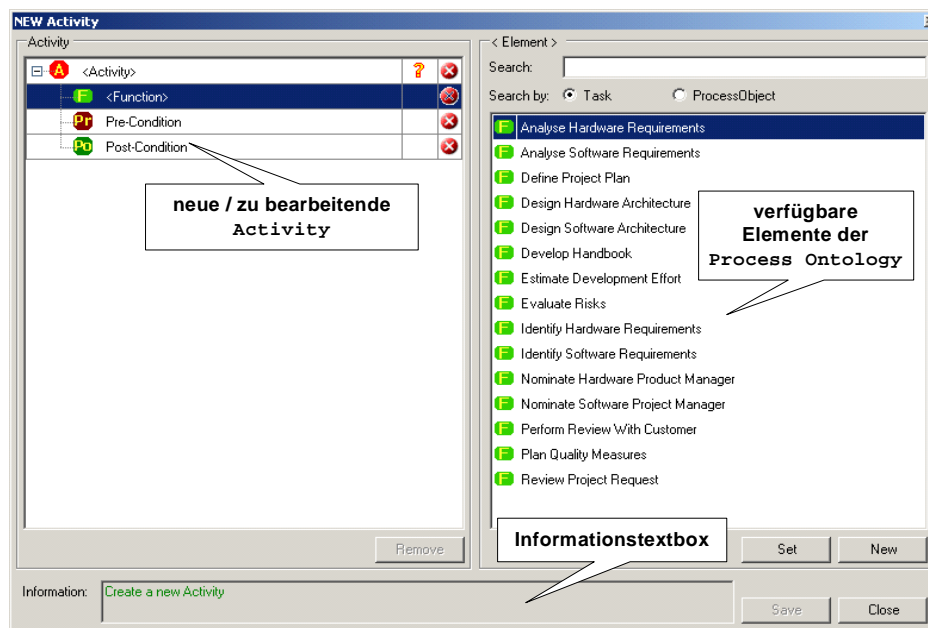


Abbildung 3.30.: Process Activity Editor

Im linken Bereich des Editors wird die Vorlage zum Anlegen einer **ProcessActivity** angezeigt, welche aus einer **Function**, der Pre- und der Post-Condition zusammengesetzt wird. Im rechten Bereich werden die bereits bestehenden **Functions** bzw. **Conditions** aufgelistet, welche entsprechend der Auswahl in der Vorlage geladen werden. Im unteren Bereich werden Informationen über den Modellierungsfortschritt bzw. Hinweise auf fehlerhafte Eingaben angezeigt.

Abbildung 3.31 zeigt die beiden Listen mit den **Functions** bzw. **Conditions**, deren Elemente einer **ProcessActivity** zugewiesen werden können. Es ist möglich

gezielt nach Elementen zu suchen, wobei die Suche auf die einzelnen Teilelemente eingeschränkt werden kann. Somit ist es möglich, **Functions** nach **Tasks** oder **Process Objects** und **Conditions** nach **Process Objects** oder **States** zu suchen. Ist das gesuchte Element nicht in der Wissensbasis enthalten, kann mit Hilfe der *New* Schaltfläche der entsprechende Editor zum Anlegen eines neuen Elements aufgerufen werden. Die Zuweisung eines Elements zu einer **ProcessActivity** erfolgt, wie bei den anderen Editoren, durch die *Set* Schaltfläche oder durch Doppelklick auf das gewünschte Element.

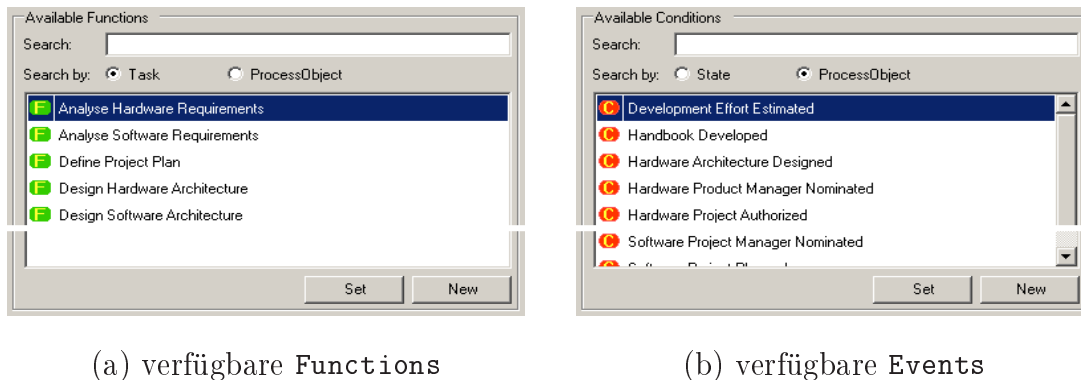



Abbildung 3.31.: Process Activity Editor - Listen verfügbarer Elemente

Da sich die Pre- bzw. Post-Condition aus mehreren **Conditions** zusammensetzen kann, muss es möglich sein, Konnektoren hinzuzufügen oder zu bearbeiten. Dies wird mittels eines Kontextmenüs realisiert, welches jedoch nur aufgerufen werden kann, wenn das Setzen eines Konnektors erlaubt ist (beispielsweise ist es im Rahmen dieses Editors nicht möglich einer **Condition** einen Konnektor zuzuweisen). Abbildung 3.32 zeigt dieses Kontextmenü, wobei lediglich jene Einträge auswählbar sind, welche zu diesem Zeitpunkt möglich sind (da in diesem Beispiel noch kein Konnektor gesetzt wurde, kann auch keiner verändert werden). Mit den ersten drei Einträgen kann ein neuer Konnektor hinzugefügt werden. Auf diese Weise ist es möglich beliebig tiefe Verschachtelungen zu definieren, um die benötigten Pre- bzw. Post-Conditions zu erstellen. Die unteren drei Einträge werden dazu verwendet, um einen bereits bestehenden Konnektorentyp zu verändern, wobei der zu diesem Zeitpunkt bestehende Typ nicht zur Auswahl steht. Es ist zu beachten, dass auf einen Konnektor immer mindestens zwei Elemente folgen müssen. Dies können **Conditions**, weitere Konnektoren oder beliebige Kombinationen von diesen sein. Wird diese Bedingung nicht erfüllt, wird dies in der Informationstextbox angezeigt und es ist nicht möglich die **ProcessActivity** zu speichern.

Wie bereits dargestellt, werden **Conditions**, wenn sie mit einer **Function** in Beziehung gesetzt werden, durch einen **ContextType** näher spezifiziert (vgl. Kap. 3.4). Dies erfolgt beim Anlegen einer **ProcessActivity** automatisch, sobald eine **Function** zugewiesen bzw. ausgetauscht wird, auf welche sich die **Conditions** beziehen können. Angezeigt werden diese Typen durch die beiden Icons  für einen nicht

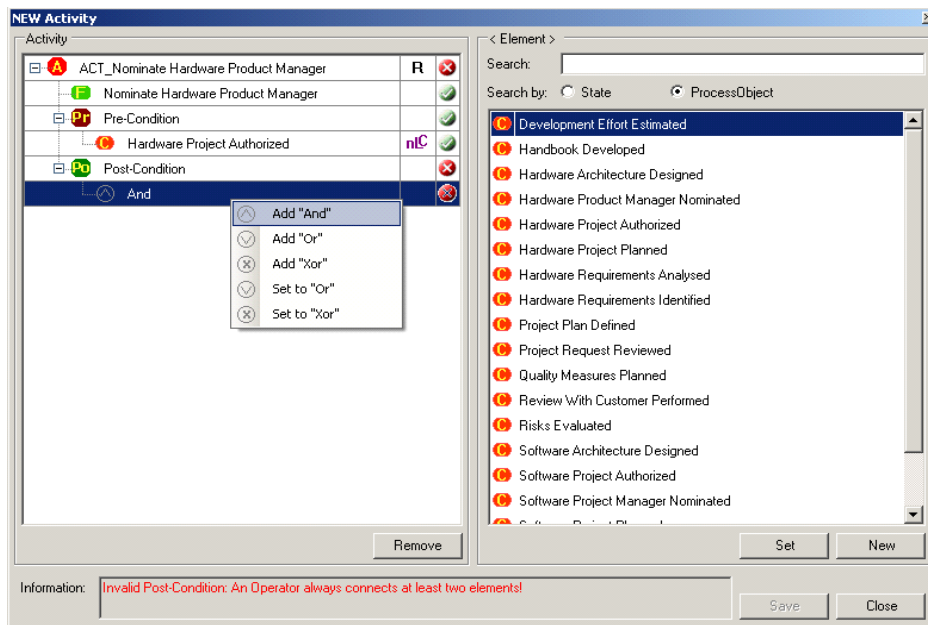



Abbildung 3.32.: Process Activity Editor - Kontextmenü logische Operatoren

lokalen und  für einen lokalen Kontext, welche neben der entsprechenden **Condition** eingeblendet werden.

Neben den **Conditions** wird auch der **ProcessActivity** selbst ein Typ zugewiesen, welcher die **Function** als eine "normale"- (**R**) oder Entscheidungsfunktion (**D**) kennzeichnet. Auch dieser Typ wird automatisch gesetzt, indem die bereits modellierte Post-Condition betrachtet wird. Wird in dieser ein Konnektor vom Typ **Or** oder **Xor** identifiziert, handelte es sich um eine Entscheidungsfunktion, anderenfalls um eine "normale" **Function**.

Um eine **ProcessActivity** in der Wissensbasis speichern zu können, wird die **Save** Schaltfläche verwendet. Diese wird nur dann aktiv, wenn alle benötigten Elemente gesetzt wurden und, falls vorhanden, auf alle Operatoren zumindest zwei Elemente folgen. Des Weiteren ist es wieder nicht möglich, eine bereits bestehende **ProcessActivity** zu speichern, da diese lediglich einmal in der Wissensbasis enthalten sein kann.

## 4. Semantische Annotierung von EPK-Modellen

Im vorangegangenen Kapitel wurde die im Rahmen dieser Arbeit entwickelte Ontologie vorgestellt, welche die Konzepte zur Annotierung von EPK-Funktionen und -Ereignissen bereitstellt, um eine automatisierte Identifikation von Prozessmustern gemäß des pModeler-Ansatzes zu ermöglichen. Aufgrund der großen Anzahl an Prozessmodellen, welche zumeist in Unternehmen vorliegen, ist eine manuelle Annotierung dieser Modelle eine zeitaufwändige und komplexe Aufgabe. Durch diesen Umstand motiviert, wird nun in diesem Kapitel ein Lösungsansatz vorgestellt, welcher eine automatisierte Instanzierung der Konzepte der Ontologie aus bestehenden EPK-Modellen ermöglicht.

Zuerst werden in Kapitel 4.1 die in den Prozessmodellen identifizierten Unklarheiten dargestellt, welche mit Hilfe der semantischen Annotierung aufgelöst werden sollen. Anschließend wird in Kapitel 4.2 die Vorgehensweise bei der automatisierten Annotierung der EPK-Funktionen und -Ereignisse beschrieben, welche außerdem die Ermittlung von Generalisierungsbeziehungen zwischen `Process Objects`, die Generierung nicht modellierter Trivialereignisse und die Aufspaltung zusammengesetzter Modellelemente umfasst. In Kapitel 4.3 wird die automatisierte Aufbereitung der `ProcessActivities` beschrieben, wobei in diesem Bereich die Ermittlung der Kontextinformationen von wesentlicher Bedeutung ist. Abschließend werden in Kapitel 4.4 die im Rahmen des Prototypen umgesetzten Komponenten zur Darstellung der Ergebnisse der semantischen Annotierung der EPK-Modelle vorgestellt.

### 4.1. Unklarheiten in EPK-Modellen

Grundlage für die semantische Annotierung dieser Arbeit bilden Prozessbeschreibungen, welche mit Hilfe von EPKs modelliert wurden. Aufgrund des hohen Freiheitsgrads bei der Modellierung von EPK-Modellen, liegen häufig Prozessmodelle mit syntaktischen und semantischen Unklarheiten vor, welche die Vergleichbarkeit dieser Modelle erschweren.

Durch die Analyse unterschiedlicher EPK-Modelle aus der Praxis konnten eine Reihe von Unklarheiten identifiziert werden, welche in Abbildung 4.1 zusammengefasst werden:

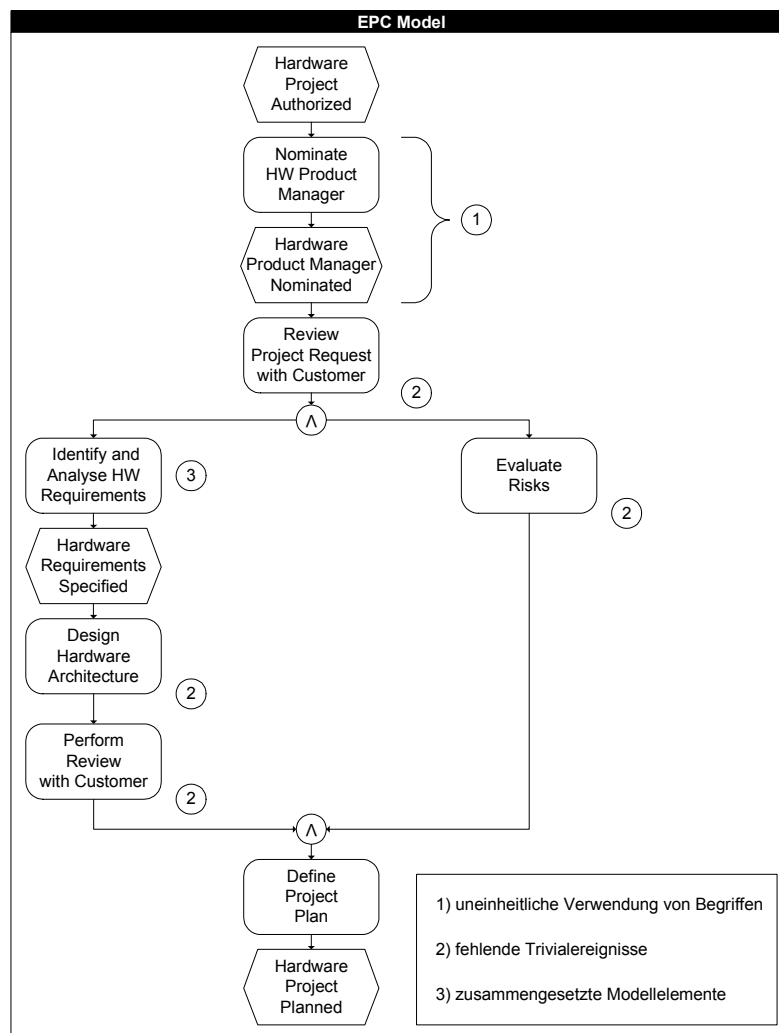


Abbildung 4.1.: Unklarheiten in EPK-Modellen

1. uneinheitliche Verwendung von Begriffen: als häufigstes Problem konnte die Verwendung unterschiedlicher Begriffe mit derselben Bedeutung identifiziert werden. Typisches Beispiel bildet die uneinheitliche Nutzung von Abkürzungen und ausgeschriebenen Begriffen. Zur Vereinheitlichung der Bezeichnungen der Instanzen der Ontologie werden bei der semantischen Aufbereitung die im **Process Dictionary** definierten Stammwörter verwendet.
2. nicht modellierte Trivialereignisse: diese werden aufgrund der Übersichtlichkeit häufig weggelassen. Da die Ereignisse allerdings für die Aufbereitung der Prozessaktivitäten sowie für die Ableitung der Prozessziele benötigt werden, werden diese automatisiert generiert.
3. zusammengesetzte Modellelemente: in den analysierten EPK-Modellen wurden teilweise mehrere Tätigkeiten in einer Funktion modelliert. Um die Vergleich-



barkeit der EPK-Modelle zu erhöhen, werden diese bei der Instanzierung der Ontologie in einzelne Funktionen zerlegt.

## 4.2. Semantische Annotierung von EPK-Funktionen und -Ereignissen

In diesem Kapitel wird die Vorgehensweise bei der automatisierten Aufbereitung der EPK-Funktionen und -Ereignisse entsprechend der Konzepte und Beziehungen der Ontologie beschrieben. Grundlage für die Instanzierung der Ontologie bildet die lexikalische Aufbereitung der EPK-Modelle, welche im **Process Dictionary** verwaltet wird. Die lexikalische Aufbereitung zerlegt die EPK-Modellelemente syntaktisch in die einzelnen Wörter, klassifiziert diese anhand ihres Worttyps und stellt eine Reihe von Beziehungen zwischen den Wörtern bereit, welche die Standardisierung der Instanzen der Ontologie sowie die automatisierte Generierung von nicht modellierten Trivialereignissen unterstützen (vgl. Kap. 3.3).

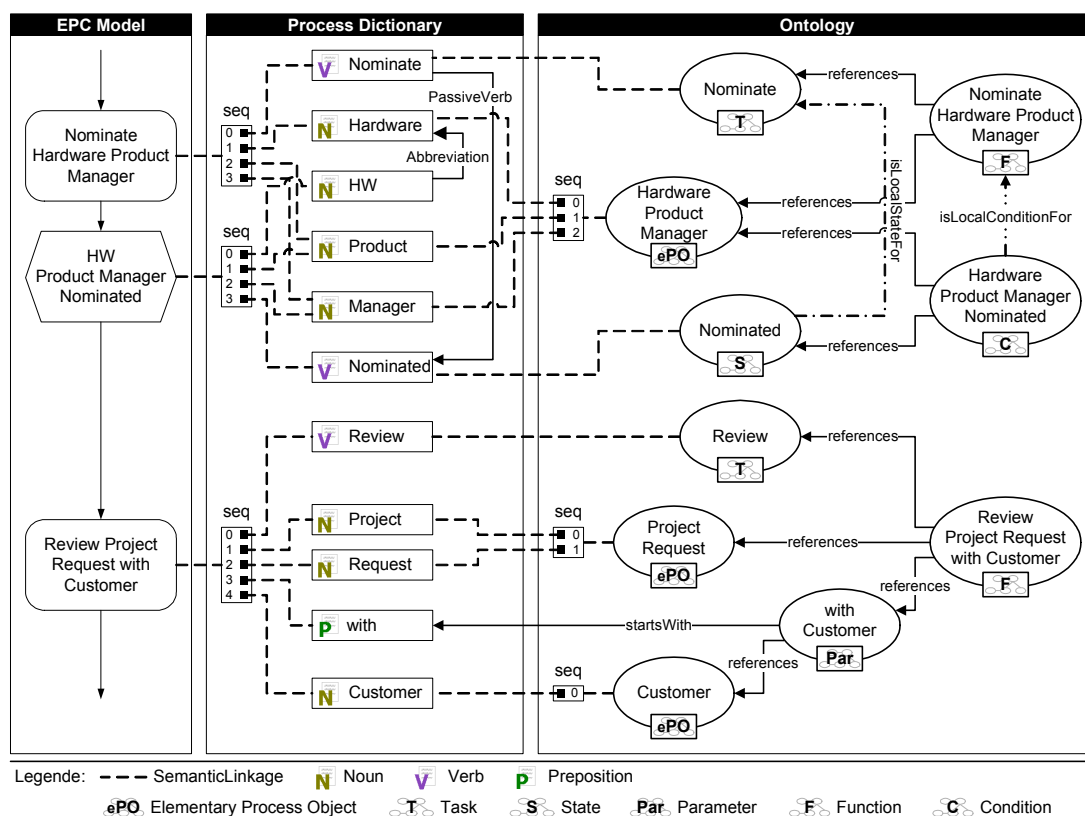


Abbildung 4.2.: Beispiel: Aufbereitung EPK-Funktion und -Ereignis

Abbildung 4.2 veranschaulicht den Zusammenhang zwischen EPK-Modell, **Process Dictionary** und Ontologie anhand eines Beispiels, welches die Beschreibung

der Vorgehensweise der automatisierten Annotierung begleitet wird. Während im linken Bereich ein Ausschnitt aus einem Prozessmodell dargestellt wird, zeigt der mittlere Bereich den entsprechenden Auszug aus dem **Process Dictionary** mit den aus den Bezeichnungen extrahierten Wörtern, deren Reihenfolge und Klassifizierung anhand ihres Worttyps. Des weiteren werden zwei Beziehungstypen des **Process Dictionary** dargestellt. Zum einen die Beziehung *Passive Verb*, welche "Nominated" als passives Verb zu "Nominate" ausweist und zum anderen die Beziehung "Abbreviation", welche "HW" als Abkürzung für "Hardware" definiert. Der rechte Bereich zeigt schließlich die entsprechenden Instanzen der Ontologie, welche auf Basis der einzelnen Wörter aus den Modellelementen extrahiert werden konnten. Wie auch bei der lexikalischen Aufbereitung ist bei den **Process Objects** die Reihenfolge der Wörter relevant (Anm. da zusammengesetzte Verben, wie bspw. "set up", als einfache Einträge im **Process Dictionary** verwaltet werden, ist auch für die Konzepte **Task** bzw. **State** eine einfache Referenz ausreichend).

Eine grundsätzliche Anforderung an die automatisierte Aufbereitung der Prozessmodelle liegt darin, gleiche Instanzen in der Ontologie lediglich einmal zu erfassen (vgl. **Process Object** "Hardware Product Manager" in Abb. 4.2). Um diese Anforderung zu erfüllen, arbeitet der Analysealgorithmus bei der Instanzierung der Teilkonzepte sowie bei der Aufbereitung der **Functions**, **Conditions** und **ProcessActivities** entsprechend dem Vorgehensmodell in Abbildung 4.3. Zuerst wird geprüft, ob das Element bereits im vorliegenden Modell aufgetreten ist. Wurde es noch nicht identifiziert, erfolgt eine Prüfung der bestehenden Wissensbasis. Konnte auch in dieser keine Übereinstimmung gefunden werden, wird eine neue Instanz angelegt und in der Ontologie integriert. Des weiteren werden die einzelnen Instanzen mit der Häufigkeit des Auftretens in den bestehenden Prozessmodellen belegt.

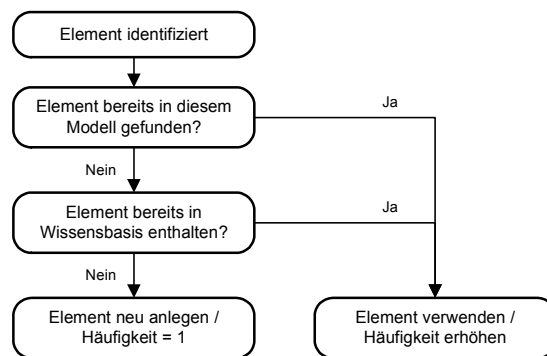


Abbildung 4.3.: Vorgehensweise: Element-Identifikation

Bei der Aufbereitung eines Prozessmodells werden zuerst alle Elemente einzeln untersucht und entsprechend der Konzepte der Ontologie aufbereitet. Hierfür erfolgt zunächst eine Unterscheidung des Modellelementtyps, wobei in der derzeitigen pModeler Ausbaustufe lediglich EPK-Funktionen und -Ereignisse analysiert werden. Der Analysealgorithmus untersucht die einzelnen Wörter der Modellelementbezeichnung entsprechend ihrer Reihenfolge (vgl. *seq* im mittleren Bereich in Abb. 4.2) und

weist diese anhand ihres Worttyps dem entsprechenden Teilkonzept zu. Für diese Zuweisung liegt dem Algorithmus eine Zuordnungsvorschrift für die einzelnen Worttypen zugrunde, welche in Tabelle 4.1 zusammengefasst wird. Diese Informationen ermöglichen es dem Analysealgorithmus, die einzelnen Teilkonzepte, wie bspw. **Process Objects** oder **Tasks** zu identifizieren und in einem weiteren Schritt zu den **Functions** und **Conditions** der Ontologie zusammenzufassen.

Worttyp	Semantik
Verb	EPK-Funktion: <b>Task</b> (Tätigkeit, welche auf ein <b>Process Object</b> ausgeführt werden kann) EPK-Ereignis: <b>State</b> (Zustand, in welchem sich das <b>Process Object</b> befinden kann)
Nomen Adjektive	Abfolgen von Nomen und Adjektiven werden zu einem <b>Process Object</b> zusammengefasst.
Artikel	Grundsätzlich Teil eines <b>Process Object</b> , wird allerdings bei der Aufbereitung weggelassen, da er keinen informativen Mehrwert enthält.
Präposition	Auslöser für die Bildung eines <b>Parameter</b> , wird mit dem folgenden <b>Process Object</b> zum <b>Parameter</b> zusammengefasst.
Konjunktion	veranlasst den Algorithmus die EPK-Funktion bzw. das EPK-Ereignis aufzuspalten (für eine detaillierte Beschreibung dieser Zerlegung vgl. Tabelle 4.2).

Tabelle 4.1.: Zuordnungsvorschrift von Worttypen

In den folgenden Absätzen wird nun die semantische Aufbereitung der EPK-Funktionen und -Ereignisse detailliert dargestellt und anhand von Beispielen veranschaulicht. Neben der grundlegenden Vorgehensweise bei der Extraktion der Teilkonzepte und der Aufbereitung der **Functions** und **Conditions** werden auch die Erweiterungen des Analysealgorithmus beschrieben, zu welchen die automatisierte Aufbereitung der Generalisierungs- und Migrationsbeziehungen der **Process Objects** sowie die automatisierte Generierung der nicht modellierten Trivialereignisse zählen.

Wird bei der Analyse der Modellelementbezeichnung ein Wort vom Typ *Verb* identifiziert, wird entsprechend der Zuordnungsvorschrift ein **Task** bzw. ein **State** instanziiert. Da das **Process Dictionary** aktive und passive Verben nicht explizit unterscheidet, wird das entsprechende Konzept aus dem Modellelementtyp (EPK-Funktion oder -Ereignis) abgeleitet.

**Beispiel:**

*Bei der EPK-Funktion und dem EPK-Ereignis in Abbildung 4.2 werden sowohl "Nominate" als auch "Nominated" lediglich als Verb klassifiziert. Aufgrund des Modellelementtyps kann jedoch abgeleitet werden, dass "Nominate" als Task und "Nominated" als State aufbereitet werden muss.*

Identifiziert der Algorithmus ein Wort vom Typ *Nomen*, *Adjektiv* oder *Artikel*, wird die Aufbereitung eines **Process Objects** eingeleitet. Hierbei werden die folgenden Wörter solange in das **Process Object** übernommen, bis entweder das Ende der Modellelementbezeichnung erreicht wird oder ein Wort vom Typ *Präposition* (Bildung eines **Parameters**), *Konjunktion* (Aufspaltung des Modellelements) oder *Verb* identifiziert wird.

**Beispiel:**

*Bei der Aufbereitung der EPK-Funktion "Nominate Hardware Product Manager" identifiziert der Algorithmus das Wort "Hardware" als Nomen und beginnt mit der Bildung eines **Process Objects**, indem diesem eine Referenz auf das Wort "Hardware" mit `seq=0` zugewiesen wird. Analog wird mit den beiden folgenden Nomen "Product" (`seq=1`) und "Manager" (`seq=2`) verfahren. Da daraufhin das Ende der Modellelementbezeichnung erreicht wurde, wird die Bildung des **Process Objects** "Hardware Product Manager" abgeschlossen (vgl. Vorgehensweise in Abb. 4.3).*

Da ein Artikel keinen informativen Mehrwert darstellt, wird dieser nicht in ein **Process Object** übernommen. Diese Vernachlässigung des Artikels trägt zur Standardisierung bei, da auf diese Weise die Semantik gleichbedeutender **Process Objects**, welche sich lediglich aufgrund des Artikel unterscheiden, explizit gemacht werden kann.

**Beispiel:**

*Die EPK-Funktionen "Define Project Plan" und "Define the Project Plan" enthalten dieselbe Information. Durch die Vernachlässigung des Artikels wird bei der Aufbereitung das entsprechende **Process Object** "Project Plan" lediglich einmal instanziiert.*

Eine weitere Standardisierung der **Process Objects** erfolgt durch die Verwendung der, im Rahmen der lexikalischen Aufbereitung definierten Stammwörter (vgl. Kap. 3.3). Auf diese Weise können semantisch gleichbedeutende **Process Objects** identifiziert werden, welche lediglich durch syntaktische Unkorrektheiten nicht explizit dargestellt werden. Diese Standardisierung unterstützt die einheitliche Verwendung von Begrifflichkeiten bei der Instanzierung der Ontologie, welche bereits im Rahmen der Bearbeitung von **Process Objects** verfolgt wurde (vgl. Kap. 3.5.2.1).

**Beispiel:**

*Bereits bei der lexikalischen Aufbereitung wurde für die Abkürzung "HW" des EPK-*

Ereignisses "HW Product Manager Nominated" das Stammwort "Hardware" definiert (siehe Abb. 4.2), welches für die Aufbereitung des `Process Objects` verwendet wird. Unter Berücksichtigung der Reihenfolge der einzelnen Wörter wird daher das `Process Object` "Hardware Product Manager" gebildet. Da bereits bei der Aufbereitung der EPK-Funktion "Nominate Hardware Product Manager" ein entsprechendes `Process Object` identifiziert wurde, wird dem EPK-Ereignis ebenfalls diese Instanz zugewiesen.

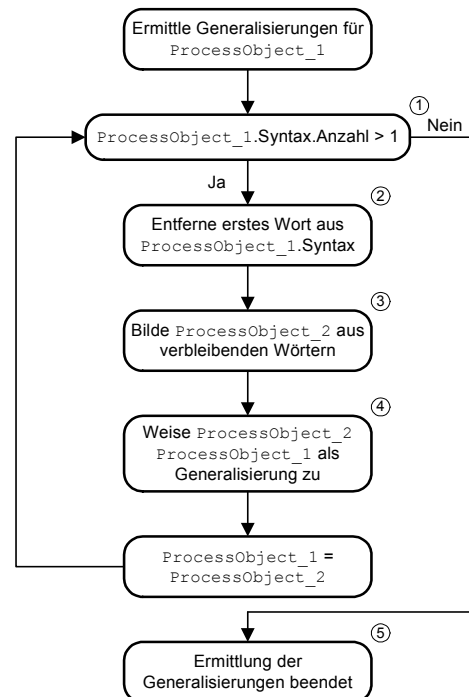


Abbildung 4.4.: Rekursion für die Ermittlung der Generalisierungshierarchien

Neben der semantischen Aufbereitung und Standardisierung der `Process Objects` werden die Generalisierungshierarchien automatisiert ermittelt. Bei der Identifikation dieser Hierarchien bedient sich der Analysealgorithmus der Besonderheit der englischen Sprache, in welcher auch zusammengesetzte Begriffe nicht zusammengeschieden werden. Da bei der Aufbereitung der `Process Objects` auch deren syntaktische Repräsentation erfasst wird, bildet diese die Grundlage für den Aufbau der Generalisierungshierarchien. Abbildung 4.4 zeigt die Vorgehensweise bei der Identifizierung der Generalisierungen zu einem `Process Object`. Umfasst die syntaktische Darstellung des `Process Objects` mehr als ein Wort (1), wird eine Rekursion gestartet. Innerhalb dieser Rekursion wird das erste Wort der syntaktischen Repräsentation entfernt (2) und aus den verbleibenden Wörtern wird ein `Process Object` gebildet (3). Dieses wird durch den Typ `Generalized Process Object` beschrieben und dem ersten bzw. dem zuvor gebildeten `Process Object` als Generalisierung zugewiesen (4). Dieser Aufruf wird solange wiederholt, bis lediglich ein Wort übrig ist (5). Jede Instanzierung eines solchen `Process Objects` folgt

dem allgemeinen Vorgehensmodell in Abbildung 4.3, d.h. es werden das vorliegende Modell bzw. die Wissensbasis nach bereits bestehenden Instanzen durchsucht. Die Häufigkeit der auf dieser Weise erhobenen **Process Objects** wird nicht erhöht, da sie nicht direkt in einem bestehenden Modell vorkommen.

### Beispiel:

*Das **Process Object** "Hardware Product Manager" in Abbildung 4.2 setzt sich aus drei Wörtern zusammen. Der Analysealgorithmus entfernt in einem ersten Schritt das erste Wort "Hardware" und bildet das generalisierte **Process Object** "Product Manager", welches durch den Typ **Generalized Process Object** beschrieben und "Hardware Product Manager" als Generalisierung zugewiesen wird. Anschließend wird von "Product Manager" wiederum das erste Wort entfernt und das **Process Object** "Manager" gebildet. Dieses wird ebenfalls durch den entsprechenden Typ beschrieben und "Product Manager" als Generalisierung zugewiesen. Da "Manager" nun aus lediglich aus einem Wort besteht, beendet der Algorithmus die Aufbereitung der Generalisierungshierarchie.*

Zusätzlich zu den Generalisierungshierarchien werden die Migrationsbeziehungen zwischen den **Process Objects** automatisiert identifiziert. Diese werden erst dann ermittelt, wenn alle Modellelemente des vorliegenden Prozessmodells verarbeitet wurden, da zu diesem Zeitpunkt sämtliche **Process Objects** verfügbar sind. Der Definition Rosemanns entsprechend (vgl. Kap. 2.3.3), beschreiben Migrationsbeziehungen, wie sich im Verlauf eines Prozessmodells die zu bearbeitenden **Process Objects** verändern. Bei der Ermittlung werden daher immer Paare von aufeinander folgenden Modellelemente betrachtet. Da dasselbe **Process Object** nicht auf sich selbst migrieren kann, werden die folgenden Modellelemente solange untersucht, bis entweder ein anderes **Process Object** gefunden oder das Ende des Prozessmodells erreicht wurde.

Abbildung 4.5 zeigt im linken Bereich einen Ausschnitt aus einem Prozessmodell und im rechten Bereich die annotierten **Process Objects** aus der Ontologie. Neben den **Process Objects** und deren Zuweisung zu den Modellelementen werden auch die automatisiert ermittelten Generalisierungshierarchien (vgl. bspw. "Hardware Requirements" *isA* "Requirements") und Migrationsbeziehungen (vgl. bspw. "Hardware Requirements" *migratesTo* "Hardware Architecture") dargestellt. Außerdem wird die Typisierung zwischen dem **Process Object** und dem nativen Modellelement ersichtlich. Im Unterschied zu den übrigen **Process Objects** wird auf diese Weise das **Process Object** "Customer" als Teil eines **Parameters** ausgewiesen.

Wie bereits bei der Beschreibung der Konzepte der Ontologie dargestellt (vgl. Kap. 3.4), können EPK-Funktionen und -Ereignisse durch einen **Parameter** näher beschrieben werden. Dieser **Parameter** wird gemäß der dieser Arbeit zugrunde liegenden Reglementierung der Worttypen durch eine *Präposition* eingeleitet und durch das darauf folgende **Process Object** beschrieben.

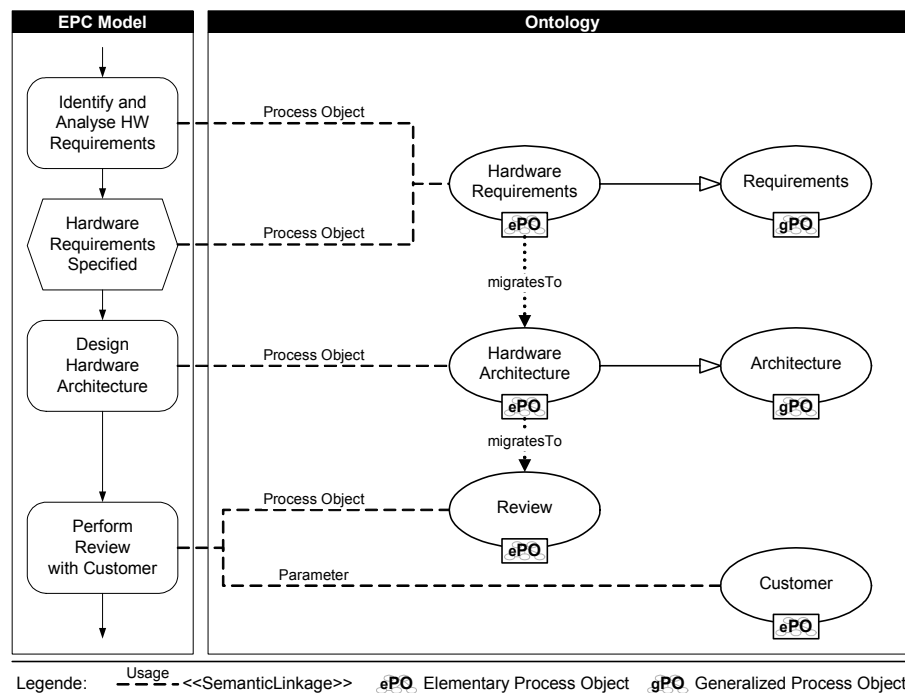


Abbildung 4.5.: Process Objects, Generalisierungen und Migrationen

**Beispiel:**

Bei der Aufbereitung der EPK-Funktion "Perform Review with Customer" identifiziert der Analysealgorithmus das Wort "with". Aus dem zugewiesenen Worttyp Präposition wird abgeleitet, dass ein Parameter instanziiert werden muss. Nachdem das auf die Präposition folgende Process Object "Customer" gebildet wurde, werden die beiden Teile zu einem Parameter "with Customer" zusammengefasst.

Nachdem die gesamte Bezeichnung eines Modellelements gemäß der lexikalischen Aufbereitung verarbeitet wurde, werden die extrahierten Teilkonzepte entsprechend des Modellelementtyps zu einer Function bzw. einer Condition der Ontologie zusammengefasst. Während bei einer EPK-Funktion ein Task, ein Process Object und gegebenenfalls ein Parameter zu einer Function verknüpft werden, werden bei einem EPK-Ereignis ein Process Object, ein Parameter, falls vorhanden, und ein State zu einer Condition zusammengefasst.

**Beispiel:**

In Abbildung 4.2 wurden aus der EPK-Funktion "Nominate Hardware Product Manager" der Task "Nominate" und das Process Object "Hardware Product Manager" extrahiert, welche zu einer entsprechenden Function zusammengefasst werden. Die Condition "Hardware Product Manager Nominated" zum EPK-Ereignis verweist auf dasselbe Process Object und auf den State "Nominated" aus der Ontologie.

Nachdem alle EPK-Funktionen und -Ereignisse des vorliegenden Prozessmodells

aufbereitet wurden, werden die *isLocalConditionFor*-Beziehungen ermittelt, indem ausgehend von einer **Function** nach dem zugehörigen Trivialereignis gesucht wird. Konnte weder im vorliegenden Modell noch in der Wissensbasis eine entsprechende **Condition** gefunden werden, wird diese automatisiert generiert. Grundlage für die Aufbereitung des Trivialereignisses bilden das **Process Object** und, falls vorhanden, der **Parameter** der **Function**. Für den **State** wird im **Process Dictionary** nach dem passiven Verb zum **Task** der **Function** gesucht. Das passive Verb wird hierbei durch den Beziehungstypen **PassiveVerb** aus dem **Process Dictionary** festgelegt.

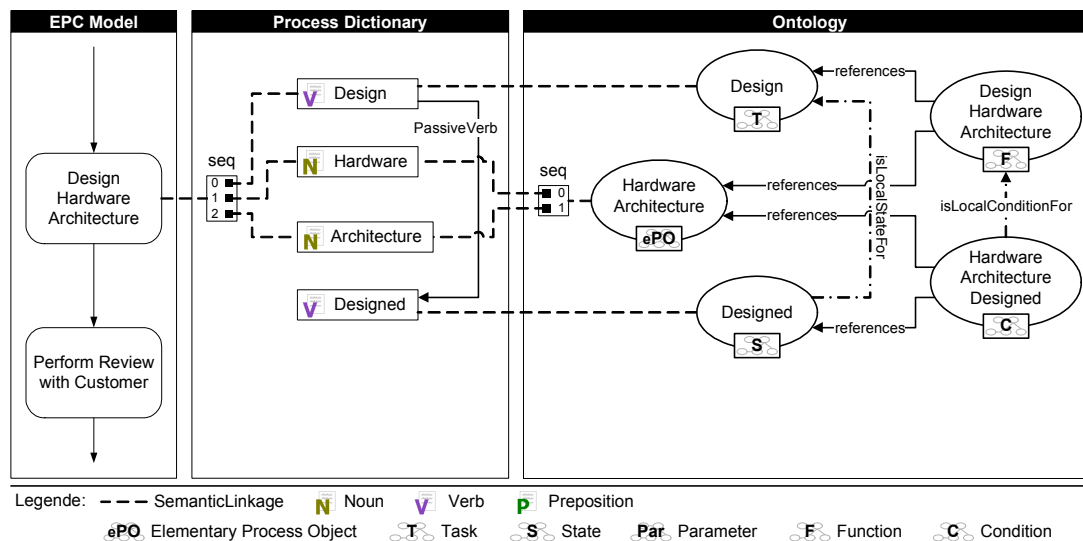


Abbildung 4.6.: Beispiel: automatisiert generiertes Trivialereignis

### Beispiel:

Für die EPK-Funktion "Design Hardware Architecture" in Abbildung 4.6 konnte kein entsprechendes Trivialereignis im bestehenden Prozessmodell gefunden werden. Der Analysealgorithmus durchsucht daher das **Process Dictionary** nach der passiven Verbform von "Design". Konnte mittels der **PassiveVerb**-Beziehung das entsprechende passive Verb "Designed" gefunden werden, wird aus diesem ein **State** gebildet, welcher mit dem **Process Object** "Hardware Architecture" der **Function** zum entsprechenden Trivialereignis "Hardware Architecture Designed" zusammengefasst wird. Diese **Condition** wird nun mittels des Beziehungstyps *isLocalConditionFor* mit der **Function** verknüpft.

### Sonderfall: zusammengesetzte Modellelemente

Neben der Aufbereitung der bestehenden Modellelemente, wird auch das Zerlegen von zusammengesetzten Modellelementen unterstützt. Sobald ein Wort vom Typ Konjunktion identifiziert wird, wird eine Zerlegung initiiert. Je nach Position der



Konjunktion, können aus einem Modellelement unterschiedliche Kombinationen der Teilkonzepte entstehen.

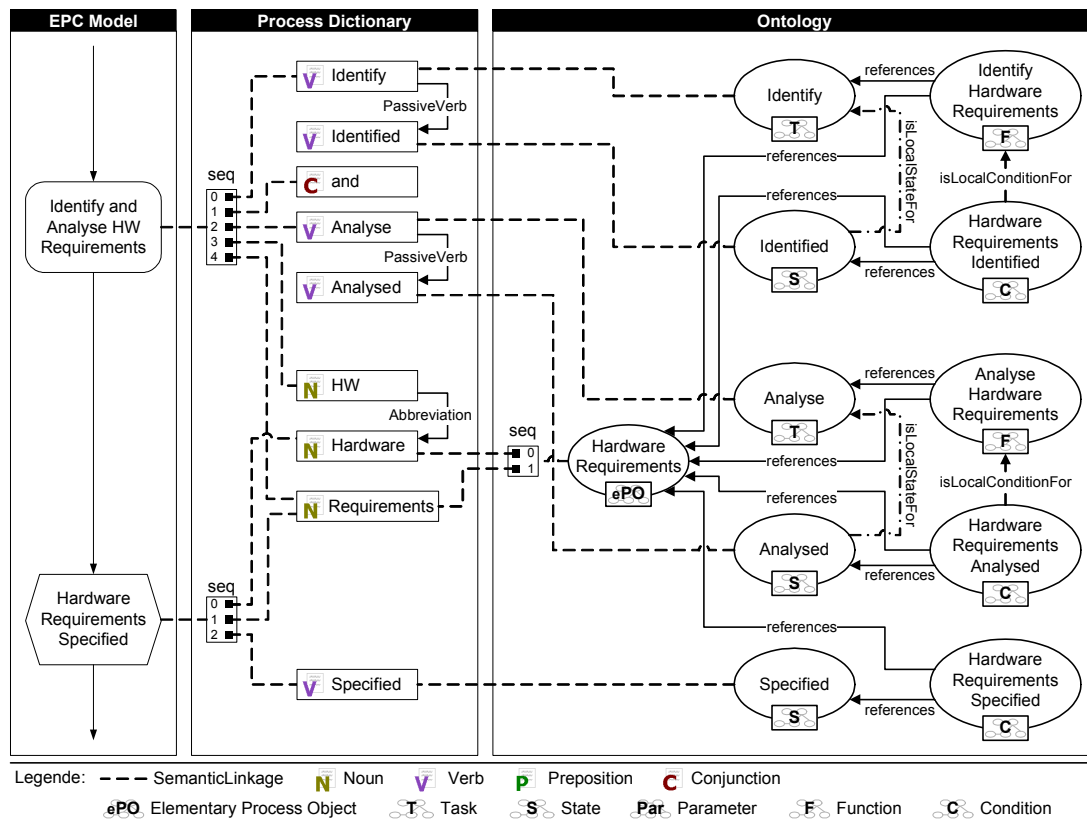


Abbildung 4.7.: Beispiel einer zusammengesetzten EPK-Funktion

### Beispiel: zusammengesetzte EPK-Funktion

Abbildung 4.7 zeigt *bspw.* eine zusammengesetzte EPK-Funktion. Mittels der EPK-Funktion "Identify and Analyse Hardware Requirements" wurden zwei Tätigkeiten in einem Modellelement modelliert. Bei der Aufbereitung wird die EPK-Funktion nun in die beiden **Functions** "Identify Hardware Requirements" bzw. "Analyse Hardware Requirements" zerlegt, welche dieser hinterlegt werden. Wie auch bei der herkömmlichen Vorgehensweise werden bei Bedarf zu beiden **Functions** die lokalen **Conditions** generiert.

In Tabelle 4.2 werden verschiedene zusammengesetzte EPK-Funktionen und deren Trennung anhand von Beispielen aufgelistet. Unter den Beispielen wird die Identifikationsreihenfolge der einzelnen Teilkonzepte dargestellt. Bei dieser Zerlegung wird allerdings die Semantik der Konjunktion nicht berücksichtigt und es wird eine sequenzielle Abfolge der beiden Tätigkeiten unterstellt. In den folgenden Absätzen werden nun die einzelnen Beispiel erläutert.


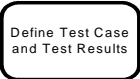
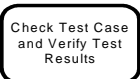
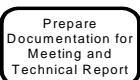
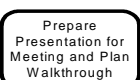
EPK-Funktion	aufbereitete Funktionen
a) 	"plan use case" "create use case"  <Task> Konj. <Task> <Process Object>
b) 	"define test case" "define test results"  <Task> <Process Object> Konj. <Process Object>
c) 	"check test case" "verify test results"  <Task> <Process Object> Konj. <Task> <Process Object>
d) 	"prepare documentation for meeting" "prepare documentation for technical report"  <Task> <Process Object> <Parameter> Konj. <Parameter>
e) 	"prepare presentation for meeting" "plan walkthrough"  <Task> <Process Object> <Parameter> Konj. <Task> <Process Object>

Tabelle 4.2.: Zerlegung zusammengesetzter Funktionen

- a) Der Algorithmus identifiziert den **Task** "plan" und weist diesen einer **Function** zu. Es folgt eine Konjunktion und für den darauf folgenden **Task** "create" wird eine zweite **Function** angelegt. Das folgende **Process Object** "use case" wird schließlich beiden **Functions** zugewiesen ("plan use case" und "create use case").
- b) Der Algorithmus identifiziert den **Task** "define" und das **Process Object** "test case". Da darauf eine Konjunktion folgt, wird die erste **Function** "define test case" gebildet. Auf die Konjunktion folgt das **Process Object** "test results" und eine weitere **Function** wird angelegt. Dieser wird ebenfalls der **Task** "define" zugewiesen, wodurch die zweite **Function** "define test results" gebildet wird.
- c) Der Algorithmus identifiziert den **Task** "check" und das **Process Object** "test case", aus welchen die erste **Function** "check test case" gebildet wird. Da auf die Konjunktion ein weiterer **Task** ("verify") und ein weiteres **Process Object** ("test results") folgen, wird eine neue **Function** "verify test results" zusammengesetzt.

- d) Der Algorithmus identifiziert den **Task** "prepare", das **Process Object** "documentation" und den **Parameter** "for meeting" und bildet daraus die entsprechende **Function**. Da vor der Konjunktion ein **Parameter** identifiziert wurde, wird aus dem folgenden **Process Object** der **Parameter** "for technical report" gebildet (d.h. es wird unterstellt, dass auf die Konjunktion ein weiterer **Parameter** folgt). Der **Task** und das **Process Object** werden aus der ersten **Function** übernommen, woraus die zweite **Function** "prepare documentation for technical report" zusammengesetzt wird.
- e) Der Algorithmus identifiziert wiederum einen **Task**, ein **Process Object** und einen **Parameter** und bildet daraus die **Function** "prepare presentation for meeting". Da auf die Konjunktion ein **Task** und ein weiteres **Process Object** folgen, wird die **Function** "plan walkthrough" gebildet.

### 4.3. Semantische Aufbereitung von Prozessaktivitäten

Im vorangegangenen Kapitel wurde die Vorgehensweise bei der automatisierten Aufbereitung der EPK-Funktionen und Ereignisse beschrieben, welche die semantisch annotierten **Functions** und **Conditions** als Ergebnis liefert. Diese stellen die wesentlichen Bestandteile der Prozessaktivitäten dar und bilden daher die Grundlage für deren automatisierte Aufbereitung.

Abbildung 4.8 zeigt die semantische Annotierung der Prozessaktivitäten anhand eines Beispiels. Während im linken Bereich der Abbildung ein Ausschnitt aus einem EPK-Modell dargestellt wird, zeigt der rechte Bereich den entsprechenden Auszug aus der Ontologie. In der Abbildung werden die den EPK-Modellelementen annotierten **Functions** und **Conditions** dargestellt, welche zu den **ProcessActivities** zusammengefasst werden. Eine **ProcessActivity** referenziert eine **Function** und wird mittels der Beziehungen *hasPreCondition* und *hasPostCondition* mit ihren Start- bzw. Endereignissen in Beziehung gesetzt. Da den **ProcessActivities** in diesem Beispiel lediglich einfache Kontextinformationen zugewiesen werden, wurde der Übersichtlichkeit wegen auf die Darstellung der **Context**-Konzepte zwischen den **Conditions** und **ProcessActivities** verzichtet.

In den folgenden Absätzen wird nun die Vorgehensweise bei der Aufbereitung der Prozessaktivitäten beschrieben, bei welcher eine EPK-Funktion mit ihren Start- und Endereignissen verknüpft wird. Einen wesentlichen Teil dieses Analyseschritts bildet daher die Ermittlung dieser Kontextinformationen, welche in weiterer Folge detailliert dargestellt wird. Hierfür wird zunächst die zu Grunde liegende Rekursion für die Identifikation der Kontextinformation auf allgemeiner Basis vorgestellt, bevor auf die spezifischen Eigenheiten bei der Bildung der Pre- bzw. Post-Conditions näher eingegangen wird. Neben der konzeptionellen Betrachtung der Vorgehenswei-

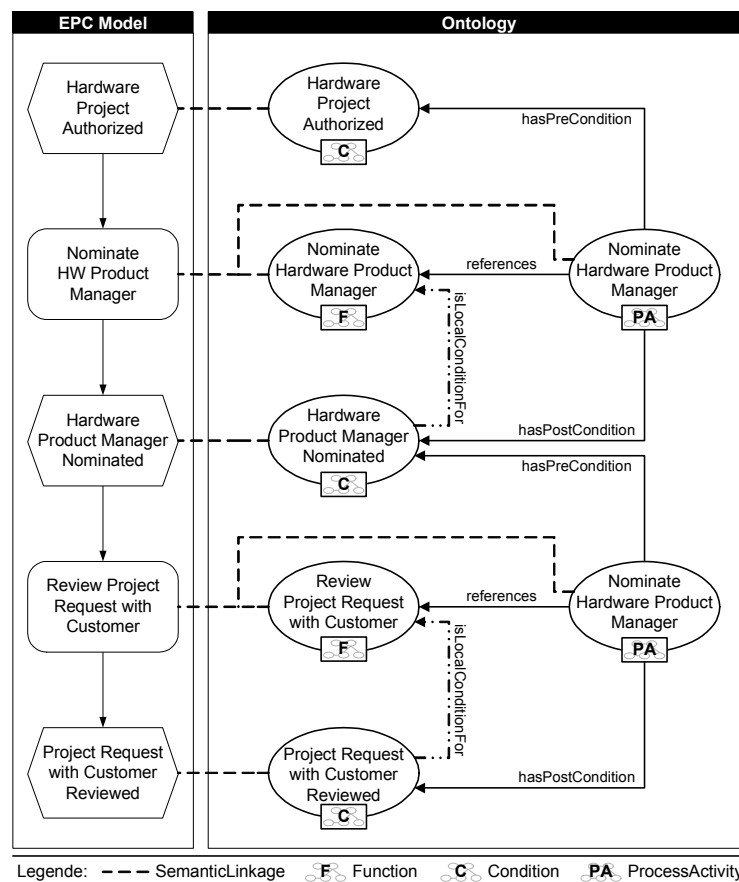


Abbildung 4.8.: Beispiel semantisch aufbereitete ProcessActivity

se wird die Zusammensetzung einer Prozessaktivität auch anhand eines allgemeinen Beispiels verdeutlicht.

Ausgangspunkt für die semantische Aufbereitung der Prozessaktivitäten bilden die EPK-Funktionen der bestehenden Prozessmodelle. In einem ersten Schritt wird also die semantisch annotierte **Function** einer **ProcessActivity** zugewiesen. Diese Zuweisung muss vor der Ermittlung der Kontextinformation erfolgen, da auf Basis der **Function** der **ContextType** für die einzelnen **Conditions** abgeleitet wird.

Bei der Aufbereitung der Kontextinformation muss berücksichtigt werden, dass mehrere EPK-Ereignisse, verbunden durch Operatoren, vor bzw. nach einer EPK-Funktion auftreten können. Aus diesem Grund ist ein rekursiver Aufruf notwendig, um, wie bei der Traversierung eines Baums, alle vorgelagerten bzw. nachfolgenden Elemente verarbeiten zu können. Die grundlegende Vorgehensweise dieser Rekursion wird in Abbildung 4.9 dargestellt, ohne dabei Rücksicht auf jene Eigenheiten zu nehmen, welche für die Bildung der Pre- bzw. Post-Condition spezifisch sind. In den folgenden Abschnitten wird nun die allgemeine Vorgehensweise für die Ermittlung der Kontextinformationen näher beschrieben, wobei an den entsprechenden Stel-

len auch auf die Besonderheiten bei der Aufbereitung der Pre- und Post-Condition hingewiesen wird. Grundlage für die Beschreibung der Rekursion bildet die Identifikation der Post-Condition.

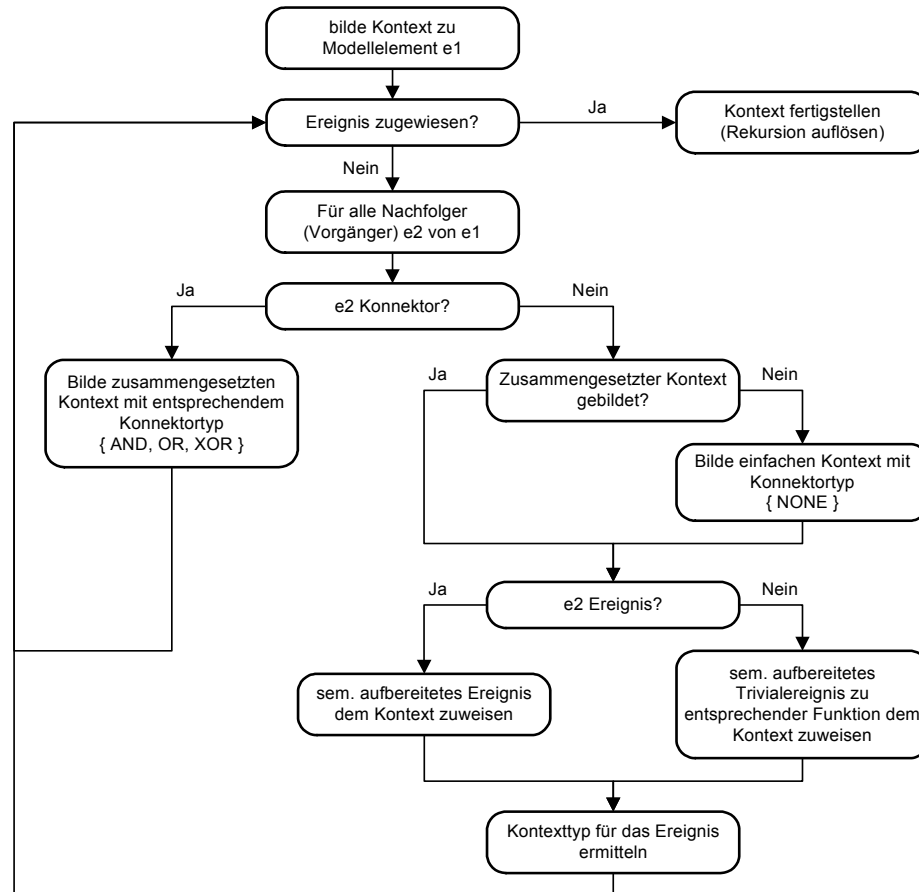


Abbildung 4.9.: Vorgehensmodell: Ermittlung der Kontextinformation

Ausgehend von einer EPK-Funktion wird das nachfolgende Modellelement im Prozessverlauf untersucht und in Abhängigkeit von dessen Typ entsprechend verarbeitet. Handelt es sich bei dem folgenden Modellelement um einen Operator, welcher mehrere ausgehende Kanten besitzt, wird ein zusammengesetzter **Context** mit dem entsprechenden **ConnectionType** gebildet. Bei der Bildung der Post-Condition legt der Typ des Operators auch den **ActivityType** fest. Wird in diesem Zusammenhang ein Operator vom Typ **Or** oder **Xor** identifiziert, wird die **Function** als Entscheidungsfunktion gekennzeichnet (**ActivityType=Decision**). Wurde kein Operator identifiziert, wird zunächst geprüft, ob bereits ein zusammengesetzter **Context** gebildet wurde. Ist dies nicht der Fall wird ein einfacher **Context** mit dem **ConnectionType** "None" gebildet. Da nicht davon ausgegangen werden kann, dass bei der Modellierung streng nach dem Muster "Ereignis-Funktion-Ereignis" vorgegangen wurde, muss auch der Modellelementtyp des folgenden Elements überprüft werden. Konnte ein EPK-Ereignis identifiziert werden, wird dem **Context** die ent-

sprechende, semantisch annotierte **Condition** zugewiesen. Wurde jedoch eine EPK-Funktion gefunden, wird die **Condition** auf unterschiedliche Weise ermittelt. Bei der Bildung der Post-Condition wird das Trivialereignis jener **Function** verwendet, für deren EPK-Funktion gerade die Kontextinformation ermittelt wird. Für die Pre-Condition hingegen wird das Trivialereignis jener **Function** verwendet, deren EPK-Funktion gerade durch die Rekursion ermittelt wurde. Nachdem die semantisch aufbereitete **Condition** zugewiesen wurde, wird der **ContextType** ermittelt, indem die **Function** und die **Condition** verglichen werden (für eine Beschreibung der unterschiedlichen **ContextTypes** vgl. Kapitel 3.4). Als Abbruchbedingung der Rekursion wird geprüft, ob bereits eine **Condition** zugewiesen wurde. Ist dies der Fall, wird die Rekursion aufgelöst, wodurch alle nachfolgenden Modellelemente der EPK-Funktion verarbeitet werden können.

Wie bei allen anderen Konzepten der Ontologie wird auch bei der Bildung der **ProcessActivities** und der Kontextinformationen darauf geachtet diese lediglich einmal zu erfassen und mit der Häufigkeit des Auftretens in den bestehenden Modellen zu belegen. Die Vorgehensweise entspricht hierbei im Wesentlichen jener der semantischen Annotierung der EPK-Funktionen und -Ereignisse (vgl. Abb. 4.3). Ein- und dieselbe **ProcessActivity** zeichnet sich in diesem Zusammenhang durch dieselbe **Function** und dieselben Kontextinformationen aus. Daher muss bereits bei der Bildung der Kontextinformation untersucht werden, ob der entsprechende **Context** bereits erfasst wurde. Grundsätzlich müssen bei einem **Context** die **Condition** und der **ContextType**, welcher durch die **Function** festgelegt wird, übereinstimmen. Bei einfachen Kontextinformationen gestaltet sich diese Prüfung sehr einfach, da lediglich ein **Condition-ContextType**-Paar untersucht werden muss. Die Identifikation bei zusammengesetzten Kontextinformationen hingegen gestaltet sich deshalb schwieriger, da zum Einen mehrere **Conditions** referenziert werden und zum Anderen Verschachtelungen der Kontextinformationen auftreten können. Die Prüfung von zusammengesetzten Kontextinformationen erfolgt daher erst nach Verarbeitung aller nachfolgenden Elemente eines Operators, d.h. nachdem die vollständige **Context**-Instanz zu diesem aufbereitet wurde. Bei der folgenden Überprüfung werden schließlich alle, diesem **Context** zugewiesenen **Condition-ContextType**-Paare und gegebenenfalls zugewiesene, zusammengesetzte **Contexts** untersucht. Konnte eine **Context** mit derselben Zusammensetzung im vorliegenden Prozessmodell identifiziert oder in der bestehenden Wissensbasis gefunden werden, wird dieser verwendet und dessen Häufigkeit erhöht, anderenfalls wird eine neue Instanz in die Ontologie integriert.

Abbildung 4.10 zeigt die Aufbereitung der Post-Condition anhand eines allgemeinen Beispiels, indem eine EPK-Funktion mit ihren Endereignissen und der semantische Repräsentation der entsprechenden Kontextinformation in der Ontologie gegenübergestellt werden. Um die Vorgehensweise des rekursiven Aufrufs zu veranschaulichen, wurde die Aufbereitung in drei wesentliche Schritte unterteilt.

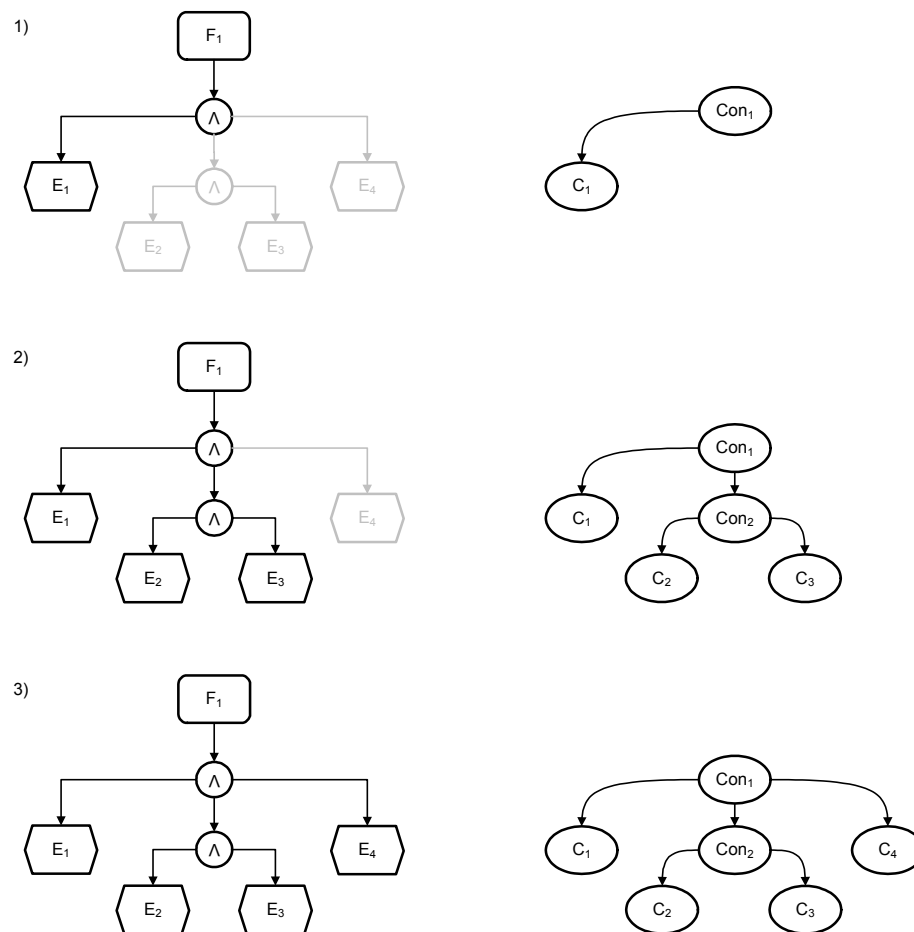


Abbildung 4.10.: Aufbereitung der Post-Condition

### Beispiel: Bildung eines zusammengesetzten Kontext

- 1) Der Algorithmus für die Aufbereitung der Kontextinformation hat einen "And"-Operator identifiziert und einen **Context** "Con<sub>1</sub>" mit dem entsprechenden **ConnectionType** gebildet. Diesem **Context** wird die **Condition** "C<sub>1</sub>" (semantische Repräsentation des EPK-Ereignisses "E<sub>1</sub>") zugewiesen. In einem realen Modell würde an dieser Stelle die Ermittlung des **ContextType** erfolgen, welcher der Beziehung zwischen dem **Context** und der **Condition** zugewiesen würde.
- 2) Nach Verarbeitung des EPK-Ereignisses "E<sub>1</sub>" identifiziert der Algorithmus einen weiteren Operator, für welchen der entsprechende **Context** "Con<sub>2</sub>" (ebenfalls mit **ConnectionType** "And") gebildet und dem zuvor angelegten **Context** "Con<sub>1</sub>" zugewiesen wird. In den folgenden Schritten werden die beiden **Conditions** "C<sub>2</sub>" und "C<sub>3</sub>" (semantische Repräsentationen der EPK-Ereignisse "E<sub>2</sub>" und "E<sub>3</sub>") mit den entsprechenden **ContextTypes** dem **Context** "Con<sub>2</sub>" zugewiesen. Daraufhin wurden sämtliche nachfolgenden Elemente des zweiten "And"-Operators verarbeitet und es wird geprüft, ob der zusammengesetzte **Context**

*"Con<sub>2</sub>" bereits an einer anderen Stelle identifiziert werden konnte (im vorliegenden Prozessmodelle oder in der Wissensbasis). Konnte eine Übereinstimmung gefunden werden, wird die bestehende Instanz verwendet und deren Häufigkeit um eins erhöht. Anderenfalls wird der eben gebildete Context neu angelegt.*

- 3) *Im dritten Schritt wird dem Context "Con<sub>1</sub>" die Condition "C<sub>4</sub>" (semantische Repräsentation des EPK-Ereignisses "E<sub>4</sub>") zugewiesen. Da nun auch sämtliche nachfolgenden Elemente des ersten "And"-Operators verarbeitet wurden, kann die Prüfung des zusammengesetzten Context "Con<sub>1</sub>" erfolgen. Diese Prüfung erfolgt nur, wenn "Con<sub>2</sub>" bereits existiert, weil anderenfalls auch "Con<sub>1</sub>" nicht existieren kann. Konnte der Context "Con<sub>1</sub>" gefunden werden, wird dessen Häufigkeit um eins erhöht und der ProcessActivity zugewiesen. Anderenfalls wird die neu erstellte Instanz verwendet.*

### Sonderfall: zusammengesetzte Modellelemente

Wie bereits bei der semantischen Aufbereitung der EPK-Funktionen und -Ereignisse angesprochen (vgl. Kap. 4.2), unterstützt die Analyse die Trennung zusammengesetzter Modellelemente. Diese Zerlegung wird auch bei der Aufbereitung der ProcessActivities weitergeführt, wobei in der vorliegenden Umsetzung die Semantik der Konjunktion nicht berücksichtigt, sondern eine sequenzielle Abfolge der einzelnen Tätigkeiten unterstellt wird. Die Reihenfolge, in welcher die Tätigkeiten innerhalb der EPK-Funktion auftreten, ist jedoch relevant und wird bei der Aufbereitung der ProcessActivity berücksichtigt.

### Beispiel: zusammengesetzte EPK-Funktion

*Abbildung 4.11 zeigt die zusammengesetzte EPK-Funktion "Identify and Analyse Hardware Requirements", welche bereits im Rahmen der Aufbereitung der Modellelemente in zwei eigenständige Functions zerlegt wurde (vgl. Kap. 4.2). Dieser Zerlegung entsprechend werden auch im Zuge der Aufbereitung der Prozessaktivitäten zwei eigenständige ProcessActivities gebildet, welche der EPK-Funktion hinterlegt werden. Die automatisch generierte Condition "Hardware Requirements Identified", welche das Trivialereignis zur ersten Teilfunktion darstellt, wird in diesem Zusammenhang entsprechend der Vorgehensweise bei der Aufbereitung der ProcessActivities als Post-Condition für die ProcessActivity "Identify Hardware Requirements" und als Pre-Condition für die ProcessActivity "Analyse Hardware Requirements" verwendet.*



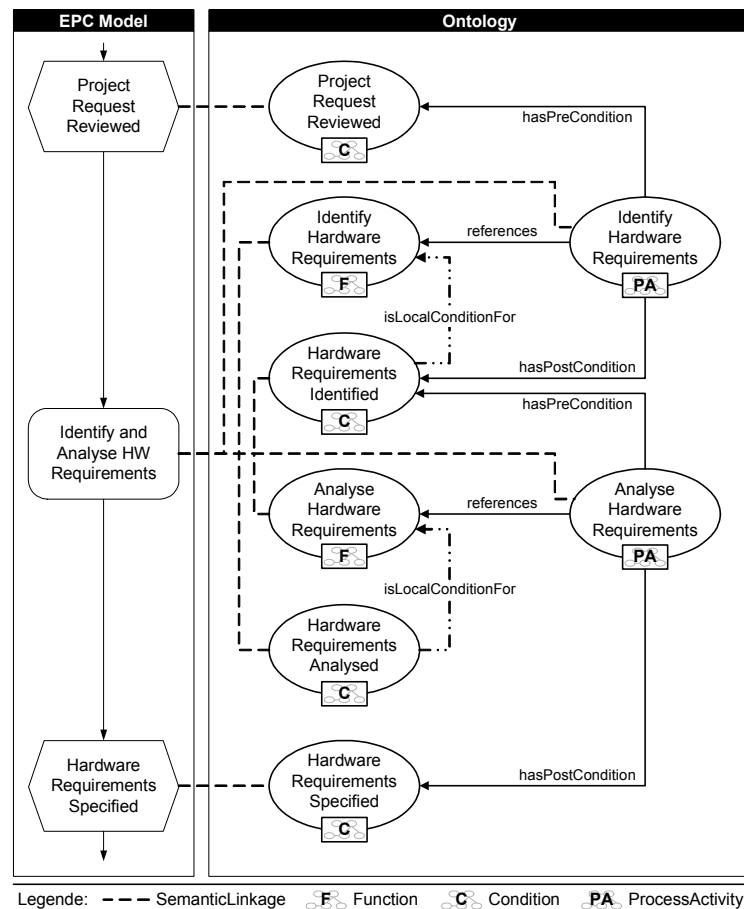


Abbildung 4.11.: Beispiel: zusammengesetzte EPK-Funktion mit semantisch annotierten ProcessActivities

## 4.4. Darstellung der semantischen Annotierung in pModeler

Nachdem in den vorangegangenen Kapiteln die Vorgehensweise bei der semantischen Aufbereitung der Prozessmodelle gemäß der Konzepte der Ontologie dargestellt wurde, werden in diesem Kapitel die Komponenten zur Darstellung der Ergebnisse der semantischen Annotierung in pModeler beschrieben.

Nach einer kurzen Beschreibung der Analysekomponente von pModeler, mit welcher die semantische Aufbereitung der Prozessmodelle gestartet werden kann, werden die beiden im Rahmen dieser Arbeit implementierten Komponenten zur Darstellung der semantischen Annotierung vorgestellt. Zuerst wird jene Benutzeroberfläche beschrieben, mit welcher die einem EPK-Modell annotierten **Process Objects** dargestellt werden. Abschließen wird die Komponente zur Anzeige der in einem EPK-Modell identifizierten **ProcessActivities** vorgestellt, mit welcher auch die

aufbereiteten **Functions**, **Conditions** und deren Teilkonzepte eingesehen werden können.

#### 4.4.1. pModeler Analysekomponente

In den folgenden Abschnitten wird kurz die Analysekomponente des pModeler Prototypen vorgestellt, welche zum Starten von Analysen sowie zur Darstellung von Analyseergebnissen, wie bspw. der Auflistung der semantisch annotierten **Process Objects** eines EPK-Modells, verwendet wird.

Abbildung 4.12 zeigt die Analysekomponente, welche über den Menüeintrag **2 Model Analysis » Pattern Mining Service** aufgerufen werden kann. Die Komponente untergliedert sich in einen Explorer zur systematischen Gliederung der EPK-Modelle im linken Bereich und einen Hauptbereich zur Darstellung des Analyseverlaufs bzw. von Analyseergebnissen.

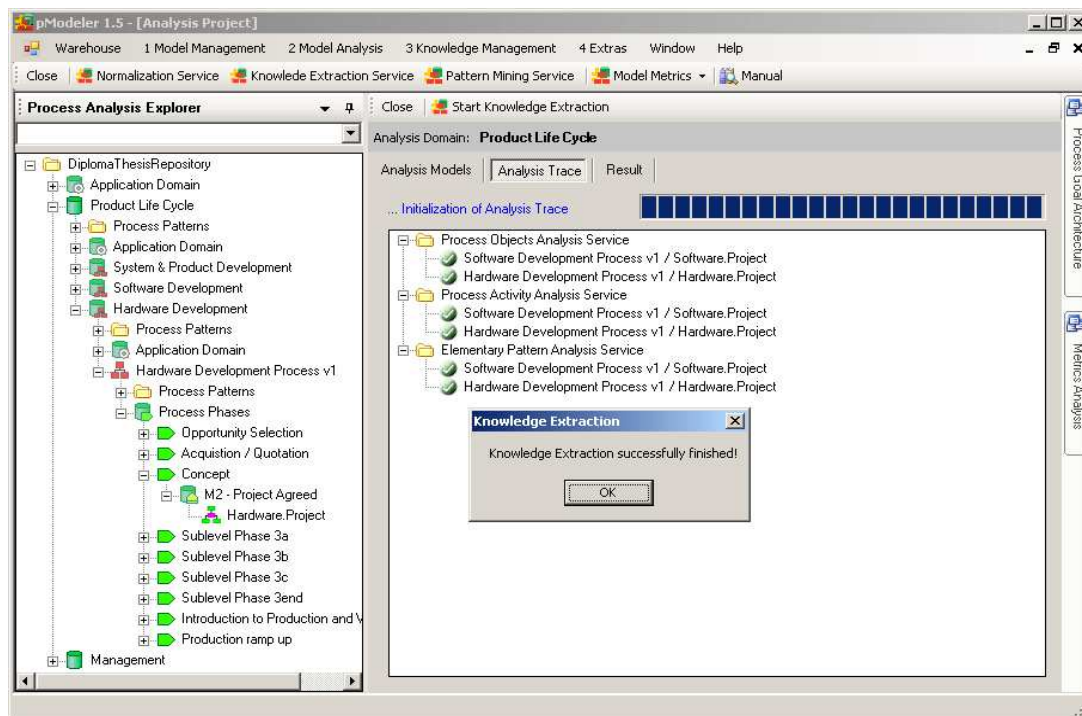


Abbildung 4.12.: Analysekomponente

Analyseservices werden durch Rechtsklick auf eine Hauptgruppe, wie bspw. **Product Life Cycle**, und die Auswahl eines entsprechenden Unterpunkts des Kontextmenüeintrags **Analysis Services** gestartet. Die beiden Analysen dieser Arbeit können durch Auswahl des Eintrags **Knowledge Extraction Service** aufgerufen werden. Daraufhin werden im rechten Bereich sämtliche, lexikalisch aufbereiteten EPK-Modelle aufgelistet, für welche noch keine semantische Aufbereitung durch-

geführt wurden. Durch Betätigung der **Start Knowledge Extraction**-Schaltfläche werden die aufgelisteten Modelle semantisch aufbereitet und die Ergebnisse werden in die Ontologie integriert.

Um Analyseergebnisse anzuzeigen, muss im Explorer ein EPK-Modell ausgewählt werden. Durch Rechtsklick auf dieses Modell erscheint ein Kontextmenü dessen Eintrag **Show Analysis Result** weiter untergliedert ist, um die Ergebnisse der lexikalischen Aufbereitung (vgl. [Grö06]), die semantisch annotierten **Process Objects** (vgl. Kap. 4.4.2) sowie Prozessaktivitäten (vgl. Kap. 4.4.3) anzeigen zu können.

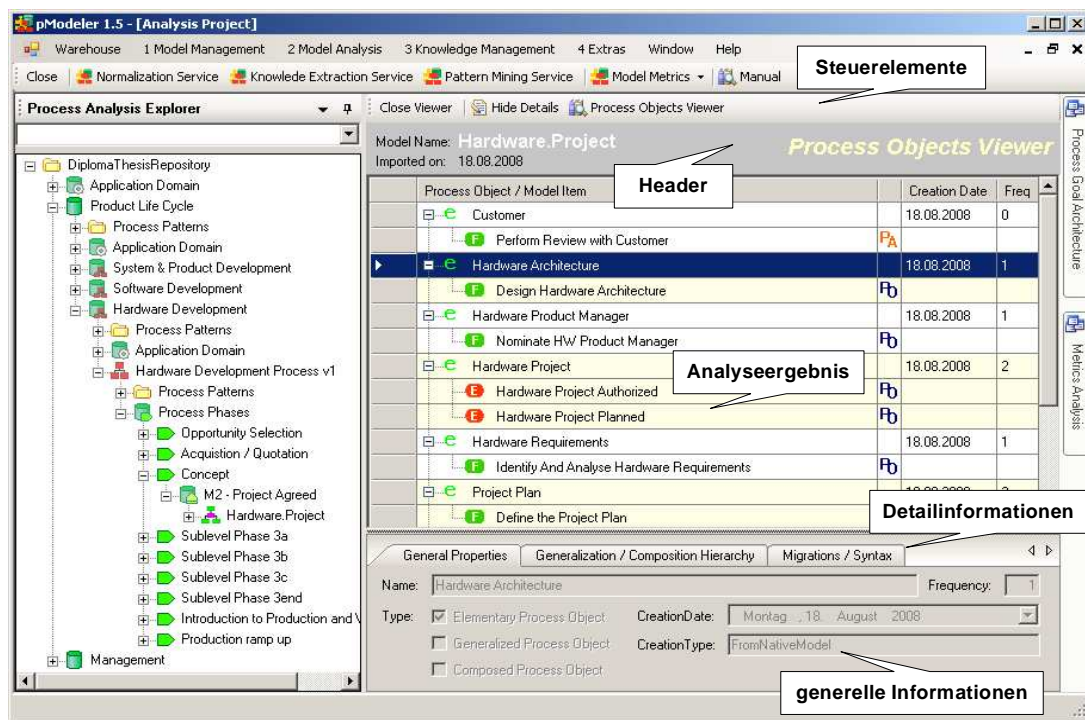


Abbildung 4.13.: Darstellung annotierter Process Objects in pModeler

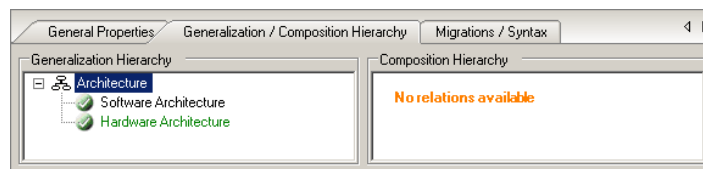
#### 4.4.2. Darstellung der annotierten Process Objects

Ziel dieses Kapitels ist die Beschreibung der Komponente zur Darstellung der einem EPK-Modell semantisch annotierten **Process Objects**, wie sie im pModeler Prototypen implementiert wurde (siehe Abb. 4.13). Der Aufruf der Benutzeroberfläche erfolgt durch einen Rechtsklick auf das gewünschte EPK-Modell im Explorer der Analysekomponente und durch Auswahl des Kontextmenüeintrags **Show Analysis Result** » **Process Object**.

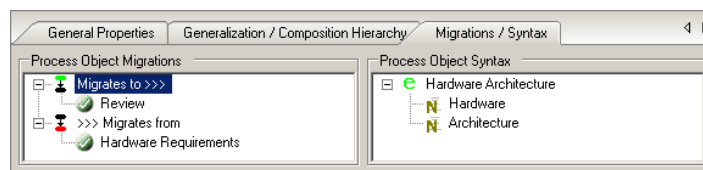
Mit Hilfe der Steuerelemente kann die Komponente geschlossen, die Detailansicht zu den **Process Objects** ein- bzw. ausgeblendet und der **Process Object Viewer** (vgl. Kap. 3.5.1) geöffnet werden. Im Header werden grundlegende Informationen

zum ausgewählten Modell bereitgestellt. Im Hauptbereich der Komponente werden die identifizierten **Process Objects** zu den einzelnen Modellelementen aufgelistet. In der Detailansicht unter dieser Liste können erweiterte Informationen zu einem ausgewählten **Process Object** eingesehen werden.

Im Hauptbereich werden die, dem ausgewählten EPK-Modell annotierten **Process Objects** aufgelistet und durch das vorangestellte Icon klassifiziert (für eine Beschreibung der Icons vgl. Kap. 3.5.1). Unter den einzelnen **Process Objects** werden jene Modellelemente aufgelistet, aus welchen diese extrahiert wurden. Neben dem Modellelement zeigt die Verwendung, ob es sich um das eigentliche **Process Object** handelt, oder ob es Teil des **Parameters** ist.



(a) Generalisierungen und Kompositionen



(b) Migrationsbeziehungen und syntaktische Repräsentation

Abbildung 4.14.: Detailinformationen zu annotierten **Process Objects**

Die erweiterten Informationen zu einem ausgewählten **Process Object** werden mit Hilfe von drei Reitern organisiert. Auf dem ersten Reiter werden die generellen Informationen angezeigt (vgl. Kap. 3.5.1.2). Die beiden anderen Reiter werden für die Darstellung der semantischen Beziehungen verwendet. Während der zweite Reiter der Darstellung der Generalisierungs- und Kompositionshierarchien dient (siehe Abb. 4.14.a), werden auf dem letzten Reiter die Migrationsbeziehungen sowie die syntaktische Repräsentation des **Process Objects** angezeigt (siehe Abb. 4.14.b). Die Migrationsbeziehungen werden in "Migrates to" und "Migrates from" unterteilt. Während unter "Migrates to" jene **Process Objects** aufgelistet werden, welche auf das gewählte folgen, werden unter "Migrates from" die vorgelagerten zusammengefasst. In der Liste für die syntaktische Repräsentation werden die einzelnen Wörter, aus welchen sich das **Process Object** zusammensetzt, aufgelistet und durch ein Icon der entsprechende Worttyp angezeigt.

### 4.4.3. Darstellung der annotierten Process Activities

In diesem Kapitel wird die Komponente zur Darstellung der semantisch annotierten Prozessaktivitäten der einzelnen EPK-Modelle beschrieben (siehe Abb. 4.15). Der Aufruf der Komponente erfolgt durch Rechtsklick auf das gewünschte EPK-Modell im Explorer der Analysekomponente und durch die Auswahl des Kontextmenüeintrags **Show Analysis Result » Process Activities**.

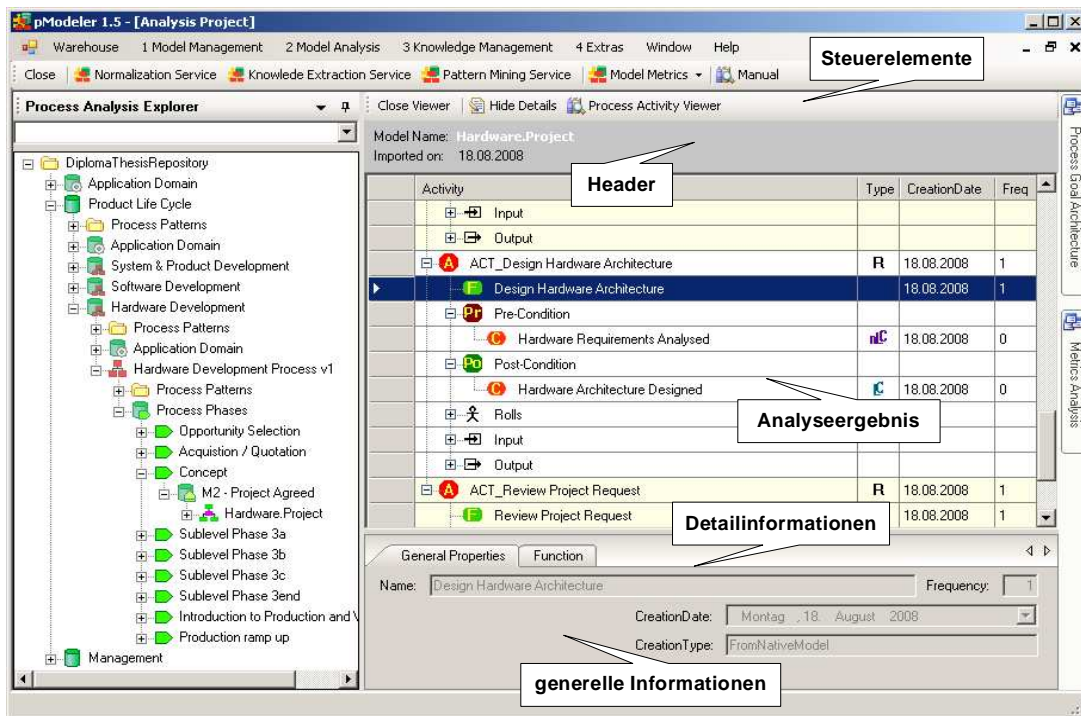
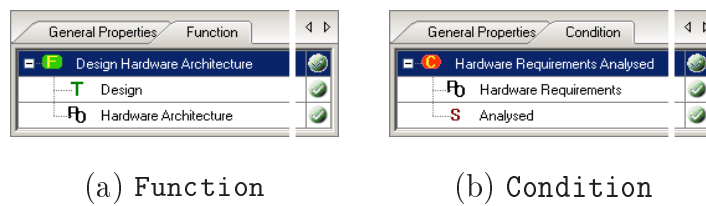


Abbildung 4.15.: Darstellung annotierter Process Activities in pModeler

Mit Hilfe der Steuerelemente kann die Komponente geschlossen, die Detailansicht zu den Elementen ein- bzw. ausgeblendet und der **Process Activity Viewer** (vgl. Kap. 3.5.3) geöffnet werden. Im Header werden allgemeine Informationen zum ausgewählten Modell angezeigt. Neben dem Hauptbereich, in welchem die semantisch annotierten Prozessaktivitäten und deren Teilkonzepte aufgelistet werden, verfügt die Komponente über einen Bereich, in welchem detailliertere Informationen zu einem Element eingesehen werden können.

Im Hauptbereich werden die semantisch annotierten **Process Activities** des gewählten EPK-Modells aufgelistet, deren Darstellung an jener des **Process Activity Viewer** (vgl. Kap. 3.5.3) angelehnt ist. Zu einer **ProcessActivity** werden demzufolge die auszuführende **Function**, und die Pre- und Post-Condition angezeigt. Außerdem werden in der rechten Spalte neben dem entsprechenden Element die **ActivityTypes** bzw. **ContextTypes** angeführt (vgl. Kap. 3.4).



(a) Function

(b) Condition

Abbildung 4.16.: Detailansicht zu annotierten Process Activities

Die erweiterten Informationen zu einem ausgewählten Element im Hauptbereich werden auf zwei Reitern im unteren Bereich präsentiert. Während der erste Reiter generelle Informationen zum ausgewählten Element anzeigt, stellt der zweite Reiter eine Auflistung der Teilelemente desselben dar. Auf letzterem werden also die einzelnen Teilkonzepte angezeigt, aus welchen die ausgewählte **ProcessActivity**, **Function**, **Pre-/Post-Condition** oder **Condition** zusammengesetzt wird (Abb. 4.16 zeigt dies bspw. für eine **Function** und eine **Condition**).

# 5. Implementierung

Parallel zur Entwicklung der Konzepte für die semantische Aufbereitung von Prozessmodellen und der Identifikation von häufig auftretenden Prozessmuster, wurde ein Prototyp als Proof of Concept entwickelt. Die Implementierung, an welcher mehrere Studenten beteiligt waren bzw. sind, wurde unter der Leitung von Herrn Mag. Andreas Bögl durchgeführt. Der Prototyp wurde für den Forschungspartner, die Siemens AG CT SE 3 München, im Rahmen mehrerer Diplomarbeiten entwickelt und im Zuge einer Weiterentwicklung schließlich um die Ausstehenden Komponenten ergänzt.

Die Entwicklung des Prototypen basiert auf der Microsoft .NET Plattform, wobei die Implementierung mittels der Programmiersprache C# durchgeführt wurde. Als Datenbank wurde Microsoft Access 2000 verwendet.

In den folgenden Unterkapiteln wird nun die Implementierung des pModeler Prototypen beschrieben. Ausgehend von einem Überblick über die gesamte Architektur des Prototypen in Kapitel 5.1, werden die Kernbereiche dieser Arbeit ausführlicher beschreiben. In Kapitel 5.2 wird das, der Ontologie zugrunde liegende Datenmodell beschrieben. Kapitel 5.3 beschäftigt sich schließlich mit den Objektmodellen, welche für die Verwaltung der Ontologie, sowie die beiden Analysen dieser Arbeit zuständig sind.

## 5.1. pModeler Architekturüberblick

Ziel dieses Kapitels ist es, einen Überblick über die gesamte Architektur des pModeler Prototypen zu geben. Hierfür werden kurz die einzelnen Module und deren Aufgaben dargestellt. Des Weiteren erfolgt eine Einordnung der Kernbereiche dieser Arbeit in die Gesamtarchitektur. Außerdem kann dieses Kapitel als eine Art Wegweiser verstanden werden. Da, wie bereits erwähnt, mehrere Diplomanden an der Entwicklung des pModeler Prototypen beteiligt waren, wird bei den Beschreibungen der einzelnen Module auf die entsprechenden Arbeiten verwiesen, in welchen diese ausführlich dargestellt werden.

Abbildung 5.1 zeigt die Gesamtarchitektur des pModeler Prototypen mit den wichtigsten Modulen, wobei die Kernbereiche dieser Arbeit in den hinterlegten Teilbereichen einzuordnen sind. Der `DataAccess` ist für die Kommunikation mit der

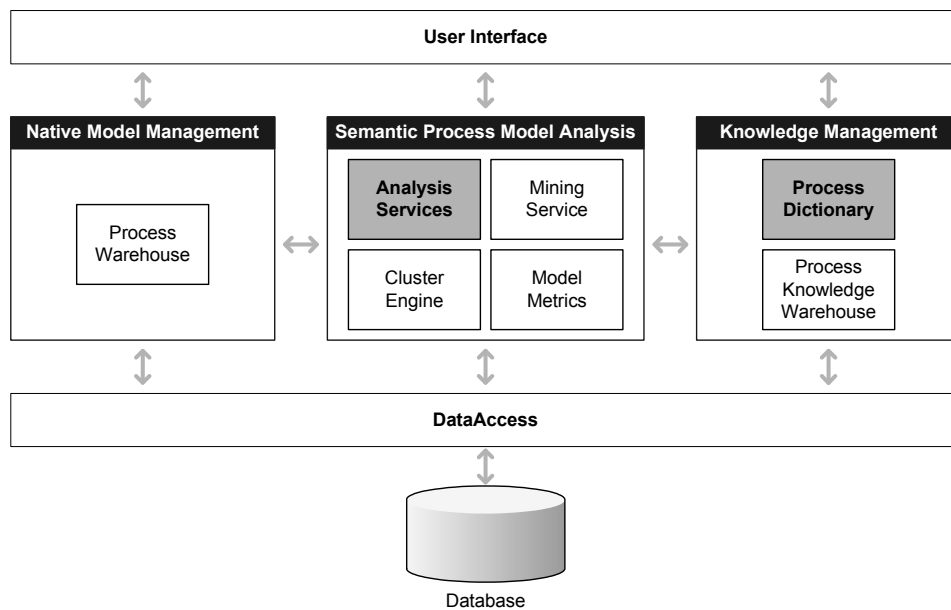


Abbildung 5.1.: pModeler Architekturüberblick

Datenbank zuständig und stellt grundlegende Zugriffsmethoden bereit. Diese werden von den Modulen der darüberliegenden Schicht verwendet bzw. erweitert, um spezifische Datenzugriffe zu realisieren. Die mittlere Schicht enthält die logischen Komponenten des Systems, welche für die Verwaltung der nativen Prozessmodelle (**Native Model Management**), die Verwaltung der Wissensbasen (**Knowledge Management**) und die Analysen (**Semantic Process Model Analysis**) zuständig sind. Diese bilden den Kern des Prototypen und werden daher in weiterer Folge ausführlicher dargestellt. Die oberste Schicht, das **User Interface** fasst schließlich sämtliche Komponenten zusammen, welche für die Interaktion mit dem Benutzer benötigt werden.

Für den Zugriff auf die Datenbank wurde der von Frau Mag.<sup>a</sup> Christina Lachner im Rahmen ihrer Arbeit realisierte **DataAccess** verwendet, welcher Methoden zur Transaktionskontrolle, für den Zugriff auf Datensätze sowie zur Manipulation von Datensätzen bereitstellt. Die Implementierung umfasst eine abstrakte Basisklasse, welche durch unterschiedliche Implementierungen die Anbindung verschiedener Datenbanksysteme ermöglicht, wobei in der gegenwärtigen Version der Zugriff auf eine MS Access 2000 Datenbank umgesetzt wurde [Lac08, S. 131ff].

Im Sinne einer einheitlichen Architektur wurde auf die Verwendung eines Object-Relational Mapping (ORM) Frameworks, wie bspw. *NHibernate*<sup>1</sup> für die Microsoft.Net Plattform, verzichtet. ORM Frameworks unterstützen das Mapping von objekt-orientierten Modellen auf relationale Datenbanken, mittels bspw. XML, und stellen Methoden für die Instanzierung und Manipulation von Objekten bereit.

<sup>1</sup><http://nhforge.org/Default.aspx>



## Native Model Management

Das **Native Model Management** ist für den Import und die Verwaltung der Prozessmodelle zuständig. Die Verwaltung umfasst in diesem Zusammenhang auch die Organisation der Prozessarchitektur, in welche die Prozessmodelle integriert werden. Die für diesen Teilbereich zuständigen Klassen werden im Modul **ProcessWarehouse** zusammengefasst, welches von Herrn Thomas Hostnik im Rahmen seiner Diplomarbeit implementiert wurde (vgl. [Hos10]).

## Knowledge Management

Der Teilbereich **Knowledge Management** wird in die beiden Module **ProcessDictionary** und **ProcessKnowledgeWarehouse** untergliedert, welche für die Verwaltung der Wissensbasen in pModeler zuständig sind.

Im **ProcessDictionary** werden die Klassen für die Verwaltung des Wörterbuchs sowie der Ontologie zusammengefasst. Innerhalb des Moduls erfolgt eine weitere Untergliederung in die Bereiche **Syntax-**, **ProcessObject-** und **ActivityDictionary**, welche in den folgenden Absätzen kurz beschrieben werden:

- Das **SyntaxDictionary** umfasst sämtliche Klassen, welche für die Verwaltung des Wörterbuchs in pModeler benötigt werden. Dieser Teilbereich wurde von Frau Mag.<sup>a</sup> Alexandra Grömer im Zuge ihrer Diplomarbeit implementiert (vgl. [Grö06]).
- Im **ProcessObjectDictionary** werden alle Klassen zusammengefasst, welche für die Verwaltung der **ProcessObjects** und derer semantischen Beziehungen zuständig sind. Dieser Teilbereich wurde im Rahmen dieser Arbeit entwickelt und wird daher in Kapitel 5.3 noch detaillierter dargestellt.
- Das **ActivityDictionary** ist für die Verwaltung der übrigen Elemente der Ontologie, wie bspw. der **Functions** oder **ProcessActivities**, zuständig. Da dieser Teil ebenfalls im Rahmen dieser Arbeit realisiert wurde, wird dieser in Kapitel 5.3 noch ausführlicher beschrieben.

Das Modul **ProcessKnowledgeWarehouse** wird in den **Goal-** und den **SolutionSpace** untergliedert. Dieser Teilbereich wurde von Frau Mag.<sup>a</sup> Christina Lachner im Rahmen ihrer Diplomarbeit implementiert (vgl. [Lac08]) und im Rahmen der Weiterentwicklung des pModeler Prototypen erweitert.

- Der **GoalSpace** dient der Verwaltung der Prozessziele, wobei anzumerken ist, dass die Verwaltung des Zielbaums im Modul **MiningService** implementiert wurde, welches in einem der folgenden Absätze beschrieben wird. Die Verwaltung umfasst in diesem Bereich die elementaren Prozessziele (**Process Goals**) sowie der generischen Prozessziele (**Generic Process Goals**).
- Der **SolutionSpace** umfasst sämtliche Klassen für die Verwaltung der Prozessmuster. Hierzu zählen sowohl instanziierte- (**Elementary Process Pattern**)

und generische Prozessmuster (**Generic Process Pattern**) als auch jene Datenstrukturen zur Abbildung von Graphen zusammengesetzter Prozessmuster.

### Semantic Process Model Analysis

Die **Semantic Process Model Analysis** fasst schließlich sämtliche Module zusammen, welche für die Aufbereitung und Analysen der Prozessmodelle zuständig sind. Die Module **AnalysisServices** und **MiningService** kapseln die einzelnen Analyseschritt vom nativen Prozessmodell bis zu den häufig auftretenden, komponierten Prozessmustern und bilden somit den Kern des pModeler Prototypen. Im Unterschied dazu haben die Module **ClusterEngine** und **ModelMetrics** unterstützende Funktionen und liefern keinen direkten Beitrag zur Aufbereitung der Prozessmodelle. Diese werden daher lediglich der Vollständigkeit halber kurz erwähnt.

Das Modul **ClusterEngine** kapselt alle Klassen für das Clustering der Prozessmodelle. Zielsetzung dieser Analyse ist die Identifikation von ähnlichen Prozessmodellen mithilfe des Clusteringalgorithmus *CLARANS*. Dieses Modul wurde von Frau Mag.<sup>a</sup> Alexandra Grömer im Rahmen ihrer Diplomarbeit implementiert (vgl. [Grö06]).

Das Modul **ModelMetrics** ist für die Berechnung von standardisierten Prozessmodellmetriken zuständig und fasst entsprechende Klassen für diese Aufgaben zusammen. Implementiert wurde dieser Teilbereich von Herrn Mag. Peter Zierl im Rahmen seiner Diplomarbeit (vgl. [Zie07]).

Wie bereits einleitend erwähnt werden im Modul **AnalysisServices** die Klassen für die einzelnen Analyseschritt zusammengefasst, welche für Aufbereitung der Prozessmodelle und die Extraktion von Prozessmustern benötigt werden. Zur Vereinfachung des Analyseablaufs wurden die Analyseschritte in Gruppen zusammengefasst, wobei der **Normalization-**, **Knowledge Extraction-** und der **Pattern Mining Service** unterschieden werden können. Für diese drei Hauptanalyseschritte wurden entsprechende Controller-Klassen implementiert, welche die einzelnen Analyse-Klassen verwenden und den Ablauf der Analyse steuern. In den folgenden Absätzen werden nun die drei Services und die von ihnen verwendeten Analyse-Klassen kurz vorgestellt:

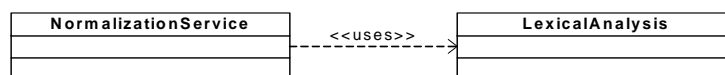


Abbildung 5.2.: Klassen - Normalization Service

- Der **Normalization Service** (vgl. Abb. 5.2) ist für die lexikalische Aufbereitung der Prozessmodelle zuständig. Der Klasse **LexicalAnalysis** werden hierfür die zu analysierenden Prozessmodelle übergeben. Diese verarbeitet die

einzelnen Modellelemente (derzeit lediglich EPK-Funktionen und -Ereignisse), indem sie diese in die einzelnen Wörter zerlegt und die Wortklassifikation vornimmt. Dieser Teilbereich wurde von Frau Mag.<sup>a</sup> Alexandra Grömer im Rahmen ihrer Diplomarbeit implementiert (vgl. [Grö06]).

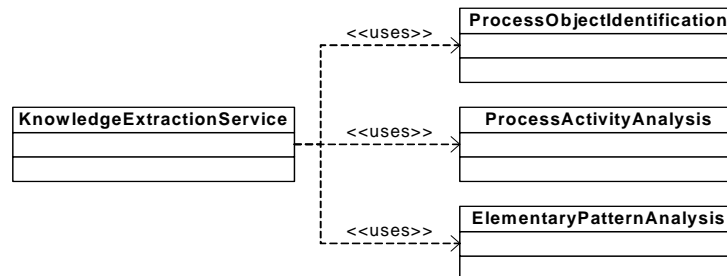


Abbildung 5.3.: Klassen - Knowledge Extraction Service

- Der **Knowledge Extraction Service** umfasst mehrere Analyseschritte, welche sequentiell abgearbeitet werden (vgl. Abb. 5.3). Die Aufgabe dieses Services ist die semantische Aufbereitung der Prozessmodelle entsprechend der Konzepte der Ontologie, sowie die Generierung der instanziierten, komponierten Prozessmuster (**Process Pattern Model**) für die einzelnen Prozessmodelle. Der Service kapselt die folgenden Analyse-Klassen:
  - **ProcessObjectIdentification**: dieser Analyseschritt dient der Identifikation der **Process Objects** und deren Beziehungen, sowie der Aufbereitung der EPK-Funktionen und -Ereignisse entsprechend der Konzepte der Ontologie. Der Klasse werden die zu analysierenden Prozessmodelle übergeben, verarbeitet die einzelnen Modellelemente auf Basis der Ergebnisse der lexikalischen Analyse und instanziiert die Konzepte der Ontologie (**ProcessObjects**, **Functions**, **Tasks**, usw.). Da diese im Rahmen dieser Arbeit implementiert wurde, wird sie in Kapitel 5.3.3 noch näher beschrieben.
  - **ProcessActivityAnalysis**: Ziel dieses Analyseschritts ist die Aufbereitung der **ProcessActivities**, d.h. die Ermittlung der Kontextinformationen der EPK-Funktionen. Ausgangspunkt für diesen Schritt bilden die nativen Prozessmodelle, sowie die hinterlegten, semantisch aufbereiteten **Functions** und **Conditions** der Ontologie. Diese Klasse wurde ebenfalls im Rahmen dieser Arbeit entwickelt und wird daher in Kapitel 5.3.3 näher betrachtet.
  - **ElementaryPatternAnalysis**: Zielsetzung dieses Schritts ist die Entkopplung der Prozessstruktur von den nativen Prozessmodellen, sowie die Generierung der Prozessziele (**Process Goals**). Input für die Klasse **ElementaryPatternAnalysis** bilden die zu analysierenden Prozessmodelle. Für jedes Prozessmodell werden aus den, den EPK-Funktionen hinter-

legten `ProcessActivities` die instanziierten Prozessmuster (`Elementary Process Pattern`) gebildet und zu einem Graphen, dem `Process Pattern Model` zusammengefasst. Weiters werden zu allen `Elementary Process Patterns` die entsprechenden `Process Goals` generiert und diesen hinterlegt. Diese Analyse-Klasse wurde im Rahmen der Weiterentwicklung des pModeler Prototypen implementiert.

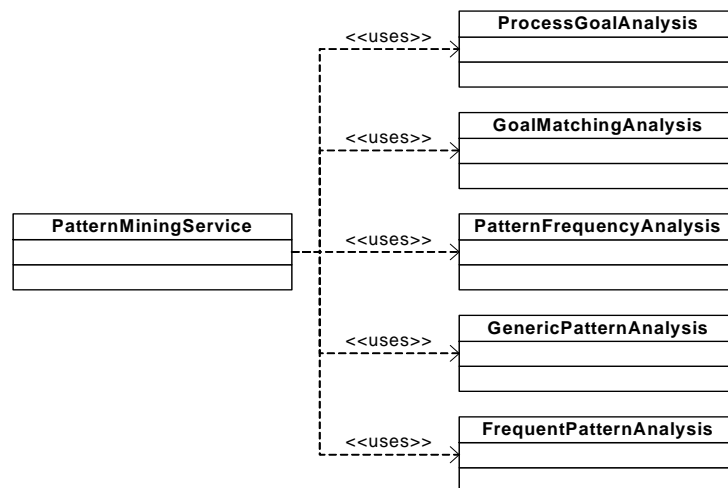


Abbildung 5.4.: Klassen - Pattern Mining Service

- Der `Pattern Mining Service` ist schließlich für den Aufbau der Zielstruktur in Form einer hierarchischen Baumstruktur (`Process Goal Tree`), sowie für die Konstruktion der häufig auftretenden, komponierten Prozessmuster, den sogenannten `Composite Process Patterns` zuständig. In Abbildung 5.4 werden die einzelnen Analyse-Klassen zusammengefasst, welche hierfür durchlaufen werden müssen. Dieser Service wurde zur Gänze im Rahmen der Weiterentwicklung des pModeler Prototypen realisiert.

Wesentlich für die folgenden Analyseschritt ist das Modul `MiningService`, welches die Datenstrukturen für den `Process Goal Tree` bereitstellt. Neben diverser grundlegender Methoden zur Navigation durch die Baumstruktur, sowie für den Aufbau derselben, kapselt dieses Modul einige Methoden zur Manipulation des `Process Goal Trees`. Zu diesen zählen der Pruning-Algorithmus, sowie die Algorithmen für die Berechnung der Traces, welche für die Konstruktion der häufigen Prozessmuster von wesentlicher Bedeutung sind und daher an entsprechender Stelle erwähnt werden.

- `ProcessGoalAnalysis`: Ziel dieses Schrittes ist die Ermittlung der Beziehungen zwischen den `Process Goals` der einzelnen Modelle, d.h. die Generierung der Zielstruktur in Form des `Process Goal Trees`. Hierfür werden der Klasse die `Process Pattern Models` übergeben, welche aus den hinterlegten `Process Goals` die entsprechenden `Process Goals`

Trees generiert. Für den Aufbau der Baumstruktur werden die entsprechenden Methoden des `MiningService` verwendet. Abschließend werden die einzelnen `Process Goal Trees` in die bestehende Zielstruktur integriert.

- `GoalMatchingAnalysis`: Ziel dieses Analyseschrittes ist die Identifikation von ähnlichen Prozesszielen. Den Input für die Klasse bilden die `Process Pattern Models` sowie die entsprechenden `Process Goal Trees` zu diesen Modellen. Die einzelnen `Process Goals` werden paarweise, sowohl modellintern als auch modellübergreifend verglichen und es wird ein Ähnlichkeitsmaß berechnet. Auf diese Weise wird die `Matching Table` aufgebaut, welche Paare von ähnlichen `Process Goals` sowie den Grad ihrer Übereinstimmung zusammenfasst.
- `PatternFrequencyAnalysis`: Aufgabe dieser Klasse ist die Berechnung der Häufigkeit des Auftretens der `Elementary Process Patterns`. Hierfür werden dieser die `Process Pattern Models` und die entsprechende `Matching Table` übergeben. Auf Basis der, in der `Matching Table` zusammengefassten, übereinstimmenden `Process Goals` werden die entsprechenden `Elementary Process Patterns` ermittelt und deren Häufigkeit berechnet.
- `GenericPatternAnalysis`: Ziel dieser Analyse-Klasse ist die Abstraktion der häufig auftretenden Prozessmuster sowie der Prozessziele, d.h. die Generierung der generischen Prozessmuster (`Generic Process Pattern`) bzw. der generischen Prozessziele (`Generic Process Goals`). Den Input bilden wiederum die `Process Pattern Models`, sowie die `Matching Table`. Zuerst werden die, in der `Matching Table` zusammengefassten, übereinstimmenden `Process Goals` abstrahiert. Parallel dazu werden für die, den `Process Goals` entsprechenden `Elementary Process Patterns` die generischen Prozessmuster aufbereitet.
- `FrequentPatternMining`: der letzte Analyseschritt ist schließlich für die Konstruktion der häufigen, komponierten Prozessmuster zuständig. Hierfür werden der Klasse die `Process Goal Trees` übergeben, welche zuerst mit Hilfe der Methoden der Klasse `MiningService` für diesen Analyseschritt vorbereitet werden. Die Vorbereitung umfasst in diesem Zusammenhang das `Process Goal Tree Pruning` und die `Trace Calculation`. Der `Pruning-Algorithmus` reduziert hierbei die `Process Goal Trees`, indem sämtliche nicht häufig auftretende `Process Goals` entfernt werden. Daraufhin werden für die reduzierten `Process Goal Trees` die `Traces` berechnet, d.h. es werden alle möglichen, korrekten Ausführungsreihenfolgen für die Modelle ermittelt. Im Anschluss daran werden die Ergebnisse der `Trace-Berechnung` von der Klasse `FrequentPatternMining` ausgewertet und es werden die häufigen, komponierten Prozessmuster

konstruiert (**Composite Process Pattern**).

## 5.2. Datenmodell

In diesem Kapitel wird das Datenmodell beschrieben, mit welchem die Ontologie dieser Arbeit in der Datenbank abgebildet wird. Die Entscheidung die Ontologie mittels einer relationalen Datenbank und nicht mittels einer formalen Beschreibungssprache, wie bspw. RDF oder OWL, zu realisieren, kann wie folgt begründet werden:

1. bestehende Schnittstellen: die Ontologie wurde zu einem Zeitpunkt entwickelt, als bereits wesentliche Teile des Prototypen entwickelt wurden bzw. waren. Hierzu zählen vor allem die direkten Schnittstellen zur Ontologie, d.h. die Prozessmodellverwaltung und das Wörterbuch. Um eine nahtlose Anbindung an diese gewährleisten zu können, wurde auch die Ontologie mittels einer relationalen Datenbank umgesetzt.
2. einfache Ableitungsregeln: die Zielsetzung der Ontologie liegt weniger in der Ableitung nicht explizit modellierten Wissens als in der semantischen Beschreibung von Modellelementbezeichnungen und der Aufbereitung elementarer Tätigkeiten, um die automatisierte Extraktion von Prozessmustern zu ermöglichen (vgl. Kap. 3.1). Da hierfür keine "komplexen" Ableitungsregeln benötigt werden, bieten die Möglichkeiten des maschinellen Schließens, welche eine wesentliche Stärke formaler Beschreibungssprachen bilden, keinen unmittelbaren Mehrwert für den gewählten Ansatz.

Für die relationale Umsetzung der Ontologie wurden für die unterschiedlichen Konzepte und Beziehungen einzelne Tabellen angelegt, um einen möglichst einfachen Zugriff auf sämtliche Daten zu ermöglichen. Da das Datenmodell einen erheblichen Umfang erreicht hat, wird es der Übersichtlichkeit wegen in drei Teilen beschrieben. Das erste Teilmodell zeigt jenen Ausschnitt aus der Ontologie, mit welchem die **Process Objects** und sämtliche semantische Beziehungen zwischen diesen gespeichert werden können. Im zweiten Modell werden die Tabellen für jene Konzepte beschrieben, welche für die semantische Aufbereitung der EPK-Funktionen und EPK-Ereignisse benötigt werden. Den Abschluss bildet jenes Teilmodell, mit welchem die **ProcessActivities** und Kontextinformationen persistiert werden können.

### 5.2.1. Teildatenmodell Process Object

Abbildung 5.5 zeigt jenes Teildatenmodell, mit welchem die **Process Objects** und deren semantischen Beziehungen in der Datenbank abgebildet werden. Das **Process Object** erbt die grundlegenden Attribute aus der Tabelle **BaseObject**, welche auch als Basis für die übrigen Konzepte der Ontologie, wie bspw. den **Functions**, dient.

Diese allgemeinen Attribute sind eine eindeutige *OID*, ein *Name*, das Attribut *CreatedOn*, mit welchem das Erstellungsdatum der Instanz gespeichert wird, und die *Frequency*, welche die Häufigkeit des Auftretens festhält. Außerdem erhält jede Instanz eine Referenz auf die Tabelle *CreationType*, mit welcher die Art der Erstellung festgelegt wird. In diesem Zusammenhang werden die Ausprägungen *fromNative-Model*, *fromSystem* und *fromUser* unterschieden, welche bereits in Kapitel 3.5.1.2 beschrieben wurden.

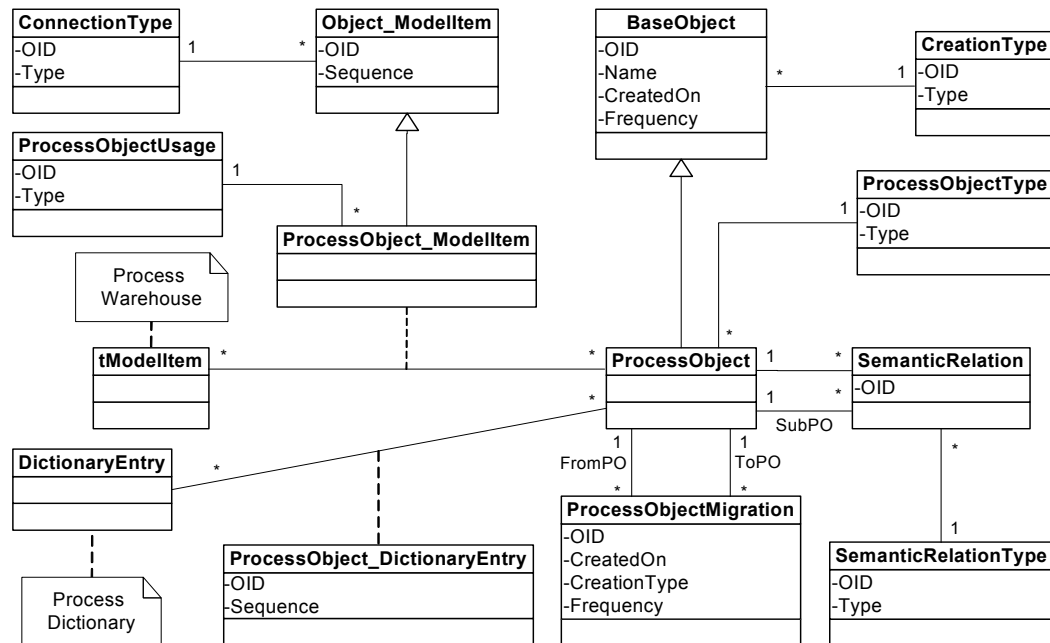


Abbildung 5.5.: Teildatenmodell Process Object und semantische Beziehungen

Neben diesen allgemeinen Attributen, wird einem *Process Object* eine Referenz auf die Tabelle *ProcessObjectType* zugewiesen. Diese Tabelle enthält die unterschiedlichen Typen zur grundlegenden Klassifizierung der *Process Objects*. Zu diesen Typen zählen *ElementaryProcessObject*, *GeneralizedProcessObject* und *ComposedProcessObject*, welche bereits im Rahmen der konzeptionellen Beschreibung definiert wurden (vgl. Kapitel 3.4). Erweitert wurde diese Enumeration um den Typ *GeneralizedComposedProcessObject*, um *Process Objects* zu kennzeichnen, welche generalisiert und komponiert sind. Dieser wurde eingeführt, um den gezielten Zugriff auf *Process Objects* dieses Typs zu ermöglichen.

Dem Namen des *Process Objects* liegt eine syntaktische Repräsentation mit Einträgen aus dem *Process Dictionary* zugrunde. Die Referenzen auf diese Einträge werden durch die Assoziationstabelle *ProcessObject\_DictionaryEntry* realisiert. Diese Tabelle enthält, neben den entsprechenden Fremdschlüsseln, die Attribute *OID* und *Sequence*, mit welcher die Reihenfolge der einzelnen Wörter für die Bildung des *Process Objects* festgelegt wird.

Die Verweise auf die Modellelemente, welchen das *Process Object* hinterlegt ist,

wird in der Assoziationstabelle `ProcessObject_ModelItem` persistiert. Dieser liegt die Basistabelle `Object_ModelItem` zugrunde, welche auch für die Verknüpfung der übrigen Konzepte der Ontologie mit den Modellelementen verwendet wird. Die Attribute dieser Tabelle sind eine eindeutige *OID*, die *Sequence* und eine Referenz auf die Tabelle `ConnectionType`. Die letzten beiden Attribute werden für den Fall benötigt, dass ein Modellelement aufgespalten werden muss (die Aufspaltung von Modellelementen wurde bereits in Kapitel 4.2 beschrieben). Die *Sequence* gibt dabei die Reihenfolge vor, in welcher die, dem Modellelement hinterlegten Instanzen auftreten. Der `ConnectionType` spezifiziert, wie die einzelnen Instanzen des Modellelements miteinander in Beziehung stehen, d.h. ob sie durch ein "und" oder ein "oder" getrennt wurden. In der derzeitigen Ausbaustufe wird dieses Attribut allerdings noch nicht berücksichtigt.

Die Assoziationstabelle `ProcessObject_ModelItem` erbt nun die Attribute dieser Basistabelle und wird um eine Referenz auf die Tabelle `ProcessObjectUsage` erweitert. In dieser wird ebenfalls eine Enumeration verwaltet, welche signalisiert, ob es sich bei der Instanz um das eigentliche `Process Object` des Modellelements handelt, oder ob sie als Teil des `Parameters` Verwendung findet.

Zur Verwaltung der Generalisierungs- und Kompositionshierarchien wurde die Assoziationstabelle `SemanticRelation` eingeführt. In dieser Tabelle werden neben einer eindeutigen *OID* die Fremdschlüssel des Basis-`Process Objects` und des untergeordneten `Process Objects` gespeichert. Eine Referenz auf die Tabelle `SemanticRelationType` legt dabei fest, ob es sich dabei um eine Generalisierungs- (*Generalization*) oder eine Kompositionsbeziehung (*Composition*) handelt.

Die letzte Tabelle dieses Teilmodells stellt die Tabelle `ProcessObjectMigration` dar, in welcher die Migrationsbeziehungen zwischen den `Process Objects` verwaltet werden. Diese Tabelle enthält, neben den Attributen *OID*, *CreatedOn* und *Frequency*, die Fremdschlüssel auf die beiden beteiligten `Process Objects`. Diese Referenzen bilden den Ausgangspunkt (*FromPO*) und das Ziel (*ToPO*) der Migration.

### 5.2.2. Teildatenmodell semantisch aufbereitete Modellelemente

Mit dem Teilmodell in Abbildung 5.6 werden jene Konzepte in der Datenbank abgebildet, welche für die Darstellung der semantisch aufbereiteten EPK-Funktionen und -Ereignisse benötigt werden. Die Konzepte zur semantischen Darstellung der EPK-Elemente erben, wie das `Process Object` die grundlegenden Attribute der Basistabelle `BaseObject`, welche bereits im vorangegangenen Kapitel beschrieben wurde. Die spezifischen Erweiterungen für die einzelnen Konzepte werden nun in den folgenden Absätzen kurz zusammengefasst.

Die Tabellen `Task` und `State` werden um eine Referenz auf einen Eintrag im



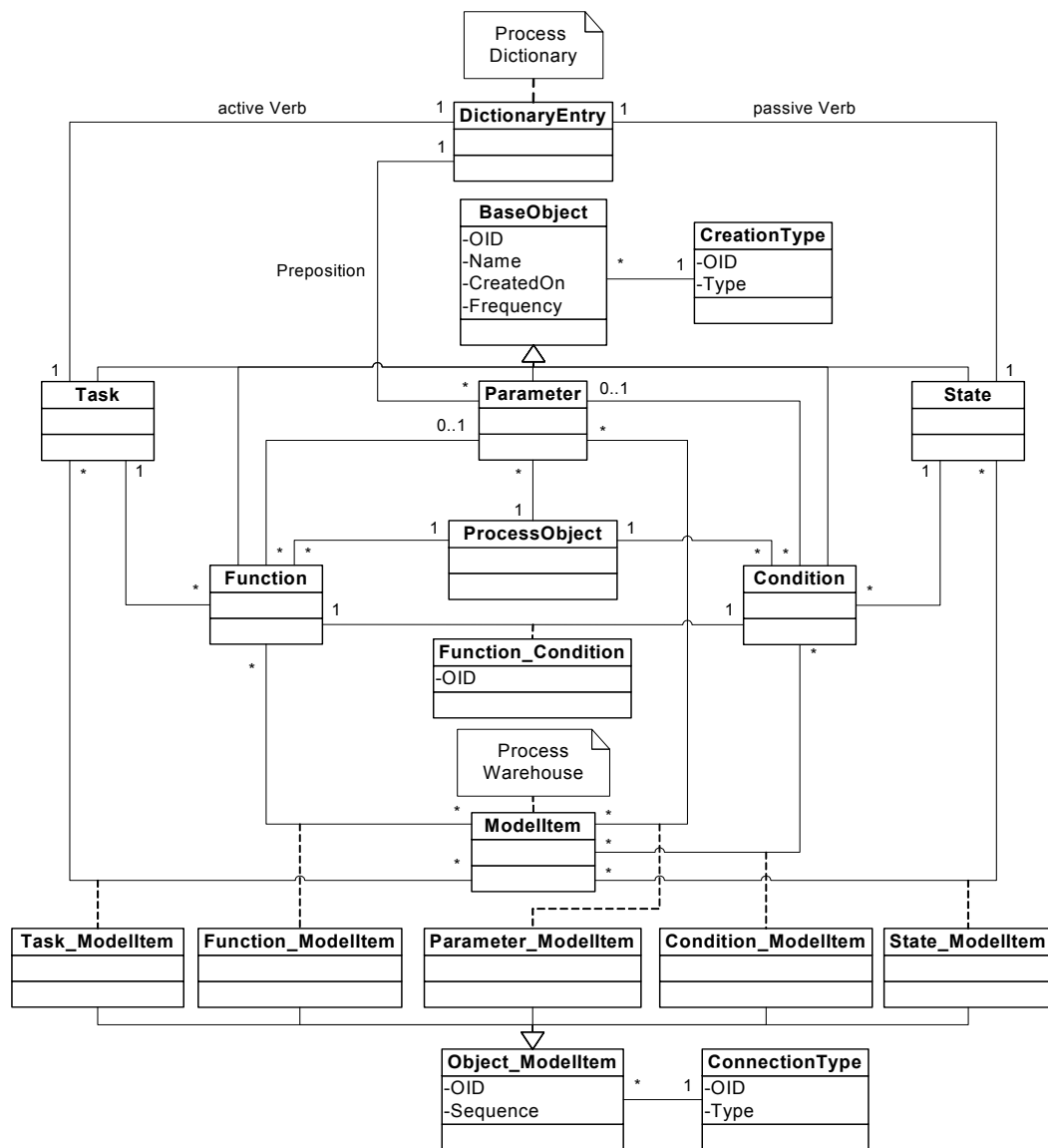


Abbildung 5.6.: Teildatenmodell semantisch aufbereitete Modellelemente

**Process Dictionary** erweitert. Im Unterschied zum **Process Object** ist hier eine einfache Referenz ausreichend, da beispielsweise auch zusammengesetzte Verben wie "set up" als ein einzelner Eintrag im **Process Dictionary** verwaltet werden. Da der **Task** eine Tätigkeit darstellt, welche auf ein **Process Object** ausgeführt wird, kann diese Referenz lediglich auf ein aktives Verb gesetzt werden. Im Gegensatz dazu referenziert ein **State** ein passives Verb. Diese Einschränkungen werden seitens des Programms gewährleistet. Der Grund für die Einführung dieser beiden Tabelle im Gegensatz zu einer einfachen Referenz auf die entsprechenden Einträge bei der Bildung der **Functions** bzw. **Conditions** liegt darin, dass versucht wurde, eine klare Trennung zwischen dem **Process Dictionary** und der Ontologie zu erzielen.

Die Tabelle `Parameter` referenziert eine Präposition im `Process Dictionary` und ein `Process Object` aus der Ontologie. Die Referenz auf die Präposition wird wiederum vom Programm geregelt.

Mit den Tabellen `Function` und `Condition` werden die semantisch aufbereiteten Modellelemente verwaltet. Der dieser Arbeit zugrunde liegenden Definition zufolge, setzt sich eine `Function` aus einem `Task`, einem `Process Object` und einem optionalen `Parameter` zusammen, welche über entsprechende Referenzen innerhalb der Tabelle `Function` miteinander kombiniert werden. Dem entsprechend enthält die Tabelle `Condition` die Fremdschlüssel für das `Process Object`, den optionalen `Parameter` und den `State`. Der *Name* dieser Elemente wird aus den einzelnen Teilkonzepten zusammengesetzt. Um allerdings nicht bei jeder Abfrage alle Teilkonzepte direkt laden zu müssen, wird der Name explizit in der Datenbank gespeichert.

Für die Verknüpfung der einzelnen Teilkonzepte mit den nativen Modellelemente im `Process Warehouse` wurden entsprechende Assoziationstabellen angelegt. Diese erben die Basisattribute von `Object_ModelItem` (vgl. Kapitel 5.2.1) und werden um die entsprechenden Fremdschlüsselattribute erweitert (so wurde beispielsweise für die Verknüpfung zwischen der EPK-Funktion und der semantisch aufbereiteten `Function` die Tabelle `Function_ModelItem` angelegt. Für die Prüfung des korrekten Modellelementtyps ist erneut das Programm zuständig).

### 5.2.3. Teildatenmodell `ProcessActivity`

Den Abschluss zur Beschreibung des Datenmodells bildet das Teilmodell zur Persistierung der `ProcessActivities` und der Kontextinformationen (siehe Abbildung 5.7). Die `ProcessActivity` erbt die Basisattribute von `BaseObject` und wird um die Referenzen auf die auszuführende `Function` und die Kontextinformationen `Pre-` und `Post-Condition` erweitert. Neben diesen Referenzen wird die `ProcessActivity` durch einen Typ beschrieben, welcher in der Tabelle `ActivityType` verwaltet wird. Dieser Typ kennzeichnet die `Function` als reguläre- (*Regular*) oder Entscheidungsfunktion (*Decision*).

Da sich eine `Pre-` bzw. `Post-Condition` aus mehreren `Conditions` zusammensetzen kann, ist es nicht möglich die Kontextinformation mittels einer einfachen Referenz zu realisieren. Deshalb wurde die Tabelle `Context` eingeführt, mit welcher auch komplexere Kontextinformationen abgebildet werden können. Die grundsätzlichen Attribute dieser Tabelle sind eine *OID*, die Häufigkeit des Auftretens (*Frequency*) und die Referenzen auf die Tabellen `BaseType` und `ConnectionType`. Während der `BaseType` die Kontextinformation als `Pre-` oder `Post-Condition` kennzeichnet, beschreibt der `ConnectionType`, wie die, mit dem `Context` in Beziehung stehenden Teile, miteinander verknüpft werden. In diesem Zusammenhang werden die Ausprägungen *None*, *And*, *Or* und *Xor* unterschieden.

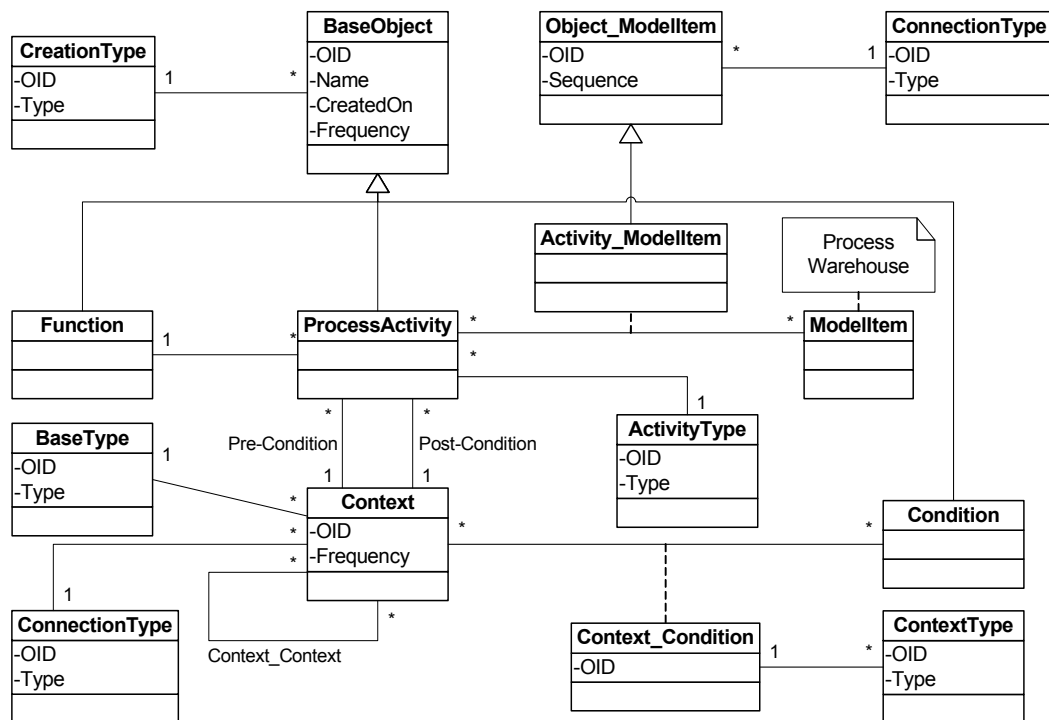


Abbildung 5.7.: Teildatenmodell ProcessActivity

Die **Conditions** werden mittels der Tabelle **Context\_Condition** mit dem **Context** verknüpft. Diese Tabelle enthält außerdem eine Referenz auf die Tabelle **ContextType**, in welcher die Typen zur Kennzeichnung der Beziehung zwischen der **Condition** und der **Function** verwaltet werden. Die entsprechenden Ausprägungen *LocalInitialContext*, *NonLocalInitialContext*, *LocalResultingContext* und *NonLocalResultingContext* dieser Enumeration wurden bereits in Kapitel 3.4 beschrieben.

Für die Abbildung verschachtelter Kontextinformationen wird schließlich die Tabelle **Context\_Context** verwendet, welche lediglich die Fremdschlüssel der beiden betroffenen **Contexts** enthält.

## 5.3. Objektmodell

In diesem Kapitel werden die Objektmodelle für die Verwaltung der Ontologie und der Analysen beschrieben. Auch an dieser Stelle wird das Gesamtmodell aufgrund des Umfangs in einzelnen Teilbereichen beschrieben, wobei sich die Ausführungen auf lediglich einige ausgewählte Schnittstellen, Klassen und Methoden beschränken. Hierfür wird zunächst jenes Objektmodell näher betrachtet, welches für die Verwaltung der Ontologie verwendet wird. Die Hauptaufgaben der Methoden dieses Teilbereichs sind der Zugriff auf die in der Datenbank gespeicherten Instanzen der Ontologie (z.B. **ProcessObjects**, **Functions**, etc.) sowie die Persistierung derselben. Zum

Abschluss dieses Kapitels werden die Objektmodelle der **ProcessObjectAnalysis** und **ProcessActivityAnalysis** kurz dargestellt, welche durch die Aufbereitung der Elemente der Prozessmodelle die Ontologie instanzieren.

Abbildung 5.8 zeigt den schematischen Aufbau des Objektmodells zur Verwaltung der Ontologie. Das in der Abbildung dargestellte **<OntologyElement>** bildet die Grundlage dieses Objektmodells und steht für die Darstellung der Datenobjekte zur Abbildung der verschiedenen Konzepte der Ontologie, wie bspw. der **ProcessObjects**, **Functions**, etc.. Diese Datenobjekte erweitern das **OntologyBaseObject**, welches die gemeinsamen Attribute aller Konzepte der Ontologie kapselt. Diese Objekte stellen reine Datencontainer dar und enthalten keine Methoden für den Zugriff bzw. die Persistierung.

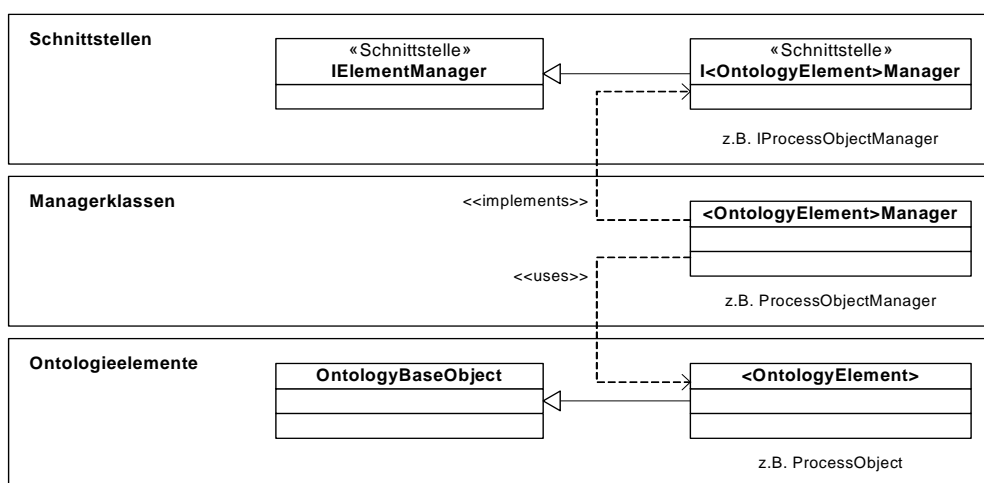


Abbildung 5.8.: Schematisches Objektmodell für die Verwaltung der Ontologie

Für den Zugriff auf die Instanzen der Ontologie wurden eigene Managerklassen implementiert (in Abb. 5.8 als **<OntologyElement>Manager** bezeichnet), um eine klare Trennung zwischen Datenobjekten und Verarbeitungslogik zu erzielen. Konkrete Umsetzungen dieser Klassen stellen bspw. **ProcessObjectManager** oder **FunctionManager** dar. Die einzelnen Managerklassen implementieren entsprechende Schnittstellen (in Abb. 5.8 als **I<OntologyElement>Manager** bezeichnet), welche die öffentlich zugänglichen Methoden festlegen. Allgemeine Methoden werden im **IElementManager** gekapselt, welchen die übrigen Schnittstellen erweitern. Konkrete Instanzen zu den vorher genannten Beispielen stellen bspw. **IProcessObjectManager** und **IFunctionManager** dar. Diese verwenden die entsprechenden Datenobjekte und stellen Methoden für den Zugriff, das Speichern und Löschen, usw. bereit.

In den folgenden Unterkapiteln werden nun schrittweise die verschiedenen Objektmodelle beschrieben. Ausgehend von einer ausführlichen Beschreibung der Datenobjekte in Kapitel 5.3.1, werden in Kapitel 5.3.2 die entsprechenden Managerklassen anhand von ausgewählten Beispielen überblicksmäßig dargestellt. Abschließend werden in Kapitel 5.3.3 jene beiden Klassen beschrieben, welche die Methoden für die

beiden Analysen dieser Arbeit kapseln.

### 5.3.1. Datenobjekte der Ontologie

Ziel dieses Teilbereichs ist es, die entwickelte Ontologie aus dem Blickpunkt der Implementierung darzustellen. Hierfür werden zunächst die Klassen, welche die grundlegenden Attribute der Ontologiekonzepte kapseln im Überblick dargestellt, bevor diese anhand eines Beispiels präzisiert werden. Darauf aufbauend werden die verschiedenen Klassen zur Abbildung der Beziehungen zwischen den Konzepten der Ontologie, wie bspw. die semantischen Beziehungen der `ProcessObjects`, ausführlich beschrieben und den entsprechenden Datenobjekten zugewiesen.

#### Ontologiekonzepte im Überblick

Abbildung 5.9 zeigt einen Überblick über die implementierten Klassen, mit welchen die unterschiedlichen Konzepte der Ontologie dargestellt werden (für eine ausführliche Darstellung des Objektmodells vgl. Abb. B.2 im Anhang). Wie bereits einleitend erwähnt stellen diese Objekte reine Datencontainer dar, welche lediglich wenige Methoden enthalten, wie bspw. *Clone*-Methoden, mit welchen neue Instanzen des entsprechenden Objekts erstellt werden können.

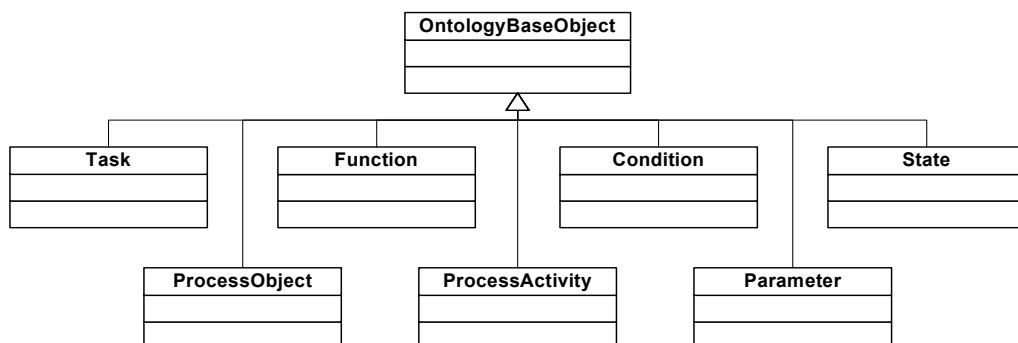
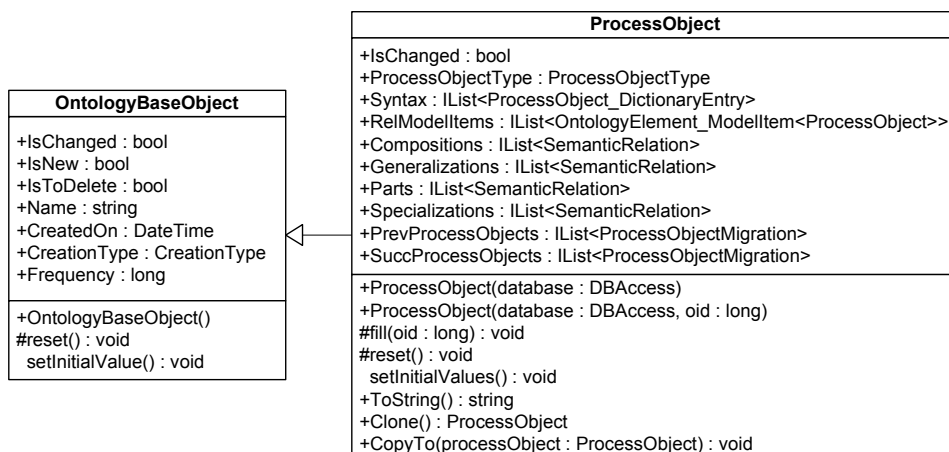


Abbildung 5.9.: Objektmodellüberblick - Datenobjekte der Ontologie

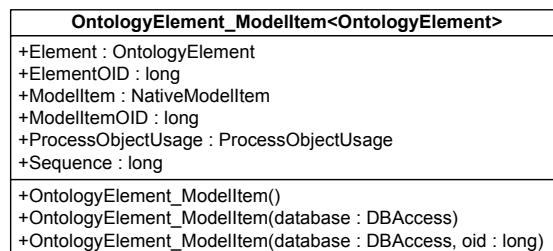
Abbildung 5.10 zeigt das `OntologyBaseObject` und die Klasse `ProcessObject`, welche dieses erweitert. Das `OntologyBaseObject` kapselt die grundlegenden Attribute der Ontologiekonzepte. Zu diesen zählen bspw. *Name*, *CreatedOn*, usw., welche bereits im Datenmodell ausführlich beschrieben wurden (vgl. Kap. 5.2). Die unterschiedlichen Klassen zur Darstellung der Ontologiekonzepte erweitern das `OntologyBaseObject` um spezifische Attribute, wie bspw. der *ProcessObjectType* des `ProcessObject`. Neben diesen Attributen werden auch die Beziehungen zu anderen Konzepten der Ontologie in den jeweiligen Klassen als Properties bereitgestellt (vgl. bspw. *Specializations* im `ProcessObject`, welches eine Liste mit allen Spezialisierungen enthält). Für diese Referenzen wurden, entsprechend der Fremdschlüsseltabellen

Abbildung 5.10.: Klassen - `OntologyBaseObject` & `ProcessObject`

in der Datenbank, eigene Klassen implementiert, welche die Attribute dieser Beziehungen kapseln. Diese werden in den folgenden Absätzen ausführlich beschrieben, wobei bei der Darstellung der Klassen lediglich die wichtigsten Attribute abgebildet werden. Neben der Beschreibung der Attribute werden diese den entsprechenden Ontologie-Klassen zugeordnet.

### Referenzen zu nativen Modellelementen

Werden die Instanzen der Ontologiekonzepte durch die Analyse bestehender Prozessmodelle erhoben, werden sie den entsprechenden Modellelementen hinterlegt. Diese Beziehung wird durch die Klasse `OntologyElement_NativeModelItem<OntologyElement>` dargestellt (vgl. Abb. 5.11). Bei dieser handelt es sich um eine generische Klasse, welche es ermöglicht dieselbe Implementierung für sämtliche Elemente der Ontologie zu verwenden. Der generische Typparameter `OntologyElement` wurde hierfür auf Klassen vom Basistyp `OntologyBaseObject` eingeschränkt (vgl. Abb. 5.12.a) und legt bei der Instanzierung fest, für welches Element die Instanz gebildet werden soll (vgl. Abb. 5.12.b).

Abbildung 5.11.: Klasse - `OntologyElement_NativeModelItem<OntologyElement>`

Die Klasse verbindet also ein natives Modellelement (`NativeModelItem`) mit einer Instanz der Ontologie (`Element`). Der Typ des `Element`-Properties wird durch den

```

/// <summary>
/// Defines the reference between an OntologyElement with the given typeparameter
/// and a NativeModelItem.
/// </summary>
/// <typeparam name="OntologyElement">The OntologyElement.</typeparam>
public class OntologyElement_ModelItem<OntologyElement> : pModeler.Utils.pModelerDataObject
    where OntologyElement : OntologyBaseObject
{
    ...
}

```

(a) Einschränkung - Typparameter *OntologyElement*

```

State state = new State();

OntologyElement_ModelItem<State> reference = new OntologyElement_ModelItem<State>();
reference.Element = state;
reference.ModelItem = nativeModelItem;

state.RelatedModelItems.Add(reference);

```

(b) Instanziierung und Zuweisung

Abbildung 5.12.: Verwendung eines Typparameter anhand eines Beispiels

Typparameter festgelegt. Wird die Klasse bspw. für einen *State* instanziiert, nimmt auch das Property diesen Typ an. Die Referenzen zu den nativen Modellelementen werden in entsprechenden Listen aller Ontologie-Klassen als Property gekapselt (Property *IList*<*OntologyElement\_NativeModelItem*<*OntologyElement*>> *RelatedModelItems* in allen Ontologie-Klassen, vgl. bspw. Abb. 5.12.b für einen *State*).

### Syntaktische Darstellung des *ProcessObject*

Die *ProcessObjects* setzen sich aus einem oder mehreren *DictionaryEntries* aus dem *ProcessDictionary* zusammen. Für die Darstellung dieser Beziehung wird die Klasse *ProcessObject\_DictionaryEntry* in Abbildung 5.13 verwendet. Diese verknüpft das *ProcessObject* mit einem *DictionaryEntry*, wobei die *Sequence* die Reihenfolge der einzelnen Wörter bestimmt. Die syntaktische Repräsentation wird in der Klasse *ProcessObject* in einer List gekapselt, welche die Einträge entsprechend der *Sequence* sortiert (Property *IList*<*ProcessObject\_DictionaryEntry*> *Syntax* in der *ProcessObject*-Klasse).

<b>ProcessObject_DictionaryEntry</b>
+DictionaryEntry : DictionaryEntry
+DictionaryEntryOID : long
+ProcessObject : ProcessObject
+ProcessObjectOID : long
+Sequence : long
+ProcessObject_DictionaryEntry(database : DBAccess)
+ProcessObject_DictionaryEntry(database : DBAccess, oid : long)

Abbildung 5.13.: Klasse - *ProcessObject\_DictionaryEntry*

### Semantische Beziehungen des ProcessObject

Zu den `ProcessObjects` können sowohl Generalisierungs- und Kompositionshierarchien als auch Migrationsbeziehungen definiert werden. Die beiden Klassen, welche die entsprechenden Referenzen kapseln werden in Abbildung 5.14 dargestellt.

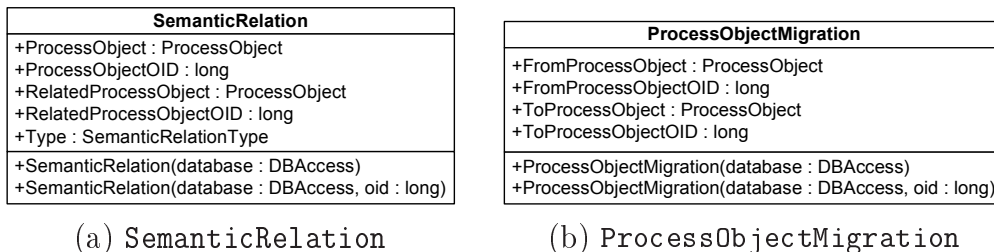


Abbildung 5.14.: Klassen - Semantische Beziehungen des ProcessObject

Abbildung 5.14.a zeigt die Klasse zur Darstellung von Generalisierungs- und Kompositionshierarchien. Während die beiden Properties *ProcessObject* und *RelatedProcessObject* die in Beziehung stehenden Instanzen definieren, spezifiziert der *Type* mit den Ausprägungen `Generalization` und `Composition` die Art der Hierarchie. Mittels dieser Klasse werden in unterschiedlichen Listen in der `ProcessObject`-Klasse die Generalisierungen, Spezialisierungen, Kompositionen und Teil-Beziehungen zu diesem verwaltet (Properties *IList<SemanticRelation> Generalizations*, *IList<SemanticRelation> Specializations*, *IList<SemanticRelation> Compositions* und *IList<SemanticRelation> Parts* in der `ProcessObject`-Klasse).

Mit der Klasse `ProcessObjectMigration` (vgl. Abb. 5.14.b) werden die Migrationsbeziehungen der `ProcessObjects` abgebildet. Diese Klasse kapselt den Start (*FromProcessObject*) und das Ziel (*ToProcessObject*) einer Migrationsbeziehung. Mittels entsprechender Listen zur Unterscheidung von ein- bzw. ausgehenden Migrationsbeziehungen werden diese mit dem `ProcessObject` verknüpft (Properties *IList<ProcessObjectMigration> PrevProcessObjectList* bzw. *IList<ProcessObjectMigration> SuccProcessObjectList* in der `ProcessObject`-Klasse).

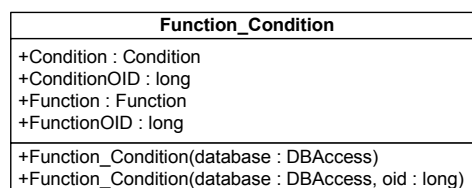


Abbildung 5.15.: Klasse - Function\_Condition

### Beziehung zwischen Funktion und Trivialereignis

Da für die Bildung der `ProcessActivities` die Beziehung zwischen den Funktionen und ihren Trivialereignissen eine bedeutende Rolle spielt, wurde auch für diese eine



eigene Klasse `Function_Condition` implementiert (vgl. Abb. 5.15). Diese verknüpft eine `Function` und einen `Condition` und wird jeweils den beiden Ontologieelementen mittels eines Properties zugewiesen (Property `Function_Condition LocalCondition` in der `Function`-Klasse bzw. `Function_Condition GeneratingFunction` in der `Condition`-Klasse).

### Die Context-Klassen

Für die Bildung einer `ProcessActivity` ist vor allem der `Context` von Bedeutung, welcher die, der `Function` vor- bzw. nachgelagerten `Conditions` kapselt. Da es in diesem Zusammenhang möglich ist, dass mehrere `Conditions` durch Konnektoren mit der `Function` verknüpft sein können, sind für die Bildung des `Contexts`, entsprechend der unterschiedlichen Tabellen der Datenbank (vgl. Kap. 5.2.3), mehrere Klassen notwendig, welche in Abbildung 5.16 dargestellt werden.

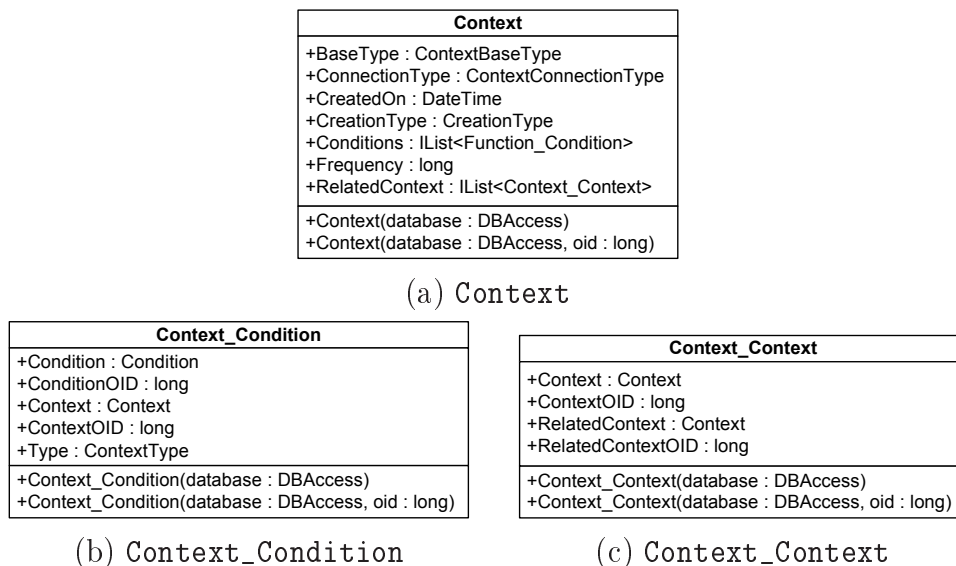


Abbildung 5.16.: Klassen - Context

Grundlage bildet die Klasse `Context` (vgl. Abb. 5.16.a), welche die grundlegenden Kontextinformationen enthält und mit der `ProcessActivity` in Beziehung gesetzt wird (Properties `Context InitialContext` bzw. `Context ResultingContext` der `ProcessActivity`-Klasse). Das Property `BaseType` liefert bzw. setzt den grundlegenden Typ des `Contexts`, welcher durch die Enumeration `ContextBaseType` mit den Ausprägungen `Initial` und `Resulting` beschrieben wird. Der `ConnectionType` bestimmt, ob und wie zusammengesetzte Kontextinformationen verknüpft sind. Die Enumeration `ContextConnectionType` unterscheidet in diesem Zusammenhang `None` (einfacher Kontext) und die Konnektorentypen `And`, `Or` und `Xor`.

Das Property `Conditions` liefert bzw. setzt eine Liste von, mit dieser `Context`-Instanz in Beziehung stehender `Conditions`, welche mittels der Klasse `Context_Conditions` dargestellt werden (vgl. Abb. 5.16.b). Während die Properties

*Context* und *Condition* Instanzen des jeweiligen Objekts liefern bzw. setzen, wird mittels *Type* der entsprechende Kontexttyp festgelegt. Hierbei handelt es sich wiederum um eine Enumeration (*ContextType*), welche die Ausprägungen *LocalInitialContext*, *NonLocalInitialContext*, *LocalResultingContext* oder *NonLocalResultingContext* unterscheidet (für eine Beschreibung der unterschiedliche Kontexttypen vgl. Kap. 3.4).

Die Klasse *Context\_Context* (vgl. Abb. 5.16.c) wird für die Bildung verschachtelter Kontextinformationen benötigt, wenn mehrere Konnektoren aufeinander folgen. Diese enthält lediglich die Properties *Context* und *RelatedContext*, welche die miteinander in Beziehung stehenden *Contexts* zurückgeben bzw. setzen. Auch diese Klasse wird in einer List im *Context* gekapselt und über das Property *RelatedContext* gesetzt bzw. zurückgegeben.

### 5.3.2. Managerklassen der Ontologie

Die Methoden für den Zugriff auf die Ontologieelemente sowie für deren Persistierung werden in als *<OntologyElement>Manager* bezeichneten Klassen gekapselt, welche über eine allgemeine und spezifische Schnittstellenbeschreibungen verfügen. Abbildung 5.17 fasst die implementierten Managerklassen und Schnittstellen zusammen, welche in den folgenden Absätzen anhand einiger ausgewählter Beispiele kurz dargestellt werden (für eine ausführliche Darstellung des Objektmodells siehe Abb. B.3 bis B.6 im Anhang).

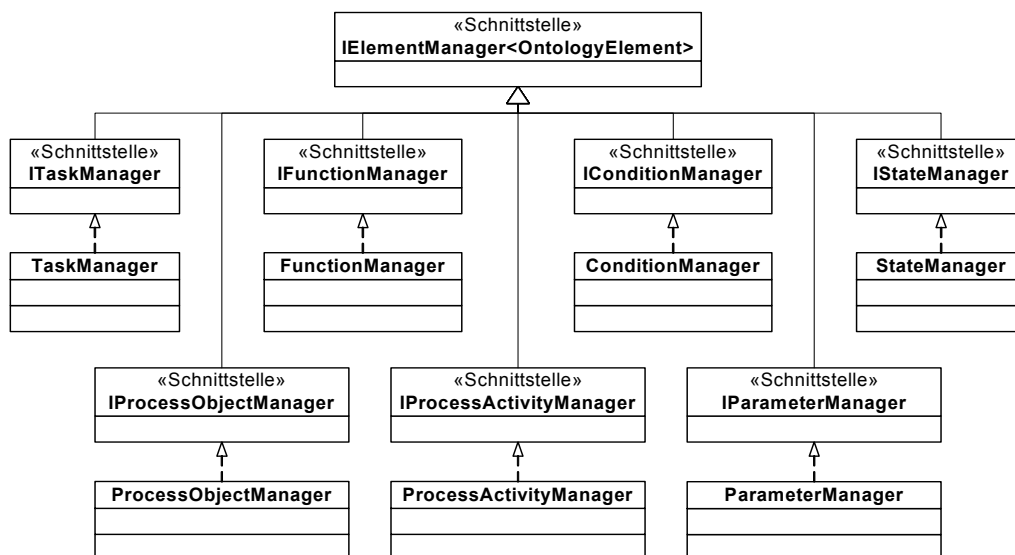


Abbildung 5.17.: Objektmodellüberblick - Managerklassen der Ontologie

Abbildung 5.18 zeigt die allgemeine Schnittstelle *IElementManager*, welche die gemeinsamen Methoden für sämtliche Managerklassen der Ontologiekonzepte kap-

selt. Es handelt sich wiederum um eine generische Implementierung, deren Typparameter *OntologyElement* ein Element der Ontologie sein muss (d.h. der Typparameter muss vom Basistyp `OntologyBaseObject` sein). Die Instanzierung erfolgt bei der Definition einer konkreten Schnittstelle, wie bspw. des `IProcessObjectManager`, bei welchem der Typparameter `ProcessObject` verwendet wird (vgl. Abb. 5.19).

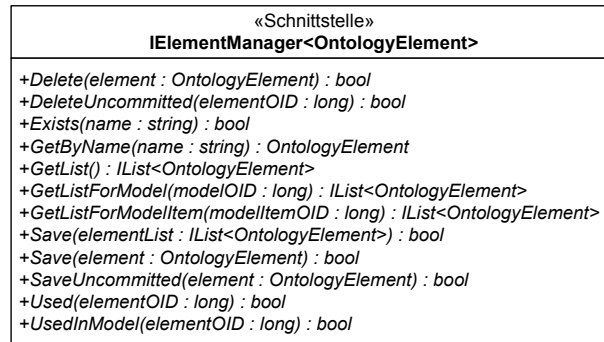


Abbildung 5.18.: Schnittstelle - `IElementManager`

Anhand dieser grundlegenden Schnittstelle werden in den folgenden Absätzen kurz die zur Verfügung gestellten Methoden vorgestellt. Ziel ist keine erschöpfende Aufzählung bzw. Beschreibung sämtlicher Methoden, sondern vielmehr eine überblicksmäßige Darstellung. Es sollen vor allem die gewählten Namenskonventionen für die Bezeichnungen der Methoden vorgestellt werden, welche für alle Implementierungen der Managerklassen dieser Arbeit gültig sind (für eine ausführliche Beschreibung der Methoden der unterschiedlichen Manager-Schnittstellen vgl. Tab. B.1 bis B.8 im Anhang).

```

/// <summary>
/// Defines the interface for ProcessObjectManager classes.
/// </summary>
public interface IProcessObjectManager : IElementManager<ProcessObject>
{
    ...
}
  
```

Abbildung 5.19.: Instanzierung - `IElementManager<ProcessObject>`

### Zugriffsmethoden

Methoden, welche einzelne Ontologieinstanzen liefern, werden mit *Get* eingeleitet und durch die nachfolgende Bezeichnung näher spezifiziert. So liefert bspw. die Methode *GetByName(string name)* eine Ontologieinstanz mit dem gegebenen Namen. Als weitere Beispiele können die Methoden *GetByDictionaryEntry(long dictionaryEntryOID)* des `Task-` bzw. `StateManager` genannt werden, welche den entsprechenden `Task` bzw. den entsprechenden `State` für den gegebenen `DictionaryEntry` zurückgeben.

Methoden, welche Listen von Ontologieinstanzen liefern werden mit *GetList* eingeleitet und durch die nachfolgende Bezeichnung näher spezifiziert. Während bspw. die Methode *GetList()* alle in der Ontologie vorhandenen Instanzen des betreffenden Typs liefert, gibt *GetListForNativeModel(long modelOID)* lediglich jene Instanzen zurück, welche dem gegebenen nativen Prozessmodell hinterlegt sind. Auch in diesem Bereich stellen die einzelnen Managerklassen eine Vielzahl an spezifischen Methoden zur Verfügung, wie bspw. die Methode *GetListForTask(long taskOID)* des *FunctionManager*, welche eine Liste von *Functions* für den gegebenen *Task* zurückgibt.

### Existenz- und Verwendungsabfragen

Methoden zur Prüfung, ob eine bestimmte Instanz bereits existiert, werden mit *Exists* eingeleitet. Die Methode *Exists(string name)* der allgemeinen Schnittstelle liefert demzufolge *true*, wenn das Element mit dem gegebenen Namen bereits in der Ontologie integriert worden ist, anderenfalls *false*.

Um die Konsistenz der Ontologie zu gewährleisten, dürfen lediglich jene Instanzen bearbeitet bzw. gelöscht werden, welche in keinem anderen Element verwendet werden. Methoden für diese Abfragen werden mit *Used* eingeleitet. Demzufolge liefert die Methode *Used(long elementOID)* *true*, wenn das gegebene Element verwendet bzw. referenziert wird. Wird diese Methode bspw. im *TaskManager* für einen *Task* aufgerufen, wird geprüft, ob dieser auf ein natives Modellelement verweist oder für die Bildung einer *Function* verwendet wird. Nur wenn beide Kriterien nicht erfüllt sind, darf der *Task* gelöscht bzw. bearbeitet werden. Neben dieser allgemeinen Methode werden eine Reihe weiterer, spezifische Methoden angeboten, um die Verwendung von Elementen zu ermitteln. So bspw. die Methode *UsedInFunction(long processObjectOID)* des *ProcessObjectManager*, mit welcher festgestellt werden kann, ob das gegebene *ProcessObject* in einer *Function* verwendet wird.

### Speicherung

Des Weiteren definiert die allgemeine Schnittstelle Methoden für die Speicherung und das Löschen von Elementen. Diese werden sowohl mit (*bool Save(...)* bzw. *bool Delete(...)*) als auch ohne Transaktionskontrolle (*bool SaveUncommitted(...)* bzw. *bool DeleteUncommitted(...)*) angeboten. Die Methoden ohne Transaktionskontrolle werden benötigt, da bei der Persistierung der Ontologielemente neben den grundsätzlichen Attributen auch sämtliche von diesen referenzierten Elemente bzw. Listen gespeichert werden. Die allgemeine Vorgehensweise bei der Speicherung von Elementen wird nun in den folgenden Absätzen anhand eines Beispiels veranschaulicht.

Die Abbildungen 5.20 und 5.21 zeigen die Methoden *Save(...)* bzw. *SaveUncommitted(...)* des *FunctionManager*, an welchen die Speicherung der Ontologielemente stellvertretend für alle Managerklassen gezeigt werden soll. Die *Save*-Methode verwaltet die Transaktionskontrolle und ruft die *SaveUncommitted*-Methode auf, welche

```

/// <summary>
/// Saves the given Function and all its relations to database.
/// Transaction control is used.
/// </summary>
/// <param name="function">The Function to save.</param>
/// <returns>true, if saving was successful, otherwise false.</returns>
public bool Save(Function function)
{
    if (function != null)
    {
        database.Begin();

        if (!this.SaveUncommitted(function))
        {
            database.Rollback();
            return false;
        }

        database.Commit();
    }
    return true;
}

```

Abbildung 5.20.: Methode - FunctionManager *Save(...)*

die eigentliche Speicherung durchführt. Wie bereits einleitend erwähnt, werden bei der Speicherung einer Ontologieinstanz sämtliche, mit dieser in Beziehung stehende Objekte ebenfalls gespeichert. Hierfür verarbeitet die Methode die folgenden Schritte:

- Zuerst wird die Methode *saveRelatedObjects(...)* aufgerufen, welche sämtliche mit der Ontologieinstanz in Beziehung stehende Objekte speichert. Für das Beispiel *FunctionManager* werden daher der *Task*, das *Process Object* und gegebenenfalls der *Parameter*, aus welchen sich die *Function* zusammensetzt, gespeichert, indem die *SaveUncommitted*-Methoden der entsprechenden Managerklassen aufgerufen werden.

```

/// <summary>
/// Saves the given Function an all its relations to database.
/// NO Transaction control is used.
/// </summary>
/// <param name="function">The Function to save.</param>
/// <returns>true, if saving was successful, otherwise false.</returns>
public bool SaveUncommitted(Function function)
{
    // save related objects (Method, ProcessObject, Parameter).
    if (!this.saveRelatedObjects(function))
        return false;

    // save Basic Properties
    if (!this.saveBasicProperties(function))
        return false;

    // save references (List of NativeModelItems, LocalEvent).
    if (!this.saveReferences(function))
        return false;

    return true;
}

```

Abbildung 5.21.: Methode - FunctionManager *SaveUncommitted(...)*

- Wurden die referenzierten Objekte erfolgreich gespeichert, werden mittels der Methode *saveBasicProperties(...)* die grundlegenden Attribute der **Function** gespeichert.
- Abschließend werden, nach erfolgreicher Speicherung der grundlegenden Attribute, sämtliche Beziehungen des Ontologieelements persistiert. Für eine **Function** werden hierbei die Referenzen zu den nativen Modellelementen sowie die Beziehung zum Trivialereignis gespeichert, indem wiederum die entsprechenden *SaveUncommitted*-Methoden der Managerklassen aufgerufen werden.

Wurden alle zuvor aufgelisteten Speicheraufrufe erfolgreich abgeschlossen, wird die *Commit*-Anweisung der *Save*-Methode ausgeführt, und die Speicherung ist abgeschlossen.

```

/// <summary>
/// Gets or creates the Condition for the given Elements.
/// </summary>
/// <param name="processObject">The ProcessObject of the Condition.</param>
/// <param name="parameter">The Parameter of the Condition (optional).</param>
/// <param name="state">The State of the Condition.</param>
/// <returns>The existing or newly created Condition.</returns>
public Condition Create(ProcessObject processObject, Parameter parameter, State state)
{
    Condition condition = null;
    // if no element of the Condition is new, check if it already exists.
    if (!processObject.IsNew && !state.IsNew &&
        (parameter == null || !parameter.IsNew))
    {
        long parameter_OID = -1;
        if (parameter != null)
            parameter_OID = parameter.OID;
        condition = this.GetByAllParts(processObject.OID, parameter_OID, state.OID);
    }
    // if the Condition does not exist, create new one.
    if (condition == null)
    {
        condition = new Condition(database);
        // set name.
        condition.Name = processObject.Name;
        if (parameter != null)
            condition.Name += " " + parameter.Name;
        condition.Name += " " + state.Name;
    }
    // set current references anyway.
    condition.ProcessObject = processObject;
    condition.Parameter = parameter;
    condition.State = state;

    return condition;
}

```

Abbildung 5.22.: Methode - ConditionManager *Create(...)*

## Factory-Methoden

Um neue Ontologieelemente anzulegen werden sogenannte Factory-Methoden von den jeweiligen Managerklassen angeboten. Diesen werden die, zur Bildung benötigten Elemente übergeben und liefern eine entsprechende Instanz, des zu generierenden Elements. Hierfür wird zuerst in der Datenbank nach diesem Element gesucht.

Konnte das geforderte Ontologieelement gefunden werden, wird diese Instanz zurückgegeben. Wurde das benötigte Element noch nicht in die Ontologie integriert, wird eine neue Instanz gebildet und zurückgegeben.

Abbildung 5.22 veranschaulicht dies anhand der *Create*-Methode des **ConditionManager**, welcher für ein **ProcessObject**, einen optionalen **Parameter** und einen **State** die entsprechende **Condition** zurückgibt. Ist keines dieser Element neu, sucht die Methode *GetByAllParts(...)* des **ConditionManager** nach einem Element mit den gegebenen Parametern in der Datenbank. Konnte dieses gefunden werden wird dieses zurückgegeben. Konnte die **Condition** nicht gefunden werden, wird eine neue Instanz erzeugt und zurückgegeben.

Einen besonderen Stellenwert unter den Factory-Methoden nehmen die Methode *CreateLocalCondition(Function)* des **ConditionManager** und die Methode *CreateGeneratingFunction(Condition)* des **FunctionManager** ein. Diese Methoden generieren zum gegebenen Element das lokale Gegenstück, d.h. die Methode *CreateLocalCondition* erzeugt für die gegebene **Function** das entsprechende Trivialereignis (vgl. Abb. 5.23). Die Methode ermittelt zuerst den entsprechenden **State** für den **Task** der gegebenen **Function**. Dies erfolgt mittels der Methode *CreateLocalState* des **StateManager**, welcher für einen gegebenen **Task** das entsprechende Gegenstück erzeugt bzw. die bestehende Instanz aus der Ontologie zurückgibt. Daraufhin wird die, im vorhergehenden Absatz beschriebene *Create*-Methode des **ConditionManager** aufgerufen, welche entweder eine bestehende Instanz zurückgibt oder eine neue erzeugt.

```

/// <summary>
/// Gets or creates the local Condition for the given Function.
/// </summary>
/// <param name="function">The Function to get or create the local Condition for.</param>
/// <returns>The local Condition for the given Function, or null.</returns>
public Condition CreateLocalCondition(Function function, out bool passiveVerbExists)
{
    passiveVerbExists = true;
    Condition condition = null;

    // check if necessary elements are set.
    if (function != null && function.Method != null && function.ProcessObject != null)
    {
        // get local State
        if (this.stateManager == null)
            this.stateManager = new StateManager(database);

        State state = this.stateManager.CreateLocalState(function.Method, out passiveVerbExists);
        if (state != null)
        {
            condition = this.Create(function.ProcessObject, function.Parameter, state);
        }
    }
    return condition;
}

```

Abbildung 5.23.: Methode - **ConditionManager** *CreateLocalCondition(...)*

### 5.3.3. Analyseklassen

In diesem Unterkapitel werden die beiden Klassen, welche die Methoden für die Analysen dieser Arbeit kapseln, überblicksmäßig vorgestellt. Wie bereits in Kapitel 5.1 dargestellt, wurden diese im Modul `AnalysisServices` im Bereich der `Semantic Process Model Analysis` implementiert. Sie bilden die ersten beiden Schritte des `Knowledge Extraction Service` und dienen der semantischen Aufbereitung der Prozessmodelle.

#### ProcessObjectIdentification

Ziel dieses Analyseschrittes ist sowohl die Identifikation der `Process Objects` und deren Beziehungen als auch die semantische Aufbereitung der EPK-Funktionen und -Ereignisse entsprechend der Konzepte der Ontologie. Die Klasse `ProcessObjectIdentification` verarbeitet hierfür die nativen Prozessmodelle und die Ergebnisse der lexikalischen Analyse gemäß der Vorgehensweise in Kapitel 4.2. Ziel der folgenden Absätze ist es, anhand von ausgewählten Programmausschnitten die wesentlichen Teile dieser Analyse-Klasse zu beschreiben und somit einen Überblick über die Implementierung dieses Schrittes zu geben.

Die Analyse wird durch die Methode `ExecuteAnalysisService()` gestartet, woraufhin die einzelnen Modellelemente verarbeitet werden. Abbildung 5.24 zeigt den Kern des Analysealgorithmus, welcher für jedes Modellelement ausgeführt wird. Zunächst wird die syntaktische Representation des Modellelements aus dem `ProcessDictionary` geladen (`getDictionaryEntriesForModelItem(...)`), welche sequenziell abgearbeitet wird. Entsprechend der definierten Zuordnungsvorschrift (vgl. Kap. 4.2 Tab. 4.1) werden durch die Unterscheidung der verschiedenen Worttypen die jeweiligen Methoden zur Aufbereitung der unterschiedlichen Ontologieelemente aufgerufen, welche in entsprechenden Listen gesammelt werden. Die allgemeine Vorgehensweise dieser Methoden sieht vor, dass zunächst die entsprechende Liste für das Ontologieelement durchsucht wird, um gegebenenfalls die bereits im Rahmen dieses Modells extrahierte Instanz zu verwenden. Konnte das Element in dieser Liste nicht gefunden werden, wird die entsprechende Factory-Methode der Managerklasse aufgerufen, welche entweder die bereits existierende Instanz der Ontologie oder eine neue Instanz zurückgibt (vgl. Kap. 4.2 Abb. 4.3). Neben der Generierung des Elements selbst, wird auch die Referenz auf das Modellelement gesetzt. Die folgende Aufzählung stellt nun dar, welche Methoden durch die unterschiedlichen Worttypen aufgerufen werden und fasst kurz zusammen, welche Elemente erstellt werden. Im Anschluss daran wird anhand einer konkreten Methode die Vorgehensweise nochmals bspw. dargestellt.

- Der Worttyp **Verb** ruft die Methode `processVerb(...)` auf, welche in Abhängigkeit vom Typ des Modellelements ein `Task` (EPK-Funktion) oder einen `State` (EPK-Ereignis) bildet.



```

// get the syntactical representation of the ModelItem
IList<DictionaryEntry> modelItemSyntaxList =
    Dictionary.getDictionaryEntriesForModelItem(database, nativeModelItem.OID);

for (int i = 0; i < modelItemSyntaxList.Count; i++)
{
    DictionaryEntry dictionaryEntry = modelItemSyntaxList[i];
    switch (dictionaryEntry.Type)
    {
        case WordType.Verb:
            // depending on the current ModelItemType a Method or a State is set.
            this.processVerb(nativeModelItem, dictionaryEntry);
            break;
        case WordType.Adjective:
        case WordType.Article:
        case WordType.Noun:
            // initiate the creation of a ProcessObject.
            this.processProcessObject(nativeModelItem, modelItemSyntaxList, ref i);
            break;
        case WordType.Preposition:
            // initiates the creation of a Parameter.
            this.processParameter(nativeModelItem, modelItemSyntaxList, ref i);
            break;
        case WordType.Conjunction:
            // initiates the split of the ModelItem. If required elements are set so far
            // a Function or an Event are set, otherwise preparing of elements continues.
            this.processConjunction(nativeModelItem, modelItemSyntaxList, ref i);
            break;
        default: break;
    }
}

if (nativeModelItem.ModelItemTypeForAnalysis == ModelItemTypeForAnalysis.Function)
{
    // initiates the creation of a Function.
    this.processFunction(nativeModelItem);
}
else if (nativeModelItem.ModelItemTypeForAnalysis == ModelItemTypeForAnalysis.Event)
{
    // initiates the creation of an Condition.
    this.processCondition(nativeModelItem);
}
}

```

Abbildung 5.24.: Programmausschnitt - Aufbereitung der Modellelement-Syntax

- Die Worttypen **Adjective**, **Article** und **Noun** rufen die Methode *processProcessObject(...)* auf, welche ein `Process Object` und die Generalisierungshierarchie für dieses bildet.
- Ein Wort vom Typ **Preposition** ruft die Methode *processParameter(...)* auf, welche für die Generierung eines `Parameters` zuständig ist. Diese Methode löst auch die Aufbereitung des folgenden `Process Objects` aus, da dieses für den Parameter benötigt wird.
- Ein Wort vom Typ **Conjunction** ruft die Methode *processConjunction(...)* auf. Tritt eine Konjunktion im Modellelement auf, wird dieses aufgeteilt (vgl. Kap. 4.2 Tab. 4.2 für eine Übersicht über die Aufteilungsregeln am Beispiel einer EPK-Funktion).

Abbildung 5.25 zeigt die Methode *processProcessObject(...)*, mit welcher die allgemeine Vorgehensweise anhand eines Beispiels dargestellt werden soll. Als Parameter werden der Methode das Modellelement (`NativeModelItem`), dessen syntaktische

```

/// <summary>
/// Initiates processing of a ProcessObject.
/// </summary>
/// <param name="nativeModelItem">The NativeModelItem.</param>
/// <param name="modelItemSyntaxList">The syntactical representation.</param>
/// <param name="index">The current index.</param>
private void processProcessObject(NativeModelItem nativeModelItem,
                                  IList<DictionaryEntry> modelItemSyntaxList, ref int index)
{
    // create Process Object.
    ProcessObject processObject = this.buildProcessObject(modelItemSyntaxList, ref index);

    // set reference to the current NativeModelItem.
    setPO_ModelItem_Relation(processObject, nativeModelItem,
                             ProcessObjectUsage.ProcessObject, this.currProcessObjectSeq);

    // add Process Object to list for current NativeModelItem
    this.processObjectForModelItemList.Add(processObject);
    this.currProcessObjectSeq++;
    this.lastSetElement = processObject;
}

```

Abbildung 5.25.: Methode - ProcessObjectIdentification *processProcessObject(...)*

Representation (`IList<DictionaryEntry>`) und der aktuelle Index innerhalb dieser Liste übergeben. Zuerst wird die Methode *buildProcessObject(...)* aufgerufen, welche das `ProcessObject` extrahiert (vgl. Abb. 5.26 und den folgenden Absatz). Im Anschluss daran wird mit der Methode *setPO\_ModelItem\_Relation(...)* die Beziehung zwischen dem `ProcessObject` und dem Modellelement gesetzt. Der Parameter `currProcessObjectSeq` legt dabei die Reihenfolge des `ProcessObjects` innerhalb des Modellelements fest.

Abbildung 5.26 zeigt nun jene Methode, welche für die eigentliche Generierung des `Process Objects` zuständig ist. Mit der Methode *getProcessObjectSyntax(...)* werden zunächst jene Einträge aus der syntaktischen Representation des Modellelements extrahiert, welche für die Bildung des `Process Objects` benötigt werden. Ferner setzt diese Methode bereits die Bezeichnung des Elements zusammen (`out name`). Im Anschluss daran durchsucht die Methode *getProcessObjectFromList(...)* jene Liste mit `Process Objects`, welche bereits im Rahmen der Analyse des vorliegenden Prozessmodells extrahiert wurden und gibt dieses gegebenen falls zurück. Konnte keine Instanz gefunden werden, wird die Factory-Methode des `ProcessObjectManager` aufgerufen, welche entweder eine bestehende Instanz aus der Ontologie oder eine neu generierte zurückgibt. Abschließend werden die Generalisierungshierarchie gebildet (*buildGeneralizations(...)*) und die Häufigkeit des `ProcessObjects` erhöht.

Nachdem die gesamte syntaktische Representation eines Modellelements auf diese Weise verarbeitet wurde, werden entsprechend dem vorliegenden Modellelementtyp die aufbereiteten Ontologieelemente zu einer `Function` (*processFunction(...)*) bzw. einer `Condition` (*processCondition(...)*) zusammengesetzt.

Wurden sämtliche Modellelemente des vorliegenden Modells auf diese Weise ver-

```

/// <summary>
/// Initiates the process of setting the ProcessObject for the given SyntaxList.
/// </summary>
/// <param name="modelItemSyntaxList">The SyntaxList of the ModelItem.</param>
/// <param name="startIndex">Startindex for ProcessObject within the ModelItemSyntax.</param>
/// <returns>The ProcessObject.</returns>
private ProcessObject buildProcessObject(IList<DictionaryEntry> modelItemSyntaxList,
                                         ref int startIndex)
{
    string name;
    // get syntactical presentation and name for ProcessObject.
    IList<DictionaryEntry> processObjectSyntaxList =
        this.getProcessObjectSyntax(out name, modelItemSyntaxList, ref startIndex);

    ProcessObject processObject = getProcessObjectFromList(name);
    if (processObject == null)
    {
        // create or get ProcessObject.
        processObject = this.processObjectManager.Create(name, processObjectSyntaxList);
        processObject.CreationType = CreationType.FromNativeModel;
        this.processObjectList.Add(processObject);

        // build Generalizations.
        if (processObject.Syntax.Count > 1)
            buildGeneralizations(processObject);
    }
    processObject.Frequency++;

    return processObject;
}

```

Abbildung 5.26.: Methode - ProcessObjectIdentification *buildProcessObject(...)*

arbeitet werden abschließend die Migrationsbeziehungen und die Beziehungen zwischen **Function** und lokaler **Condition** gesetzt.

Die Methode *setProcessObjectMigration()* ist für die Generierung der Migrationsbeziehungen zuständig. Diese ermittelt für jede EPK-Funktion die nachfolgende bzw. die nachfolgenden EPK-Funktion(en). Die den EPK-Funktionen hinterlegten **Process Objects** werden daraufhin zur Migrationsbeziehung zusammengefasst, welche den beiden **Process Objects** zugewiesen wird (vgl. Beschreibung der Migrationsbeziehungen in Kap. 5.3).

Den Abschluss dieses Analyseschritts bildet die Generierung der Beziehungen zwischen den **Functions** und ihren lokalen **Conditions** (Trivialereignisse). Die Methode *setFunction\_ConditionRelations()* prüft für jede aufbereitete **Function**, ob eine entsprechende lokale **Condition** im vorliegenden Modell gefunden wurde und setzt die entsprechende Beziehung **Function\_Condition**, welche sowohl der **Function** als auch der **Condition** zugewiesen wird. Konnte keine entsprechende **Condition** gefunden werden, wird dieser mittels der **Factory-Methode** des **ConditionManager** (vgl. Kapitel 5.3.2 Abb. 5.23) automatisch generiert.

### ProcessActivityAnalysis

Ziel dieses Analyseschritts ist die Ermittlung der Kontextinformationen für die EPK-Funktionen, d.h. die Generierung der **ProcessActivities** der Ontologie. Den Input

für diesen Analyseschritt bilden die nativen Prozessmodelle sowie die, den Modell-elementen annotierten `Functions` und `Conditions`.

Die Klasse `ProcessActivityAnalysis` kapselt die Methoden, welche für diesen Analyseschritt notwendig sind. Sie implementiert im Wesentlichen das Vorgehensmodell, welches bereits im Rahmen der konzeptionellen Beschreibung beschrieben wurde (vgl. Kap. 4.3).

```

/// <summary>
/// Sets the ProcessActivity/Activities for the given NativeModelItem.
/// </summary>
/// <param name="nativeModelItem">NativeModelItem to set ProcessActivity/Activities for.</param>
private void processProcessActivity(NativeModelItem nativeModelItem)
{
    this.functionForModelItemList =
        this.functionManager.GetListForNativeModelItem(nativeModelItem.OID);

    Function predFunction = null;

    // split Functions are built as a sequence.
    for (int index = 0; index < this.functionForModelItemList.Count; index++)
    {
        this.currFunction = this.functionForModelItemList[index];

        this.enumActivityType = ActivityType.Regular;

        // InitialContext
        this.processInitialContext(nativeModelItem, predFunction, index);

        // ResultingContext
        this.processResultingContext(nativeModelItem, index);

        predFunction = this.currFunction;

        // set AnalysisActivity
        this.buildProcessActivity(nativeModelItem, index);
    }
}

```

Abbildung 5.27.: Methode - `ProcessActivityAnalysis processProcessActivity(...)`

Die Analyse wird durch die Methode `ExecuteAnalysisService()` angestoßen. Diese lädt die einzelnen Modellelemente des vorliegenden Prozessmodells und stößt für jede EPK-Funktion die Generierung der entsprechenden `ProcessActivity` an. Hierfür wird die Methode `processProcessActivity(...)` aufgerufen, welche in Abbildung 5.27 dargestellt wird. Zuerst werden die, der EPK-Funktion hinterlegten `Functions` geladen, welche in weiterer Folge einzeln verarbeitet werden. Für jede `Function` werden durch die Methoden `processInitialContext(...)` und `processResultingContext(...)` die entsprechenden Kontextinformationen ermittelt und in globalen Variablen zwischengespeichert (vgl. folgender Absatz). Daraufhin wird die `ProcessActivity` aus der `Function` und diesen Kontextinformationen zusammengesetzt. Hierfür ist die Methode `buildProcessActivity(...)` zuständig, welche, entsprechend der Vorgehensweise bei der Ermittlung der übrigen Ontologieelemente, zuerst prüft, ob die `ProcessActivity` bereits im vorliegenden Modell identifiziert wurde. Konnte keine entsprechende Instanz gefunden werden, wird die Factory-Methode des `ProcessActivityManager` aufgerufen, welche entweder eine bestehende Instanz aus der Ontologie oder eine neu generierte zurückgibt. Dieser Instanz wird schließlich eine Referenz auf die

EPK-Funktion zugewiesen und in einer Liste zwischengespeichert.

Wesentlicher Bestandteil dieses Analyseschritts ist die Ermittlung der Kontextinformationen. Hierfür sind wie bereits erwähnt die Methoden *processInitialContext(...)* bzw. *processResultingContext(...)* zuständig, welche die vorgelagerten bzw. nachfolgenden Modellelemente der EPK-Funktion untersuchen und zusammenfassen. Da die Vorgehensweise beider Algorithmen ähnlich ist und bereits im Rahmen der konzeptionellen Beschreibung sehr ausführlich dargestellt wurde (vgl. Kap. 4.3), wird diese in den folgenden Abschnitten lediglich für die Generierung des **InitialContexts** kurz skizziert.

Im folgenden Programmausschnitt tritt bereits die Klasse **AnalysisCondition** auf. Bei dieser handelt es sich um ein Hilfskonstrukt, welches für die Bildung eines zusammengesetzten Kontexts (**ComplexContext**) benötigt und daher erst an dieser Stelle näher beschrieben wird.

```

/// <summary>
/// Processes the InitialContext for the current Function.
/// </summary>
/// <param name="nativeModelItem">The NativeModelItem of the current Function.</param>
/// <param name="predFunction">The previous Function of the current Function
/// (if the NativeModelItem has been split), or null.</param>
/// <param name="index">The index of the current Function within the NativeModelItem.</param>
private void processInitialContext(NativeModelItem nativeModelItem,
                                   Function predFunction, int index)
{
    if (index == 0)
    {
        // initialize temporary Analysis lists.
        this.currAnalysisEventList = new List<AnalysisCondition>();
        this.currAnalysisContextList = new List<AnalysisContext>();
        // set ComplexContext for ProcessActivity.
        this.setInitialContext(nativeModelItem, false, 0);
    }
    else
    {
        // if the original Function was split, local Condition of the previous Function is used.
        Condition initialCondition = this.getLocalConditionForFunction(predFunction);
        // set AnalysisCondition.
        AnalysisCondition analysisCondition =
            this.buildAnalysisCondition(initialCondition, this.currFunction, 0, true);
        // set SimpleContext for ProcessActivity.
        this.initialAnalysisContext =
            this.setSimpleContext(analysisCondition, ContextBaseType.Initial);
    }
}

```

Abbildung 5.28.: Methode - *ProcessActivityAnalysis processInitialContext(...)*

Ausgangspunkt für die Ermittlung des **InitialContexts** bildet die Methode *processInitialContext(...)*, welche in Abbildung 5.28 dargestellt wird. Diese prüft zuerst, ob es sich bei der aktuell zu bearbeitenden **Function** um die erste innerhalb des Modellelements handelt (**index == 0**). Ist dies der Fall wird die Ermittlung der Kontextinformation auf Basis der vorgelagerten Modellelemente mittels der rekursiven Methode *setInitialContext(...)* durchgeführt, welche im folgenden Abschnitt beschrieben wird. Ist der **index** größer als 0, sind dem Modellelement mehrere **Functions** hinterlegt. Für diese wird jeweils ein einfacher Kontext aus der loka-

len *Condition* der im Modellelement vorgelagerten *Function* gebildet (Methode *setSimpleContext(...)*).

Abbildung 5.29 zeigt die rekursive Methode, welche den *InitialContext* auf Basis des vorliegenden Prozessmodells ermittelt. Die Rekursion durchwandert solange die vorgelagerten Modellelemente des gegebenen Elements (*NativeModelItem nativeModelItem*), bis entweder keine Vorgänger mehr gefunden werden oder eine *Condition* identifiziert wurde. Grundsätzlich werden zuerst alle vorgelagerten Modellelemente besucht und die *Conditions* in einer temporären Listen gesammelt. Hierfür werden zunächst die einzelnen Modellelementtypen unterschieden:

```

/// <summary>
/// Sets the initial Context for the given Function.
/// </summary>
/// <param name="nativeModelItem">The current NativeModelItem.</param>
/// <param name="conditionSet">Determines, if an Condition has been set.</param>
/// <param name="level">Determines the number of visited Connectors.</param>
private void setInitialContext(NativeModelItem nativeModelItem, bool conditionSet, long level)
{
    if (nativeModelItem.PredecessorIDs == null || conditionSet)
    {
        conditionSet = false;
        return;
    }
    foreach (long predOID in nativeModelItem.PredecessorIDs)
    {
        NativeModelItem predModelItem = this.getModelItemFromList(predOID);

        if (predModelItem.ModelItemTypeForAnalysis == ModelItemTypeForAnalysis.Event)
        {
            // get last Condition (last, because of split Functions/Conditions)
            Condition condition = this.getLastCondition(predModelItem.OID);

            this.setAnalysisCondition(condition, level, true);
            conditionSet = true;
        }
        else if (predModelItem.ModelItemTypeForAnalysis == ModelItemTypeForAnalysis.Function)
        {
            // get local Condition of predecesing Function (last, because of split Functions)
            Condition condition = this.getLocalCondition(predModelItem.OID);
            if (condition != null)
            {
                this.setAnalysisCondition(condition, level, true);
                conditionSet = true;
            }
        }
        else if (predModelItem.ModelItemTypeForAnalysis == ModelItemTypeForAnalysis.Operator)
        {
            // if Connector has more than one incoming arcs, set Type and increase level.
            conditionSet = false;
            if (predModelItem.PredecessorIDs.Length > 1)
            {
                level++;
            }
        }
        // recursive call.
        this.setInitialContext(predModelItem, conditionSet, level);
        // generate Context with collected Events.
        this.buildInitialContext(predModelItem, ref level);
    }
}

```

Abbildung 5.29.: Methode - *ProcessActivityAnalysis setInitialContext(...)*

- Tritt ein Operator auf (*ModelItemType.Operator*), welcher mehrere eingehende Kanten besitzt (Join-Operator), wird die Variable *level* um eins erhöht.

Diese Variable zählt die Anzahl relevanter Operatoren zwischen der EPK-Funktion und den vorgelagerten Elementen, um diese bei der Bildung des zusammengesetzten Kontexts der entsprechenden Ebene zuordnen zu können (vgl. Beispiel in Abb. 5.30). Als irrelevant können bei der Bildung des **InitialContexts** jene Operatoren gesehen werden, welche lediglich eine eingehende Kante besitzen (Split-Operatoren) und daher vernachlässigt werden können.

- Für Modellelemente vom Typ EPK-Ereignis (`ModelItemType.Event`) bzw. EPK-Funktion (`ModelItemType.Function`) werden die entsprechenden **Conditions** aus der Ontologie gesetzt. Für ein EPK-Ereignis wird hierbei die letzte hinterlegte **Condition** verwendet (d.h. bei einem aufgespaltenen EPK-Ereignis wird die letzte, hinterlegte **Condition** verwendet). Handelt es sich bei dem Modellelement hingegen um eine EPK-Funktion, wird die automatisch generierte lokale **Condition** der letzten hinterlegten **Function** verwendet. Die auf diese Weise ermittelte **Condition** wird mittels der Methode `setAnalysisCondition(...)` in die Liste mit bereits für die aktuell bearbeitete EPK-Funktion identifizierten **Conditions** aufgenommen. Hierbei wird die bereits angesprochene **AnalysisCondition** gebildet, welcher neben der **Condition** die aktuelle Ebene, d.h. die Variable `level` kapselt.

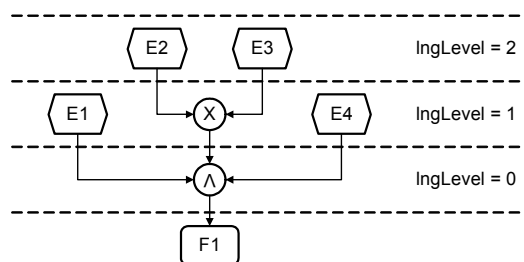


Abbildung 5.30.: Beispiel - Ebenen für Generierung des Context

Wie bereit erwähnt, bricht die Rekursion ab, wenn eine **Condition** gesetzt wurde. Beim Auflösen der Rekursion wird die Method `buildInitialContext(...)` aufgerufen, welche aus den gesammelten Daten den **InitialContext** zusammensetzt. Der erstellte Kontext wird daraufhin einer globalen Variable zugewiesen, welche für die Bildung der **ProcessActivity** herangezogen wird. Die Methode generiert den Kontext wenn eine der folgenden Bedingungen auftritt:

- Wurde kein Join-Operator identifiziert (d.h. `level == 0`) und es wurde eine **Condition** gesetzt, wird ein einfacher Kontext gebildet und der globalen Variable zugewiesen.
- Wurden ein oder mehrere Operatoren identifiziert, wird die Erstellung des zusammengesetzten Kontexts erst dann gestartet, wenn beim Auflösen der Rekursion das aktuelle Modellelement `predModelItem` vom Typ **Operator**, genauer gesagt ein Join-Operator ist. Dadurch wird ein zusammengesetzter

Kontext erst dann gebildet, wenn alle diesem vorgelagerten Elemente bereits verarbeitet wurden.

Nachdem ein zusammengesetzter Kontext gebildet wurde, wird dieser ebenfalls in einem Hilfskonstrukt, dem `AnalysisContext` gekapselt, welchem die entsprechende Ebene zugewiesen wird. Diese `AnalysisContexts` werden für jeden, zu bildenden `InitialContext` in einer eigenen Liste gesammelt.

Bei der Bildung des Kontexts durchsucht der Algorithmus die beiden Listen mit den `AnalysisConditions` und den `AnalysisContexts` nach jenen Elementen, welche auf der, dem Operator übergeordneten Ebene liegen. Mit diesen `Conditions` und den bereits generierten, verschachtelten `Contexts` wird nun der `Context` für den aktuellen Operator gebildet.

In Tabelle 5.1 werden die einzelnen Schritte zur Bildung des `InitialContexts` für das allgemeine Beispiel in Abbildung 5.30 dargestellt. In der ersten Spalte werden die Parameter der rekursiven Methode `setInitialContext` beim Durchwandern des nativen Prozessmodells dargestellt. Die beiden Spalten für die Methoden `setAnalysisCondition` und `buildInitialContext` zeigen die Listen mit den bereits identifizierten `AnalysisConditions` bzw. `AnalysisContexts`, wobei neue Instanzen hervorgehoben werden.

- 1. Schritt:** Es werden die anfänglichen Werte der Methode `setInitialContext` dargestellt. Die Rekursion wird mit der EPK-Funktion `F1` gestartet, für welche der `InitialContext` gebildet werden soll. Die anderen Parameter werden zu Beginn mit `false` für `ConditionSet` und `0` für den `Level` belegt. Die beiden Listen zum Sammeln der `AnalysisConditions` bzw. `AnalysisContexts` werden neu initialisiert und sind daher leer.
- 2. Schritt:** Die Rekursion wandert zum nächsten vorgelagerten Element im Prozessmodell. Da es sich bei diesem um einen AND-Operator handelt, wird die aktuelle Ebene um eins erhöht (`Level = 1`). Da `ConditionSet` weiterhin `false` ist, wandert die Rekursion eine Ebene tiefer.
- 3. Schritt:** Die Rekursion erreicht das EPK-Ereignis `E1`, woraufhin die entsprechende `AnalysisCondition` mit der aktuellen Ebene gebildet und in der Liste zwischengespeichert wird. `ConditionSet` wird auf `true` gesetzt, wodurch die Rekursion abbricht. Da beim Auflösen der Rekursion das aktuelle `NativeModelItem` ein EPK-Ereignis ist, wird kein Kontext generiert.
- 4. Schritt:** Als nächstes Element wird ein XOR-Operator identifiziert, woraufhin die Variable `Level` wiederum um eins erhöht wird und die Rekursion eine Ebene tiefer wandert.
- 5. & 6. Schritt:** Es werden die beiden EPK-Ereignisse `E2` und `E3` erreicht. Aus diesen werden wiederum zwei `AnalysisConditions` mit `Level = 2` gebildet und der entsprechenden Liste zugewiesen.



<b>setInitialContext ( NMI, CS, Lvl )</b>	<b>setAnalysisCondition</b>	<b>buildInitialContext</b>
1. ( F1, false, 0 )	ACondL = { }	ACL = { }
2. ( And, false, 1 )	ACondL = { }	ACL = { }
3. ( E1, true, 1 )	ACondL = { <b>ACond [ Condition(E1), Lvl(1) ]</b> }	ACL = { }
4. ( Xor, false, 2 )	ACondL = { ACond [ Condition(E1), Lvl(1) ] }	ACL = { }
5. ( E2, true, 2 )	ACondL = { ACond [ Condition(E1), Lvl(1) ], <b>ACond [ Condition(E2), Lvl(2) ]</b> }	ACL = { }
6. ( E3, true, 2 )	ACondL = { ACond [ Condition(E1), Lvl(1) ], ACond [ Condition(E2), Lvl(2) ], <b>ACond [ Condition(E3), Lvl(2) ]</b> }	ACL = { }
7. ( Xor, false, 1 )	ACondL = { ACond [ Condition(E1), Lvl(1) ] }	ACL = { <b>AC [ C<sub>XOR</sub>(E2, E3), Lvl(1) ]</b> }
8. ( E4, true, 1 )	ACondL = { ACond [ Condition(E1), Lvl(1) ], <b>Cond [ Condition(E4), Lvl(1) ]</b> }	ACL = { AC [ C <sub>XOR</sub> (E2, E3), Lvl(1) ] }
9. ( And, false, 0 )	ACondL = { }	ACL = { <b>AC [ C<sub>AND</sub>(E1, E4, C<sub>XOR</sub>(E2, E3)), Lvl(0) ]</b> }

Legende: NMI ... NativeModellItem      ACond ... AnalysisCondition      AC ... AnalysisContext  
 CS ... ConditionSet                    ACondL ... AnalysisConditionList      ACL ... AnalysisContextList  
 Lvl ... Level                            C ... Context

Tabelle 5.1.: Beispiel - Generierung des InitialContext

- 7. Schritt:** Nachdem die beiden EPK-Ereignisse E2 und E3 verarbeitet wurden, wird die Rekursion einen Schritt weiter aufgelöst, da alle Element dieser Ebene bereits verarbeitet wurden. Beim Auflösen der Rekursion wird wieder der XOR-Operator erreicht, wodurch die Bildung eines zusammengesetzten Kontexts angestoßen wird. Hierfür durchsucht der Algorithmus die beiden Listen nach jenen Elementen, welche eine Ebene über dem Operator liegen. Da der Operator auf Ebene 1 identifiziert wurde, werden also jene Elemente gesucht, welchen Level == 2 zugewiesen wurden. Im vorliegenden Beispiel sind dies die beiden Conditions zu E2 und E3. Diese beiden werden nun zu einem Context mit dem ConnectionType = XOR zusammengefasst und aus der Liste mit bereits identifizierten AnalysisConditions entfernt. Für den Context wird ein AnalysisContext mit dem aktuellen Level = 1 gebildet und in der entsprechenden Liste zwischengespeichert.
- 8. Schritt:** Für das EPK-Ereignis E4 wird die AnalysisCondition mit Level = 1 gebildet und der entsprechenden Liste zugewiesen.

**9. Schritt:** Nachdem das EPK-Ereignis **E4** verarbeitet wurde, wurden alle Elemente der Ebene 1 besucht. Dadurch wird die Rekursion soweit aufgelöst, bis der AND-Operator erreicht und die Bildung des zusammengesetzten Kontexts ausgelöst wird. Hierfür sucht der Algorithmus nach allen Elementen, welche auf `Level == 1` identifiziert wurden. Im vorliegenden Beispiel sind das die beiden `Conditions` zu den EPK-Ereignissen **E1** und **E4** sowie der zuvor gebildete, zusammengesetzte `Context`. Da die Ebene 0 erreicht wurde, wird der gebildete `Context` der globalen Variable zugewiesen und die Ermittlung des `InitialContext` für die EPK-Funktion **F1** ist abgeschlossen.

## 6. Schlussbetrachtung

Die Bereitstellung von Prozessmuster, d.h. bewährte Prozesslösungen für bestimmte Problemstellungen, bildet ein geeignetes Mittel zur Unterstützung der Modellierung bzw. der Anpassung von Geschäftsprozessen. Da eine manuelle Identifikation bei einer Vielzahl an Prozessmodellen allerdings eine sehr komplexe und zeitaufwändige Aufgabe darstellt, stellt pModeler einen Ansatz zur automatisierten Extraktion von Prozessmustern aus bestehenden Prozessmodellen vor.

Prozessmodelle liegen zumeist in Form von Prozessbeschreibungen, wie bspw. der in dieser Arbeit verwendeten EPK Notation vor, deren Modellelemente mittels natürlicher Sprache modelliert werden. Ein automatisierter Vergleich der Modellelemente ist allerdings mit dem Problem der Mehrdeutigkeit der natürlichen Sprache konfrontiert, welche durch Sprachdefekte, wie bspw. Synonyme und Homonyme, auftritt.

Die Zielsetzung dieser Diplomarbeit lag daher in der Definition einer Ontologie, mit welcher die EPK-Funktionen und -Ereignisse und deren implizite Semantik auf eine klar definierte Weise beschrieben und den Modellelementen annotiert werden können, um die Vergleichbarkeit von EPK-Modellen zu erhöhen. Die Spezifikation der Ontologie umfasst im wesentlichen zwei Teilbereiche. Ersterer zur semantischen Aufbereitung der natürlich-sprachlichen Bezeichnungen von EPK-Funktionen und -Ereignissen und zweiterer zur Beschreibung von elementaren Tätigkeiten, d.h. von Prozessaktivitäten, welche Funktionen mit ihren vor- bzw. nachgelagerten Ereignissen in Beziehung setzen.

Beim Entwurf der Ontologie zur semantischen Beschreibung der EPK-Funktionen und -Ereignisse wurden Konzepte definiert, aus welchen sich deren natürlich-sprachlichen Bezeichnungen zusammensetzen. Hierfür wurden Konzepte zur Annotierung von Prozessobjekten (**Process Object**), Tätigkeiten (**Task**), Zuständen (**State**) und Parametern (**Parameter**) spezifiziert. Des weiteren wurden eine Reihe von Beziehungen zwischen den einzelnen Konzepten spezifiziert, um bspw. Generalisierungs- oder Kompositionshierarchien von Prozessobjekten abbilden zu können. Zur semantischen Beschreibung der Modellelemente der EPK-Modelle werden die zuvor genannten Konzepte miteinander verknüpft. Während eine EPK-Funktion (**Function**) aus einer Tätigkeit, einem Prozessobjekt und einem optionalen Parameter zusammengesetzt wird, wird ein EPK-Ereignis (**Condition**) durch ein Prozessobjekt, einen optionalen Parameter und einen Zustand beschrieben.

Der zweiten Teil der Ontologie umfasst jene Konzepte und Beziehungen zur semantischen Aufbereitung von Prozessaktivitäten (`ProcessActivity`). Eine Prozessaktivität bildet den kleinst-möglichen Modellierungsbaustein zur Beschreibung eines Prozessmusters und verknüpft eine Funktion mit ihren vor- bzw. nachgelagerten Ereignissen. Des Weiteren wird mit der Prozessaktivität sichergestellt, dass allen Funktionen der bestehenden Prozessmodelle ein Ergebnis zugewiesen wird, welches für die Ableitung der Prozessziele benötigt wird.

Die zweite Aufgabenstellung dieser Diplomarbeit lag in der automatisierten Aufbereitung der Modellelemente bestehender EPK-Modelle, welche in zwei Schritten durchgeführt wird. In einem ersten Schritt werden basierend auf den Ergebnissen einer lexikalischen Analyse mit Hilfe einer Reihe von Ableitungsregeln die einzelnen Teilkonzepte instanziiert, zu den Konzepten zur Beschreibung von EPK-Funktionen und -Ereignissen zusammengefasst und diesen semantisch annotiert. Eine wesentliche Zielsetzung dieser Aufbereitung liegt in einer Normalisierung der Prozessmodelle, indem zum einen durch die Verwendung eines standardisierten Vokabulars die Bezeichnungen der den Modellelementen annotierten Instanzen der Ontologie vereinheitlicht und zum anderen nicht modellierte Trivialereignisse automatisch generiert werden. Die semantische Annotierung des ersten Schritts bildet die Grundlage für die Aufbereitung der Prozessaktivitäten, in welcher die semantisch aufbereiteten Funktionen mit ihren vor- bzw. nachgelagerten semantisch aufbereiteten oder automatisiert generierten Ereignissen in Beziehung gesetzt werden.

Im Rahmen der prototypischen Implementierung von pModeler wurden die Ontologie, Komponenten für die Verwaltung der Ontologie sowie die spezifizierten Algorithmen zur semantischen Aufbereitung und Annotierung bestehender EPK-Modelle umgesetzt. Aufgrund der prototypischen Entwicklung und bereits bestehender Komponenten wurde die Ontologie mittels einer relationalen Datenbank (d.h. Microsoft Access) realisiert.

Auf Basis der spezifizierten Unklarheiten in EPK-Modellen, welche auf Grundlage von Prozessmodellen des Projektpartners Siemens erhoben werden konnten, wurden eine Reihe von Testmodellen erstellt. Bereits diese ersten Tests haben gezeigt, dass basierend auf einer sorgfältig durchgeführten, lexikalischen Analyse, eine automatisierte Aufbereitung und Annotierung von EPK-Modellen möglich ist. Des Weiteren wurde ersichtlich, dass durch die Verwendung eines standardisierten Vokabulars und der damit einhergehenden Beseitigung von syntaktischen Unklarheiten in den Modellelementbezeichnungen die Menge an Prozessobjekten reduziert werden kann.

Die prototypische Weiterentwicklung, welche nicht mehr Teil dieser Arbeit war, hat gezeigt, dass es mit den spezifizierten Konzepten grundsätzlich möglich ist, Prozessmuster aus bestehenden EPK-Modellen zu extrahieren. Allerdings liegen auf Grund des gegenwärtigen, sehr frühen Entwicklungsstands der implementierten Algorithmen zur automatisierten Identifikation von Prozessmustern noch keine um-

fangreichen Testergebnisse vor und es wurden vorerst lediglich einfache Modellvergleiche durchgeführt.

Die prototypische Entwicklung von pModeler bietet Raum für Verbesserungen und Erweiterungen. Im Zusammenhang mit dieser Arbeit sind diese sicherlich im Bereich der Ontologie zu sehen, wobei eine wesentlich Aufgabe in der Formalisierung der Ontologie mittels einer formalen Beschreibungssprache, wie bspw. OWL, liegt. Andere mögliche Erweiterungen seien im Folgenden lediglich kurz skizziert:

- Erweiterung der Ontologie um Konzepte und Beziehungen zur Abbildung der erweiterten Modellierungskonstrukte von eEPKs, d.h. zur Darstellung von Organisationseinheiten, Informationsobjekten und Anwendungssystemen, gegebenenfalls unterstützt durch die Einbindung bestehender Ontologien, wie bspw. einer Organisationsontologie.
- Erweiterung des Konzepts zur Darstellung von Prozessaktivitäten, um die Einbindung beteiligter Organisationseinheiten bzw. Anwendungssysteme, sowie von Input- und Outputobjekten zu ermöglichen.
- Erweiterung der Prozessnormalisierung um Mechanismen zur strukturellen Standardisierung. Die Analysen setzen wohl strukturierte EPK-Modelle voraus, d.h. bspw. dass zu jedem öffnenden Konnektor ein entsprechender schließender Konnektor vorhanden sein muss. Im Rahmen der Normalisierung könnten EPK-Modelle in einem ersten Schritt auf diese Anforderungen hin überprüft und (semi-)automatisch angepasst werden.

Die hohe Bedeutung, welche dem semantischen Geschäftsprozessmanagement beigemessen wird, lässt sich an der Vielzahl an Arbeit aus diesem Bereich in den letzten Jahren erkennen. pModeler liefert einen Beitrag zur semantischen Prozessmodellierung, indem auf Grundlage semantisch annotierter EPK-Modelle Prozessmuster automatisiert identifiziert werden, welche den Prozess der Modellierung unterstützen können. Die prototypische Implementierung von pModeler hat einerseits gezeigt, dass eine solche Identifikation grundsätzlich möglich ist, andererseits allerdings noch wesentlicher Forschungs- und Entwicklungsbedarf besteht, um diese auf solide Beine zu stellen.

# Literaturverzeichnis

- [Aal99] AALST, Wil M. P. d.: Formalization and Verification of Event-driven Process Chains. In: *Information & Software Technology* 41 (1999), S. 639 – 650
- [Aal04] AALST, Wil M. P. d.: Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management. In: DESEL, Jörg (Hrsg.) ; REISIG, Wolfgang (Hrsg.) ; ROZENBERG, Grzegorz (Hrsg.): *Lectures on Concurrency and Petri Nets* Bd. 3098, Springer, 2004 (Lecture Notes in Computer Science), S. 1–65
- [ABHK00] AALST, Wil M. P. d. ; BARROS, Alistair P. ; HOFSTEDE, Arthur H. M. ; KIEPUSZEWSKI, Bartek: Advanced Workflow Patterns. In: *Conference on Cooperative Information Systems*, 2000
- [AHW03] AALST, Wil M. P. d. ; HOFSTEDE, Arthur H. M. ; WESKE, Mathias: Business Process Management: A Survey. In: *International Conference on Business Process Management (BPM 2003)*, 2003
- [All05] ALLWEYER, Thomas: *Geschäftsprozessmanagement - Strategie, Entwurf, Implementierung, Controlling*. Herdecke [u.a.] : W3I Verlag, 2005
- [ATH05] ATHENA: Deliverable DA1.3.1 Report on Methodology description and guideline definition / Integrated Project ATHENA. 2005. – Deliverable
- [BCD<sup>+</sup>07] BELECHEANU, Roxana ; CABRAL, Liliana ; DOMINGUE, John ; GAALLOUL, Walid ; HEPP, Martin ; FILIPOWSKA, Agata ; KACZMAREK, Monika ; KACZMAREK, Tomasz ; NITZSCHE, Jörg ; NORTON, Barry ; PEDRINACI, Carlos ; ROMAN, Dumitru ; STOLLBERG, Michael ; STEIN, Sebastian: Business Process Ontology Framework / SUPER Project. 2007. – Deliverable
- [BDKK02] BECKER, Jörg ; DELFMANN, Patrick ; KNACKSTEDT, Ralf ; KUROPKA, Dominik: Konfigurative Referenzmodellierung. In: BECKER, Jörg (Hrsg.) ; KNACKSTEDT, Ralf (Hrsg.): *Wissensmanagement mit Referenzmodellen*. Physica Verlag, 2002, S. 25 – 144
- [BDW07] BORN, Matthias ; DÖRR, Florian ; WEBER, Ingo: User-friendly Semantic Annotation of Business Process Modeling. In: WESKE, Mathias

- (Hrsg.) ; HACID, Mohand-Saïd (Hrsg.) ; GODART, Claude (Hrsg.): *Web Information Systems Engineering - WISE 2007 Workshops*. Springer Verlag, 2007, S. 260–271
- [Bög07] BÖGL, Andreas: pModeler: Ein System zur semantischen Prozessmodellanalyse / Data And Knowledge Engineering Johannes Kepler Universität Linz. 2007. – Forschungsbericht
- [BK05] BECKER, Jörg ; KAHN, Dieter: Der Prozeß im Fokus. In: BECKER, Jörg (Hrsg.) ; KUGELER, Martin (Hrsg.) ; ROSEMAN, Michael (Hrsg.): *Prozessmanagement - Ein Leitfaden zur prozessorientierten Organisationsgestaltung*. Springer Verlag, 2005, S. 3–16
- [BKS08] BÖGL, Andreas ; KOBLE, Maximilian ; SCHREFL, Michael: Knowledge Acquisition from EPC Models for Extraction of Process Patterns in Engineering Domains. In: BICHLER, Martin (Hrsg.) ; HESS, Thomas (Hrsg.) ; KRUMHOLTZ, Helmut (Hrsg.) ; LECHNER, Ulrike (Hrsg.) ; MATTHEIS, Florian (Hrsg.) ; PICOT, Arnold (Hrsg.) ; SPEITKAMP, Benjamin (Hrsg.) ; WOLF, Petra (Hrsg.): *Proceedings der Multikonferenz Wirtschaftsinformatik 2008 (MKWI 2008)*, 2008, S. 1601–1612
- [Bor97] BORST, Willem N.: *Construction of Engineering Ontologies for Knowledge Sharing and Reuse*, University of Enschede, Diss., 1997
- [CFLGP06] CORCHO, Oscar ; FERNANDEZ-LOPEZ, Mariano ; GOMEZ-PEREZ, Asuncion: Ontological Engineering: Principles, Methods, Tools and Languages. In: *Ontologies for Software Engineering and Software Technology*. Springer Verlag, 2006, S. 1–48
- [Dit07] DITTMANN, Lars U. ; CORSTEN, Hans (Hrsg.) ; REISS, Michael (Hrsg.) ; STEINLE, Claus (Hrsg.) ; ZELEWSKI, Stephan (Hrsg.): *OntoFMEA: Ontologiebasierte Fehlermöglichkeits- und Einflussanalyse*. Gabler, 2007
- [FBGL98] FOX, Mark S. ; BARBUCEANU, Mihai ; GRUNINGER, Michael ; LIN, Jinxin: An Organization Ontology for Enterprise Modelling. In: PRIETULA, Michael J. (Hrsg.) ; CARLEY, Kathleen M. (Hrsg.) ; GASSER, Les (Hrsg.): *Simulating Organizations: Computational Models of Institutions and Groups*. AAAI Press / The MIT Press, 1998, S. 131–152
- [FCF93] FOX, Mark S. ; CHIONGLO, John F. ; FADEL, Fadi G.: A Common-Sense Model of the Enterprise. In: *Proceedings of the 2nd Industrial Engineering Research Conference*, 1993, S. 425–429
- [FE03] FÖRSTER, Alexander ; ENGELS, Gregor: Quality Ensuring Development of Software Processes. In: OQUENDO, Flávio (Hrsg.): *EWSPPT 2003*, 2003, S. 62–73

- [Fen04] FENSEL, Dieter: *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*. Springer-Verlag, 2004
- [FFG94] FADEL, Fadi G. ; FOX, Mark S. ; GRUNINGER, Michael: A Generic Enterprise Resource Ontology. In: *Proceedings of the 3rd Workshop on Enabling Technologies - Infrastructure for Collaborative Enterprises*, 1994, S. 117–128
- [FL02] FETTKE, Peter ; LOOS, Peter: Methoden zur Wiederverwendung von Referenzmodellen - Übersicht und Taxonomie. In: BECKER, Jörg (Hrsg.) ; KNACKSTEDT, Ralf (Hrsg.): *Referenzmodellierung 2002 - Methoden - Modelle - Erfahrungen. Tagungsband zur 6. Fachtagung Referenzmodellierung 2002*. Nürnberg, 2002, S. 9–33
- [FL03] FRANK, Ulrich ; LAAK, Bodo L.: Anforderungen an Sprachen zur Modellierung von Geschäftsprozessen / Institut für Wirtschaftsinformatik. Universität Koblenz-Landau, 2003 (34). – Arbeitsbericht
- [FSM<sup>+</sup>07] FANTINI, Paola ; SAVOLDELLI, Alberto ; MILANESI, Micaela ; CARIZZONI, Giulio ; KOEHLER, Jana ; STEIN, Sebastian ; ANGELI, Ralf ; HEPP, Martin ; ROMAN, Dumitru ; BRELAGE, Christian ; BORN, Matthias: Semantic Business Process Life Cycle / SUPER Project. 2007 (D2.2). – Deliverable
- [Gad05] GADATSCH, Andreas: *Grundkurs Geschäftsprozessmanagement*. Vieweg Verlag, 2005
- [GCSP06] GRANGEL, Reyes ; CHALMETA, Ricardo ; SCHUSTER, Stefan ; PENA, Inaki: Exchange of Business Process Models Using the POP\* Metamodel. In: BUSSLER, Christoph (Hrsg.) ; HALLER, Armin (Hrsg.): *Business Process Management Workshops: BPM 2005 International Workshops*. Springer-Verlag, 2006, S. 233–244
- [GDD06] GASEVIC, Dragan ; DJURIC, Dragan ; DEVEDZIC, Vladan: *Model Driven Architecture and Ontology Development*. Springer, 2006
- [GF94] GRUNINGER, Michael ; FOX, Mark S.: An Activity Ontology for Enterprise Modelling / Universität Toronto. 1994. – Forschungsbericht
- [GG95] GUARINO, Nicola ; GIARETTA, Pierdaniele: Ontologies and Knowledge Bases: Towards a Terminological Clarification. In: MARS, Nicolaas J. I. (Hrsg.): *Towards Very Large Knowledge Bases*. IOS Press, 1995, S. 25–32
- [GHA07] GRIMM, Stephan ; HITZLER, Pascal ; ABDECKER, Andreas: Knowledge Representation and Ontologies. In: STUDER, Rudi (Hrsg.) ; GRIMM, Stephan (Hrsg.) ; ABDECKER, Andreas (Hrsg.): *Semantic Web Services: Concepts, Technology and Applications*. Springer, 2007, S. 51–



106

- [GM03] GRÜNINGER, Michael ; MENZEL, Christopher: The Process Specification Language (PSL) - Theory and Applications. In: *AI Magazine* 24 (2003), Nr. 3, S. 63–74
- [GN87] GENESERETH, Michael R. ; NILSSON, Nils J.: *Logical Foundations of Artificial Intelligence*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1987
- [GPCFL04] GOMEZ-PEREZ, Asuncion ; CORCHO, Oscar ; FERNANDEZ-LOPEZ, Mariano: *Ontological Engineering: with examples from the areas of Knowledge Management, e-Commerce and the Semantic Web*. Springer, 2004
- [Grö06] GRÖMER, Alexandra: *Analyse von Prozessmodellen mit Hilfe von Data Mining Methoden*. Linz, Johannes Kepler Universität Linz, Diplomarbeit, 2006
- [Gru93a] GRUBER, Thomas R.: Toward Principles for the Design of Ontologies Used for Knowledge Sharing. In: GUARINO, Nicola (Hrsg.) ; POLI, Roberto (Hrsg.): *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Kluwer Academic Publishers, 1993, S. 907–928
- [Gru93b] GRUBER, Thomas R.: A Translation Approach to Portable Ontology Specifications. In: *Knowledge Acquisition*, 1993, S. 199–220
- [Gua97] GUARINO, Nicola: Semantic Matching: Formal Ontological Distinctions for Information Organization, Extraction and Integration. In: *Summer School of Information Extraction*, 1997
- [Gua98] GUARINO, Nicola: Formal Ontology and Information Systems. In: *International Conference On Formal Ontology In Information Systems FOIS'98*. Trento, Italy : Amsterdam, IOS Press, 1998, S. 3–15
- [Hag05] HAGEN, Mariele: *Definition einer Sprache zur Beschreibung von Prozessmustern zur Unterstützung agiler Softwareentwicklungsprozesse*, Universität Leipzig, Diss., 2005
- [Hes02] HESSE, Wolfgang: Aktuelles Schlagwort Ontologie(n). In: *Informatik Spektrum* Bd. 25, Heft 6 (2002), S. 477–480
- [HHR04] HEINRICH, L.J. ; HEINZL, A. ; ROITHMAYR, F.: *Wirtschaftsinformatik Lexikon*. 7. Auflage. München : Oldenbourg Wissenschaftsverlag GmbH, 2004
- [HLD<sup>+</sup>05] HEPP, Martin ; LEYMANN, Frank ; DOMINGUE, John ; WAHLER, Alexander ; FENSEL, Dieter: Semantic Business Process Management: A Vision Towards Using Semantic Web Services for Business Process Ma-

- agement. In: *IEEE International Conference on e-Business Engineering (ICEBE 2005)*, IEEE Computer Society, 2005, S. 535–540
- [HM03] HERMAN, George A. ; MALONE, Thomas W.: What is in the Process Handbook? In: MALONE, Thomas W. (Hrsg.) ; CROWSTON, Kevin (Hrsg.) ; HERMAN, George A. (Hrsg.): *Organizing Business Knowledge: The MIT Process Handbook*. MIT Press, 2003, S. 221–258
- [Hol95] HOLLINGSWORTH, David: The Workflow Reference Model / Workflow Management Coalition. 1995. – Forschungsbericht
- [Hos10] HOSTNIK, Thomas: *Realisierung der Nativen Prozessmodellverwaltung für pModeler, einem System zur semantischen Prozessmodellanalyse*, Johannes Kepler Universität Linz, Diplomarbeit, 2010
- [HR07] HEPP, Martin ; ROMAN, Dumitru: An Ontology Framework for Semantic Business Process Management. In: OBERWEIS, Andreas (Hrsg.) ; WEINHARDT, Christof (Hrsg.) ; GIMPEL, Henner (Hrsg.) ; KOSCHMIDDER, Agnes (Hrsg.) ; PANKRATIUS, Victor (Hrsg.) ; SCHNIZLER, Björn (Hrsg.): *Wirtschaftsinformatik (1)*, Universitätsverlag Karlsruhe, 2007, S. 423–440
- [Hüs03] HÜSSELMANN, C.: *Fuzzy-Geschäftsprozessmanagement*. Lohmar - Köln : Josef Eul Verlag GmbH, 2003
- [HSW97] HEIJST, G. van ; SCHREIBER, A. T. ; WIELINGA, B. J.: Using Explicit Ontologies in KBS Development. In: *International Journal of Human-Computer Studies* 46 (1997), S. 183–292
- [HZJ04] HENKEL, Martin ; ZDRAVKOVIC, Jelena ; JOHANNESSON, Paul: Service-based processes: design for business and technology. In: *IC-SOC '04: Proceedings of the 2nd international conference on Service oriented computing*, ACM Press, 2004, S. 21 – 29
- [Jab95] JABLONSKI, S.: *Workflow-Management-Systeme - Modellierung und Architektur*. Bonn[u.a.] : Internat. Thompson Publ., 1995
- [KFG99] KIM, H. M. ; FOX, Mark S. ; GRUNINGER, Michael: An Ontology for Quality Management - Enabling Quality Problem Identification and Tracing. In: *BT Technology Journal* 17 (1999), Nr. 4, S. 131–140
- [Kin03] KINDLER, Ekkart: On the Semantics of EPCs: A Framework for Resolving the Vicious Circle (extended abstract). In: NÜTTGENS, Markus (Hrsg.) ; RUMP, Frank J. (Hrsg.): *EPK 2003 - Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, Proceedings des GI-Workshops und Arbeitskreistreffens*. Bamberg, 2003, S. 7 – 18
- [KK97] KUENG, Peter ; KAWALEK, Peter: Goal-Based Business Process Mo-

- dels: Creation and Evaluation. In: *Business Process Management Journal* 3 (1997), Nr. 1, S. 17–38
- [KNS92] KELLER, G. ; NÜTTGENS, Markus ; SCHEER, August W.: Semantische Prozeßmodellierung auf der Grundlage "Ereignisgesteuerter Prozeßketten (EPK)". In: SCHEER, August W. (Hrsg.): *Veröffentlichungen des Instituts für Wirtschaftsinformatik*. Saarbrücken, 1992. – Heft 89
- [Lac08] LACHNER, Christina: *Entwurf und Implementierung eines Process Knowledge Warehouses für pModeler, einem System zur semantischen Prozessmodellanalyse*, Johannes Kepler Universität Linz, Diplomarbeit, 2008
- [LBS08] LAUTENBACHER, Florian ; BAUER, Bernhard ; SEITZ, Christian: Semantic Business Process Modeling - Benefits and Capability. In: HINKELMANN, Knut (Hrsg.): *AAAI 2008 Stanford Spring Symposium - AI Meets Business Rules and Process Management (AIBR)*. Stanford University, California, USA, March 26-28 2008, S. 71–76
- [LFB96] LIN, Jinxin ; FOX, Mark S. ; BILGIC, Taner: A Requirement Ontology for Engineering Design. In: *Concurrent Engineering: Research and Applications* 4 (1996), S. 279–291
- [LGJ<sup>+</sup>98] LEE, Jintae ; GRUNINGER, Michael ; JIN, Yan ; MALONE, Thomas ; TATE, Austin ; YOST, Gregg ; PIF WORKING GROUP other Members of t.: The PIF Process Interchange Format and Framework Version 1.2. In: *The Knowledge Engineering Review* 13 (1998), Nr. 1, S. 91–120
- [Lin08] LIN, Yun: *Semantic Annotation for Process Models: Facilitating Process Knowledge Management via Semantic Interoperability*, University of Trondheim, Diss., 2008
- [LM01] LASSILA, Ora ; MCGUINNESS, Deborah: The Role of Frame-Based Representation on the Semantic Web / Knowledge Systems Laboratory. Stanford, California, 2001 (KSL-01-02). – Technical Report
- [LYPWG94] LEE, Jintae ; YOST, Gregg ; PIF WORKING GROUP the: The PIF Process Interchange Format and Framework / MIT Center for Coordination Science. 1994 (Working Paper 180). – Forschungsbericht
- [MBB<sup>+</sup>01] MERTENS, P. ; BACK, A. ; BECKER, J. ; KÖNIG, W. ; KRALLMANN, H. ; RIEGER, B. ; SCHEER, A.-W. ; SEIBT, D. ; STAHLKNECHT, P. ; STRUNZ, H. ; THOME, R. ; WEDEKIND, H.: *Lexikon der Wirtschaftsinformatik*. Springer Verlag, 2001
- [MCL<sup>+</sup>99] MALONE, Thomas W. ; CROWSTON, Kevin ; LEE, Jintae ; PENTLAND, Brian ; DELLAROCAS, Chrysanthos ; WYNER, George ; QUIMBY, John ; OSBORN, Charles S. ; BERNSTEIN, Abraham ; HERMAN, George ;

- KLEIN, Mark ; O'DONNELL, Elissa: Tools for Inventing Organizations: Toward a Handbook of Organizational Processes. In: *Management Science* 45 (1999), March, Nr. 3, S. 425–443
- [Mül05] MÜLLER, Joachim: *Workflow-based Integration*. Springer Verlag, 2005
- [MVC<sup>+</sup>01] MENDES, Ricardo ; VASCONCELOS, Andre ; CAETANO, Artur ; NEVES, Joao ; SINOGAS, Pedro ; TRIBOLET, Jose ; TRIBOLET, Sinogas J.: Understanding Strategy: a Goal Modeling Methodology. In: WANG, Ying-xu (Hrsg.) ; PATEL, Shushma (Hrsg.) ; JOHNSTON, Ronald (Hrsg.): *7th International Conference on Object Oriented Information Systems (OOIS 2001)*, 2001, S. 437–446
- [Nip95] NIPPA, Michael: Anforderungen an das Management von prozeßorientierten Unternehmen. In: NIPPA, Michael (Hrsg.) ; PICOT, Arnold (Hrsg.): *Prozeßmanagement und Reengineering - Die Praxis im deutschsprachigen Raum*. Campus Verlag, 1995, S. 39 – 58
- [NPW05] NEUMANN, Stefan ; PROBST, Christian ; WERNSMANN, Clemens: Kontinuierliches Prozessmanagement. In: BECKER, Jörg (Hrsg.) ; KUGELER, Martin (Hrsg.) ; ROSEMANN, Michael (Hrsg.): *Prozessmanagement - Ein Leitfaden zur prozessorientierten Organisationsgestaltung*. Springer Verlag, 2005, S. 299–325
- [NR02] NÜTTGENS, Markus ; RUMP, Frank J.: Syntax und Semantik Ereignisgesteuerter Prozessketten (EPK). In: DESEL, Jörg (Hrsg.) ; WESKE, Mathias (Hrsg.) ; Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam (Veranst.): *Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen (Promise 2002)*. Bonn, 2002, S. 64 – 77
- [PD07] PEDRINACI, Carlos ; DOMINGUE, John: Towards an Ontology for Process Monitoring and Mining. In: HEPP, Martin (Hrsg.) ; HINKELMANN, Knut (Hrsg.) ; KARAGIANNIS, Dimitris (Hrsg.) ; KLEIN, Rüdiger (Hrsg.) ; STOJANOVIC, Nenad (Hrsg.): *Proceedings of the Workshop on Semantic Business Process and Product Lifecycle Management (SBPM-2007)* Bd. 251, 2007 (CEUR Workshop Proceedings)
- [Ros96] ROSEMANN, Michael: *Komplexitätsmanagement in Prozeßmodellen: methodenspezifische Gestaltungsempfehlungen für die Informationsmodellierung*. Wiesbaden : Gabler Verlag, 1996
- [Ros02] ROSENKRANZ, Friedrich: *Geschäftsprozesse - Modell- und computer-gestützte Planung*. Springer Verlag, 2002
- [RSD05] ROSEMANN, Michael ; SCHWEGMANN, Ansgar ; DELFMANN, Patrick: Vorbereitung der Prozessmodellierung. In: BECKER, Jörg (Hrsg.) ; KUGELER, Martin (Hrsg.) ; ROSEMANN, Michael (Hrsg.): *Prozessmanage-*

- ment - Ein Leitfaden zur prozessorientierten Organisationsgestaltung.* Springer Verlag, 2005, S. 45–103
- [Rum99] RUMP, F. J.: *Geschäftsprozeßmanagement auf der Basis ereignisgesteuerter Prozeßketten - Formalisierung, Analyse und Ausführung von EPKs.* Stuttgart[u.a.] : Teubner Verlag, 1999
- [SBF98] STUDER, Rudi ; BENJAMINS, V. R. ; FENSEL, Dieter: Knowledge Engineering: Principles and Methods. In: *Data Knowledge Engineering* 25 (1998), Nr. 1-2, S. 161–197
- [Sch97] SCHEER, A. W.: *Wirtschaftsinformatik - Referenzmodelle für industrielle Geschäftsprozesse.* Springer Verlag, 1997
- [Sch98a] SCHEER, A. W.: *ARIS - Vom Geschäftsprozeß zum Anwendungssystem.* Springer Verlag, 1998
- [Sch98b] SCHEER, August W.: *ARIS - Modellierungsmethoden, Metamodelle, Anwendungen.* Springer Verlag, 1998
- [Sch98c] SCHÜTTE, R.: *Grundsätze ordnungsmäßiger Referenzmodellierung. Konstruktion konfigurations- und anpassungsorientierter Modelle.* Wiesbaden : Gabler Verlag, 1998
- [Sei06] SEIDLMEIER, Heinrich: *Prozessmodellierung mit ARIS - Eine beispieleorientierte Einführung für Studium und Praxis.* Vieweg Verlag, 2006
- [SGCL99] SCHLENOFF, Craig ; GRUNINGER, Michael ; CIOCOIU, Mihai ; LEE, Jintae: The Essence of the Process Specification Language. In: *Transactions of the Society for Computer Simulation International* 16 (1999), Nr. 4, S. 204–216
- [SNZ95] SCHEER, August W. ; NÜTTGENS, Markus ; ZIMMERMANN, Volker: Rahmenkonzept für ein integriertes Geschäftsprozessmanagement. In: *Wirtschaftsinformatik* 37 (1995), S. 426 – 434
- [SS07] SCHMELZER, Hermann J. ; SESSELMANN, Wolfgang: *Geschäftsprozessmanagement in der Praxis.* Hanser Fachbuch, 2007
- [Sta06] STAUD, Josef: *Geschäftsprozessanalyse - ereignisgesteuerte Prozessketten und objektorientierte Geschäftsprozessmodellierung für betriebswirtschaftliche Standardsoftware.* Berlin[u.a.] : Springer Verlag, 2006
- [TF06] THOMAS, Oliver ; FELLMANN, Michael: Semantische Integration von Ontologien und Ereignisgesteuerten Prozessketten. In: NÜTTGENS, Markus (Hrsg.) ; RUMP, Frank J. (Hrsg.) ; MENDLING, Jan (Hrsg.): *EPK*, 2006, S. 7–24
- [TF07] THOMAS, Oliver ; FELLMANN, Michael: Semantic EPC: Enhanc-

- cing Process Modeling Using Ontology Languages. In: HEPP, Martin (Hrsg.) ; HINKELMANN, Knut (Hrsg.) ; KARAGIANNIS, Dimitris (Hrsg.) ; KLEIN, Rüdiger (Hrsg.) ; STOJANOVIC, Nenad (Hrsg.): *Semantic Business Process and Product Lifecycle Management. Proceedings of the Workshop SBPM 2007*, 2007, S. 64–75
- [TF09] THOMAS, Oliver ; FELLMANN, Michael: Semantische Prozessmodellierung - Konzeption und informationstechnische Unterstützung einer ontologiebasierten Repräsentation von Geschäftsprozessen. In: *Wirtschaftsinformatik* 51 (2009), Nr. 6, S. 506–517
- [TFG94] THAM, K. D. ; FOX, Mark S. ; GRUNINGER, Michael: A Cost Ontology for Enterprise Modelling. In: *Proceedings of the 3rd Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, IEEE Computer Society Press, 1994, S. 197–210
- [UG96] USCHOLD, Mike ; GRÜNINGER, Michael: Ontologies: Principles, Methods and Applications. In: *Knowledge Engineering Review* 11 (1996), Nr. 2, S. 93–155
- [UKMZ98] USCHOLD, Mike ; KING, Martin ; MORALEE, Stuart ; ZORGIOS, Yannis: The Enterprise Ontology. In: *Knowledge Engineering Review* 13 (1998), S. 31–89
- [Wes07] WESKE, Mathias: *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag, 2007
- [WMF<sup>+</sup>07] WETZSTEIN, Branimir ; MA, Zhilei ; FILIPOWSKA, Agata ; KACZMAREK, Monika ; BHIRI, Sami ; LOSADA, Silvestre ; LOPEZ-COBO, Jose-Manuel ; CICUREL, Laurent: Semantic Business Process Management: A Lifecycle Based Requirements Analysis. In: HEPP, Martin (Hrsg.) ; HINKELMANN, Knut (Hrsg.) ; KARAGIANNIS, Dimitris (Hrsg.) ; KLEIN, Rüdiger (Hrsg.) ; STOJANOVICS, Nenad (Hrsg.): *Workshop on Semantic Business Process and Product Lifecycle Management (SBPM 2007)*, Innsbruck: CEUR Workshop Proceedings, 2007, S. 1 – 11
- [Zie07] ZIERL, Peter: *Realisierung der Prozessmodellmetriken für pModeler: einem System zur semantischen Prozessmodellanalyse*, Johannes Kepler Universität Linz, Diplomarbeit, 2007

## A. Anhang

## A.1. Annotierte Process Objects



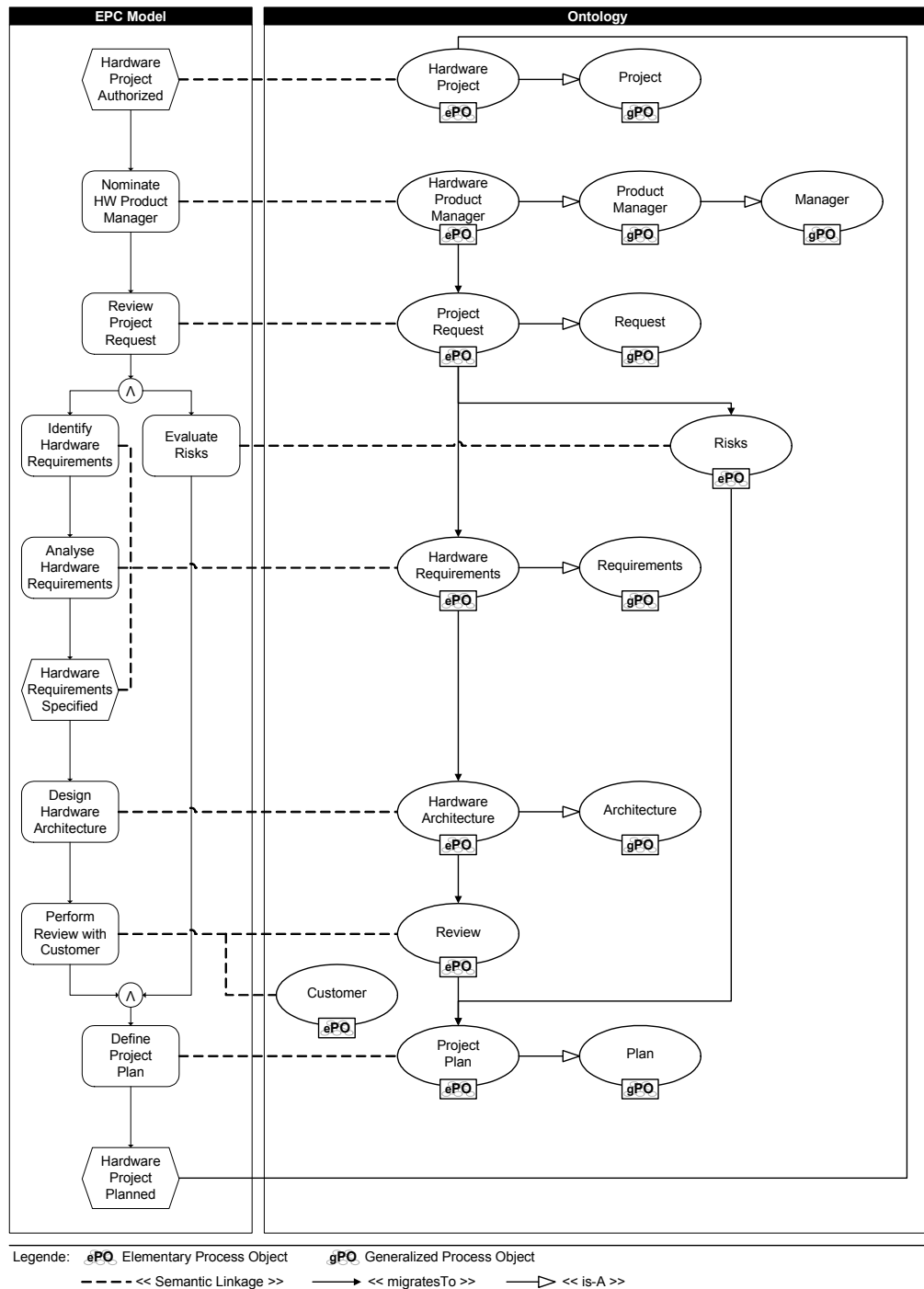


Abbildung A.1.: Beispielmodell Hardware - annotierte Process Objects

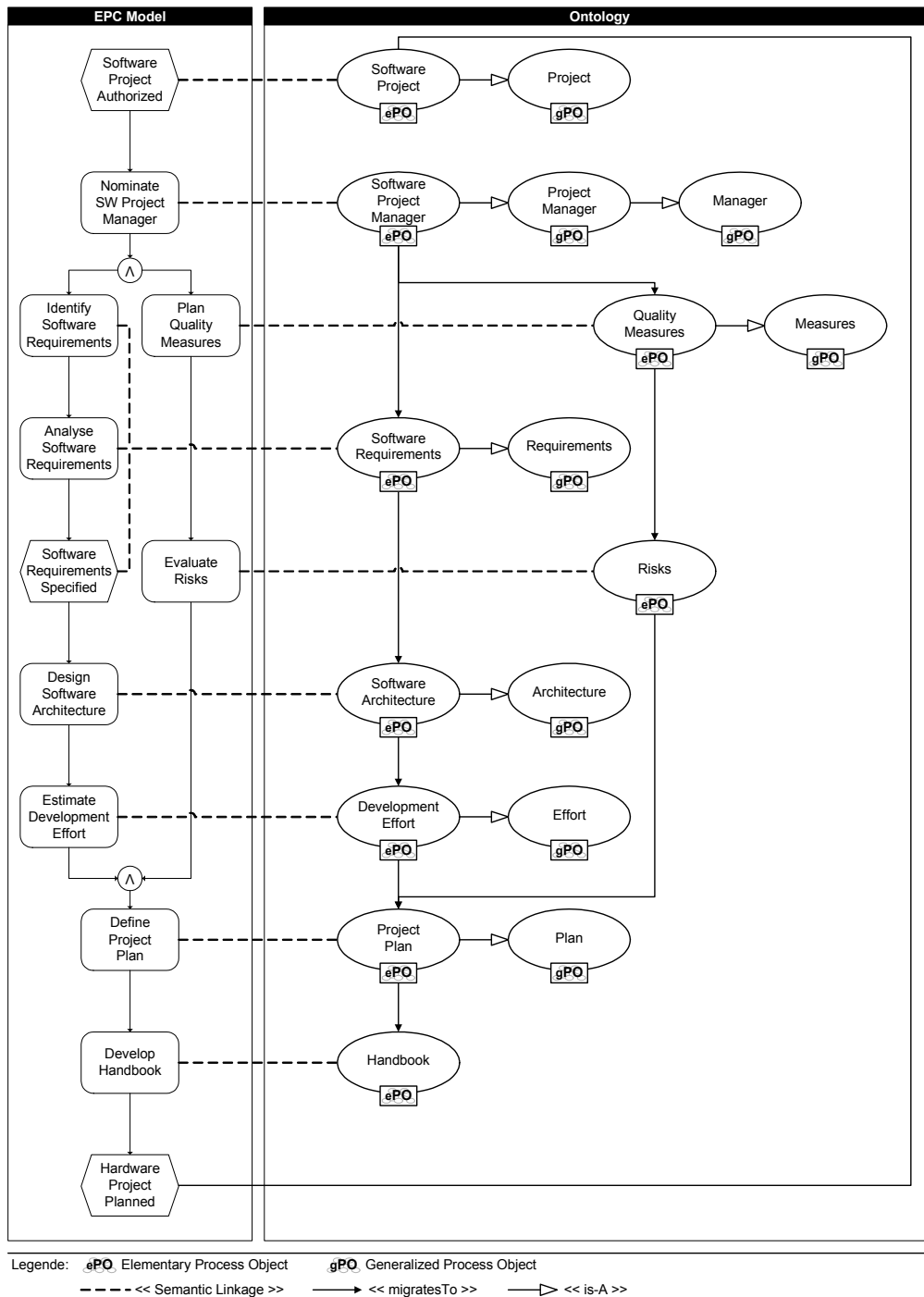


Abbildung A.2.: Beispielmmodell Software - annotierte Process Objects

## A.2. Annotierte Functions & Conditions

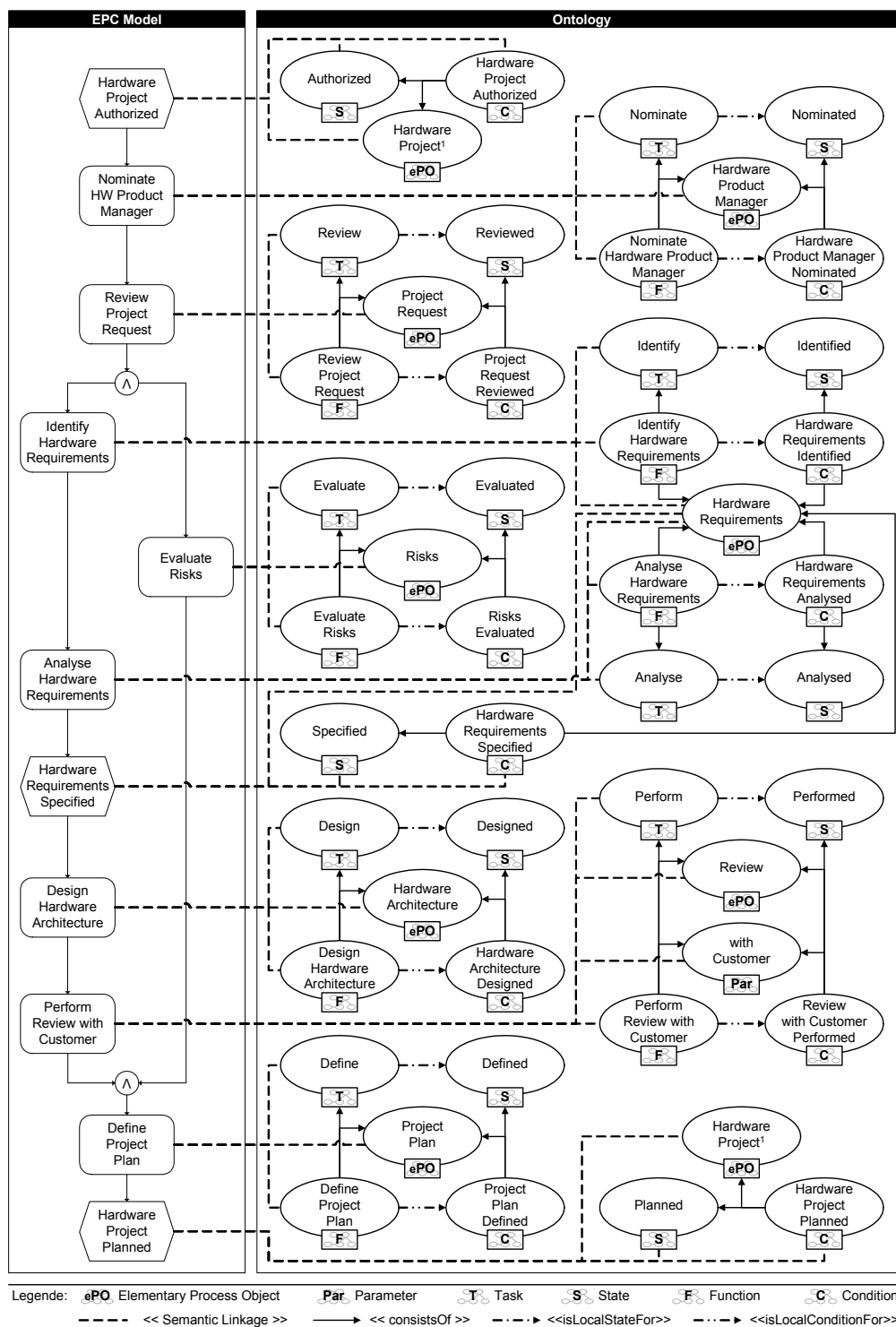


Abbildung A.3.: Beispielmmodell Hardware - annotierte Functions & Conditions

<sup>1</sup>Zur Vermeidung zusätzlicher Komplexität wird die in der Ontologie lediglich einmal vorhandene ePO-Instanz *Hardware Project* doppelt dargestellt.

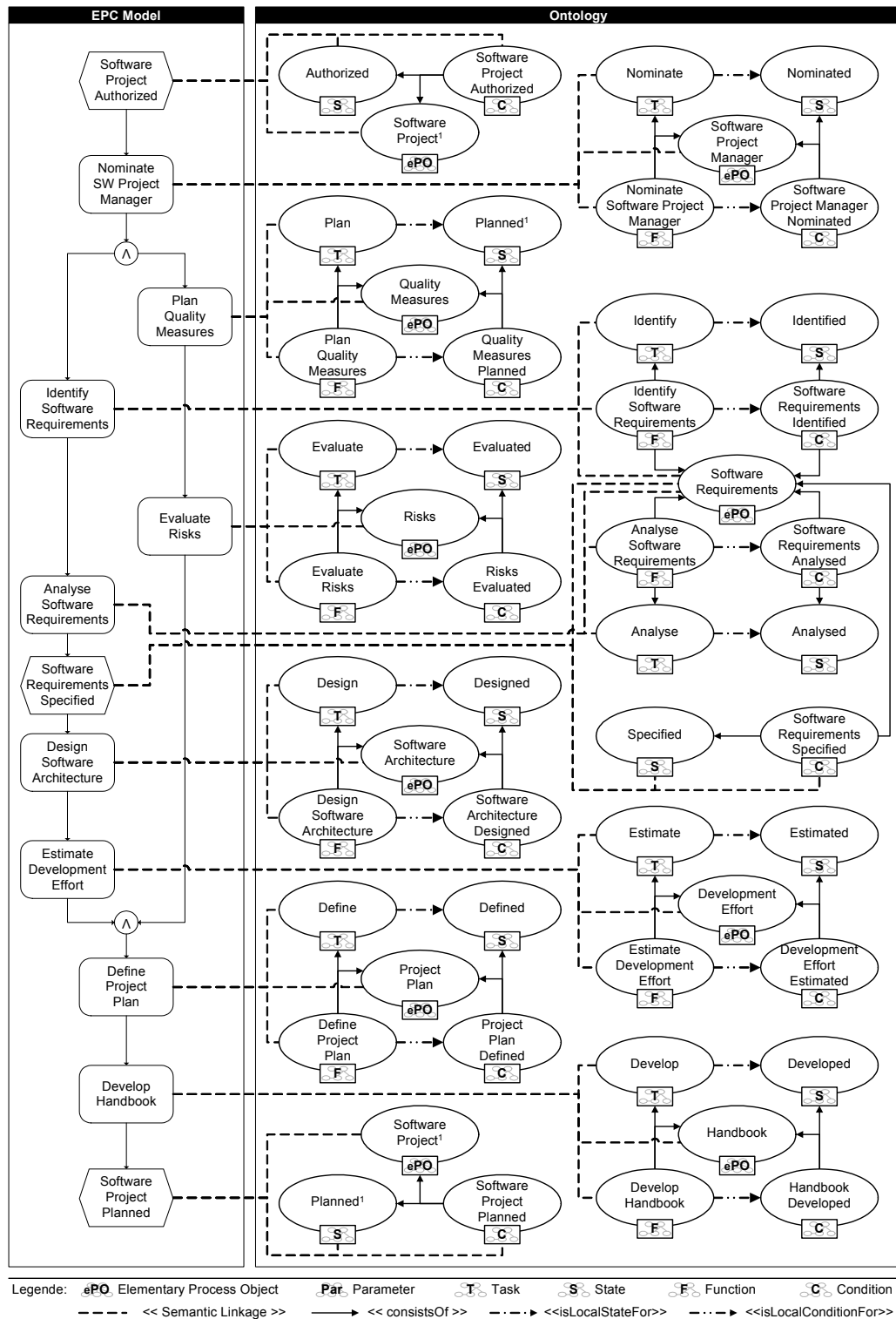


Abbildung A.4.: Beispielmodell Software - annotierte Functions & Conditions

<sup>1</sup>Zur Vermeidung zusätzlicher Komplexität werden die in der Ontologie jeweils lediglich einmal vorhandene ePO-Instanz *Hardware Project* und State-Instanz *Planned* doppelt dargestellt.

## A.3. Annotierte Activities

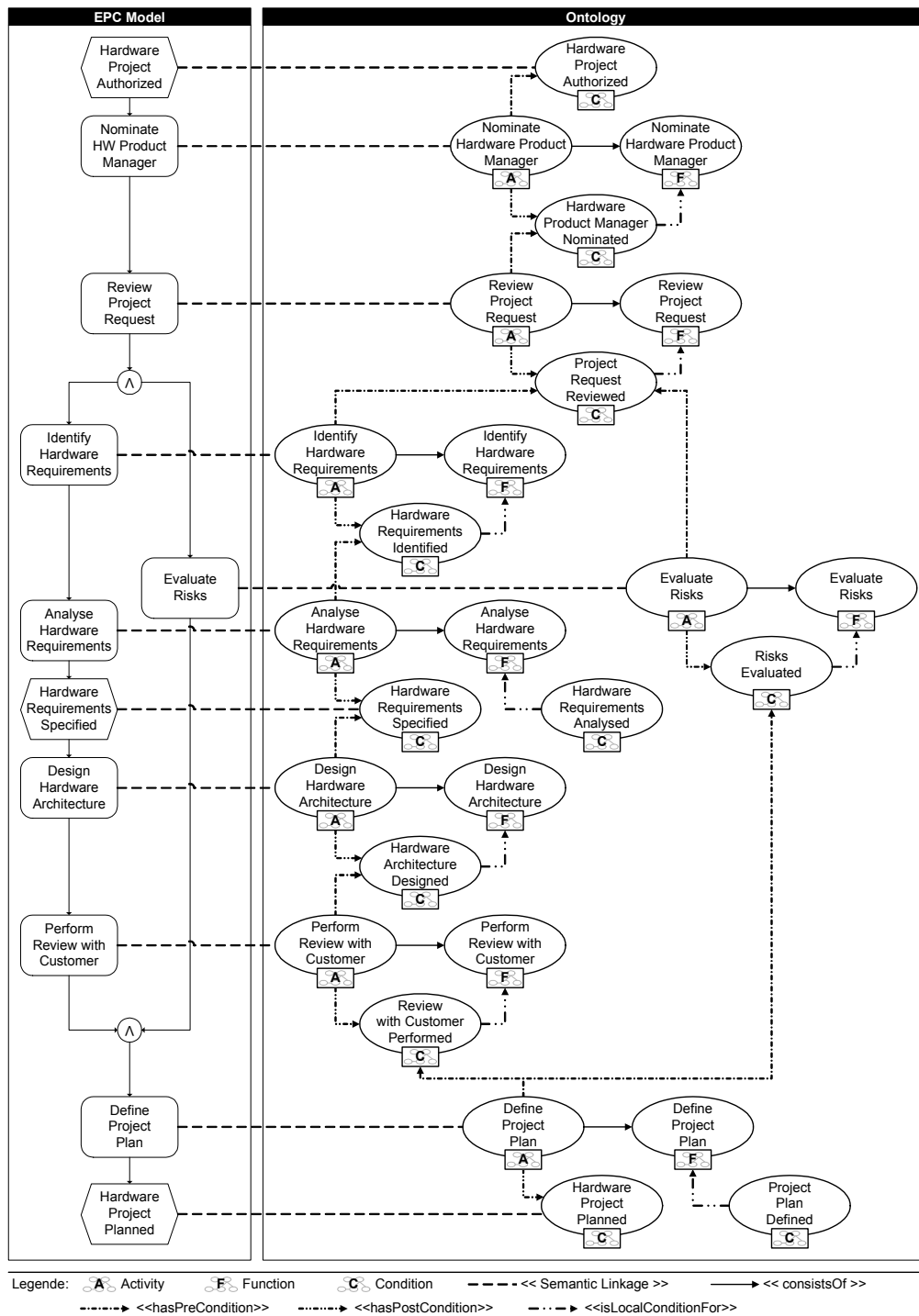


Abbildung A.5.: Beispielmodell Hardware - annotierte Activities

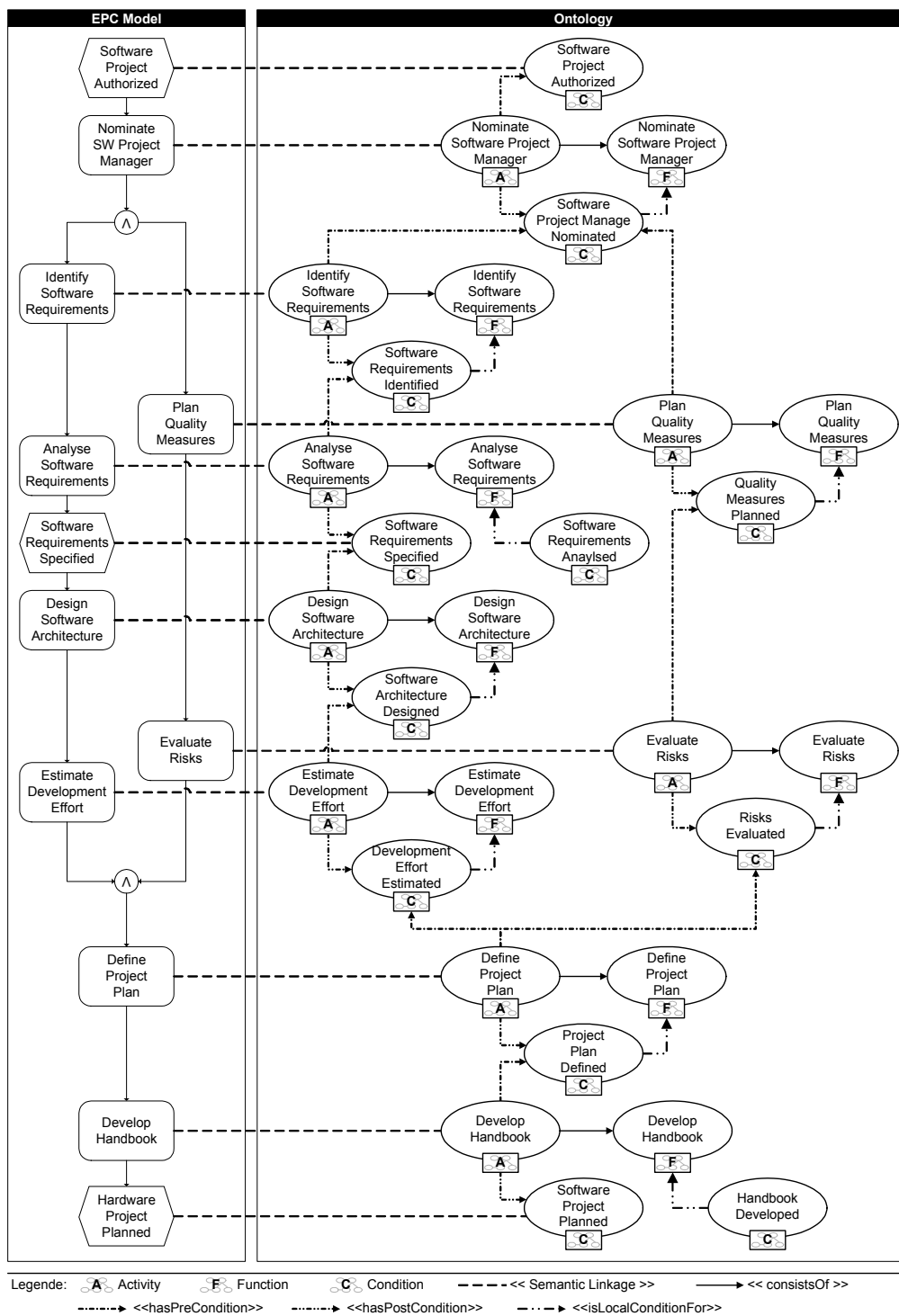


Abbildung A.6.: Beispielmodell Software - annotierte Activities



## B. Anhang

# B.1. Objektmodelle und Schnittstellenbeschreibungen

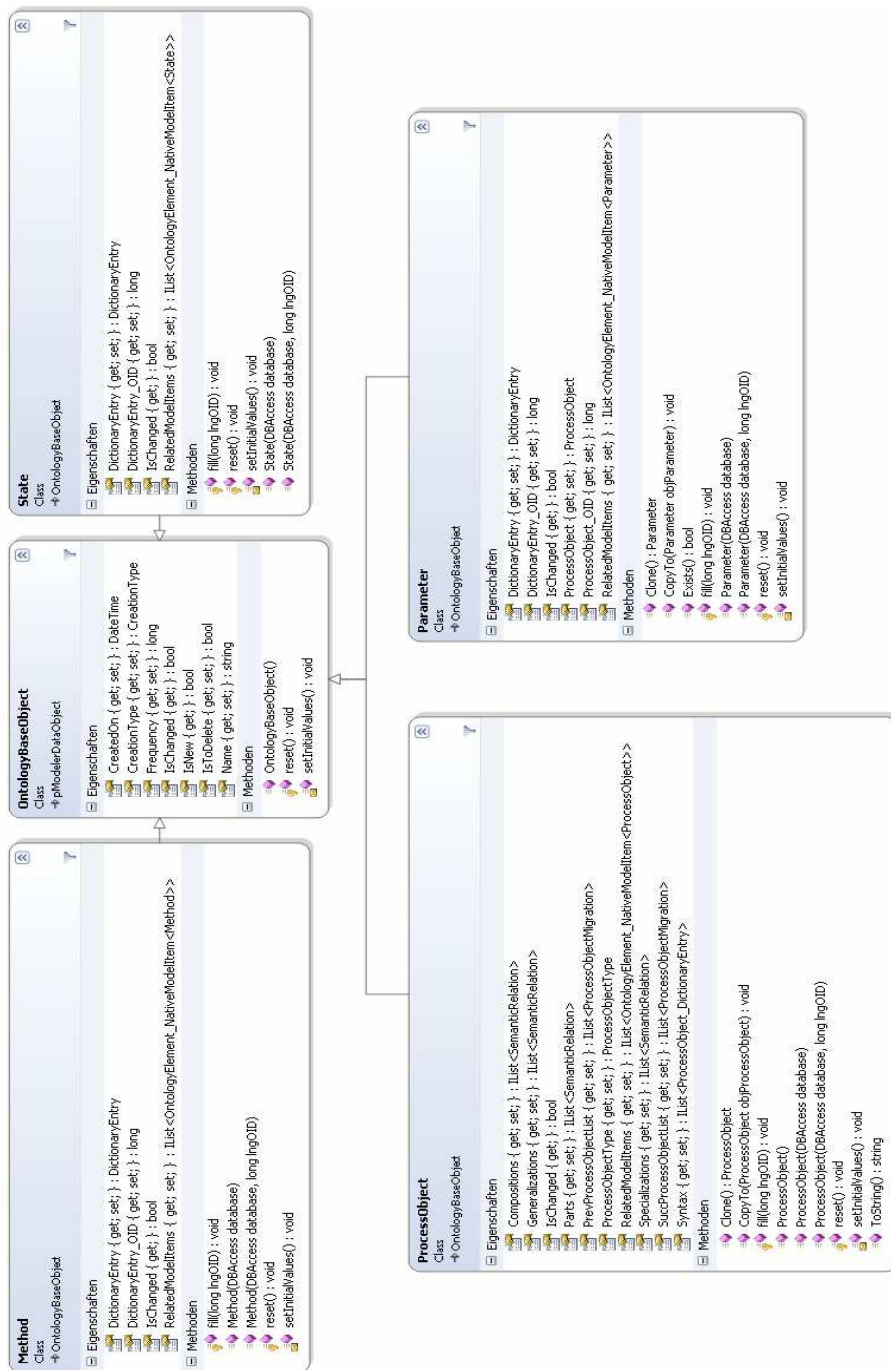


Abbildung B.1.: Objektmodell - Datenobjekte der Ontologie

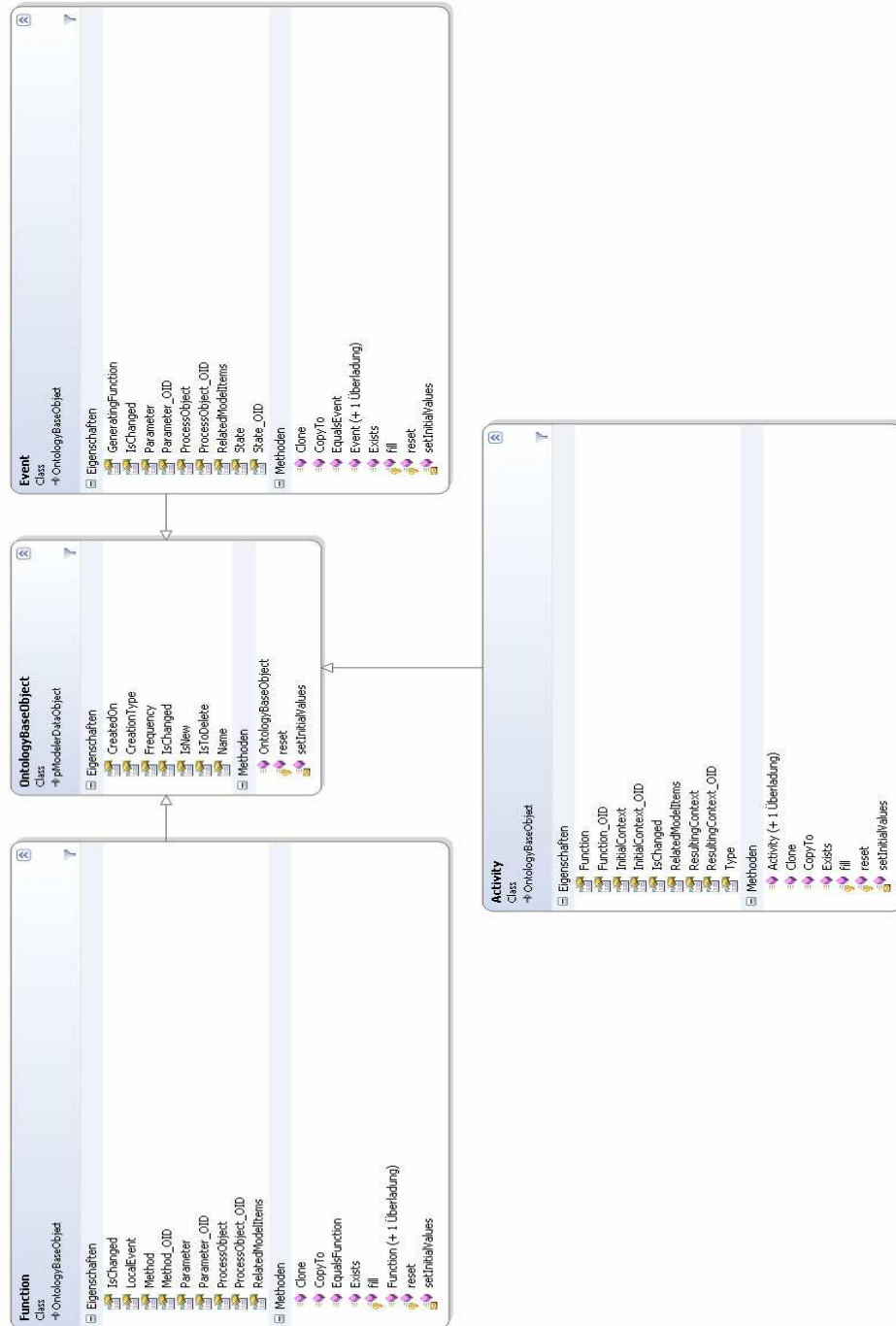


Abbildung B.2.: Objektmodell - Datenobjekte der Ontologie



Abbildung B.3.: Objektmodell - Managerklassen IElementManager &amp; ProcessObject

IElementManager	
<i>bool Delete(Element element)</i>	
	Löschen des gegebenen Elements mit Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Löschen erfolgreich war, anderenfalls <b>false</b> .
<i>bool DeleteUncommitted(OntologyElement element)</i>	
	Löschen des gegebenen Elements ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Löschen erfolgreich war, anderenfalls <b>false</b> .
<i>bool Exists(string name)</i>	
	Gibt <b>true</b> zurück, wenn das Ontologieelement mit dem gegebenen Namen existiert, ansonsten <b>false</b> .
<i>OntologyElement GetByName(string name)</i>	
	Gibt das Ontologieelement mit den gegebenen Name zurück, oder null.
<i>ICollection&lt;OntologyElement&gt; GetList()</i>	
	Gibt eine Liste mit allen Ontologieelementen des spezifizierten Typs zurück, oder null.
<i>ICollection&lt;OntologyElement&gt; GetListForNativeModel(long nativeModelOID)</i>	
	Gibt eine Liste mit allen Ontologieelementen des spezifizierten Typs für das Prozessmodell mit der gegebenen OID zurück, oder null.
<i>ICollection&lt;OntologyElement&gt; GetListForNativeModelItem(long nativeModelItemOID)</i>	
	Gibt eine Liste mit allen Ontologieelementen des spezifizierten Typs für das Modellelement mit der gegebenen OID zurück, oder null.
<i>bool Save(OntologyElement element)</i>	
	Speichert das gegebene Ontologieelement mit Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Speichern erfolgreich war, anderenfalls <b>false</b> .
<i>bool Save(ICollection&lt;OntologyElement&gt; elementList)</i>	
	Speichert die gegebene Liste mit Ontologieelementen mit Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Speichern erfolgreich war, anderenfalls <b>false</b> .

IElementManager (Fortsetzung)
<i>bool SaveUncommitted(OntologyElement element)</i>
Speichert das gegebene Ontologieelement ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Speichern erfolgreich war, anderenfalls <b>false</b> .
<i>bool Used(long elementOID)</i>
Gibt <b>true</b> zurück, wenn das Ontologieelement mit der gegebenen OID verwendet wird, anderenfalls <b>false</b> .
<i>bool UsedInNativeModel(long elementOID)</i>
Gibt <b>true</b> zurück, wenn das Ontologieelement mit der gegebenen OID einem Prozessmodell hinterlegt ist, anderenfalls <b>false</b> .

Tabelle B.1.: Schnittstelle - IElementManager

IProcessObjectManager
<p><i>SemanticRelation AddSemanticRelation(ProcessObject processObject, ProcessObject relatedProcessObject, ProcessObjectSemRelType semRelType)</i></p>
<p>Fügt eine <b>SemanticRelation</b> zwischen den gegebenen <b>ProcessObjects</b> mit dem gegebenen <b>ProcessObjectSemRelType</b> hinzu und gibt diese zurück.</p>
<p><i>ProcessObject Create(string name, IList&lt;DictionaryEntry&gt; processObjectSyntaxList)</i></p>
<p>Gibt das <b>ProcessObject</b> mit dem gegebenen Namen und der gegebenen Syntax zurück. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz.</p>
<p><i>IList&lt;ProcessObject&gt; GetListForType(ProcessObjectType processObjectType)</i></p>
<p>Gibt eine Liste mit <b>ProcessObjects</b> mit dem gegebenen <b>ProcessObjectType</b> zurück, oder null.</p>
<p><i>IList&lt;ProcessObject&gt; GetListOfRootsBySemanticRelationType(SemanticRelationType semanticRelationType)</i></p>
<p>Gibt eine Liste mit <b>ProcessObjects</b> zurück, welche in einer Hierarchie mit dem gegebenen <b>SemanticRelationType</b> das Wurzelement bilden, oder null.</p>
<p><i>IList&lt;ProcessObject&gt; GetListWithoutCompositions(long processObjectOID)</i></p>
<p>Gibt eine Liste mit <b>ProcessObjects</b> zurück, wobei das <b>ProcessObject</b> mit der gegebenen <b>OID</b> und dessen Kompositionen nicht enthalten sind, oder null.</p>
<p><i>IList&lt;ProcessObject&gt; GetListWithoutGeneralizations(long processObjectOID)</i></p>
<p>Gibt eine Liste mit <b>ProcessObjects</b> zurück, wobei das <b>ProcessObject</b> mit der gegebenen <b>OID</b> und dessen Generalisierungen nicht enthalten sind, oder null.</p>

IProcessObjectManager (Fortsetzung)
<i>IList&lt;ProcessObject&gt; GetListWithoutMigratesTo(long processObjectOID)</i>
Gibt eine Liste mit <b>ProcessObjects</b> zurück, wobei das <b>ProcessObject</b> mit der gegebenen OID und dessen Migrationsziele nicht enthalten sind, oder null.
<i>IList&lt;ProcessObject&gt; GetListWithoutParts(long processObjectOID)</i>
Gibt eine Liste mit <b>ProcessObjects</b> zurück, wobei das <b>ProcessObject</b> mit der gegebenen OID und dessen Teile nicht enthalten sind, oder null.
<i>IList&lt;ProcessObject&gt; GetListWithoutSpecializations(long processObjectOID)</i>
Gibt eine Liste mit <b>ProcessObjects</b> zurück, wobei das <b>ProcessObject</b> mit der gegebenen OID und dessen Spezialisierungen nicht enthalten sind, oder null.
<i>void RemoveSemanticRelation(ProcessObject processObject, SemanticRelation removeSemanticRelation, ProcessObjectSemRelType semRelType)</i>
Entfernt die gegebene <b>SemanticRelation</b> vom gegebenen <b>ProcessObject</b> . Der <b>ProcessObjectSemRelType</b> spezifiziert, welche Beziehung entfernt werden soll.
<i>bool UsedInEvent(long processObjectOID)</i>
Gibt <b>true</b> zurück, wenn das <b>ProcessObject</b> mit der gegebenen OID in einem <b>Event</b> verwendet wird, ansonsten <b>false</b> .
<i>bool UsedInFunction(long processObjectOID)</i>
Gibt <b>true</b> zurück, wenn das <b>ProcessObject</b> mit der gegebenen OID in einer <b>Function</b> verwendet wird, ansonsten <b>false</b> .
<i>bool UsedInParameter(long processObjectOID)</i>
Gibt <b>true</b> zurück, wenn das <b>ProcessObject</b> mit der gegebenen OID in einem <b>Parameter</b> verwendet wird, ansonsten <b>false</b> .

Tabelle B.2.: Schnittstelle - IProcessObjectManager



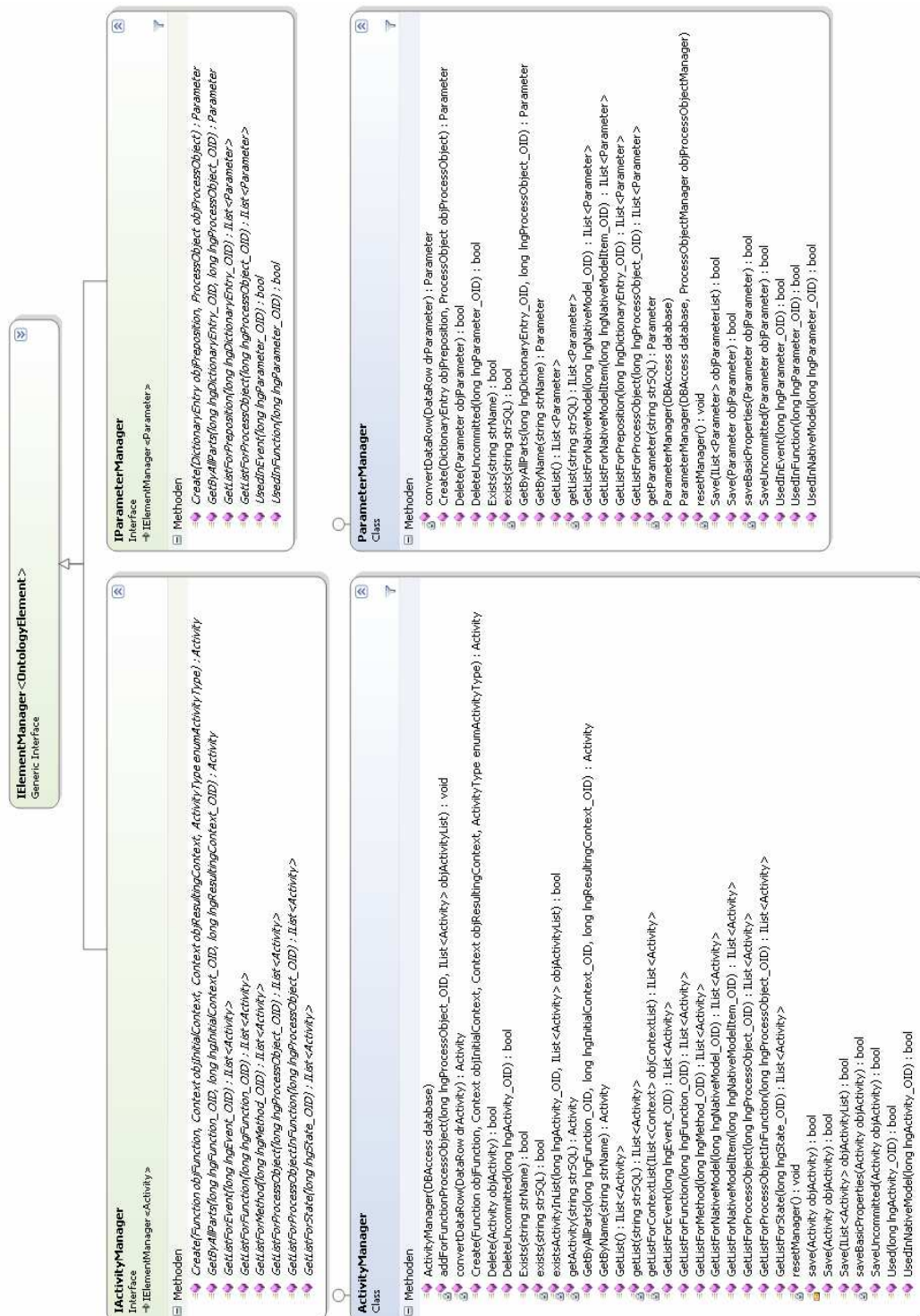


Abbildung B.4.: Objektmodell - Managerklassen Activity & Parameter

IActivityManager
<i>Activity Create(Function function, Context initialContext, Context resultingContext)</i>
Gibt die <b>Activity</b> mit den gegebenen Elementen zurück. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz.
<i>Activity GetByAllParts(long functionOID, long initialContextOID, long resultingContextOID)</i>
Gibt die <b>Activity</b> mit den gegebenen Elementen zurück, oder null.
<i>IList&lt;Activity&gt; GetListForEvent(long eventOID)</i>
Gibt eine Liste mit <b>Activities</b> zurück, welche entweder im <b>Initial-</b> oder im <b>ResultingContext</b> den <b>Event</b> mit der gegebenen <b>OID</b> enthalten, oder null.
<i>IList&lt;Activity&gt; GetListForFunction(long functionOID)</i>
Gibt eine Liste mit <b>Activities</b> zurück, welche die <b>Function</b> mit der gegebenen <b>OID</b> enthalten, oder null.
<i>IList&lt;Activity&gt; GetListForMethod(long methodOID)</i>
Gibt eine Liste mit <b>Activities</b> zurück, welche eine <b>Function</b> mit der <b>Method</b> mit der gegebenen <b>OID</b> enthalten, oder null.
<i>IList&lt;Activity&gt; GetListForProcessObject(long processObjectOID)</i>
Gibt eine Liste mit <b>Activities</b> zurück, welche eine <b>Function</b> bzw. einen <b>Event</b> mit einem <b>ProcessObject</b> mit der gegebenen <b>OID</b> enthalten, oder null.
<i>IList&lt;Activity&gt; GetListForState(long stateOID)</i>
Gibt eine Liste mit <b>Activities</b> zurück, welche entweder im <b>Initial-</b> oder im <b>ResultingContext</b> einen <b>Event</b> mit dem <b>State</b> mit der gegebenen <b>OID</b> enthalten, oder null.

Tabelle B.3.: Schnittstelle - IActivityManager

IParameterManager
<i>Parameter Create(DictionaryEntry preposition, ProcessObject processObject)</i>
Gibt den <b>Parameter</b> mit dem gegebenen <b>DictionaryEntry</b> (Preposition) und dem gegebenen <b>ProcessObject</b> zurück. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz.
<i>Parameter GetByAllParts(long dictionaryEntryOID, long processObjectOID)</i>
Gibt den <b>Parameter</b> mit dem gegebenen <b>DictionaryEntry</b> (Preposition) und dem gegebenen <b>ProcessObject</b> zurück. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz.
<i>IList&lt;Parameter&gt; GetListForPreposition(long dictionaryEntryOID)</i>
Gibt eine Liste mit <b>Parametern</b> zurück, welche den <b>DictionaryEntry</b> (die Preposition) mit der gegebenen <b>OID</b> enthalten, oder null.
<i>IList&lt;Parameter&gt; GetListForProcessObject(long processObjectOID)</i>
Gibt eine Liste mit <b>Parametern</b> zurück, welche das <b>ProcessObject</b> mit der gegebenen <b>OID</b> enthalten, oder null.
<i>bool UsedInEvent(long parameterOID)</i>
Gibt <b>true</b> zurück, wenn der <b>Parameter</b> mit der gegebenen <b>OID</b> in einem <b>Event</b> verwendet wird, ansonsten <b>false</b> .
<i>bool UsedInFunction(long parameterOID)</i>
Gibt <b>true</b> zurück, wenn der <b>Parameter</b> mit der gegebenen <b>OID</b> in einer <b>Function</b> verwendet wird, ansonsten <b>false</b> .

Tabelle B.4.: Schnittstelle - IParameterManager

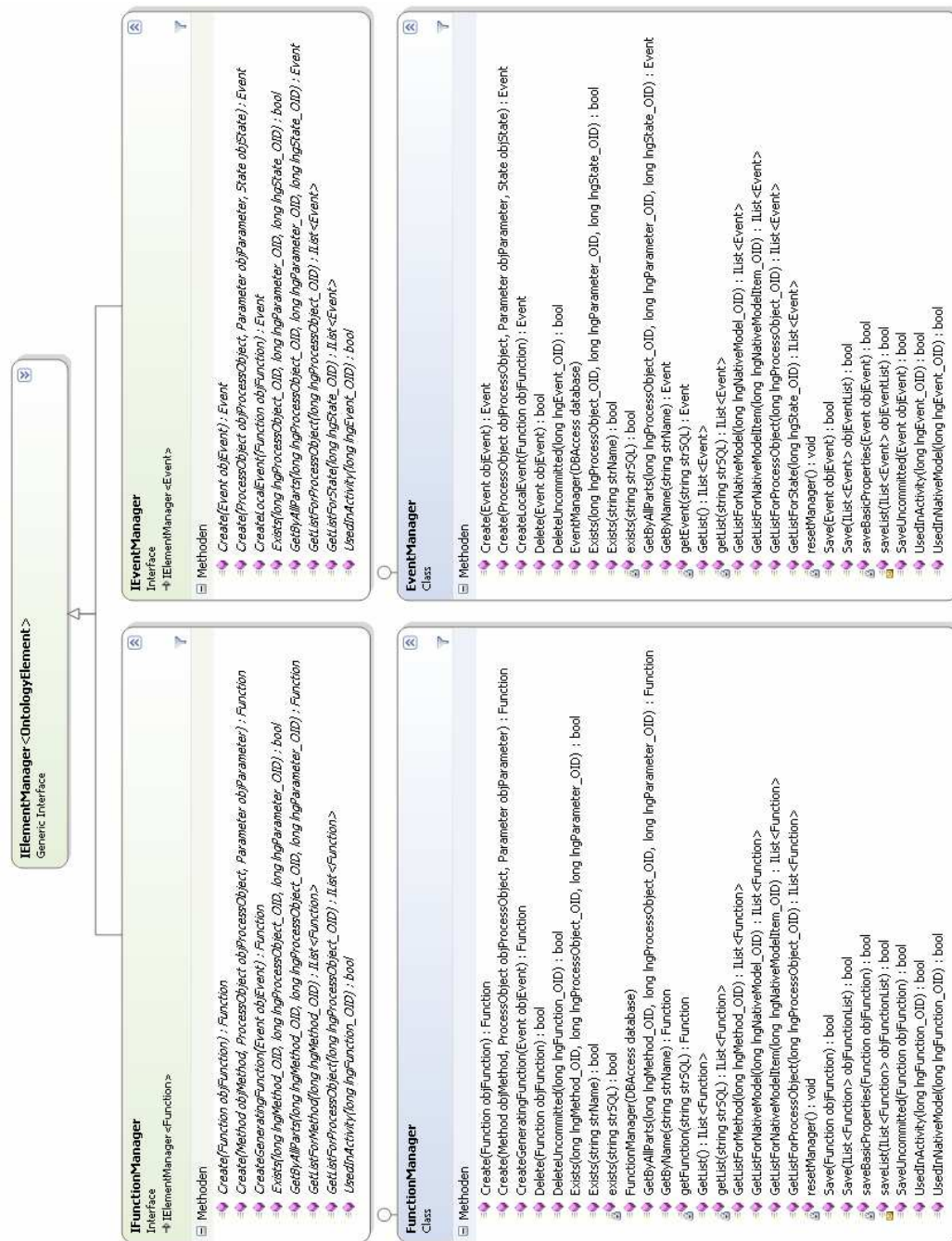


Abbildung B.5.: Objektmodell - Managerklassen Function & Event

IFunctionManager
<i>Function Create(Function function)</i>
Gibt eine <b>Function</b> für die gegebenen <b>Function</b> zurück. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz.
<i>Function Create(Method method, ProcessObject processObject, Parameter parameter)</i>
Gibt eine <b>Function</b> für die gegebenen Elemente ( <b>Method</b> , <b>ProcessObject</b> und <b>Parameter</b> (optional)) zurück, oder null. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz.
<i>Function CreateGeneratingFunction(Event event)</i>
Gibt die "lokale" <b>Function</b> für den gegebenen <b>Event</b> zurück, oder null. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz.
<i>bool Exists(long methodOID, long processObjectOID, long parameterOID)</i>
Gibt <b>true</b> zurück, wenn eine <b>Function</b> mit den gegebenen Elementen bereits existiert, anderenfalls <b>false</b> .
<i>Function GetByAllParts(long methodOID, long processObjectOID, long parameterOID)</i>
Gibt die <b>Function</b> mit den gegebenen Elementen zurück, oder null.
<i>IList&lt;Function&gt; GetListForMethod(long methodOID)</i>
Gibt eine Liste mit <b>Functions</b> zurück, welche die <b>Method</b> mit der gegebenen <b>OID</b> enthalten, oder null.
<i>IList&lt;Function&gt; GetListForProcessObject(long processObjectOID)</i>
Gibt eine Liste mit <b>Functions</b> zurück, welche das <b>ProcessObject</b> mit der gegebenen <b>OID</b> enthalten, oder null. ( <b>ProcessObjects</b> in <b>Parameter</b> werden ignoriert)
<i>bool UsedInActivity(long functionOID)</i>
Gibt <b>true</b> zurück, wenn die <b>Function</b> mit der gegebenen <b>OID</b> in einer <b>Activity</b> verwendet wird, ansonsten <b>false</b> .

Tabelle B.5.: Schnittstelle - IFunctionManager

IEventManager
<i>Event Create(Event event)</i>
Gibt einen <b>Event</b> für den gegebenen <b>Event</b> zurück. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz.
<i>Event Create(ProcessObject processObject, Parameter parameter, State state)</i>
Gibt einen <b>Event</b> für die gegebenen Elemente ( <b>ProcessObject</b> , <b>Parameter</b> (optional) und <b>State</b> ) zurück, oder null. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz.
<i>Event CreateLocalEvent(Function function)</i>
Gibt den "lokale" <b>Event</b> für die gegebene <b>Function</b> zurück, oder null. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz.
<i>bool Exists(long processObjectOID, long parameterOID, long stateOID)</i>
Gibt <b>true</b> zurück, wenn ein <b>Event</b> mit den gegebenen Elementen bereits existiert, anderenfalls <b>false</b> .
<i>Event GetByAllParts(long processObjectOID, long parameterOID, long stateOID)</i>
Gibt den <b>Event</b> mit den gegebenen Elementen zurück, oder null.
<i>ICollection&lt;Event&gt; GetListForProcessObject(long processObjectOID)</i>
Gibt eine Liste mit <b>Events</b> zurück, welche das <b>ProcessObject</b> mit der gegebenen <b>OID</b> enthalten, oder null. ( <b>ProcessObjects</b> in <b>Parameter</b> werden ignoriert)
<i>ICollection&lt;Event&gt; GetListForState(long stateOID)</i>
Gibt eine Liste mit <b>Events</b> zurück, welche den <b>State</b> mit der gegebenen <b>OID</b> enthalten, oder null.
<i>bool UsedInActivity(long eventOID)</i>
Gibt <b>true</b> zurück, wenn der <b>Event</b> mit der gegebenen <b>OID</b> in einer <b>Activity</b> verwendet wird, ansonsten <b>false</b> .

Tabelle B.6.: Schnittstelle - IEventManager

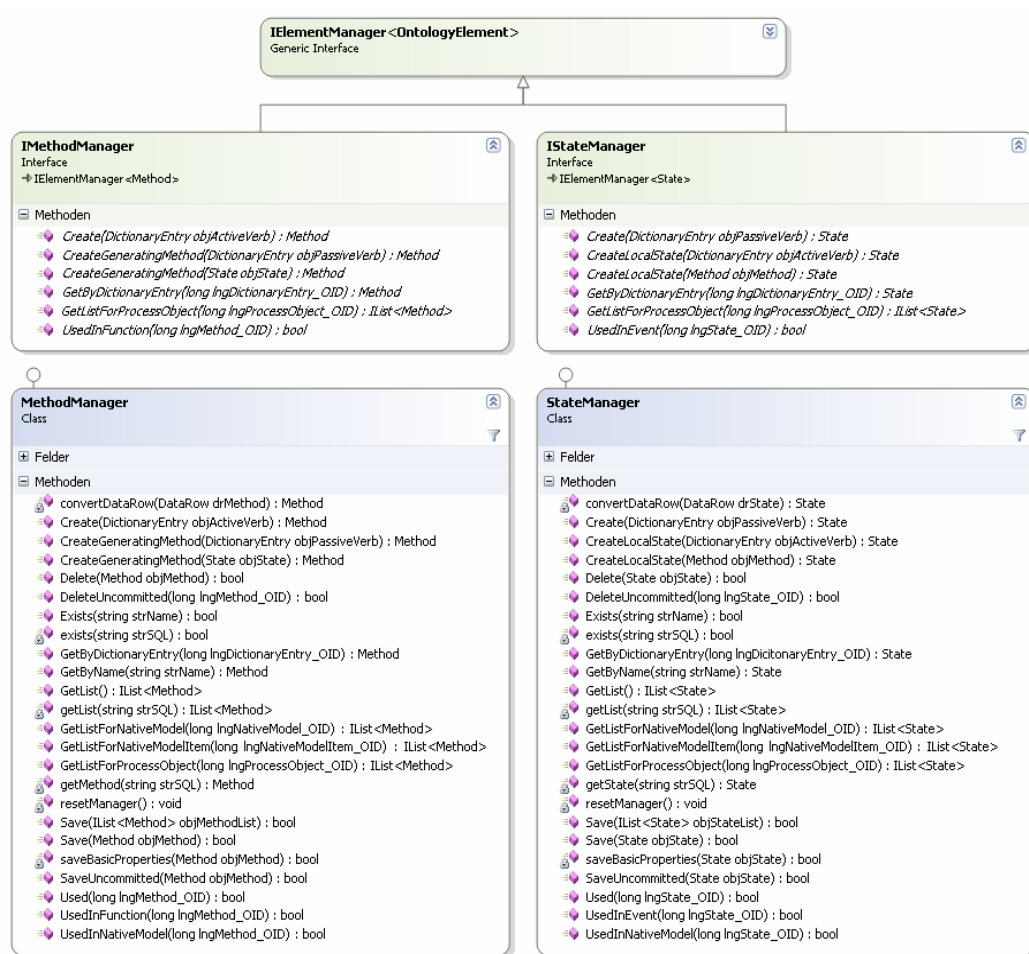


Abbildung B.6.: Objektmodell - Managerklassen Method & State

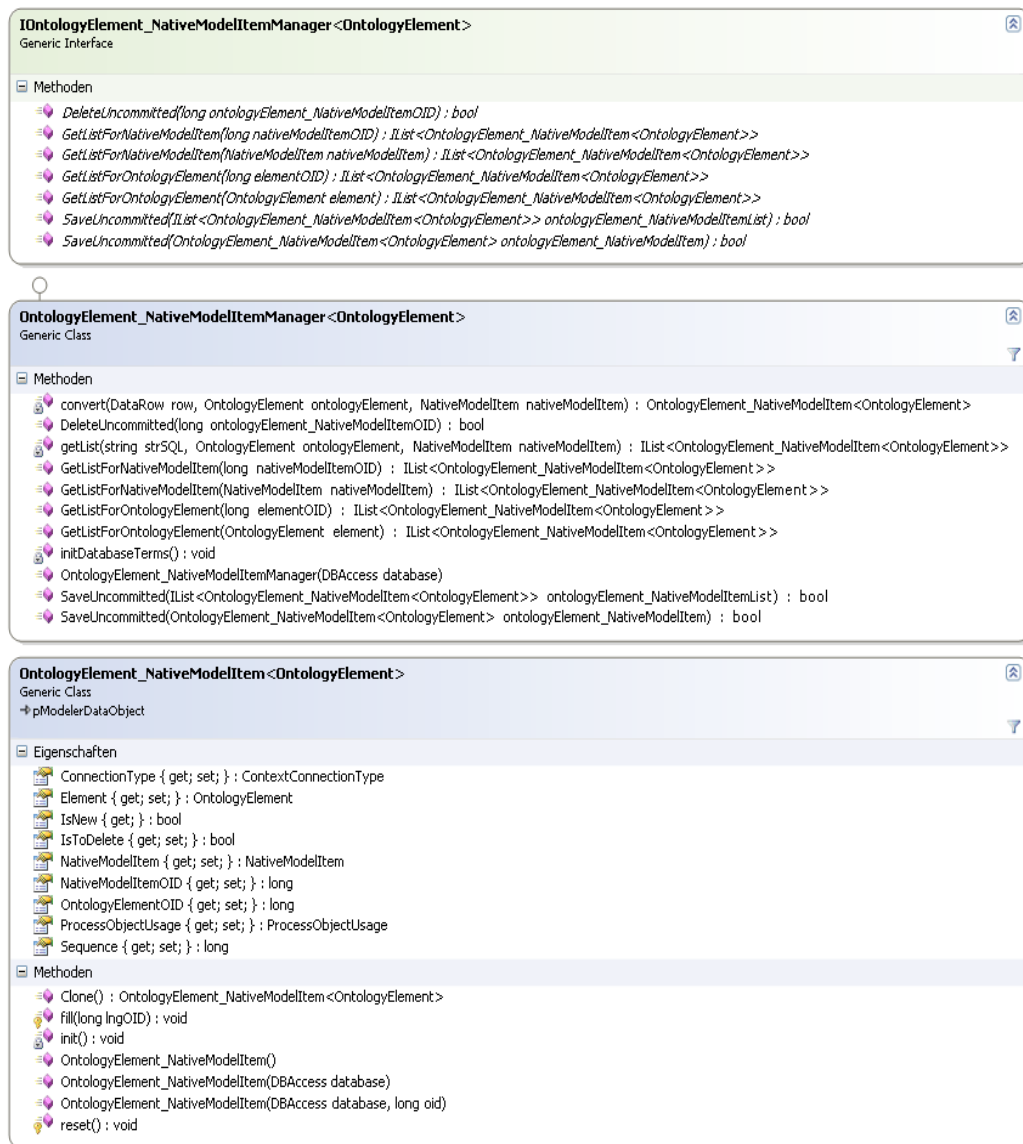
IMethodManager
<i>Method Create(DictionaryEntry activeVerb)</i>
Gibt eine <b>Method</b> für den gegebenen <b>DictionaryEntry</b> (aktives Verb) zurück, oder null. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz.
<i>Method CreateGeneratingMethod(DictionaryEntry passiveVerb)</i>
Gibt die "lokale" <b>Method</b> für den gegebenen <b>DictionaryEntry</b> (passives Verb) zurück (z.B. für "nominated" die <b>Method</b> "nominate"), oder null. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz.
<i>Method CreateGeneratingMethod(State state)</i>
Gibt die "lokale" <b>Method</b> für den gegebenen <b>State</b> zurück (z.B. für "nominated" die <b>Method</b> "nominate"), oder null. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz.
<i>Method GetByDictionaryEntry(long dictionaryEntryOID)</i>
Gibt die <b>Method</b> für den <b>DictionaryEntry</b> mit der gegebenen <b>OID</b> zurück, oder null.
<i>IList&lt;Method&gt; GetListForProcessObject(long processObjectOID)</i>
Gibt eine Liste mit <b>Methods</b> , welche zusammen mit dem <b>ProcessObject</b> mit der gegebenen <b>OID</b> eine <b>Function</b> bilden, zurück, oder null.
<i>bool UsedInFunction(long methodOID)</i>
Gibt <b>true</b> zurück, wenn die <b>Method</b> mit der gegebenen <b>OID</b> in einer <b>Function</b> verwendet wird, ansonsten <b>false</b> .

Tabelle B.7.: Schnittstelle - IMethodManager



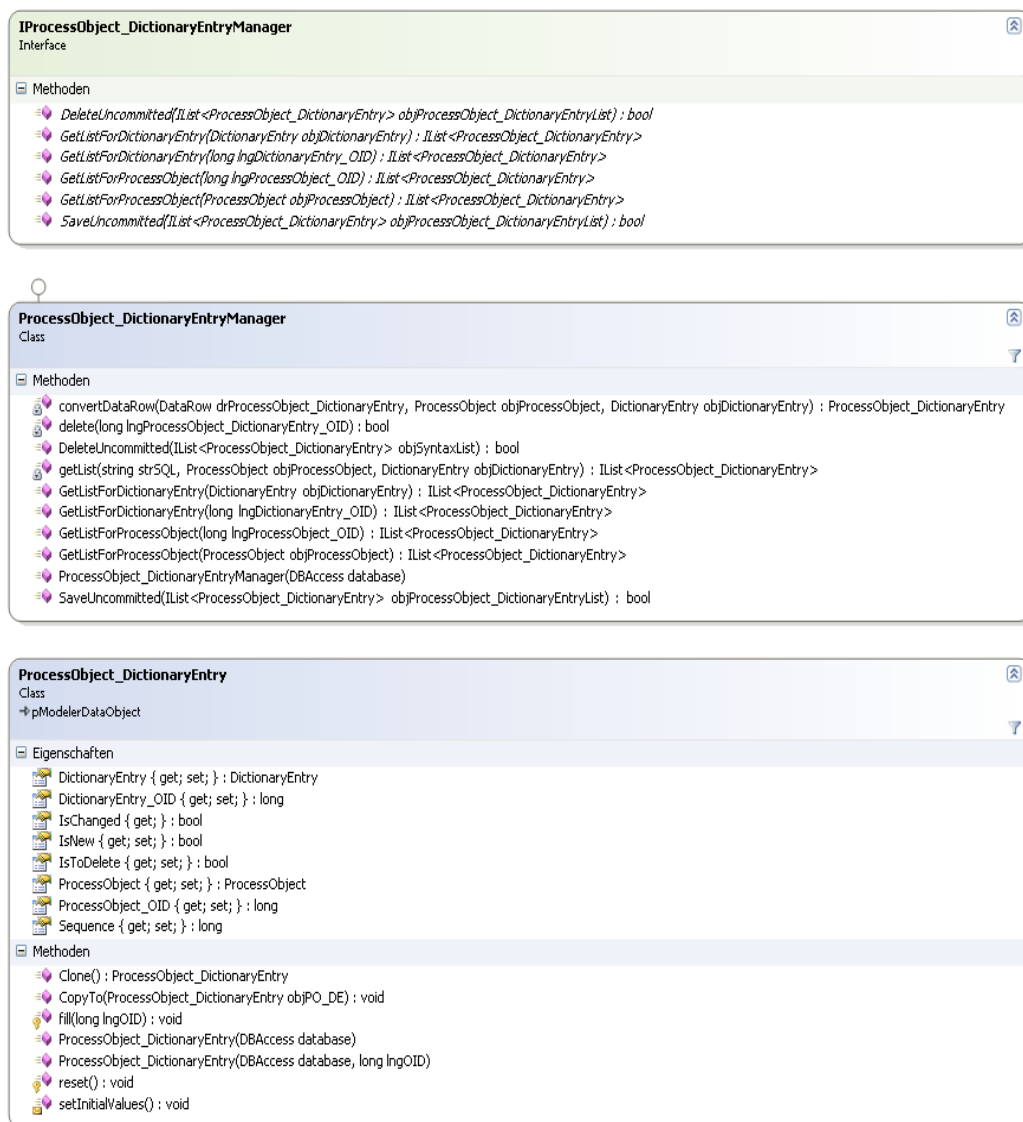
IStateManager
<i>State Create(DictionaryEntry passiveVerb)</i>
Gibt einen <b>State</b> für den gegebenen <b>DictionaryEntry</b> (passives Verb) zurück, oder null. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz.
<i>State CreateLocalState(DictionaryEntry activeVerb)</i>
Gibt den "lokale" <b>State</b> für den gegebenen <b>DictionaryEntry</b> (aktives Verb) zurück (z.B. für "nominate" den <b>State</b> "nominated"), oder null. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz.
<i>State CreateLocalState(Method method)</i>
Gibt den "lokale" <b>State</b> für die gegebenen <b>Method</b> zurück (z.B. für "nominate" den <b>State</b> "nominated"), oder null. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz.
<i>State GetByDictionaryEntry(long dictionaryEntryOID)</i>
Gibt den <b>State</b> für den <b>DictionaryEntry</b> mit der gegebenen <b>OID</b> zurück, oder null.
<i>IList&lt;State&gt; GetListForProcessObject(long processObjectOID)</i>
Gibt eine Liste mit <b>States</b> , welche zusammen mit dem <b>ProcessObject</b> mit der gegebenen <b>OID</b> einen <b>Event</b> bilden, zurück, oder null.
<i>bool UsedInEvent(long stateOID)</i>
Gibt <b>true</b> zurück, wenn der <b>State</b> mit der gegebenen <b>OID</b> in einem <b>Event</b> verwendet wird, ansonsten <b>false</b> .

Tabelle B.8.: Schnittstelle - IStateManager

Abbildung B.7.: Objektmodell - Klassen `OntologyElement_NativeModelItem`

IOntologyElement_NativeModelItemManager
<i>bool DeleteUncommitted(long ontologyElement_NativeModelItemOID)</i>
Löscht die Referenz mit der gegebenen OID ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Löschen erfolgreich war, anderenfalls <b>false</b> .
<i>IList&lt;OntologyElement_NativeModelItem&lt;OntologyElement&gt;&gt; GetListForNativeModelItem(long nativeModelItemOID)</i>
Gibt eine Liste mit Referenzen für das Modellelement mit der gegebenen OID zurück.
<i>IList&lt;OntologyElement_NativeModelItem&lt;OntologyElement&gt;&gt; GetListForNativeModelItem(NativeModelItem nativeModelItem)</i>
Gibt eine Liste mit Referenzen für das gegebene Modellelement zurück und setzt die <i>NativeModelItem</i> -Referenz.
<i>IList&lt;OntologyElement_NativeModelItem&lt;OntologyElement&gt;&gt; GetListForProcessObject(long elementOID)</i>
Gibt eine Liste mit Referenzen für das Ontologieelement mit der gegebenen OID zurück.
<i>IList&lt;OntologyElement_NativeModelItem&lt;OntologyElement&gt;&gt; GetListForProcessObject(OntologyElement element)</i>
Gibt eine Liste mit Referenzen für das gegebene Ontologieelement zurück und setzt die <i>Element</i> -Referenz.
<i>bool SaveUncommitted( IList&lt;OntologyElement_NativeModelItem&lt;OntologyElement&gt;&gt; ontologyElement_NativeModelItemList)</i>
Speichert die List mit Referenzen ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Speichern erfolgreich war, anderenfalls <b>false</b> .
<i>bool SaveUncommitted(OntologyElement_NativeModelItem&lt;Ontology- Element&gt; ontologyElement_NativeModelItem)</i>
Speichert die Referenz ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Speichern erfolgreich war, anderenfalls <b>false</b> .

Tabelle B.9.: Schnittstelle - IOntologyElement\_NativeModelItemManager

Abbildung B.8.: Objektmodell - Klassen `ProcessObject_DictionaryEntry`

IProcessObject_DictionaryEntryManager
<i>bool DeleteUncommitted(IList&lt;ProcessObject_DictionaryEntry&gt; processObject_DictionaryEntryList)</i>
Löscht die Liste mit Referenzen ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Löschen erfolgreich war, anderenfalls <b>false</b> .
<i>IList&lt;ProcessObject_DictionaryEntry&gt; GetListForDictionaryEntry(DictionaryEntry dictionaryEntry)</i>
Gibt eine Liste mit Referenzen für den gegebenen <b>DictionaryEntry</b> zurück und setzt die <i>DictionaryEntry</i> -Referenz.
<i>IList&lt;ProcessObject_DictionaryEntry&gt; GetListForDictionaryEntry(long dictionaryEntryOID)</i>
Gibt eine Liste mit Referenzen für den <b>DictionaryEntry</b> mit der gegebenen <b>OID</b> zurück.
<i>IList&lt;ProcessObject_DictionaryEntry&gt; GetListForProcessObject(long processObjectOID)</i>
Gibt eine Liste mit Referenzen für das <b>ProcessObject</b> mit der gegebenen <b>OID</b> zurück.
<i>IList&lt;ProcessObject_DictionaryEntry&gt; GetListForProcessObject(ProcessObject processObject)</i>
Gibt eine Liste mit Referenzen für das gegebenen <b>ProcessObject</b> zurück und setzt die <i>ProcessObject</i> -Referenz.
<i>bool SaveUncommitted(IList&lt;ProcessObject_DictionaryEntry&gt; processObject_DictionaryEntryList)</i>
Speichert die Liste mit Referenzen ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Speichern erfolgreich war, anderenfalls <b>false</b> .

Tabelle B.10.: Schnittstelle - IProcessObject\_DictionaryEntryManager

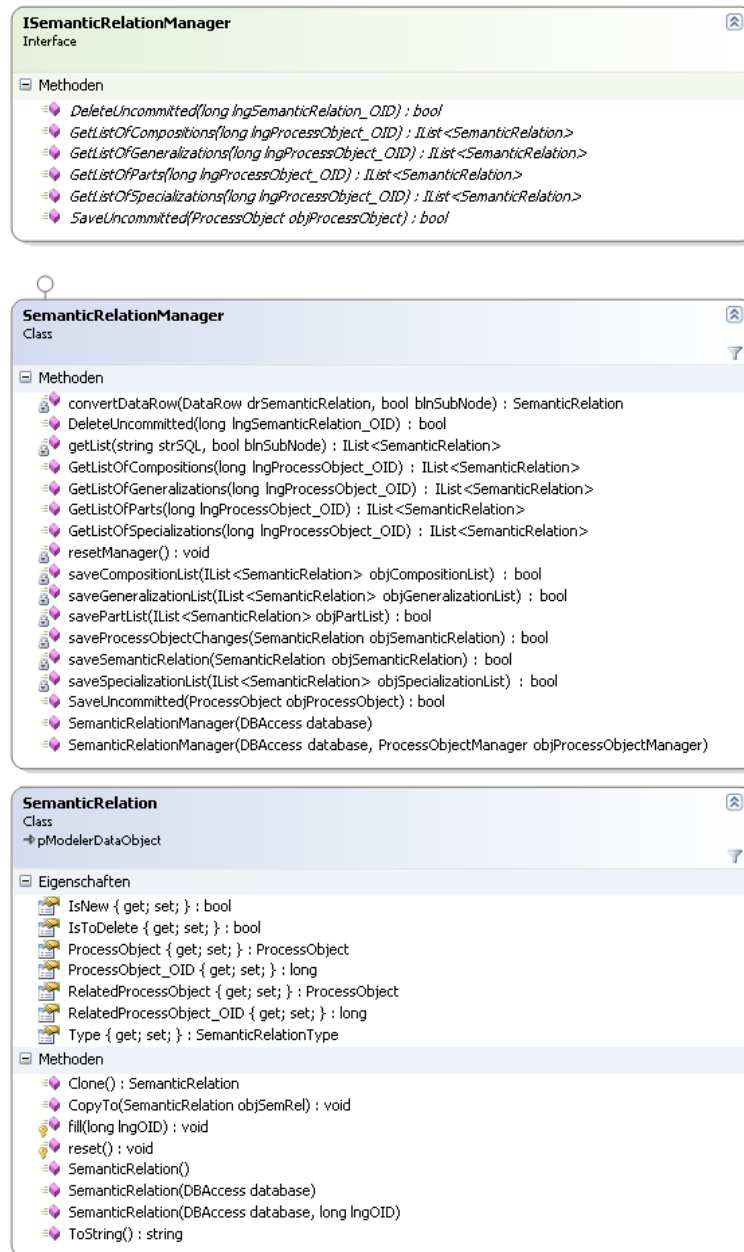


Abbildung B.9.: Objektmodell - Klassen SemanticRelation

ISemanticRelationManager
<i>bool DeleteUncommitted(long semanticRelationOID)</i>
Löscht die <b>SemanticRelation</b> mit der gegebenen OID ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Löschen erfolgreich war, anderenfalls <b>false</b> .
<i>IList&lt;SemanticRelation&gt; GetListOfCompositions(long processObjectOID)</i>
Gibt eine Liste mit allen Kompositionsbeziehungen für das <b>ProcessObject</b> mit der gegebenen OID zurück.
<i>IList&lt;SemanticRelation&gt; GetListOfGeneralizations(long processObjectOID)</i>
Gibt eine Liste mit allen Generalisierungsbeziehungen für das <b>ProcessObject</b> mit der gegebenen OID zurück.
<i>IList&lt;SemanticRelation&gt; GetListOfParts(long processObjectOID)</i>
Gibt eine Liste mit allen Teils-Beziehungen für das <b>ProcessObject</b> mit der gegebenen OID zurück.
<i>IList&lt;SemanticRelation&gt; GetListOfSpecializations(long processObjectOID)</i>
Gibt eine Liste mit allen Spezialisierungsbeziehungen für das <b>ProcessObject</b> mit der gegebenen OID zurück.
<i>bool SaveUncommitted(ProcessObject processObject)</i>
Speichert alle <b>SemanticRelations</b> für das gegebene <b>ProcessObject</b> . Gibt <b>true</b> zurück, wenn Speichern erfolgreich war, anderenfalls <b>false</b> .

Tabelle B.11.: Schnittstelle - ISemanticRelationManager

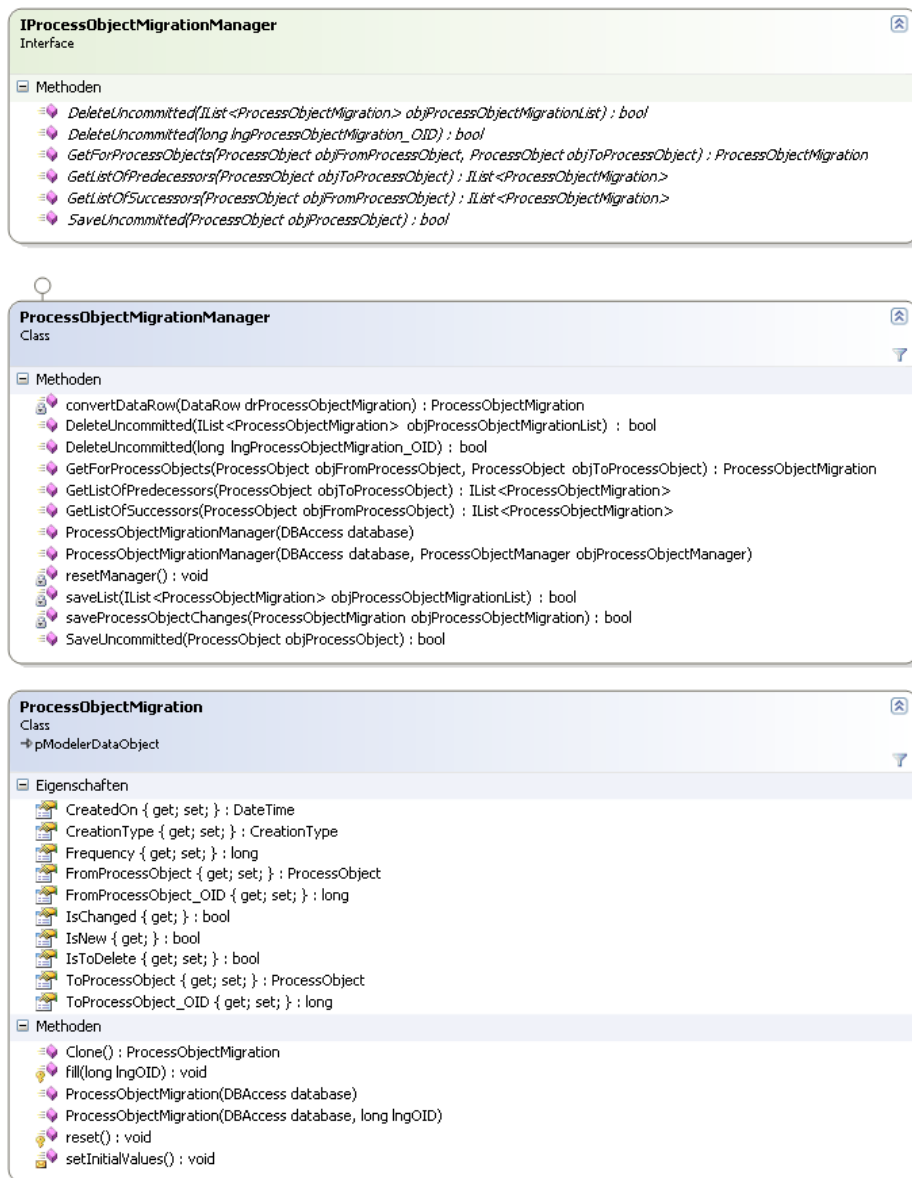
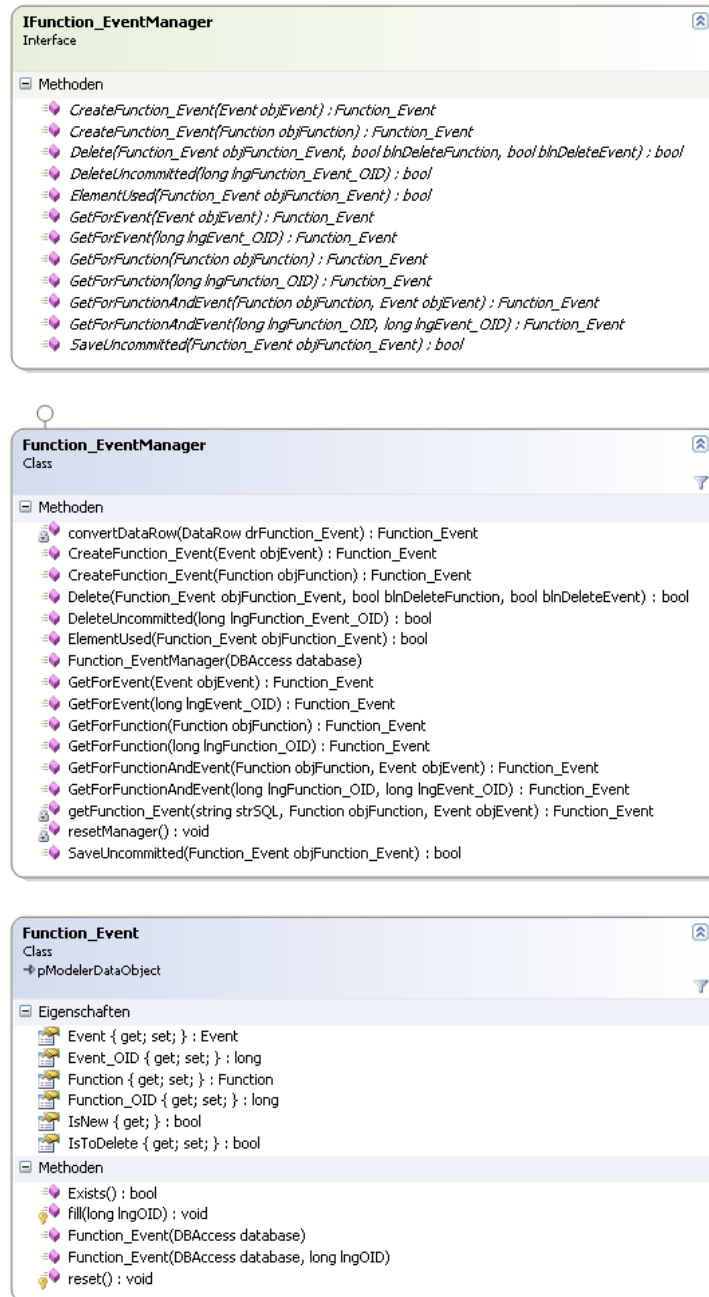


Abbildung B.10.: Objektmodell - Klassen ProcessObjectMigration



IProcessObjectMigrationManager
<i>bool DeleteUncommitted( IList&lt;ProcessObjectMigration&gt; processObjectMigrationList)</i>
Löscht die Liste mit <b>ProcessObjectMigrations</b> ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Löschen erfolgreich war, anderenfalls <b>false</b> .
<i>bool DeleteUncommitted(long processObjectMigrationOID)</i>
Löscht die <b>ProcessObjectMigration</b> mit der gegebenen <b>OID</b> ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Löschen erfolgreich war, anderenfalls <b>false</b> .
<i>ProcessObjectMigration GetForProcessObjects( ProcessObject fromProcessObject, ProcessObject toProcessObject)</i>
Gibt die <b>ProcessObjectMigration</b> mit dem gegebenen Start- und dem gegebenen Ziel- <b>ProcessObject</b> zurück.
<i>IList&lt;ProcessObjectMigration&gt; GetListOfPredecessors( ProcessObject toProcessObject)</i>
Gibt eine Liste mit <b>ProcessObjectMigrations</b> zurück, in welchen das gegebene <b>ProcessObject</b> das Ziel ist und setzt die <b>ToProcessObject</b> -Referenz.
<i>IList&lt;ProcessObjectMigration&gt; GetListOfSuccessors( ProcessObject fromProcessObject)</i>
Gibt eine Liste mit <b>ProcessObjectMigrations</b> zurück, in welchen das gegebene <b>ProcessObject</b> der Start ist und setzt die <b>FromProcessObject</b> -Referenz.
<i>bool SaveUncommitted(ProcessObject processObject)</i>
Speichert alle <b>ProcessObjectMigrations</b> für das gegebene <b>ProcessObject</b> . Gibt <b>true</b> zurück, wenn Speichern erfolgreich war, anderenfalls <b>false</b> .

Tabelle B.12.: Schnittstelle - IProcessObjectMigrationManager

Abbildung B.11.: Objektmodell - Klassen `Function_Event`

IFunction_EventManager
<i>Function_Event CreateFunction_Event(Event event)</i>
Gibt eine <b>Function_Event</b> -Beziehung für den gegebenen <b>Event</b> zurück. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz (Generierung umfasst gegebenenfalls auch die Generierung der <b>Function</b> und/oder des <b>Events</b> ).
<i>Function_Event CreateFunction_Event(Function function)</i>
Gibt eine <b>Function_Event</b> -Beziehung für die gegebenen <b>Function</b> zurück. Existiert die gewünschte Instanz bereits in der Ontologie, wird diese zurückgegeben, anderenfalls eine neu generierte Instanz (Generierung umfasst gegebenenfalls auch die Generierung der <b>Function</b> und/oder des <b>Events</b> ).
<i>bool Delete(Function_Event function_Event, bool deleteFunction, bool deleteEvent)</i>
Löscht die gegebene <b>Function_Event</b> -Beziehung mit Transaktionskontrolle. Die booleschen Parameter spezifizieren, ob die referenzierte <b>Function</b> bzw. der referenzierte <b>Event</b> ebenfalls gelöscht werden sollen. Gibt <b>true</b> zurück, wenn Löschen erfolgreich war, anderenfalls <b>false</b> .
<i>bool DeleteUncommitted(long function_EventOID)</i>
Löscht die <b>Function_Event</b> -Beziehung ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Löschen erfolgreich war, anderenfalls <b>false</b> .
<i>bool ElementUsed(Function_Event function_Event)</i>
Gibt <b>true</b> zurück, wenn die <b>Function</b> und/oder der <b>Event</b> verwendet werden, anderenfalls <b>false</b> .

IFunction_EventManager (Fortsetzung)
<i>Function_Event GetForEvent(Event event)</i>
Gibt die <b>Function_Event</b> -Beziehung für den gegebenen <b>Event</b> zurück und setzt die <i>Event</i> -Referenz.
<i>Function_Event GetForEvent(long eventOID)</i>
Gibt die <b>Function_Event</b> -Beziehung für den <b>Event</b> mit der gegebenen <b>OID</b> zurück.
<i>Function_Event GetForFunction(Function function)</i>
Gibt die <b>Function_Event</b> -Beziehung für die gegebene <b>Function</b> zurück und setzt die <i>Function</i> -Referenz.
<i>Function_Event GetForFunction(long functionOID)</i>
Gibt die <b>Function_Event</b> -Beziehung für die <b>Function</b> mit der gegebenen <b>OID</b> zurück.
<i>Function_Event GetForFunctionAndEvent(Function function, Event event)</i>
Gibt die <b>Function_Event</b> -Beziehung für die gegebene <b>Function</b> und den gegebenen <b>Event</b> zurück und setzt die <i>Function</i> - und <i>Event</i> -Referenz.
<i>Function_Event GetForFunctionAndEvent(long functionOID, long eventOID)</i>
Gibt die <b>Function_Event</b> -Beziehung für die <b>Function</b> mit der gegebenen <b>OID</b> und den <b>Event</b> mit der gegebenen <b>OID</b> zurück.
<i>bool SaveUncommitted(Function_Event function_Event)</i>
Speichert die <b>Function_Event</b> -Beziehung ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Speichern erfolgreich war, anderenfalls <b>false</b> .

Tabelle B.13.: Schnittstelle - IFunction\_EventManager

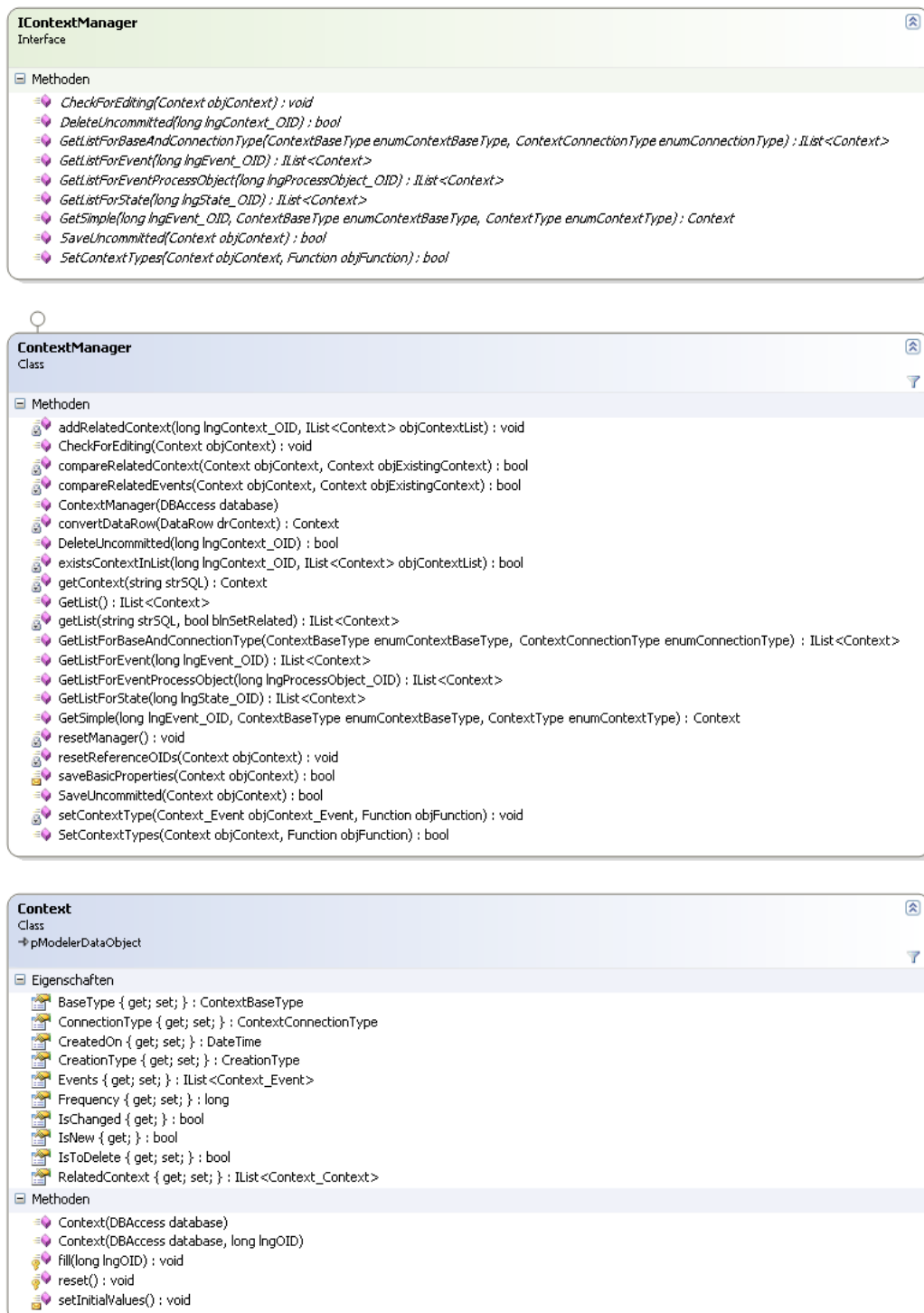


Abbildung B.12.: Objektmodell - Klassen Context

<b>IContextManager</b>
<i>void CheckForEditing(Context context)</i>
Prüft, ob der gegebene <b>Context</b> bereits existiert und setzt gegebenenfalls die OIDs der bestehenden Instanz(en). Die Prüfung bezieht gegebenenfalls verschachtelte Kontextinformationen mit ein.
<i>bool DeleteUncommitted(long contextOID)</i>
Löscht den <b>Context</b> mit der gegebenen OID ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Löschen erfolgreich war, anderenfalls <b>false</b> .
<i>ICollection&lt;Context&gt; GetListForBaseAndConnectionType( ContextBaseType baseType, ContextConnectionType connectionType)</i>
Gibt eine Liste mit <b>Contexts</b> mit dem gegebenen <b>ContextBaseType</b> und dem gegebenen <b>ContextConnectionType</b> zurück.
<i>ICollection&lt;Context&gt; GetListForEvent(long eventOID)</i>
Gibt eine Liste mit <b>Contexts</b> mit dem <b>Event</b> mit der gegebenen OID zurück.
<i>ICollection&lt;Context&gt; GetListForEventProcessObject(long processObjectOID)</i>
Gibt eine Liste mit <b>Contexts</b> zurück, welche <b>Events</b> mit dem <b>Process-Object</b> mit der gegebenen OID enthalten.
<i>ICollection&lt;Context&gt; GetListForState(long stateOID)</i>
Gibt eine Liste mit <b>Contexts</b> zurück, welche <b>Events</b> mit dem <b>State</b> mit der gegebenen OID enthalten.
<i>Context GetSimple(long eventOID, ContextBaseType baseType, ContextType contextType)</i>
Gibt einen einfachen <b>Context</b> mit dem <b>Event</b> mit der gegebenen OID, dem gegebenen <b>ContextBaseType</b> und <b>ContextType</b> zurück.
<i>bool SaveUncommitted(Context context)</i>
Speichert den gegebenen <b>Context</b> ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Speichern erfolgreich war, anderenfalls <b>false</b> .
<i>bool SetContextTypes(Context context, Function function)</i>
Setzt den <b>ContextType</b> in den <b>Context_Event</b> -Beziehungen des gegebenen <b>Context</b> für die gegebene <b>Function</b> (der <b>ContextType</b> wird durch Vergleich von <b>Function</b> und <b>Event</b> ermittelt).

Tabelle B.14.: Schnittstelle - IContextManager

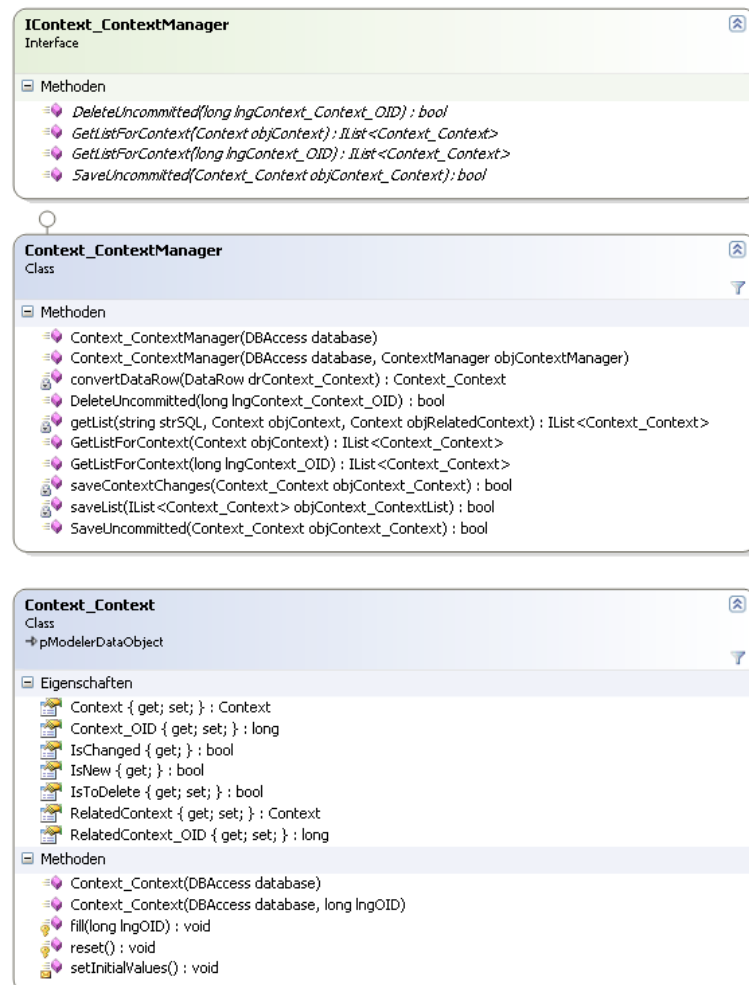


Abbildung B.13.: Objektmodell - Klassen Context\_Context

IContext_ContextManager
<i>bool DeleteUncommitted(long context_ContextOID)</i>
Löscht die <b>Context_Context</b> -Beziehung mit der gegebenen OID ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Löschen erfolgreich war, anderenfalls <b>false</b> .
<i>ICollection&lt;Context_Context&gt; GetListForContext(Context context)</i>
Gibt eine Liste von <b>Context_Context</b> -Beziehungen für den gegebenen <b>Context</b> zurück und setzt die <i>Context</i> -Referenz.
<i>ICollection&lt;Context_Context&gt; GetListForContext(long contextOID)</i>
Gibt eine Liste mit <b>Context_Context</b> -Beziehung mit dem <b>Context</b> mit der gegebenen OID zurück.
<i>bool SaveUncommitted(Context_Context context_Context)</i>
Speichert die gegebene <b>Context_Context</b> -Beziehung ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Speichern erfolgreich war, anderenfalls <b>false</b> .

Tabelle B.15.: Schnittstelle - IContext\_ContextManager



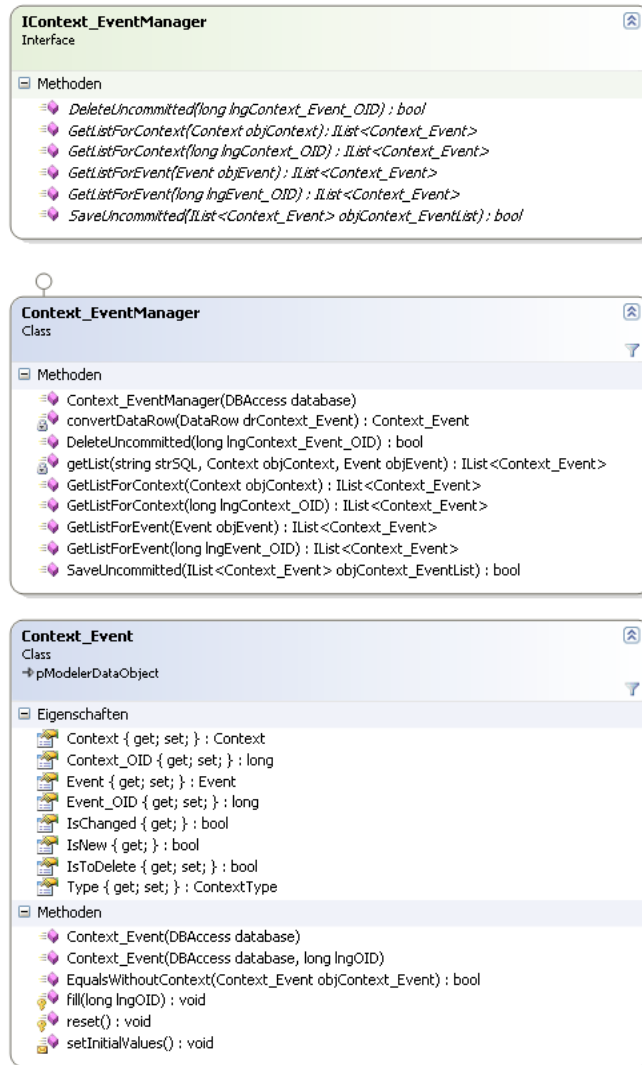


Abbildung B.14.: Objektmodell - Klassen Context\_Event

IContext_EventManager
<i>bool DeleteUncommitted(long context_EventOID)</i>
Löscht die <b>Context_Event</b> -Beziehung mit der gegebenen OID ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Löschen erfolgreich war, anderenfalls <b>false</b> .
<i>IList&lt;Context_Event&gt; GetListForContext(Context context)</i>
Gibt eine Liste mit <b>Context_Event</b> -Beziehungen für den gegebenen <b>Context</b> zurück und setzt die <i>Context</i> -Referenz.
<i>IList&lt;Context_Event&gt; GetListForContext(long contextOID)</i>
Gibt eine Liste mit <b>Context_Event</b> -Beziehungen für den <b>Context</b> mit der gegebenen OID zurück.
<i>IList&lt;Context_Event&gt; GetListForEvent(Event event)</i>
Gibt eine Liste mit <b>Context_Event</b> -Beziehungen für den gegebenen <b>Event</b> zurück und setzt die <i>Event</i> -Referenz.
<i>IList&lt;Context_Event&gt; GetListForEvent(long eventOID)</i>
Gibt eine Liste mit <b>Context_Event</b> -Beziehungen für den <b>Event</b> mit der gegebenen OID zurück.
<i>bool SaveUncommitted(IList&lt;Context_Event&gt; context_EventList)</i>
Speichert die gegebene Liste mit <b>Context_Event</b> -Beziehungen ohne Transaktionskontrolle. Gibt <b>true</b> zurück, wenn Speichern erfolgreich war, anderenfalls <b>false</b> .

Tabelle B.16.: Schnittstelle - IContext\_EventManager