Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3429

Modellierung von Scientific Workflows mit Choreographien

Oliver Sonnauer

Studiengang:	Softwaretechnik
Prüfer:	Jun.-Prof. Dr.-Ing Dimka Karastoyanova
Betreuer:	Dipl.-Inf. Dipl.-Wirt. Ing. (FH) Karolina Vukojevic
begonnen am:	06. November 2012
beendet am:	08. Mai 2013
CR-Klassifikation:	D.1.7, D.2.11, H.4.1, I.3.4

Kurzfassung

Diese Arbeit beschäftigt sich mit der Konzeption eines grafischen Editors zur Modellierung von Choreographien. Das BPEL4Chor Modell dient dabei als Grundlage. Mit dem Editor soll es zudem möglich sein, ausführbare BPEL Prozesse aus der modellierten Choreographie zu generieren. Anhand der Konzeption wird die Implementierung eines Prototyps für die Eclipse Plattform vorgestellt. Dieser Prototyp wird mit dem Graphical Modeling Framework realisiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

1 Einleitung

Das Exzellenzcluster Simulation Technology (SimTech)[1] befasst sich mit Multi-Skalen und Multi-Physik Simulationsmethoden. Multi-Skalar bedeutet dabei, auf mehreren Skalierungsebenen wie z. B. von der Zelle zum Gewebe über Knochen zum Skelett. Multi-Physik bedeutet, dass mehrere Teilbereiche der Physik wie Thermodynamik oder Quantenmechanik verwendet werden. Diese einzelnen Simulationen werden mit Hilfe der Workflowtechnologie, als einzelne Prozesse modelliert. Die Workflowtechnologie hat ihre Ursprung in der Industrie, wo sie zur Automatisierung von Geschäftsprozessen verwendet wird. Workflows für Geschäftsprozesse werden üblicherweise vom Management spezifiziert. Die IT-Abteilung kümmert sich dann um die technischen Details der Ausführung. Haben sich diese Prozesse etabliert, werden sie meistens nicht mehr abgeändert. Im Rahmen der SimTech Projekte modellieren Wissenschaftler ihre Simulationen mit Hilfe von Workflows. Dabei ist es meistens die selbe Person oder Personengruppe, welche die Workflows spezifizieren und ausführen. Hinzu kommt noch die experimentelle Natur von Simulationen. Sie müssen bei der Ausführung überwacht werden. Das Resultat wird evaluiert, darauf der Workflow angepasst und erneut ausgeführt. Diese Kreislauf wiederholt sich sehr oft, was eine ständige Anpassung des Prozesses zur Folge hat. Daher bezeichnet man die Workflows im wissenschaftlichen Kontext als *Scientific Workflows* [TDG06]. Diese *Scientific Workflows*, welche die einzelnen Simulationen ausführen, sollen zu einer ganzheitlichen Simulation zusammengeführt werden. Betrachtet man die einzelnen Simulationen als „Geschäftspartner“, so kann deren Zusammenwirken als Choreographie beschrieben werden. Daher soll die SimTech Workflow Umgebung so erweitert werden, dass Wissenschaftler Choreographien modellieren und diese ausführen können. Als Modell für diese Choreographien soll die Beschreibungssprache BPEL4Chor [KL08] dienen.

Ziele dieser Arbeit

Es soll ein grafisches Werkzeug entwickelt werden, mit welchem sich Choreographien modellieren lassen. Für diese grafischen Modelle soll einerseits die Möglichkeit bestehen, BPEL4Chor Artefakte zu generieren und andererseits, ausführbare BPEL Prozesse mit zugehörigen WSDL Dateien zu erzeugen.

[1] http://www.simtech.uni-stuttgart.de/

Gliederung

Kapitel 2 – Verwandte Arbeiten beschreibt andere Arbeiten, die sich ebenfalls mit dem Modellieren von Choreographien befassen.

Kapitel 3 – Grundlagen beschriebt die grundlegenden Dinge, die für das Verständnis dieser Arbeit wichtig sind und worauf diese aufbaut.

Kapitel 4 – Konzeption beschreibt diverse Ansätze, wie der grafische Editor realisiert werden kann und geht dann detailliert auf den gewählten Ansatz ein.

Kapitel 5 – Implementierung befasst sich einerseits mit den verwendeten Technologien, die zur Umsetzung des Editors benutzt werden und geht andererseits auf einige Details der Implementierung ein.

Kapitel 6 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt mögliche weiterführende Arbeiten vor.

2 Verwandte Arbeiten

In [DKL+08] wird beschrieben, wie Choreographien mit BPMN modelliert, und diese dann zu BPEL4Chor Artefakten exportiert werden können. Die Modellierung erfolgt grafisch mit dem Oryx Framework[1]. Dieses Framework ist in JavaScript realisiert und benutzt skalierbare Vektorgrafiken (SVG) zur Darstellung der grafischen Elemente. Somit kann Oryx in einem Web Browser ausgeführt werden. Es wurde durch ein Export Plugin erweitert, welches die Transformation von BPMN zu BPEL4Chor durchführt. Ein weiterer grafischer Editor, der das modellieren von Choreographien unterstützt, ist der in Java implementierte *Yaoqiang BPMN Editor*[2]. In [FUMK06] ist ein Eclipse Plugin beschrieben, welches einen modellbasierten Ansatz zur Verifizierung von Web Service Kompositionen und Choreographien realisiert. Dabei werden Choreographien auf einfache Weise als UML Sequenzdiagramme dargestellt.

[1] http://bpt.hpi.uni-potsdam.de/Oryx/BPMN
[2] http://sourceforge.net/projects/bpmn/

3 Grundlagen

Die Kenntnis der hier vorgestellten Grundlagen ist wichtig für das weitere Verständnis dieser Arbeit. Wir beschreiben Choreographien und BPEL nur kurz und gehen ausführlich auf BPEL4Chor ein. Wir gehen außerdem auf das *Model View Controller* Architekturmuster ein, da die meisten der verwendeten Technologien, sowie der Editor selbst, danach konstruiert sind.

3.1 Choreographien

Die folgende Beschreibung basiert auf dem Artikel [KL08]. Der Architekturstiel *Service-Oriented Architcture* (SOA) basiert auf dem Service Paradigma. Ein Service ist eine Funktionalität, welche über eine Netzwerkadresse zur Verfügung gestellt wird und immer erreichbar ist [Ley10]. Wenn ein Geschäftsprozess durch Zusammenarbeit von vielen Services realisiert wird, nennt man dies „Orchestrierung“. Eine Orchestrierung beschreibt die Interaktionen mit Services aus dem Blickpunkt dieses einen Prozesses. Interagieren mehrere Prozesse miteinander, so nennt man dies „Choreographie“. Choreographien erfassen die Zusammenarbeit mehrerer Geschäftspartner aus einer globalen Sicht. Das Design von Choreographien kann für die beteiligten Geschäftspartner sehr wichtig sein da sie, durch Abstimmung ihrer Geschäftsprozesse, eventuelle Synergieeffekte finden und nutzen können. Die Herausforderungen beim Design von Choreographien sind zum einen die Modellierung in einer geeigneten Sprache wie z. B. BPMN oder BPEL4Chor. Zum anderen muss das Choreographie Modell auf Korrektheit überprüft werden können. Es dürfen z. B. keinen Deadlocks vorkommen und es muss immer ein Endzustand erreichbar sein. Im letzten Schritt muss es eine Möglichkeit geben, die durch die Choreographie gegebene Semantik auf einzelne Prozesse zu übertragen, so dass diese ausgeführt werden können.

Um Choreographien zu modellieren gibt es zwei wesentliche Ansätze. Der *interconnection models* Ansatz beschreibt die Teilnehmer einer Choreographie als abstrakte Prozesse. Ihr Verhalten ist durch Aktivitäten spezifiziert, welche für die Kommunikation relevant sind. Andere prozessinterne Vorgänge werden ausgeblendet. Die Kommunikationsaktivitäten der einzelnen Teilnehmer sind miteinander verbunden und tauschen Nachrichten aus. Ein Beispiel dazu sehen wir in Abbildung 3.4. Der *interaction models* Ansatz beschreibt nur die Interaktionen der einzelnen Teilnehmer einer Choreographie. Hier sind die Teilnehmer nicht als abstrakte Prozesse mit Aktivitäten spezifiziert, da nur der Nachrichtenfluss im

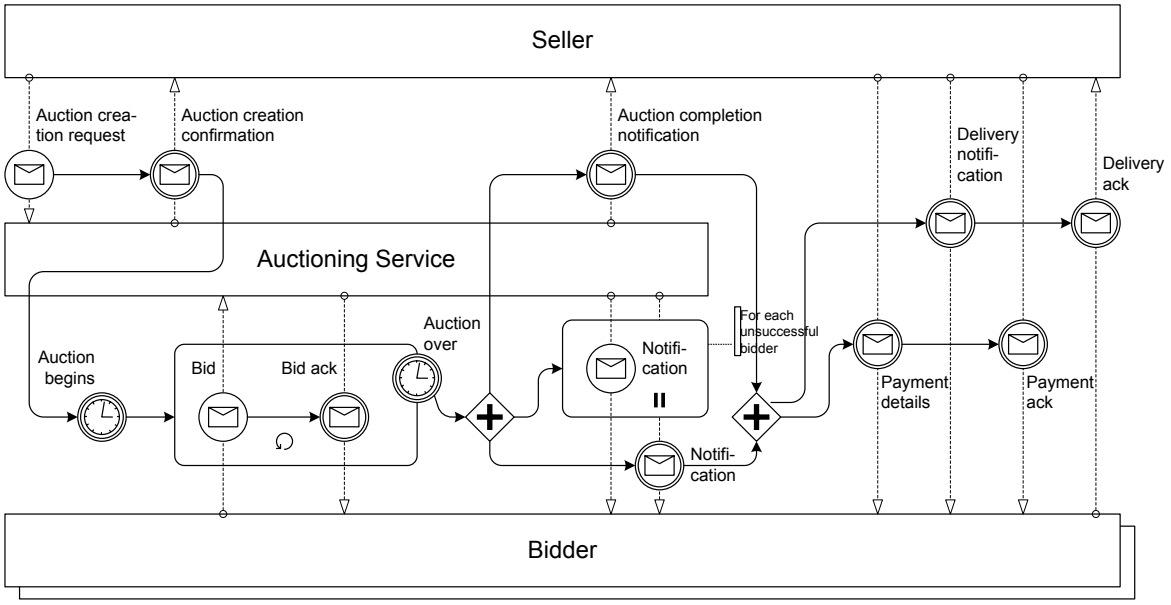

Abbildung 3.1: Auktionsszenario. Choreographie Beispiel modelliert als *interaction model* in BPMN. Quelle: [DB08]

Vordergrund steht. Ein Beispiel dazu sehen wir in Abbildung 3.1. In beiden Abbildungen ist ein Auktionsszenario modelliert, welches wir in Abschnitt 3.3.2 auf Seite 18 beschreiben werden.

3.2 BPEL

Die *Web Service Business Process Execution Language* (WS-BPEL kurz: BPEL), spezifiziert in [OAS07c], ist eine Sprache zur Beschreibung von Geschäftsprozessen. Diese BPEL Prozesse benutzen die Schnittstellen von Web Services, um ihre Funktion zu realisieren. Wie in [Ley11] beschrieben, aggregieren BPEL Prozesse Web Services. Sie sind selbst ebenfalls ein Web Service, was BPEL zu einem rekursiven Aggregationsmodell macht. BPEL ist eine Kombination aus der Graph basierten Sprache *IBM WSFL*, sowie der Kalkül basierten Sprache *MS XLANG*. Die Syntax ist in XML definiert. Die Hauptbestandteile eines BPEL Dokuments sind *Partner Links*, *Variables*, *Correlation Sets*, *Handlers* und *Activities*. Ein *Partner Link* ist ein Kommunikationskanal zwischen Prozess und einem Partner. Er verbindet maximal zwei WSDL *Port Types*. Eine *Variable* ist ein persistenter Container für Daten. Ein *Correlation Set* ist eine Sammlung von *properties*. Dies sind Werte um Prozessinstanzen eindeutig identifizieren zu können. Sie sind in Nachrichten eingebettet und können somit unter den Kommunikationspartnern ausgetauscht werden. Ein *Handler* regelt Ausnahmezustände, oder das Auftreten bestimmter

Ereignisse in laufenden Prozessen. Mit *Activities* lassen sich Kontrollfluss und Web Service Aufrufe definieren. *Activities* unterscheiden sich in Basisaktivitäten und strukturierte Aktivitäten, welche andere Aktivitäten beinhalten. Zu den Basisaktivitäten gehört ‹ `Invoke` ›, mit welcher sich Web Service Aufrufe, sowohl synchron als auch asynchron, durchführen lassen. Zu den strukturierten Aktivitäten gehört ‹ `Sequence` ›, welche die in ihr platzierten Aktivitäten in angegebener Reihenfolge ausführt. Die strukturierte ‹ `Flow` › Aktivität erlaubt hingegen eine parallele, graphbasierte Ausführung seiner Aktivitäten.

BPEL Prozesse können ausführbar oder abstrakt sein. Abstrakte Prozesse sind teilweise spezifizierte Prozesse, welche nicht zur Ausführung bestimmt sind. Sie verstecken bewusst Details und zeigen nur die wesentlichen Aspekte eines Geschäftsprozesses. Welche Aspekte das sind, wird vom Prozessersteller je nach Anwendungsfall entschieden. Wie wir im folgenden Abschnitt sehen werden, verwendet BPEL4Chor abstrakte Prozesse zur Beschreibung des Verhaltens der Teilnehmer einer Choreographie.

3.3 BPEL4Chor

Die folgenden Beschreibungen zu BPEL4Chor und seinen Artefakten basieren auf den Artikeln [DKLW07] und [DKLW09]. Andere Quellen werden gekennzeichnet. Wie wir in Abschnitt 3.1 auf Seite 14 gesehen haben, gibt es die zwei Herangehensweisen *interaction models* und *interconnection models* um Choreographien zu modellieren. BPEL4Chor basiert auf letzterem. Der WS-BPEL Standard [OAS07c] spezifiziert abstrakte BPEL Prozesse. Die Syntax dieser Prozesse ist in der "common base"[OAS07a] festgelegt. Diese "common base" gibt zwar vor, welche BPEL Konstrukte erlaubt sind, doch fehlt dem modellierten Prozess eine Semantik. Aus der Semantik geht hervor, für welchen Anwendungsfall der abstrakte Prozess definiert wurde. Sie hilft uns die Intention dieses Prozesses zu verstehen, damit daraus eine ausführbare Variante erstellt werden kann. Um abstrakten Prozessen eine Semantik zu geben, müssen Profile angegeben werden. Ein Profil definiert einerseits die zugelassenen BPEL Konstrukte und andererseits eine "executable completion" für die abstrakten Prozesse, welche zu diesem Profil gehören. Eine "executable completion" ist eine Vorschrift, wie genau der abstrakte Prozess zu einem ausführbaren erweitert werden soll. Das *Abstract Process Profile for Observable Behavior* Profil, definiert im WS-BPEL Standard, sowie dazu ein *interconnection layer* in Form einer Topologie Beschreibung, ergeben zusammen das *interconnection model* von BPEL4Chor. Durch die Verwendung von abstrakten Prozessen mit diesem Profil, ist eine Entkopplung von WSDL *Port Types*, sowie eine Fokussierung auf die Kernbestandteile einer Choreographie nämlich Kommunikationsaktivitäten, Abhängigkeiten des Verhaltens der Teilnehmer und deren Vernetzung, möglich.

In Abbildung 3.2 sind die drei verschiedenen Artefakte von BPEL4Chor zu sehen. Die *Participant Behavior Description* Dokumente definieren den Kontrollfluss der einzelnen Aktivitäten, welche zu einem bestimmten Teilnehmer gehören und bestimmen dadurch sein Verhalten. Sie

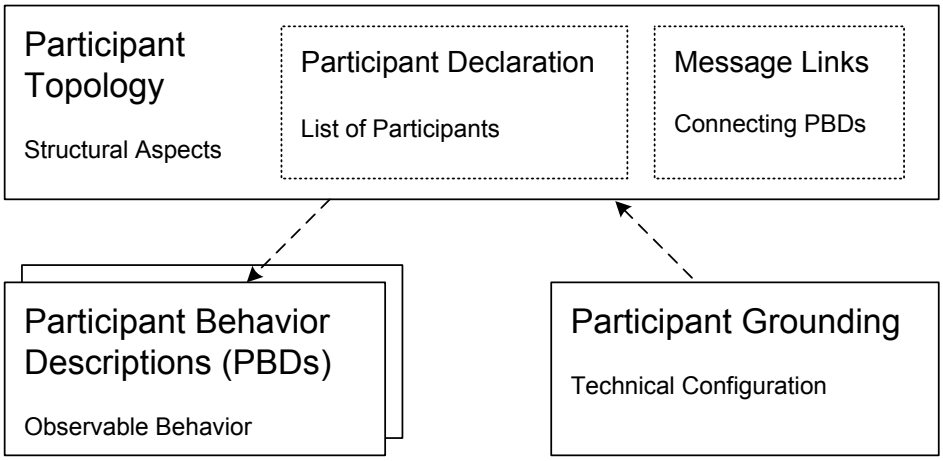

Abbildung 3.2: Die Artefakte von BPEL4Chor. Quelle: [DKLW09]

sind als abstrakte BPEL Prozesse spezifiziert. Das *Participant Topology* Dokument beschreibt, welche Teilnehmer es in einer Choreographie gibt und wie diese miteinander kommunizieren. Die Kommunikation erfolgt über einen Nachrichtenaustausch zwischen Kommunikationsaktivitäten der einzelnen Teilnehmer. BPEL4Chor unterscheidet dabei die Kommunikationsaktivitäten ‹ `Invoke` ›, ‹ `Receive` ›, ‹ `Reply` › und den ‹ `OnMessage` › Zweig aus ‹ `Pick` ›. Die *Participant Grounding* Dokumente beschreiben die technischen bzw. WSDL spezifischen Aspekte einer BPEL4Chor Choreographie. Somit kann eine Choreographie an verschiedene WSDL Definitionen gebunden werden.

3.3.1 Participant Behavior Description

In diesem Dokument wird das Verhalten eines Teilnehmers spezifiziert. Der Kontrollfluss zwischen den Kommunikationsaktivitäten bestimmt auch die Reihenfolge der Nachrichten, welche sich die Teilnehmer untereinander senden. Als syntaktische Basis dient der WS-BPEL Standard für abstrakte Prozesse. Die Eigenschaften der abstrakten Prozesse und Einschränkungen durch Profile werden in [OAS07c] genau beschrieben und hier nur zusammenfassend erwähnt. Abstrakte Prozesse bieten Opaque Erweiterungen. Diese Opaque Konstrukte haben keine Semantik sondern zeigen explizit an, dass etwas bewusst weggelassen wurde. So z. B. die ‹ `OpaqueActivity` ›, welche als Platzhalter für eine beliebige BPEL Aktivität angegeben werden kann oder der reservierte String-Wert "##opaque", um z. B. die Angabe von konkreten Variablen zu umgehen. Diese Basis ("common base"), spezifiziert in [OAS07a], wird durch das *Abstract Process Profile for Observable Behavior* Profil eingeschränkt. Dieses Profil verfolgt das Ziel, Verhalten der Prozesse im Kontext von Web Service Aufrufen zu definieren. Dabei sollen die Prozess internen Vorgänge nach außen

hin versteckt und hauptsächlich die Kommunikation mit den Partnern beschrieben werden. Diese Einschränkungen sind im WS-BPEL Standrad [OAS07c] definiert und schreiben vor, dass ‹ JoinCondition ›, welches für Aktivitäten in einem ‹ Flow › verwendet wird, das Attribut *opaque* nicht benutzen darf. Die Verwendung der ‹ Exit › Aktivität ist ebenfalls nicht erlaubt. Die Attribute *variable*, *inputVariable* und *outputVariable* von ‹ Invoke ›, ‹ Receive ›, ‹ Reply ›, ‹ OnMessage › und ‹ OnEvent ›, dürfen mit dem Wert "##opaque" deklariert werden. Das selbe gilt auch für die Attribute *part*, *toVariable* und *fromVariable* von ‹ FromPart › sowie ‹ ToPart ›. Alle anderen Attribute dürfen **nicht** mit dem Wert "##opaque" deklariert werden. Des weiteren ist das ‹ OpaqueFrom › Konstrukt erlaubt, welches in der ‹ Copy › Anweisung der ‹ Assign › Aktivität eingesetzt werden kann. Von diesem Profil ausgehend, wurde das neue Profil *Abstract Process Profile for Participant Behavior Descriptions* in [DKLW09] eingeführt, welches all dessen Eigenschaften erbt, sowie zusätzliche Einschränkungen für die Kommunikationsaktivitäten spezifiziert. So dürfen diese Aktivitäten die Attribute *partnerLink*, *portType* und *operation* **nicht** benutzen, da sonst die *Participant Behavior Description* an eine WSDL gebunden wird. Des weiteren wird das neue Attribut *wsu:id* für diese Aktivitäten eingeführt, um sie in den MessageLink Elementen eindeutig referenzieren zu können. Bei den ‹ Receive › und ‹ Reply › Aktivitäten muss zwingend das *messageExchange* Attribut gesetzt werden, um die zugehörigen Paare kennzeichnen zu können. Die Paar Identifikation wäre auch über *portType* und *operation* möglich, da aber diese Attribute nicht erlaubt sind, fällt diese Möglichkeit weg. CorrelationSets können ebenfalls in *Participant Behavior Description* benutzt werden, müssen jedoch im *properties* Attribut auf nicht qualifizierte Namen (NCName) eingeschränkt werden, da sonst eine Bindung zu PropertyAlias Elementen aus der WSDL besteht.

3.3.2 Topology

In diesem Dokument werden die strukturellen Aspekte einer Choreographie beschrieben. Dazu gehört die Angabe, welche Teilnehmer die Choreographie hat und wie viele es davon gibt. Das Verhalten der einzelnen Teilnehmer wird festgelegt. Der Nachrichtenaustausch wird durch Angaben definiert, wer mit wem kommuniziert und welche Aktivitäten dafür verwendet werden. In Abbildung 3.3 ist die *Topology* anhand eines UML Modelles beschrieben, welches auf Basis des *Topology* XSD Schemas [DK12b] entworfen wurde. Die Teilnehmer der Choreographie werden durch Participant und ParticipantSet Elemente angegeben. Das ParticipantSet wurde eingeführt, damit eine unbestimmte Anzahl von Teilnehmern mit gleichem Verhalten modelliert werden kann. Ein Beispiel dafür wäre bei einer Auktion, die unbestimmte Menge an Bietern, welche ihre Gebote auf einen Gegenstand abgeben. Die Konkrete Menge der Bieter ist erst zur Laufzeit bekannt und wird sich generell auch jedes mal beim Ausführen ändern. Wenn die Anzahl der Teilnehmer zum Zeitpunkt des Modellierens bekannt ist, werden diese Teilnehmer jeweils durch ein Participant Element angegeben. Ein Beispiel dafür wäre ein Käufer und ein Verkäufer. In Choreographien kann es häufig

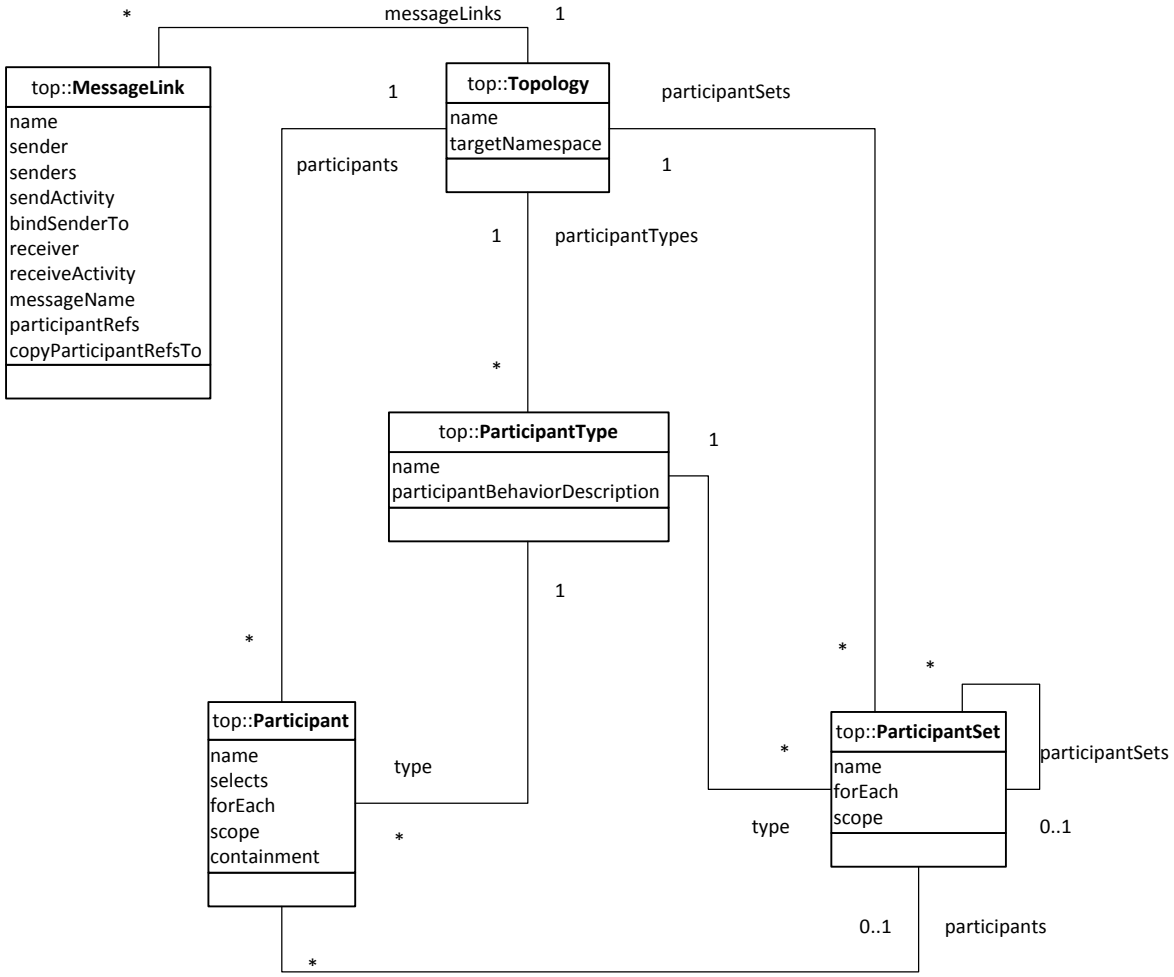

Abbildung 3.3: Topology XSD ([DK12b]) dargestellt als UML Modell

vorkommen, dass es eine bestimmte Anzahl von Teilnehmern mit gleichem Verhalten gibt wie z. B. zwei Spediteure. Damit diese Konstellation exakter beschrieben werden kann, wurde ein `ParticipantType` Element eingeführt. In BPEL4Chor gilt, dass jeder Teilnehmer einen Typ hat und dieser sein Verhalten bestimmt. So sind z. B. die Teilnehmer Firma X und Firma Y beide vom Typ Spediteur und haben somit das selbe Verhalten. Nachrichten tauschen die Teilnehmer untereinander durch `MessageLink` Elemente aus. Ein `MessageLink` beschreibt einen Kommunikationsweg vom Sender zum Empfänger.

In Abbildung 3.4 sehen wir eine Choreographie eines Auktionsszenarios, modelliert in BPMN[1]. Der Verkäufer (*Seller Service*) möchte Aktienanteile zum höchsten Gebot verkaufen.

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

Er beauftragt die Durchführung einer Auktion bei einem Makler (*Broker Service*). Wenn die Auktion startet, akzeptiert der Makler mehrere Gebote von unterschiedlichen Bietern (*Bidder Service*). Ist die Auktion nach Ablauf einer Frist beendet, benachrichtigt der Makler den Verkäufer über die Beendigung und den höchst bietenden Käufer über dessen Erfolg. Alle anderen Käufer werden vom Makler benachrichtigt, dass sie die Auktion verloren haben. Damit ist die Aufgabe des Maklers erfüllt. Der Verkäufer schickt dem Höchstbieter die Zahlungsdetails und überträgt ihm die Aktienanteile. In Listing 3.1 sehen wir einen Ausschnitt der *Topology* Beschreibung dieses Auktionsszenarios. Die Verkäufer und der Makler sind jeweils als `Participant` in Zeile **8** und **9** modelliert. Die Menge der Bieter ist von der Anzahl unbestimmt und zweigeteilt. Zum einen die Menge aller Bieter in Zeile **10** und, zum anderen, die Menge der Bieter, welche die Auktion nicht gewonnen haben in Zeile **14**. Wir sehen, dass beide `ParticipantSet` Elemente Kind Elemente haben. Dies sind Teilnehmer welche dieser Menge angehören was ebenfalls bedeutet, dass sie auch das selbe Verhalten (den selben `ParticipantType`) haben wie das Set, in welchem sie deklariert sind. In unserem Beispiel ist der `Participant` "bidder" in Zeile **11**, ein konkreter Teilnehmer, welcher ein Gebot an den ausgewählten Makler "brokerService" abgibt. Diese Auswahl wird mit dem *`selects`* Attribut getroffen. Der `Participant` "successfulBidder" in Zeile **12**, ist der Auktionsgewinner. Im anderen `ParticipantSet` "unsuccessfulBidders" aus Zeile **14**, sehen wir die Verwendung des *`forEach`* Attributs. Es verweist auf die ‹ `ForEach` › Aktivität der *Participant Behavior Description* des Maklers. Um die Idee dahinter zu verstehen, betrachten wir noch einmal Abbildung 3.4. Hier sehen wir, dass der Makler alle Verlierer der Auktion benachrichtigen muss (siehe "Send unsuccessful bid" Aktivität). In BPEL4Chor wird dies durch Referenzieren einer ‹ `ForEach` › Aktivität von einem `ParticipantSet` ausgedrückt. Diese ‹ `ForEach` › Aktivität iteriert über alle Teilnehmer des Sets. Wir sehen in Listing 3.1, dass das `ParticipantSet` aus Zeile **14** ein Kind Element (Zeile **15**) hat, welches die selbe ‹ `ForEach` › Aktivität referenziert. Dieser Teilnehmer "currentBidder" repräsentiert den aktuellen Schleifenwert in jeder Iteration und wird zur Benachrichtigung eines Auktionsverlierers benutzt. Das bedeutet, pro Schleifendurchlauf eine Benachrichtigung. Die Definition des Teilnehmers "currentBidder" ist notwendig, da er einen Kommunikationspartner darstellt.

Die `MessageLink` Elemente geben an, welche Teilnehmer untereinander kommunizieren. Dabei wird keine Reihenfolge in der Topologie vorgegeben. Das *`sender`* Attribut gibt den `Participant` an, welcher der Absender dieser Nachricht ist. Das *`senders`* Attribut hingegen bedeutet dass jedes Kind Element eines `ParticipantSet`, diese Nachricht absenden kann. Das *`receiver`* Attribut gibt an, welcher `Participant` diese Nachricht empfangen soll. Die beiden Attribute *`sendActivity`* und *`receiveActivity`* geben jeweils die Aktivität aus der *Participant Behavior Description* an, welche absendet respektive empfängt. Die Attribute *`bindSenderTo`* und *`participantRefs`* realisieren die sogenannte "link passing mobility" was bedeutet, dass über `MessageLinks` Teilnehmerreferenzen vom Sender zum Empfänger ausgetauscht werden können. Die Bekanntheit der Teilnehmer ist immer lokal für jeden Prozess. Daher muss es die Möglichkeit geben Referenzen auszutauschen, da sonst kein Antworten auf Nachrichten möglich wäre. So kann z. B. ein `Participant` **„a“**, der die

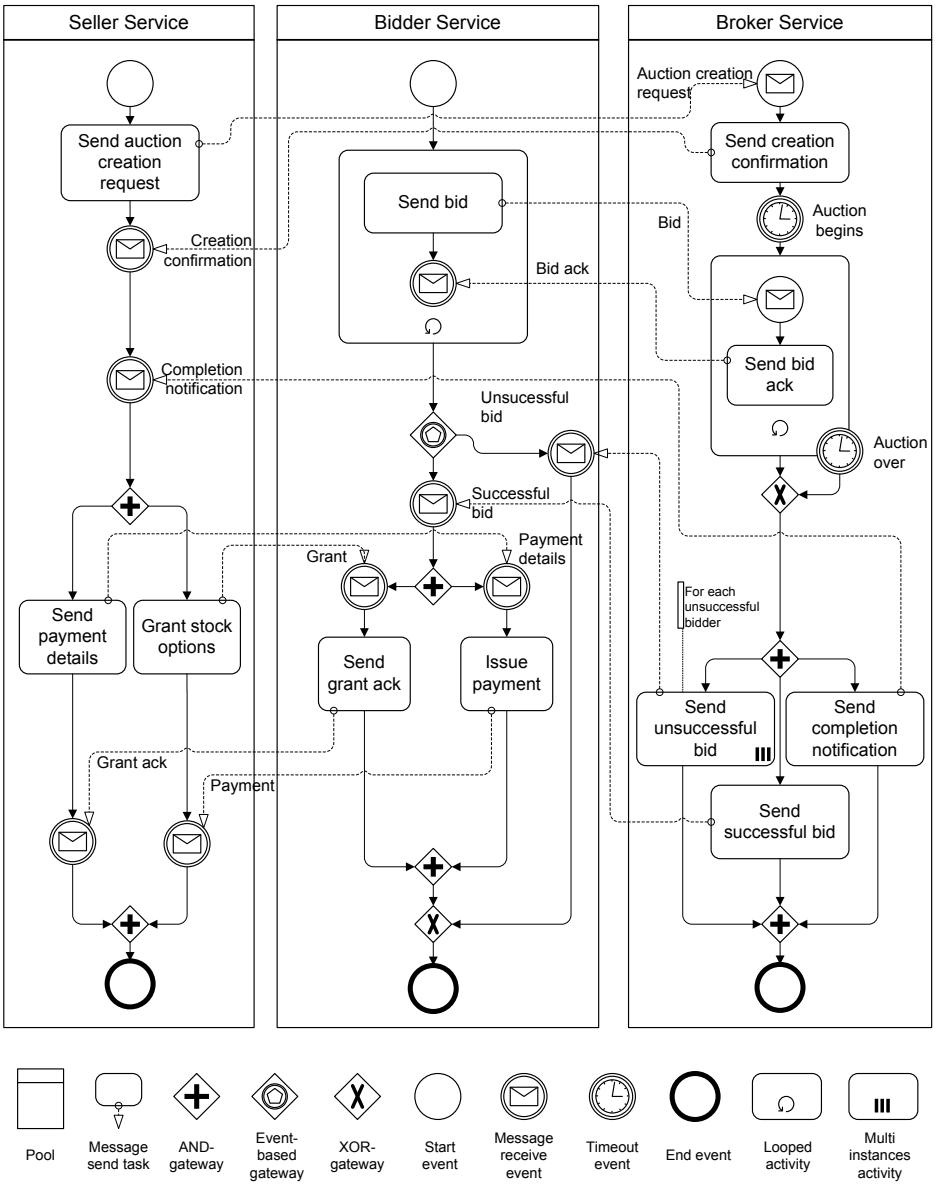

Abbildung 3.4: Auktionsszenario. Choreographie Beispiel modelliert als *interconnection model* in BPMN. Quelle: [DKLW09]

Listing 3.1 *Topology* Ausschnitt des Auktionsszenarios. Quelle: [DKLW09]

```
01 <topology name="topology" targetNamespace="urn:auction"
   xmlns:sns="urn:auction:seller" ...>
02    <participantTypes>
03       <participantType name="Seller"
   participantBehaviorDescription="sns:seller" />
04       <participantType name="BrokerService" ... />
05       <participantType name="Bidder" ... />
06    </participantTypes>
07    <participants>
08       <participant name="seller" type="Seller" selects="brokerService" />
09       <participant name="brokerService" type="BrokerService" />
10       <participantSet name="bidders" type="Bidder">
11          <participant name="bidder" selects="brokerService" />
12          <participant name="successfulBidder" />
13       </participantSet>
14       <participantSet name="unsuccessfulBidders" type="Bidder"
   forEach="as:notifyUnsuccesfulBidders">
15          <participant name="currentBidder"
   forEach="as:notifyUnsuccesfulBidders" />
16       </participantSet>
17    </participants>
18    ...
19 </topology>
```

Referenz von **„b“** kennt, diese an `Participant` **„c“** senden, indem das *`participantRefs`* Attribut auf **„b“** verweist. Jetzt kennt **„c“** die Referenz von **„b“** und kann direkt mit diesem `Participant` kommunizieren. Das *`bindSenderTo`* Attribut hat die selbe Funktion, nur dass hier der Absender die Selbstreferenz weiter gibt. Das Weitergeben von Referenzen (*`participantRefs`* und *`bindSenderTo`*) sowie das Selektieren (*`selects`*) führt zur Bindung der Teilnehmer an andere Teilnehmer.

In Abbildung 3.3 sehen wir, dass die Teilnehmer ein *`scope`* Attribut haben. Die Funktion davon wird in [Kop12] beschrieben. Es verweist auf eine ‹ `Scope` › Aktivität aus einer *Participant Behavior Description* und bedeutet, dass dieser Teilnehmer nur innerhalb des referenzierten ‹ `Scope` › bekannt ist. Ein Anwendungsbeispiel dafür ist in Listing 3.2 zu sehen. Eine Menge von Bietern **„bidders“** sendet eine Nachricht an einen Verkäufer **„s“**. Ein Teilnehmer aus der Bieter Menge ist **„b“**, welcher die ‹ `Scope` › Aktivität **„rcvScope“** referenziert. Über das *`binSenderTo`* Attribut, wird die Referenz des Bieters **„b“** an den Verkäufer **„s“** übertragen. In Listing 3.3 sehen wir einen Ausschnitt der *Participant Behavior Description* von Verkäufer **„s“**. Die Referenz des Bieters ist nur innerhalb der ‹ `Scope` › Aktivität **„rcvScope“** bekannt. Der Verkäufer bekommt von jedem Bieter die selbe Art von Nachricht, welche immer das jeweilige Gebot beinhaltet. Der Verkäufer muss also gleichartige Nachrichten von unterschiedlichen Absendern unterscheiden und abspeichern, da er später das höchste Gebot auswählen und den Höchstbieter benachrichtigen muss. Daher macht

Listing 3.2 `Participant` mit Scope. Beispiel aus [Kop12]

```
01 <participants>
02    <participant name="s" type="Seller" />
03    <participantSet name="bidders" type="Bidder">
04       <participant name="b" scope="rcvScope" containment="must-add" />
05    </participantSet>
06 </participants>
07 <messageLinks>
08    <messageLink senders="bidders" sendActivity="sendBid" bindSenderTo="b"
    receiver="s" receiveActivity="receiveBid" messageName="Bid" />
09 </messageLinks>
```

Listing 3.3 *Participant Behavior Description* Auszug des `Participant` **„s“**. Beispiel aus [Kop12]

```
01 <while>
02    <condition />
03    <scope name="rcvScope">
04       <pick><onMessage name="receiveBid" /> ... </pick>
05    </scope>
06 </while>
```

es Sinn, die Referenz auf den Absender und seine Nachricht nur innerhalb eines ‹ `Scope` › sichtbar zu machen, so dass die anderen Referenzen und Nachrichten nicht überschrieben werden.

3.3.3 Grounding

In diesem Dokument werden die Web Service spezifischen Details für *Topology* und *Participant Behavior Description* angegeben. Das *Grounding* ist notwendig, um aus *Topology* und *Participant Behavior Description* ausführbare BPEL Prozesse zu erzeugen. Wir werden darauf im nächsten Abschnitt genauer eingehen. In Abbildung 3.5 ist das Grounding Model anhand eines UML Modelles beschrieben, welches auf Basis des *Grounding* XSD Schemas [DK12a] entworfen wurde. Abbildung 3.6 zeigt die Zusammenhänge von Grounding Model, Topology Model und *Participant Behavior Description*. Dabei bedeutet der Stereotyp «becomes» an den einzelnen Kanten: „wird zugeordnet zu“. Jeder `MessageLink` aus *Topology*, muss einem `MessageLink` Eintrag im *Grounding* zugeordnet werden. `MessageLinks` repräsentieren das Versenden und Konsumieren einer Nachricht. Sie werden mit einem *`operation`* Attribut aus einer WSDL verknüpft. Dazu muss der gewünschte *`portType`* angegeben werden, welcher diese *`operation`* anbietet. Über das *`name`* Attribut, wird eine Verbindung zum entsprechenden `MessageLink` aus *Topology* hergestellt. Die Attribute *`senders`*, *`expectedPortType`*, *`expectedOperation`*, *`mediator`*, *`offeredPortType`* und *`offeredOperation`* sind zwar im Grounding Model definiert, werden aber bei der Um-

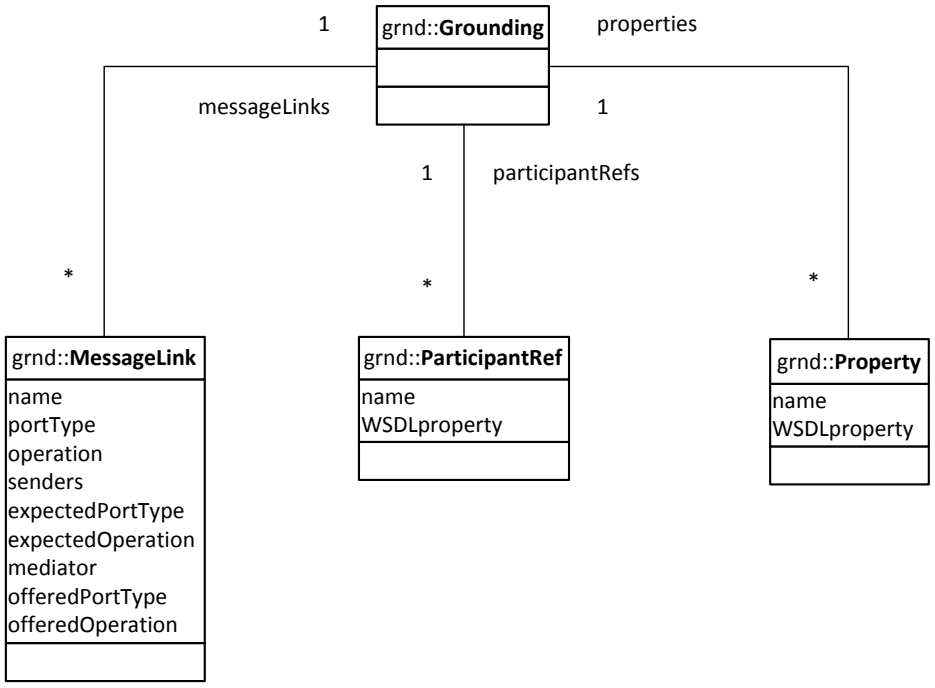

Abbildung 3.5: Grounding XSD ([DK12a]) dargestellt als UML Modell

wandlung zu ausführbaren BPEL Prozessen nicht benutzt. Sie gehören zum Thema "Message Mediation". Wie in [DKLW07] beschrieben, verknüpft das *Grounding* die in `MessageLink` definierten Sende- und Empfangsaktivität direkt mit WSDL *Port Types* und *Opeartions*. Diese Verknüpfung sollte dynamisch, wie in [Kop12] S.111 beschrieben, von einem Enterprise Service Bus erledigt werden, was bisher jedoch noch zur aktuellen Forschung gehört. Die `MessageLinks` Attribute *`bindSenderTo`* und *`participantRefs`* ("link passing mobility" siehe 3.3.2 auf Seite 18) werden jeweils einem `ParticipantRef` Eintrag im Grounding Model zugeordnet. Das *`name`* Attribut von `ParticipantRef` bezieht sich dabei auf das entsprechende Attribut des zugehörigen `Participant` aus *Topology*. Das *`WSDLProperty`* Attribut von `ParticipantRef` gibt an, wo sich ein bestimmtes Element in verschiedenen WSDL Messages befindet. Da ausführbare BPEL Prozesse Service Referenzen [OAS07b] über Messages austauschen, kann mit der Angabe des *`WSDLProperty`* Attributs die entsprechende Service Referenz, unabhängig vom Message Type, in der eingehenden Message gefunden werden. Die Werte des *`properties`* Attributs (eine Liste aus durch Leerzeichen getrennten Werten) von `CorrelationSet`, müssen ebenfalls mit einem *`WSDLProperty`* Attribut verbunden werden. Jedem Eintrag aus der *`properties`* Liste, wird im Grounding Model ein `Property` Element zugeordnet.

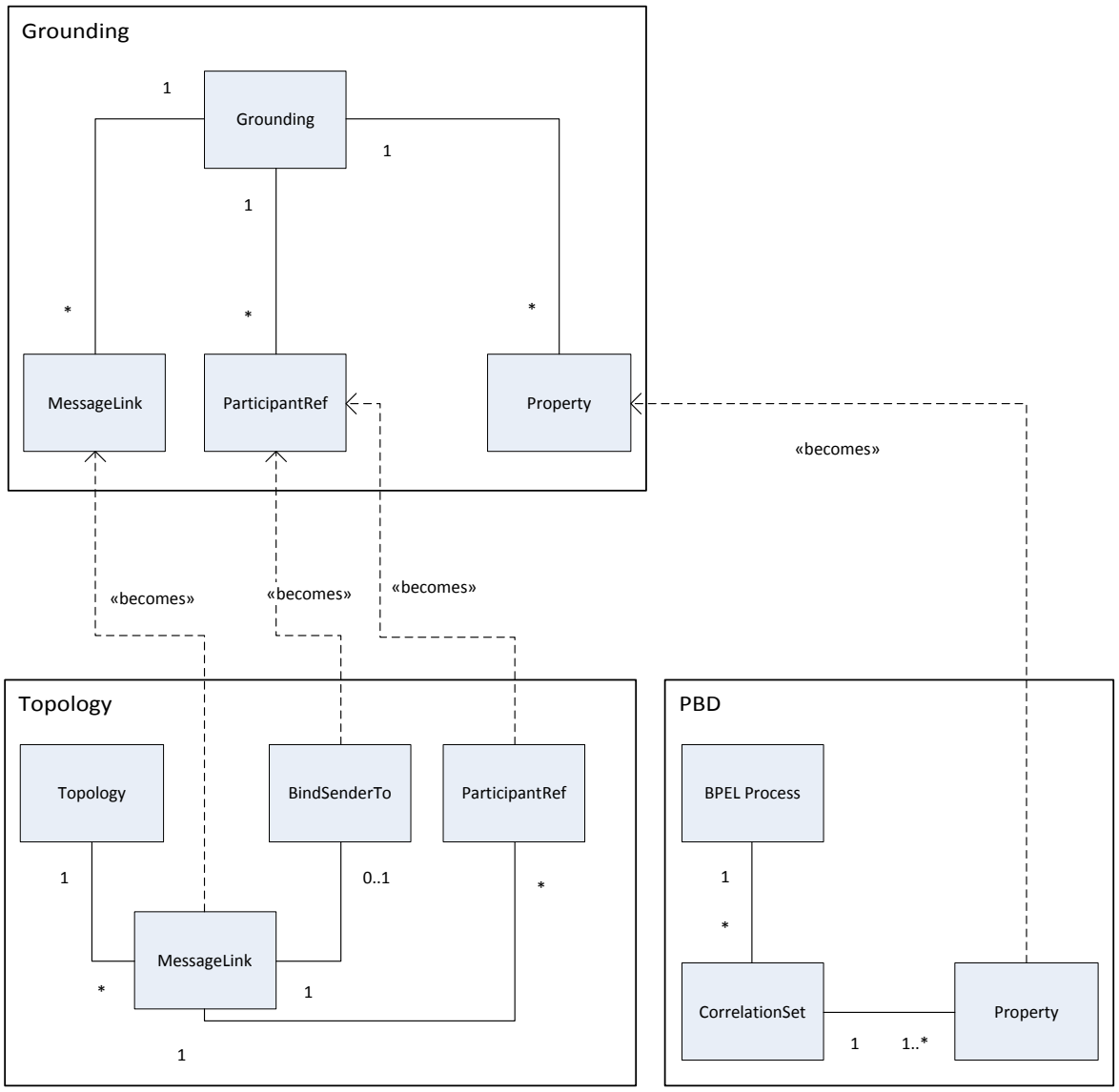

Abbildung 3.6: Das Grounding mit Verknüpfungen zu Topology und *Participant Behavior Description*

3.3.4 Von BPEL4Chor zu ausführbaren BPEL Prozessen

Choreographien an sich können nicht ausgeführt werden. Sie dienen als Vertrag zwischen Geschäftspartnern, wie diese miteinander interagieren um ihr Geschäftsziel zu erreichen. Einen Choreographie ist eine Vorlage, aus welcher ausführbare Geschäftsprozesse abgeleitet werden können. Wenn wir eine BPEL4Chor Choreographie zu ausführbaren BPEL Prozessen transformieren wollen, müssen wir die in Abbildung 3.7 dargestellten Schritte durchführen. Haben wir *Topology* und alle nötigen *Participant Behavior Description* Dokumente spezifiziert, müssen wir ein *Grounding* angeben. Dabei kann von zwei unterschiedlichen Situationen ausgegangen werden. Entweder gibt es ein einziges *Grounding*, in welchem Fall sich alle Partner auf die selben WSDL *Port Types* und *Operations* geeinigt haben oder, es gibt mehrere *Grounding* Dokumente, wenn bestimmte Partner andere WSDL Konfigurationen benötigen. Mit den Informationen aus *Topology* und *Grounding*, können die *Participant Behavior Description* Dokumente zu abstrakten BPEL Prozessen, welche zum *Abstract Process Profile for Observable Behavior* Profile konform sind, automatisch generiert werden. Dieser Transformationsprozess besteht aus vier wesentlichen Schritten. Es müssen *Partner Link Types* und *Partner Links* generiert werden. Sie geben an, welche *Port Types* respektive zum Senden und Empfangen einer Nachricht verwendet werden. Das *Abstract Process Profile for Observable Behavior* Profil fordert diese Angaben. Da die beiden Konstrukte nun bekannt sind, müssen für die Kommunikationsaktivitäten die Attribute `partnerLink`, `portType` und `operation` anhand der im *Grounding* angegebenen Details gesetzt werden. Über *Message Links* können Teilnehmerreferenzen ausgetauscht werden. BPEL kennt keine Teilnehmerreferenzen sondern Service Referenzen. Wie wir in Abbildung 3.6 sehen, wird jeder Teilnehmerreferenz eine *WSDL Property* zugeordnet. Diese *Property* gibt an, in welchem Teil einer Nachricht die passende Service Referenz zu finden ist. Diese Referenz muss in den entsprechenden *Partner Link* kopiert werden. `ParticipantSets` sind Mengen von Teilnehmern. BPEL kennt dieses Konstrukt nicht, weshalb eine Menge von Service Referenzen eingeführt werden muss. Das WS-BPEL Schema für Service Referenzen [OAS07b] spezifiziert allerdings keine Menge von Referenzen, weshalb diese als Sequenz von Referenzen zusätzlich deklariert werden muss. Des weiteren kennt BPEL keine Iteration über `ParticipantSets`, sondern nur über einen Zahlenwert. Daher müssen die betroffenen ‹ `ForEach` › Aktivitäten so umgeschrieben werden, wie in Listing 3.4 zu sehen ist. Die ‹ `ForEach` › Aktivität iteriert über eine `set` Variable, zu sehen in Zeile **3**, welche alle Service Referenzen des ursprünglichen Sets hält. In den Zeilen **9 - 16** wurde eine neue ‹ `Assign` › Aktivität eingefügt, welche den aktuellen Wert aus der `set` Variablen, in den verwendeten *Partner Link* kopiert. Die, nach diesem Transformationsprozess, entstandenen abstrakten BPEL Prozesse müssen nun mit den "executable completion" Regeln des *Abstract Process Profile for Observable Behavior* Profils, zu ausführbaren Prozessen angereichert werden. Dies geschieht in einem manuellen Verfahren.

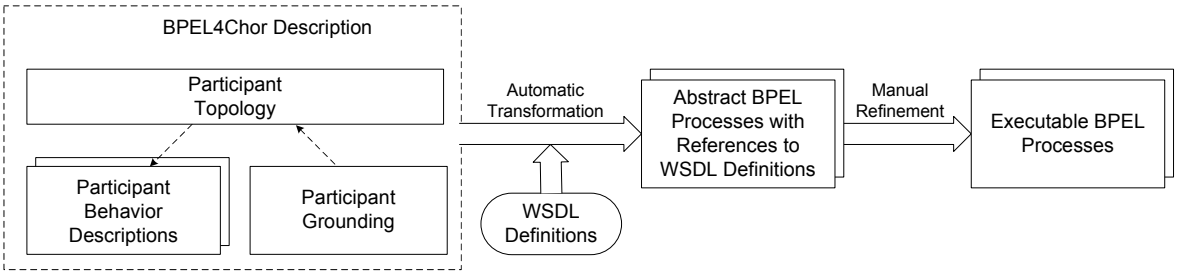

Abbildung 3.7: Von BPEL4Chor zu ausführbaren BPEL Prozessen [DKLW09]

Listing 3.4 Umgeschriebene ‹ `ForEach` › Aktivität, welche über `ParticipantSet` iteriert. Quelle basiert auf: [Rei07]

```
01 <forEach wsu_id="forEach1" counterName="i_forEach1">
02    <startCounterValue>0</startCounterValue>
03    <finalCounterValue>count($set/)1</finalCounterValue>
04    <scope>
05       <partnerLinks>
06          <partnerLink name="xy" .../>
07       </partnerLinks>
08       <sequence>
09          <assign>
10             <copy>
11                <from variable="set">
12                   <query>[$i_forEach1]</query>
13                </from>
14                <to partnerLink="xy" />
15             </copy>
16          </assign>
17          ...
18       </sequence>
19    </scope>
20 </forEach>
```

3.4 Model View Controller

Model View Controller (MVC) ist, wie in [LL07] beschrieben, ein Architekturmuster zur Trennung von Interaktion und Funktion. Es gliedert die danach realisierte Anwendung in drei Komponenten. Die *Model* Komponente realisiert die fachliche Funktion einer Anwendung. Sie kapselt Daten und stellt Methoden zur deren Manipulation zur Verfügung. Die *View* Komponente realisiert die grafische Repräsentation der Daten. Es kann mehrere verschiedene *View* Komponenten für die selben Daten geben. Die *Controller* Komponente ist einer *View* zugeordnet. Sie nimmt Benutzereingaben entgegen und veranlasst die nötigen Änderungen. Hat die Aktion eines Benutzer z. B. zur Folge, dass eine *View* aktualisiert werden muss,

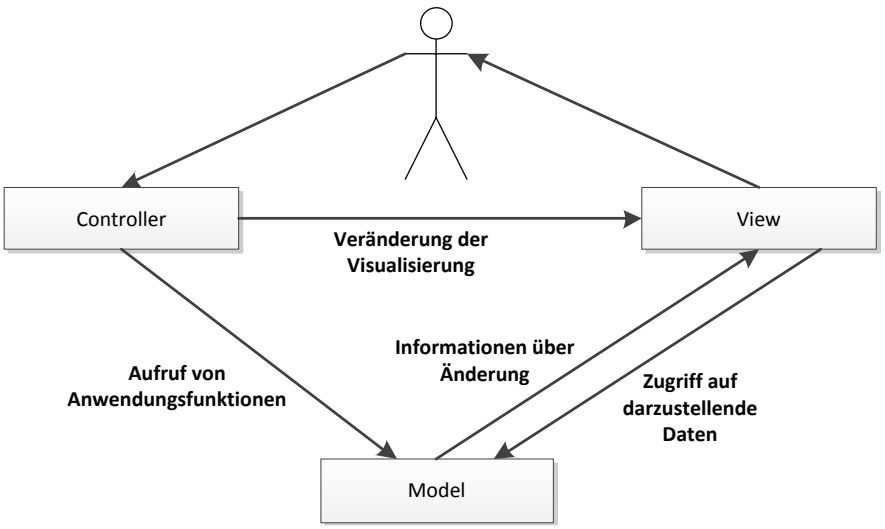

Abbildung 3.8: *Model View Controller* Architekturmuster und das Zusammenspiel seiner Komponenten. Quelle: [LL07]

stößt der zugeordnete *Controller* die Aktualisierung der grafischen Repräsentation an. Führt der Benutzer eine Aktion aus, welche eine Änderung der Daten zu Folge hat, so ruft der *Controller* die Manipulationsmethoden des *Models* auf. In Abbildung 3.8 ist die Interaktion der drei Komponenten zu sehen. Die Aktionen des Benutzers welche Daten des *Models* verändern, haben auch eine Veränderung der *View* zur Folge. Daher müssen sich alle *Views*, welche das *Model* repräsentieren, bei diesem anmelden. Das *Model* führt ein Register und benachrichtigt alle registrierten *Views* bei Änderung seines Zustandes. Dieser Ablauf wird als *Change-update* Mechanismus bezeichnet und bietet den Vorteil, das alle *Views* immer das aktuelle *Model* repräsentieren. Da es auch für das selbe *Model*, mehrere *View - Controller* Kombinationen geben kann, wird eine saubere Entkopplung der Komponenten erreicht. So können *View - Controller* Kombinationen sogar zu Laufzeit ausgetauscht werden. Ein Nachteil dieser Architektur ist, wenn sich das *Model* innerhalb sehr kurzer Zeitintervalle verändert. Dann kommt die Aktualisierung der *View* eventuell nicht mehr hinterher, da bei jeder Änderung die Daten vom *Model* erneut angefragt werden müssen.

4 Konzeption

Das Ziel dieser Arbeit ist, ein Modellierungswerkzeug für Choreographien zu entwickeln. Dabei soll es auch möglich sein, diese Choreographien in einer Workflowumgebung auszuführen. Um dies zu erreichen, müssen aus Choreographien zuerst lauffähige Prozesse generiert werden. Als Datenmodell haben wir BPEL4Chor gewählt und damit auch abstrakte BPEL Prozesse als Beschreibung des Verhaltens der Choreographie Teilnehmer. Wir haben bereits in Abschnitt 3.3.4 auf Seite 26 gesehen, dass die automatische Generierung von ausführbaren BPEL Prozessen sehr schwierig ist bzw. als manuell durchzuführender Schritt angegeben wird. Daher sehen wir für diese Arbeit davon ab, Algorithmen zu entwickeln die vollständig spezifizierte und deploy fähige BPEL Prozesse erzeugen. Auch sehen wir davon ab, für BPEL4Chor Choreographien die *Participant Behavior Description* als ausführbaren Prozess zu modellieren da sonst das Grounding Konzept verworfen bzw. übergangen wird. Daher entscheiden wir uns für den Weg eine Choreographie zu modellieren, daraus BPEL4Chor Artefakte zu exportieren um damit schließlich BPEL Prozesse zu generieren. Da unser Ziel ausführbare BPEL Prozesse sind und wir von *Participant Behavior Description* Dokumenten ausgehen, erzeugen wir zunächst mit Hilfe der Komponente *BPEL4ChorToBPEL* (siehe Abschnitt 5.1.6 auf Seite 105), abstrakte BPEL Prozesse welche zum *Abstract Process Profile for Observable Behavior* konform sind und modifizieren diese an einigen Stellen um ausführbare Prozesse zu erhalten. Diese Modifikationen werden im BPEL Standard [OAS07c] als „basic executable completion“ bezeichnet und bestehen aus folgenden Schritten:

- Der Namespace des BPEL Prozesses ändert sich von "abstract"[1] nach "executable"[2]
- Das `abstractProcessProfile` Attribut vom `Process` Element fällt weg.
- Alle Opaque Elemente, außer diese welche implizit weggelassen wurden, müssen durch ein ausführbares Element ersetzt werden.
- Sollte es keine Start Aktivität geben (mit Attribut `createInstance` = "yes"), muss eine geeignete hinzugefügt werden.
- `PartnerLink`, `Variable` und `Import` Elemente müssen zum `Process` Element hinzugefügt werden.

[1] "http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
[2] "http://docs.oasis-open.org/wsbpel/2.0/process/executable"

Um diese Schritte zu realisieren, benutzen wir die bereits erwähnte Komponente *BPEL4ChorToBPEL* sowie die neu entworfene *BasicExecutableCompletionTransformer* (siehe Abschnitt 4.5.3 auf Seite 65). In Tabelle 4.1 sind die Änderungsschritte und dazu die jeweilige Komponente, welche diese durchführt, angegeben.

Element	Änderung	Komponente
`PartnerLink`	Neue `PartnerLink` Elemente für ‹ `invoke` ›, ‹ `receive` ›, ‹ `reply` › und ‹ `onMessage` ›	*BPEL4ChorToBPEL*
`Variable`	Neue `Variable` Elemente für jedes ‹ `forEach` › über ein `ParticipantSet`	*BPEL4ChorToBPEL*
abstract Namespace	executable Namespace	*BECT*[a]
`abstractProcessProfile`	Entfernt	*BECT*
‹ `opaqueActivity` ›	Ersetzt durch ‹ `empty` ›	*BECT*
`Variable`	Neue `Variable` Elemente für jede Opaque Variable in ‹ `invoke` ›, ‹ `receive` ›, ‹ `reply` ›	*BECT*
Wert der *`inputVariable`* und *`outputVariable`* von ‹ `invoke` ›	"*`portType_operation`*_message"	*BECT*
Wert der *`variable`* von ‹ `receive` › und ‹ `reply` ›	"*`portType_operation`*_message"	*BECT*
Wert der *`variable`* von ‹ `onMessage` ›	"*`portType_operation`*_message"	*BECT*

Tabelle 4.1: Veränderung von Elementen durch die „basic executable completion“

[a] *BasicExecutableCompletionTransformer*

Die Ersetzung der Opaque Variablen durch "*`portType_operation`*_message" resultiert aus der Überlegung, da durch ein Grounding *`portType`* und die verwendete *`operation`* im jeweiligen `MessageLink` Element angegeben werden, die ausgetauschte Nachricht, welche in der Variablen transportiert wird, abgeleitet werden kann. Durch die *BPEL4ChorToBPEL* Komponente erhalten wir das für die Kommunikationsaktivität verwendete `PartnerLink` Element, welches auf den verwendeten *`portType`* verweist. Daher setzen wir den Namen der Variablen aus den Werten der *`portType`* und *`operation`* Attribute zusammen, da wir genau wissen dass die Kommunizierenden Teilnehmer den angegebenen `PartnerLink` benutzen. Den Typ der Variablen abzuleiten ist schwierig, da wir die Intention des Modellierers nicht kennen. Daher wird der Typ nicht gesetzt. Was wir nicht durchführen können ist

die Ersetzung von ‹ OpaqueFrom ›, welches in der ‹ Copy › Anweisung eingesetzt werden kann, da wir ebenfalls nicht wissen, welche ausführbare Variante von ‹ From › der Modellierer vorsieht. Das ‹ FromPart › Konstrukt in ‹ Receive ›, ‹ Invoke › , ‹ OnMessage › und ‹ OnEvent › kann ebenfalls Opaque Variablen benutzen. Wir können auch diese Werte nicht bestimmen, da wir nicht Wissen, wie die eingehende Multi-Part Message aussieht und in welchem *part* der Message die Information Steht, welche in die Variable *toVariable* kopiert werden soll. Das selbe gilt analog auch für das ‹ ToPart › Konstrukt in ‹ Reply › und ‹ Invoke ›.

In den folgenden vier Abschnitten stellen wir Ansätze vor, wie sich der Choreographie Editor realisieren lassen könnte. Wir werden uns für den Vierten Ansatz entscheiden und stellen das Konzept ausführlich in Abschnitt 4.5 auf Seite 34 vor.

4.1 Erster Ansatz

Eines der bestehenden Workflow Modellierungstools, welches in SimTech[3] eingesetzt wird, ist der *BPEL Designer*[4]. Der *BPEL Designer* ist ein grafischer Editor zu Modellierung von BPEL Prozessen und wurde als Eclipse Plugin (siehe 5.1.1 auf Seite 83) realisiert. Als grundlegendes Modell dient der WS-BPEL Standard, welcher mittels *Eclipse Modeling Framework* (siehe 5.1.2 auf Seite 86) modelliert wurde. Die grafische Repräsentation des Modells, ist mit dem *Graphical Editing Framework* (siehe 5.1.3 auf Seite 92) realisiert worden. Wie in [Vuko9] beschrieben, ist der *BPEL Designer* für die SimTech spezifische Anforderung, Simulations Workflows zu erstellen, so erweitert worden, das er den Modellierer über den gesamten Lebenszyklus des Workflows hinweg unterstützt. Das bedeutet dass der Modellierer zuerst seinen Prozess erstellt, dabei auf simulationsspezifische Aktivitäten aus einem Katalog zurück greift, den Prozess ausführt, überwacht und schließlich die angezeigten Ergebnisse auswerten kann.

In Abbildung 4.1 sehen wir die Oberfläche des *BPEL Designers*. Er zeigt eine der Prozessvorlagen, welche benutzt werden kann um eine Simulation zu modellieren. Hier ist auch zu sehen, dass der *BPEL Designer* zur Modellierung eines einzelnen BPEL Prozesses ausgelegt ist. Die Zeichenfläche repräsentiert das Wurzelelement ‹ Process › eines BPEL Prozesses. Alle weiteren grafischen Elemente Repräsentationen der zum Prozess gehörenden BPEL Konstrukte. In dieser Abbildung ist eine ‹ Sequence › ("main") zu sehen und die darin enthaltene *extension* Aktivität ‹ SimulationStartActivity › ("simulationStart"). Da wir Choreographien modellieren wollen, müssten wir den *BPEL Designer* so erweitern, dass mehrere Prozesse auf der Zeichenfläche platziert, sowie *Message Links* zwischen Kommunikationsaktivitäten gezogen werden könnten. Das Problem dabei ist, dass wir einen reinen Prozess Editor zu

[3] http://www.iaas.uni-stuttgart.de/forschung/projects/simtech/project_modeling.php
[4] http://www.eclipse.org/bpel/

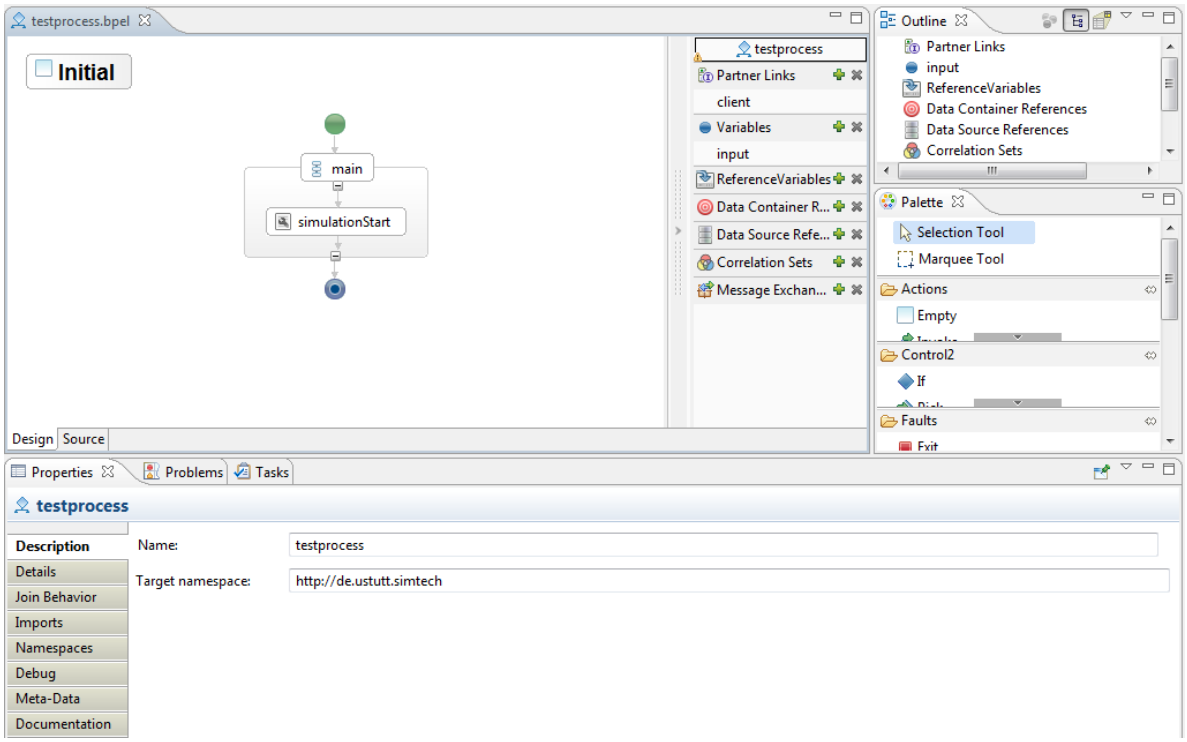

Abbildung 4.1: SimTech *BPEL Designer* mit leerer Prozessvorlage

einem Choreographie Editor umbauen müssten. Wir würden eine Mischform kreieren, in welcher keine saubere Trennung mehr besteht. Dazu müssen wir auch bedenken, dass die *Participant Behavior Description* Dokumente abstrakte BPEL Prozesse sind. Wir laufen in die Gefahr, abstrakte Prozesse mit ausführbaren in einem Editor Fenster zu mischen. Der *BPEL Designer* Code ist sehr komplex und muss zuerst langwierig analysiert und verstanden werden. Das zugrunde liegende EMF Modell des WS-BPEL Standards ist ungeeignet, um Choreographien zu modellieren. Es muss mit BPEL4Chor erweitert bzw. verwoben werden. Den Vorteil des großen Funktionsumfangs zum modellieren von BPEL Prozessen und die SimTech spezifischen Erweiterungen behalten wir mit diesem Ansatz allerdings bei. Wir entscheiden uns dennoch aus den genannten Gründen gegen die Erweiterung des *BPEL Designer* zu einem *Chor Designer*

4.2 Zweiter Ansatz

Der zweite Ansatz ist dem ersten sehr ähnlich. Er unterscheidet sich aber in dem wesentlichen Punkt, dass wir den Code *BPEL Designers* kopieren und so abändern, dass ein reiner Choreographie Editor daraus resultiert. Wir hätten damit zwei verschiedene Editoren mit

teilweise gemeinsamer Code Basis. Der Vorteil daran wäre, dass der *BPEL Designer* weiterhin ein eigenständiges Tool zum Modellieren von BPEL Prozessen bleibt und der neue *Chor Designer* hingegen, ein eigenständiges Tool zum modellieren von Choreographien. Die Nachteile sind ebenfalls die Komplexität des vorhandenen Codes und Veränderung des vorhandenen EMF Modells. Zudem muss noch bedacht werden, dass der *BPEL Designer* weiter entwickelt wird, was eine unter Umständen aufwendige Code Migration für den neu erstellten *Chor Designer* zur Folge hat. Wir entscheiden uns, wegen der genannten Gründe, gegen diesen Ansatz.

4.3 Dritter Ansatz

Wir haben bereits die Tendenz zu einem separaten Editor aus den ersten beiden Ansätzen erörtert, da wir es als Vorteil ansehen, Choreographie Editor und Prozess Editor zu trennen. Wir erstellen von Grund auf einen neuen grafischen Editor, den *Chor Designer*, in welchem wir abstrakte BPEL Prozesse importieren können. Diese abstrakten Prozesse werden zuvor z. B. mit dem *BPEL Designer* oder einem anderen BPEL Tool modelliert. Im *Chor Designer* muss alles ausgeblendet oder angepasst werden, was nicht zur *Participant Behavior Description* konform ist. Es muss also für jeden importierten Prozess, eine Sicht auf den Quellprozess erstellt werden. Zwischen den Kommunikationsaktivitäten können dann *Message Links* gezogen werden. Das Problem, welches sich daraus ergibt ist, dass sich die importierten Quellprozesse ändern können. Diese Änderungen müssen im *Chor Designer* automatisch nachgezogen werden. Noch problematischer wäre es, wenn im *Chor Designer* die importierten Prozesse ebenfalls editierbar wären. Wir hätten damit ein Synchronisationsproblem in beide Richtungen. Im Modellierungsprozess neigt der Anwender dazu, schrittweise die Choreographie zu erarbeiten. Mit diesem Ansatz müsste der Modellierer immer zwischen den zwei Editoren hin und her Wechseln. Aus diesen Gründen entscheiden wir uns gegen diesen Ansatz.

4.4 Vierter Ansatz

Wir erstellen einen neuen grafischen Editor, der speziell zum modellieren von Choreographien entworfen wird. Als Basis wird das BPEL4Chor Modell verwendet und ausgehen davon ein Modell für den Editor konzipiert. Dabei wird der Editor so aufgebaut, dass die Choreographie für den Anwender von Grund auf modellierbar ist. Das bedeutet, er kann die Teilnehmer definieren, deren Verhalten modellieren und *Message Links* ziehen. Die Modellierung des Verhaltens wird dabei nur auf die BPEL Konstrukte eingeschränkt, welche zum *Abstract Process Profile for Participant Behavior Descriptions* konform sind. Der Editor kann einerseits BPEL4Chor Artefakte exportieren und andererseits, nach Durchführung

einer Transformation, BPEL Prozesse erzeugen, welche dann mit anderen BPEL Tools weiter bearbeitet werden können. Ein Nachteil dieses Ansatzes ist, dass die umfangreichen Modellierungmöglichkeiten des *BPEL Designers* nicht in dieser Arbeit reproduziert werden können. Wir erstellen diesen Editor ebenfalls als Eclipse Plugin, damit er zusammen mit dem *BPEL Designer* in der selben Eclipse *Workbench* (siehe 5.1.1 auf Seite 86) ausgeführt werden kann. Dadurch können beide Editoren in einer gemeinsamen Umgebung benutzt werden. Aufgrund der überwiegenden Vorteile entscheiden wir uns für diesen Ansatz und erläutern die Konzeption des Editors im folgenden Abschnitt.

4.5 Choreographie Editor

Bei der Entwicklung eines grafischen Editors für Choreographien, stellt sich die Frage nach dem Modell. Wenn wir das BPEL4Chor Modell benutzen funktioniert dies einerseits nicht, da das Modell nicht exakt dafür geeignet ist, alle grafischen Elemente einer Choreographie darzustellen. Warum dies so ist, werden wir im folgenden sehen. Andererseits wäre es nicht sinnvoll den Editor so zu konzipieren, dass er eine exakte grafische Repräsentation der BPEL4Chor Artefakte anzeigt, da dazu ein XML Editor genügt. Außerdem bleibt das Editor Modell somit eigenständig und wir erhalten uns die Möglichkeit, das Modell in ein beliebiges anderes Modell zu transformieren. Allerdings eignet sich BPEL4Chor gut um Choreographien zu beschreiben. Daher nehmen wir Ideen aus BPEL4Chor und bauen so ein geeignetes Modell für den Editor auf. Dieses neue Modell nennen wir im folgenden Chor Model. Zusätzlich erstellen wir Komponenten zur Transformation vom Chor Model zu BPEL4Chor. Dies ist beliebig für andere Modelle erweiterbar, da diese Komponenten nur lose an den Editor gekoppelt sein sollen.

Bevor wir das Chor Model entwickeln, müssen wir zuerst feststellen, welche grafischen Elemente für die Modellierung von Choreographien benötigt werden. Wir betrachten dazu Abbildung 4.2, in welcher eine Beispiel Choreographie in BPMN modelliert wurde. Das Beispiel stammt aus [DKLW07]. Ein Reisender möchte ein Flug buchen und gibt dazu eine Bestellung bei der Reiseagentur auf. Die Agentur nimmt die Bestellung entgegen und stellt bei mehreren Fluglinien Preisanfragen. Die Fluglinien, welche ein Angebot abgeben möchten, senden ihren Preis der Agentur zu. Hat die Agentur alle Preise eingesammelt, bestimmt diese den besten Preis, bestellt bei der entsprechenden Fluglinie ein Ticket und überträgt zusätzlich die Kontaktadresse des Reisenden. Die ausgewählte Fluglinie kann darauf das Flugticket dem Reisenden zukommen lassen, alle andere Fluglinien warten bis der Timeout erreicht wird und brechen ab. Die Agentur sendet dem Reisenden schließlich noch den Reiseplan.

Die Grafischen Elemente, welche wir Anhand von Abbildung 4.2 identifizieren können sind, für jeden Teilnehmer der Choreographie ein Rechteck. Darin befinden sich die Aktivitäten,

welche das Verhalten definieren. Nachrichtenaustausch wird mit Verbindungspfeilen dargestellt. Wenn wir diese Abbildung mit BPEL4Chor Artefakten beschreiben, dann entspricht das, was wir hier sehen, dem *Topology* Dokument mit *Participant*, *ParticipatSet*, *Message Links*, sowie den Inhalt der *Participant Behavior Description* aller Teilnehmer. Die *Participant Types* sind implizit zu sehen. Jedes äußerste Rechteck entspricht einem Typ und gleichzeitig auch dem Teilnehmer. Würde es z. B. zwei Agenturen als Teilnehmer geben und würden diese Nachrichten untereinander austauschen, wären auch zwei Rechtecke mit dem selbem Inhalt vorhanden. In [Ste07] wurde eine BPEL4Chor Beschreibung des BPMN Modells aus Abbildung 4.2 generiert. Wir nehmen diese BPEL4Chor Artefakte als Grundlage und entwickeln, zusammen mit dem BPMN Diagramm, ein grafisches Konzept dieser Choreographie. Das Resultat ist in Abbildung 4.3 zu sehen.

4.5.1 Entwicklung des Chor Models

Alle grafischen Elemente, welche wir im Editor platzieren wollen, realisieren wir im Chor Model als eine eigene Entität bzw. Klasse. Diese Entscheidung müssen wir schon an dieser Stelle treffen, da wir den Editor mittels GMF (siehe 5.1.4 auf Seite 95) realisieren werden. Dessen Tooling Komponente (siehe 5.1.4 auf Seite 98) realisiert ein Mapping von grafischen Elementen auf Modell Klassen. Die Tooling Komponente basiert auf dem Konzept dass es im Datenmodell des Editors ein Wurzelelement gibt, welches die Zeichenfläche repräsentiert. Dieses Wurzelelement muss alle Entitäten referenzieren, welche als grafische Repräsentation auf dieser Zeichenfläche platziert werden. Die platzierbaren Entitäten sind Teilnehmer (Participant und ParticipantSet), Nachrichtenlinks (Message Links), Teilnehmer Referenzen, welche über die Nachrichtenlinks versendet werden (Paticipant Referenzen) und BPEL Aktivitäten. Dazu kommen noch die Links zwischen Aktivitäten innerhalb eines ‹ `Flow` › Containers. Die BPEL Aktivitäten der *Participant Behavior Description* realisieren wir in einem vom Chor Model separaten PBD Model. Die Teilnehmerklassen des Chor Model, referenzieren die Prozess Klasse des PBD Model. Wir verhindert damit eine komplette Integration des PBD Model ins Chor Model. Umgekehrte Referenzen von PBD Model auf Chor Model möchten wir vermeiden, da sonst die beiden Modelle in gegenseitiger Abhängigkeit stehen. Es müssten, bei der Transformation (siehe Abschnitt 4.5.2 auf Seite 48) vom PBD Model, Ausnahmebehandlungen eingebaut werden, damit referenzierte Elemente aus dem Chor Model nicht Teil der XML Serialisierung des PBD Model werden. Als Beispiel wäre eine Referenz von `ForEach` auf `ParticipantSet` denkbar um die Iteration über dieses Set zu modellieren. Genau dies möchten wir vermeiden und geben im Verlauf des folgenden Abschnitts, eine andere Lösung dafür an.

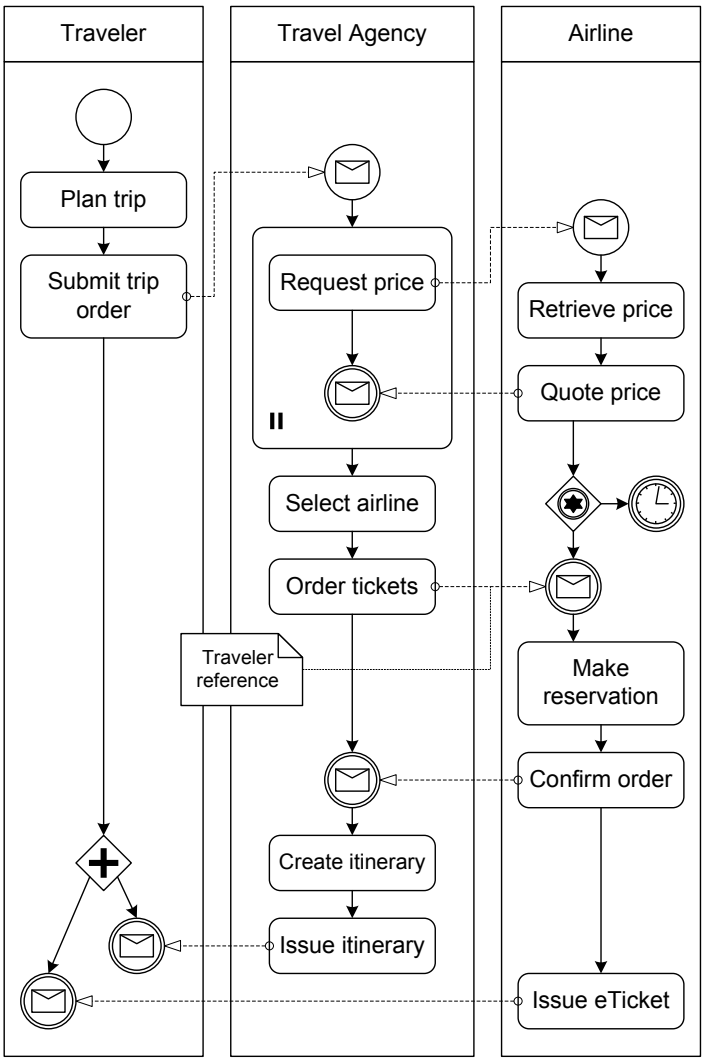

Abbildung 4.2: BPMN Choreographie „Buchung eines Flugtickets“. Quelle: [DKLW07]

Participants

Zunächst schauen wir uns das Topology Model an, welches in Abschnitt 3.3.2 auf Seite 18 erläutert und in Abbildung 3.3 dargestellt wurde. Hier werden mehrere `ParticipantType` Elemente definiert. Diese wiederum verweisen auf die in einem anderen Dokument definierte *Participant Behavior Description*. Im folgenden reden wir oft über `Participant` und `ParticipantSet` Elemente, daher bezeichnen wir beide als *participant(s)*. Ein *participant* hat genau einen `ParticipantType`. Allerdings können mehrere *participants* auf den selben `ParticipantType` verweisen. Würden wir also den Ansatz verfolgen, für jeden `ParticipantType` ein grafisches Element darzustellen, liefen wir in ein Problem

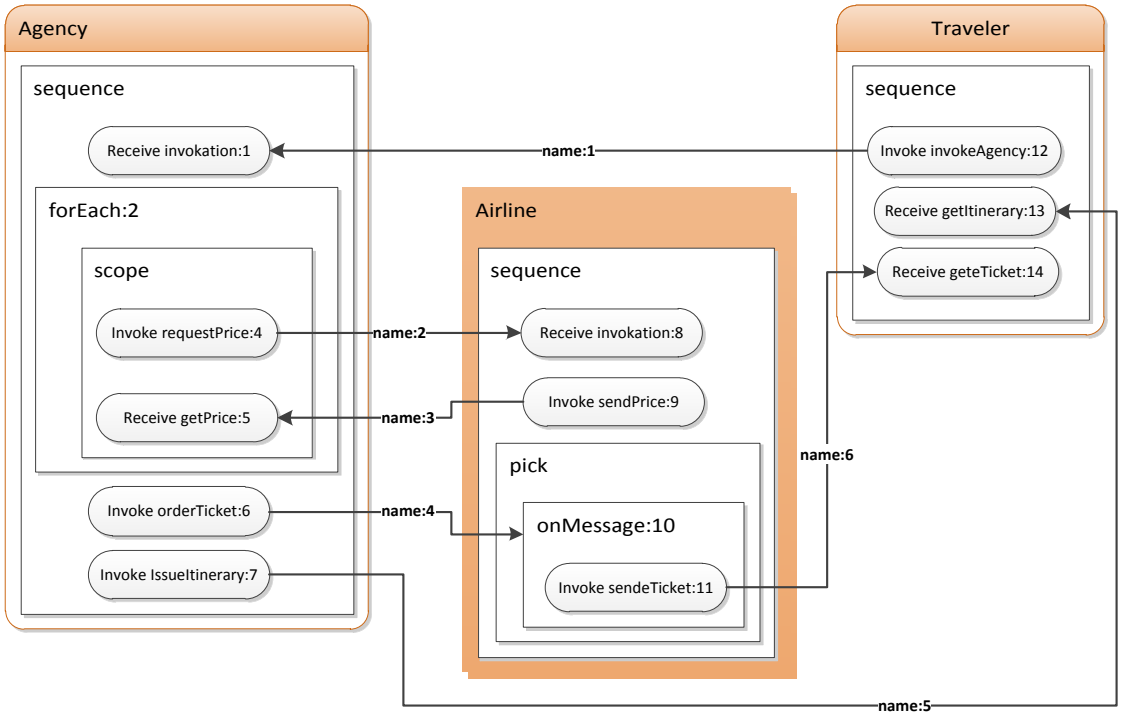

Abbildung 4.3: Choreographie „Buchung eines Flugtickets“. Konzeptionelles Modell

wenn mehrere *participants* des gleichen Typs miteinander kommunizieren. Dann hätten die `MessageLinks` als Quelle und Ziel das selbe grafische Element, was für die Darstellung einer Kommunikation zwischen mehreren Partnern ungeeignet wäre. Daher wählen wir für alle *participants* eine eigenes grafisches Element.

Jeder *participant* hat ein Verhalten, welches in der zugehörigen *Participant Behavior Description* beschrieben ist. Dieses Dokument wird durch den `ParticipantType` referenziert. Wir möchten die Möglichkeit haben, das Verhalten im selben Editor zu modellieren. Daher müssen die Elemente des PBD Model an einen sinnvollen Ort im Editor platziert werden können und zwar auch so, dass sich die Kommunikationsaktivitäten gut durch Pfeile, welche die `MessageLinks` repräsentieren, verbinden lassen. Da wir aber `ParticipantType` nicht grafisch darstellen, muss jeder *participant* direkt auf seine *Participant Behavior Description* verweisen. Im Chor Model verweisen wir auf das `Process` Element des PBD Model. Die Aktivitäten platzieren wir innerhalb der grafischen Darstellung der *participants* um auszudrücken, dass dieser *participant* das modellierte Verhalten hat. Dies führt allerdings dazu, dass wir eine Konvention bezüglich der Typisierung treffen müssen. Wir entscheiden uns dafür, für jeden *participant* automatisch eine neue *Participant Behavior Description* mit anzulegen. Selbst, wenn mehrere *participants* das selbe Verhalten haben. Wir könnten dies auch

anders realisieren, indem wir die Typisierung beibehalten. So würde z. B. beim Erstellen eines *participants A*, ein neuer Typ T_A mit angelegt werden. Für diesem Typ würden wir das Verhalten mittels BPEL Aktivitäten modellieren. Doch laufen wir dann in ein Problem, wenn wir einen weiteren *participant B* des selben Typs T_A erstellen wollen. Wir müssten zuerst eine Auswahl des Typs anbieten, bevor wir *B* im Editor platzieren können. Angenommen, wir hätten diese Auswahl. Fügen wir dann z. B. *B* eine Aktivität hinzu, ändert sich T_A was zur Folge hat, dass sich auch das Verhalten von *A* mit ändert. Der Editor muss diese Änderungen für alle *participants* des selben Typs synchronisieren. Wir erörtern die Typisierung für diese Arbeit nicht weiter aber stellen die Betrachtung dieser Möglichkeit in Aussicht für eine zukünftige Erweiterung. In Abbildung 4.4 ist zu sehen, wie wir *participants* im Chor Model darstellen. Zwecks besserer Abgrenzung zum Topology Model, nennen wir die *participants* im Chor Model `CParticipant` und `CParticipantSet`.

Es ist wichtig, dass wir `Participant` und `ParticipantSet` Elemente grafisch unterschiedlich darstellen, da das `ParticipantSet` eine Menge von Teilnehmern angibt, welche von ihrer Anzahl zur Designzeit nicht bekannt sind. `MessageLink` Elemente von `Participant` zu `ParticipantSet` stellen eine $1 : n$ Kommunikation dar, was ebenfalls grafisch angedeutet werden sollte. Was beide Elemente gemeinsam haben ist das *`name`* Attribut sowie eine Referenz auf `ParticipantType`. Die `ParticipantSet` Elemente können Kind Elemente haben welche wiederum entweder `Participant` oder `ParticipantSet` Elemente sind. Hier ist also eine Verschachtlung möglich. Von einer grafischen Darstellung der Kind Elemente eines `ParticipantSet` sehen wir ab, dennoch muss es eine Möglichkeit geben sie zu modellieren, da diese Elemente Quelle oder Ziel eines `MessageLink` sein können. Das Topology Model sieht außerdem die Möglichkeit zur Restriktion der *participants* auf eine ‹ `Scope` › Aktivität vor, was beide Elemente ebenfalls gemeinsam haben. Für alle Gemeinsamkeiten führen wir, wie in Abbildung 4.5 zu sehen, das neue Element `CParticipantCommon` ein, von welchem `CParticipant` sowie `CParticipantSet` erben. Das *`containment`* Attribut von `Participant` wird als `PContainment` Datentyp ins Chor Model übernommen. `Participant` Elemente haben zudem noch ein *`selects`* Attribut, welches `Participant` oder `ParticipantSet` Elemente referenzieren kann. Im Chor Model referenzieren wir damit `CParticipantCommon` Elemente.

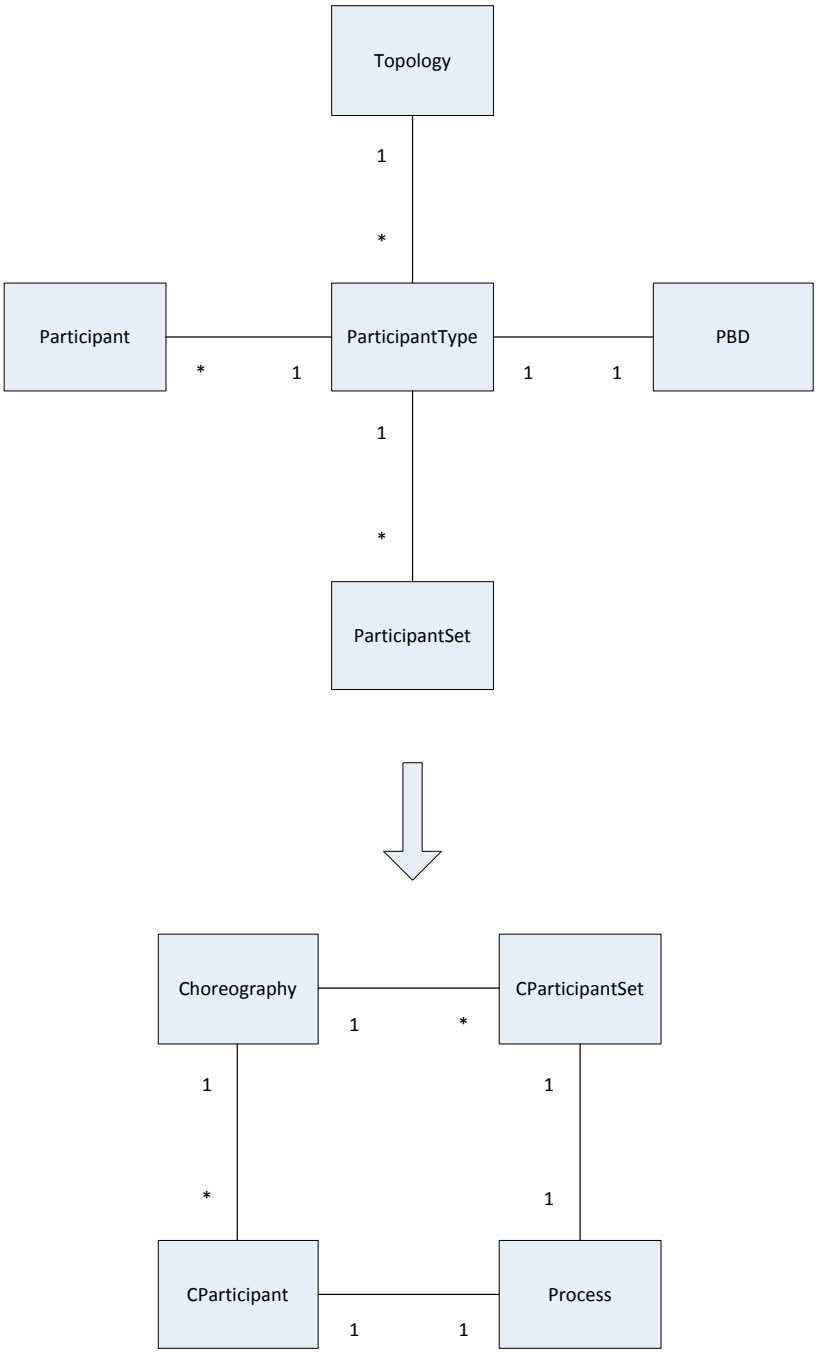

Abbildung 4.4: Vom Topology Model zum Chor Model

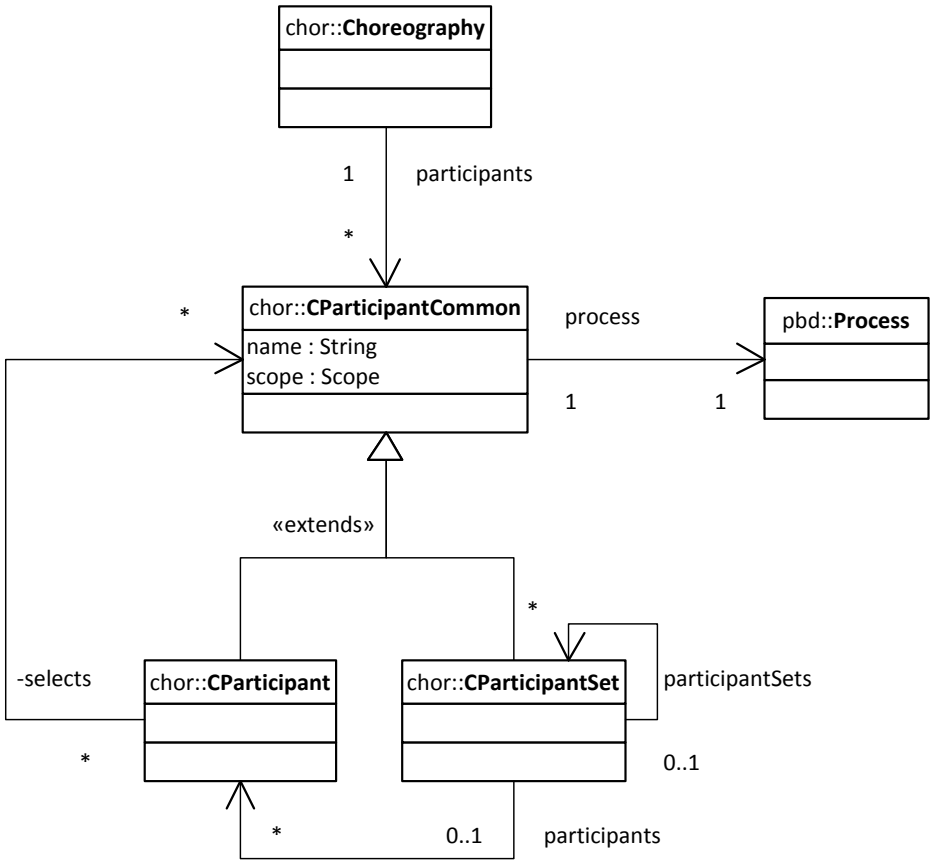

Abbildung 4.5: Chor Model mit Participants

Die ‹ `ForEach` › Aktivitäten des PBD Model können über ein `ParticipantSet` iterieren. Im Topology Model haben `ParticipantSet` und `Participant` ein *`forEach`* Attribut wobei für letztere das setzen dieses Attributes nur Sinn macht, wenn sie innerhalb eines `ParticipantSet` mit *`forEach`* Attribut sind. Im Chor Model drehen wir das Ganze um und geben dem Modellierer bei der ‹ `ForEach` › Aktivität die Möglichkeit, anzugeben über welches Set iteriert werden soll und welches Kind Element die Referenz auf den aktuellen Wert im Schleifendurchlauf hält. Wir erweitern allerdings nicht die ‹ `ForEach` › Aktivität im PBD Model, sondern definieren das neue Element `ForEachIterationSpec`, wie in Abbildung 4.6 zu sehen. Dieses Element referenziert `CParticipant` als *`iteratorValue`*, `CParticipantSet` als *`iteratorSet`* und schließlich die zugehörige ‹ `ForEach` › Aktivität selbst. Mit dieser Lösung ist es ebenfalls möglich und auch richtig, dass mehrere ‹ `ForEach` › Aktivitäten das selbe `CParticipantSet` referenzieren können. Dabei ist lediglich als Modellierer darauf zu achten, dass in jeder `ForEachIterationSpec` immer unterschiedliche `CParticipant` Kind Elemente des selben Sets, als *`iteratorValue`* angegeben werden.

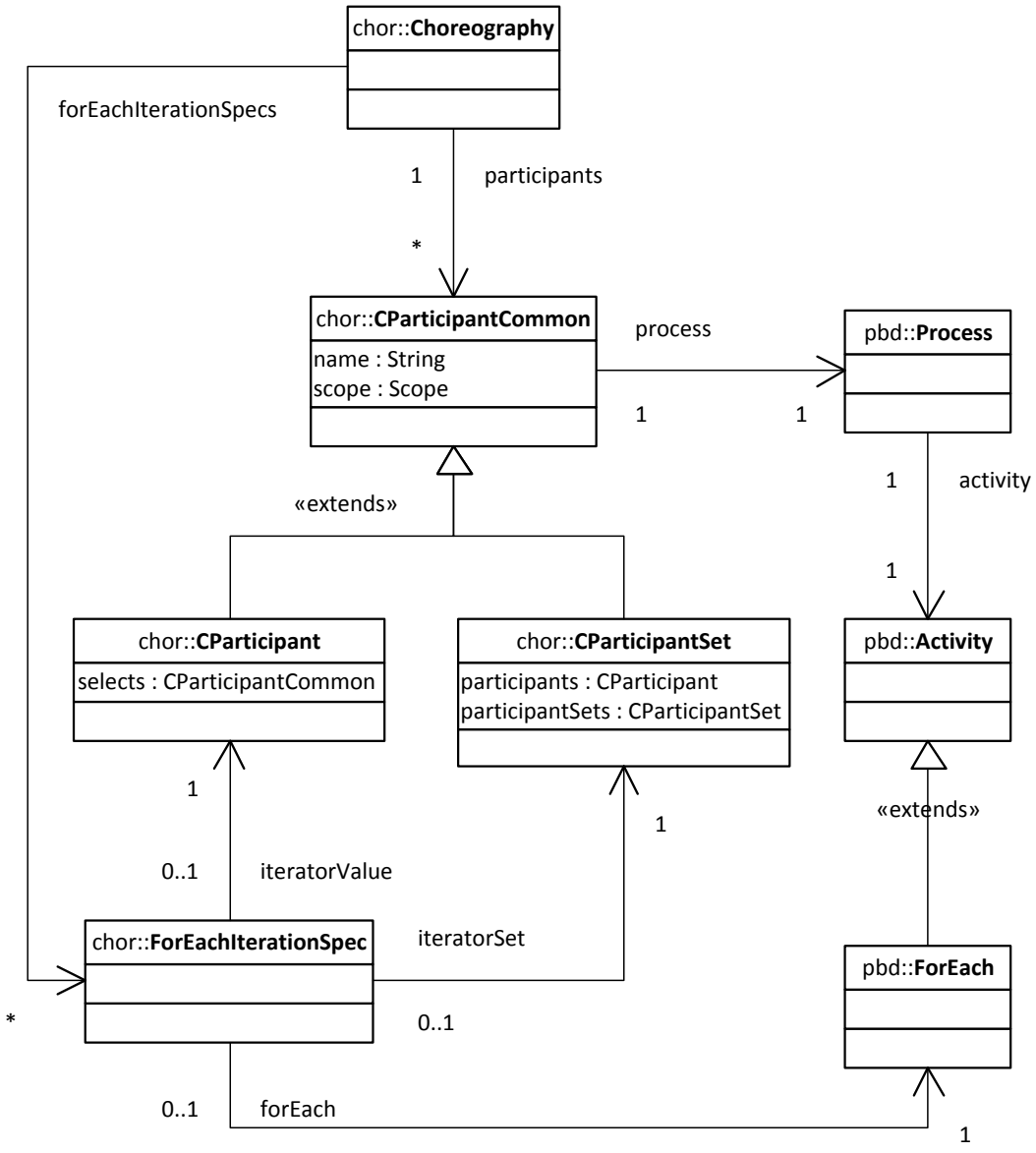

Abbildung 4.6: Chor Model mit Participants und ForEach Lösung

Message Links

Die `MessageLink` Elemente aus dem Topology Model geben mit ihren Attributen *`sendActivity`* und *`receiveActivity`* jeweils Quelle und Ziel des Links an. Für die Umsetzung im Chor Model betrachten wir Abbildung 4.7. Für `MessageLink` wird ein `CMessageLink` Element eingeführt, welches ebenfalls die selben Attribute bekommt. Wir

könnten mit *sendActivity* und *receiveActivity* jeweils Aktivitäten des PBD Model referenzieren, doch müssten wir dann beim Link setzen entsprechende Prüfungen vornehmen, da Links nur zwischen den Kommunikationsaktivitäten erlaubt sind, sowie zusätzlich noch der ‹ OnMessage › Zweig der ‹ Pick › Aktivität. Daher führen wir das neue Element CLinkable ein und referenzieren mit den Attributen *sendActivity* und *receiveActivity* jeweils darauf. Diejenigen Elemente aus dem PBD Model welche verlinkt werden dürfen, erben von besagtem Element, die anderen nicht. Wir müssen allerdings noch Prüfungen durchführen um zu bestimmen, ob ein Link tatsächlich zwischen zwei CLinkable Elementen gezogen werden darf. Die Bedingungen sind in [DKLW07] beschrieben.

Das Attribut *participantRefs* übernehmen wir ebenfalls, referenzieren aber damit ein neues Element CParticipantRef. Die Einführung dieses neuen Elements ist notwendig, da wir Participant Referenzen grafisch darstellen wollen. Das CParticipantRef Element referenziert dann schließlich den eigentlichen CParticipant, wessen Referenz über diesen CMessageLink ausgetauscht werden soll. Das Attribut *bindSenderTo* referenziert im Chor Model den CParticipant, welcher der Sender ist. Falls der CMessageLink von einem CParticipant ausgeht, ist es die Selbstreferenz, bei CParticipantSet ist es eines der Kind Elemente. Die Attribute *sender* und *receiver* referenzieren jeweils einen CParticipant, da Absender und Empfänger konkrete Teilnehmer sein müssen und nicht eine Menge von Teilnehmern. Für den Fall, dass alle Kind Elemente eines CParticipantSet Absender sein können, gibt es das *senders* Attribut, welches folglich CParticipantSet referenziert. Die Attribute *sender*, *senders*, *receiver* und *bindSenderTo* von CMessageLink sind in Abbildung 4.7, zwecks besserer Übersicht, nicht als Assoziationen dargestellt. Dies gilt auch für die Assoziationen von Choroegraphy.

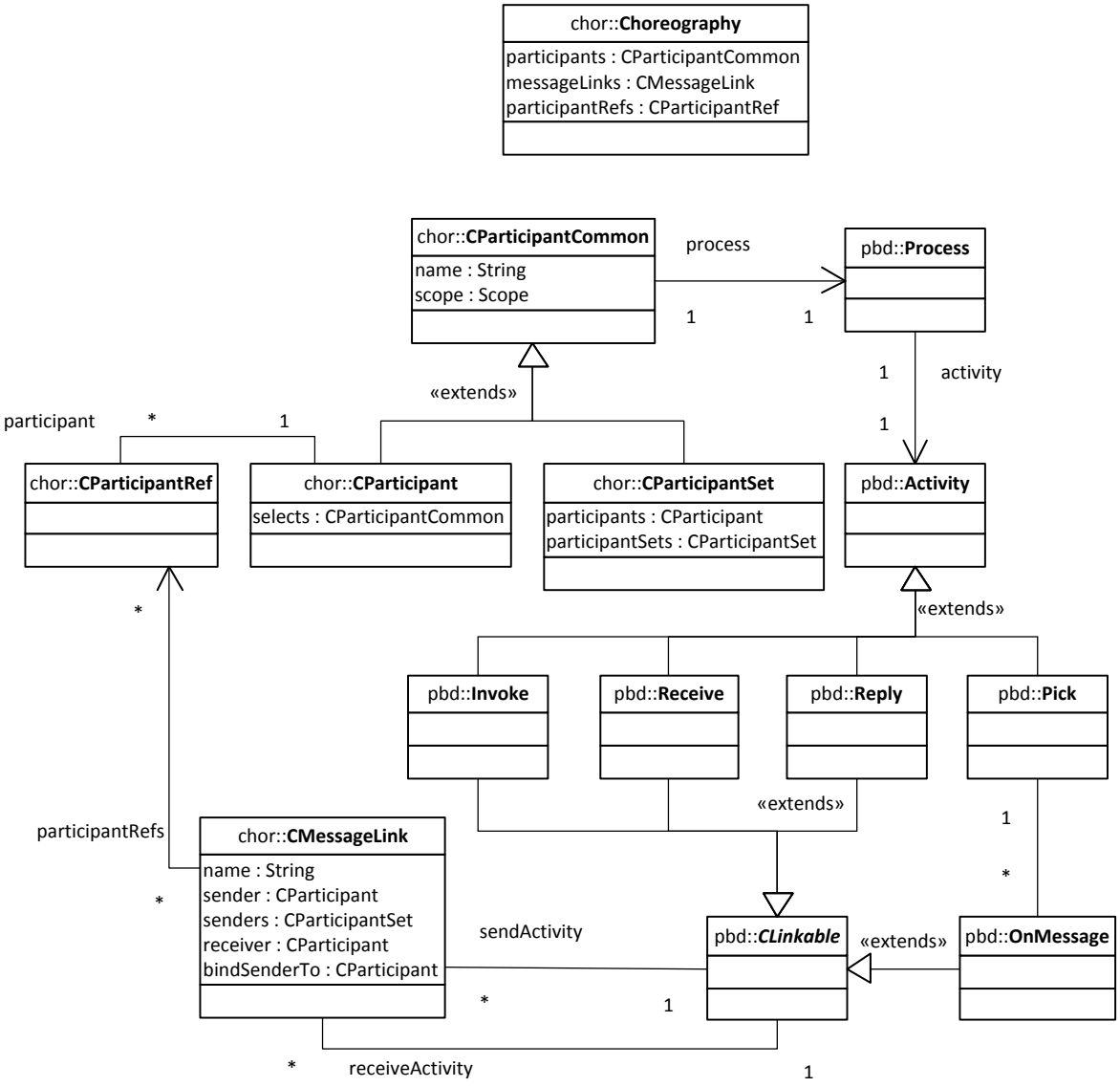

Abbildung 4.7: Chor Model mit `CMessageLink`, `CParticipantRef` und `CLinkable`

Links im Flow Container

Die ‹ `Flow` › Aktivität, definiert im WS-BPEL Standard [OAS07c], erlaubt graphbasierten Kontrollfluss der modellierten Aktivitäten. Wir möchten in unserem Editor die Aktivitäten im `Flow` Container platzieren und mit Kanten bzw. Links Verbinden. Dazu betrachten wir Abbildung 4.8. Das PBD Model stellt `Link` Elemente bereit, welche von der ‹ `Flow` › Aktivität referenziert werden. Diese `Link` Elemente haben jeweils ein *`name`* Attribut. Die Basisklasse

`Activity`, von welcher alle BPEL Aktivitäten erben, erlaubt die Angabe von `Targets` und `Sources`. Somit können sich alle Aktivitäten untereinander verlinken. `Target` und `Source` Elemente verweisen mit ihren *`linkName`* Attributen auf das *`name`* Attribut des `Link` Elements. Zusätzlich bietet `Source` noch die Angabe einer `TransitionCondition` und für alle `Target` Elemente zusammen, lässt sich eine `JoinCondition` angeben. Diese Elemente stehen uns zur Verfügung, sind aber eher unpraktisch um einen Link zwischen zwei Aktivitäten nach unseren Voraussetzungen grafisch darzustellen. Wir führen im Chor Model das neue Element `FlowActivityLink` ein. Mit den *`sourceActivity`* und *`targetActivity`* Attributen verweisen wir auf `Activity` aus dem PBD Model. Das *`name`* Attribut bezeichnet den Namen dieses Links und mit dem *`transitionCondition`* Attribut verweisen wir auf das entsprechende `TransitionCondition` Element aus PBD Model. Die `JoinCondition` können wir an jede Aktivität setzen, welche eine benötigt. Dazu erstellen wir, falls nicht schon vorhanden, eine neue Instanz des `Targets` Containers und referenzieren mit dessen *`joinCondition`* Attribut auf die entsprechende `JoinCondition` Instanz. Somit können nun Links zwischen Aktivitäten gezogen werden. Allerdings müssen wir bei der Transformation bedenken, die entsprechenden Elemente `Target`, `Source` und `Link` des PBD Model, aus dem `FlowActivityLink` Element zu generieren.

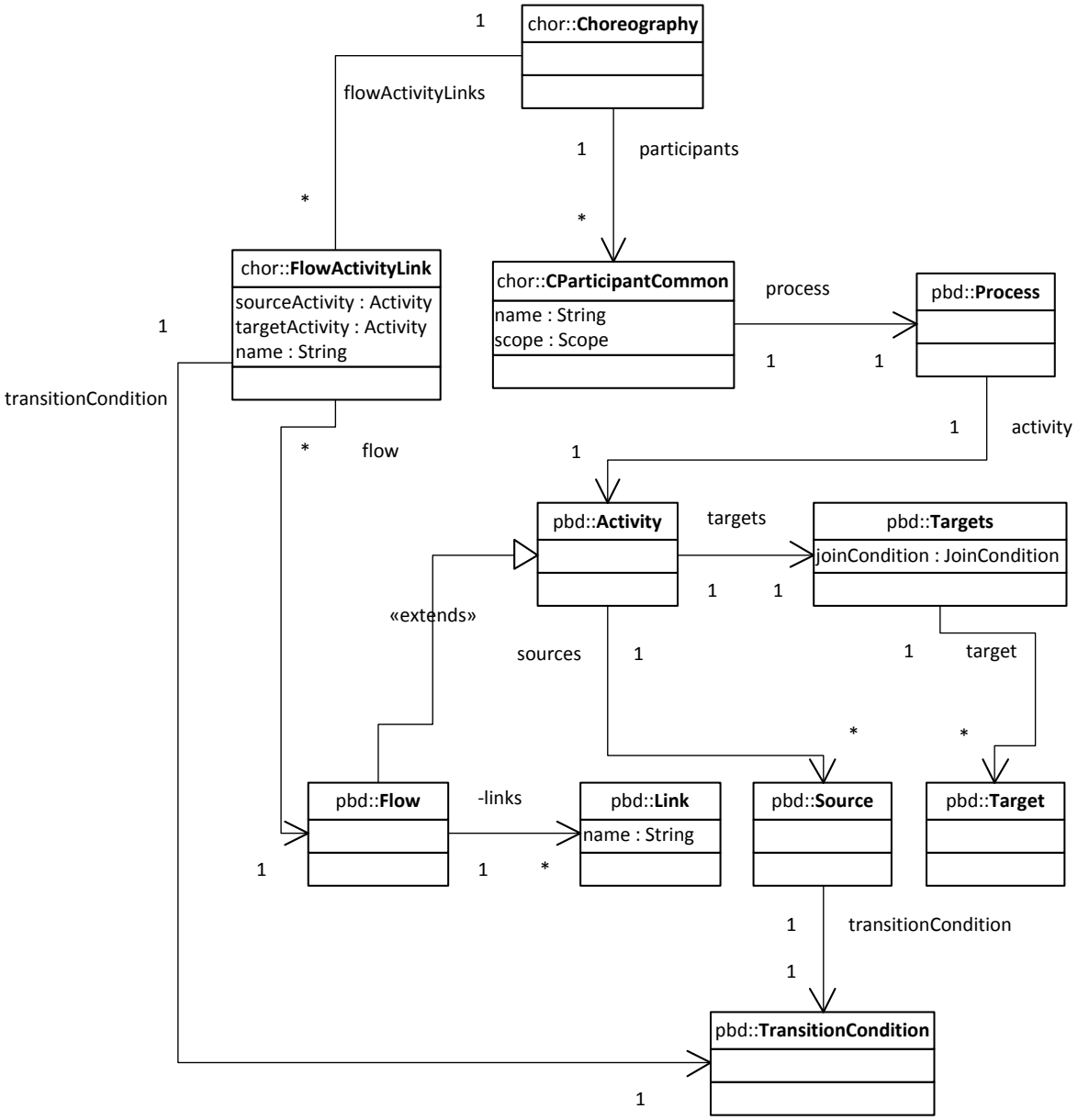

Abbildung 4.8: Chor Model mit `FlowActivityLink` und Beziehungen

Grounding

Wie wir im Abschnitt 3.3 auf Seite 16 erläutert haben ist es für eine BPEL4Chor Choreographie notwendig, ein Grounding anzugeben um BPEL Prozesse aus der Choreographie zu generieren. Wir übernehmen den Grounding Mechanismus für unser Chor

Model und führen das neue Element `CGrounding` ein, welches für ein spezifisches Grounding steht. Das Grounding Model sieht `MessageLink`, `ParticipantRef` und `Property` Elemente vor. Diese Elemente muss es auch für unser `CGrounding` geben, daher führen wir die Elemente `CMessageLinkGrounding`, `CParticipantRefGrounding` und `CorrelationSetGrounding` ein, wie in Abbildung 4.9 zu sehen ist.

Das *`cMessageLink`* Attribut von `CMessageLinkGrounding` referenziert das `CMessageLink` Element, welches mit einem Grounding verknüpft werden soll. Das *`portType`* Attribut gibt den gewählten Port Type aus einer WSDL an. Dies wird als "Qualified Name" angegeben, daher verwenden wir einen passenden Datentyp QName bestehend aus den Teilen Namespace URI, localPart und prefix. Das *`operation`* Attribut gibt die WSDL *Operation* an und das *`bSToWSDLProperty`* Attribut gibt die *WSDL Property* als QName für das *`bindSenderTo`* Attribut an. Es muss nur gesetzt werden, wenn *`bindSenderTo`* auch im referenzierten `CMessageLink` angegeben wurde.

Das *`cParticipantRef`* Attribut von `CParticipantRefGrounding` referenziert das `CParticiapntRef` Element, welches mit diesem Grounding verknüpft werden soll. Das *`WSDLProperty`* Attribut gibt die *WSDL Property* als QName für das referenzierte `CParticiapntRef` Element an.

Für `CorrelationSetGrounding` müssen wir noch ein weiteres Element einführen da im PBD Model das *`properties`* Attribut von `CorrelationSet` eine Liste von Werten, welche durch Leerzeichen getrennt sind, enthält. Jedem Eintrag muss ein QName im Grounding zugeordnet werden. Wir führen das Element `PropertyGrounding` im Chor Model ein. Ein Element davon verknüpft genau **einen** *`properties`* Eintrag, mit **einem** QName. Das *`propertyName`* Attribut von `PropertyGrounding` gibt den Namen von **einem** Eintrag aus dem *`properties`* Attribut von `CorrelationSet` an, welche mit diesem Grounding verknüpft werden soll. Das *`WSDLProperty`* Attribut gibt die *WSDL Property* als QName an, mit welcher *`propertyName`* verknüpft werden soll.

Das *`correlationSet`* Attribut von `CorrelationSetGrounding` referenziert das `CorreltationSet` aus dem PBD Model, welchen mit diesem Grounding verknüpft werden soll. Das *`propertyGroundings`* referenziert für jeden Eintrag aus dem *`properties`* Attribut des referenzierten `CorrelationSet`, ein `PropertyGrounding` Element.

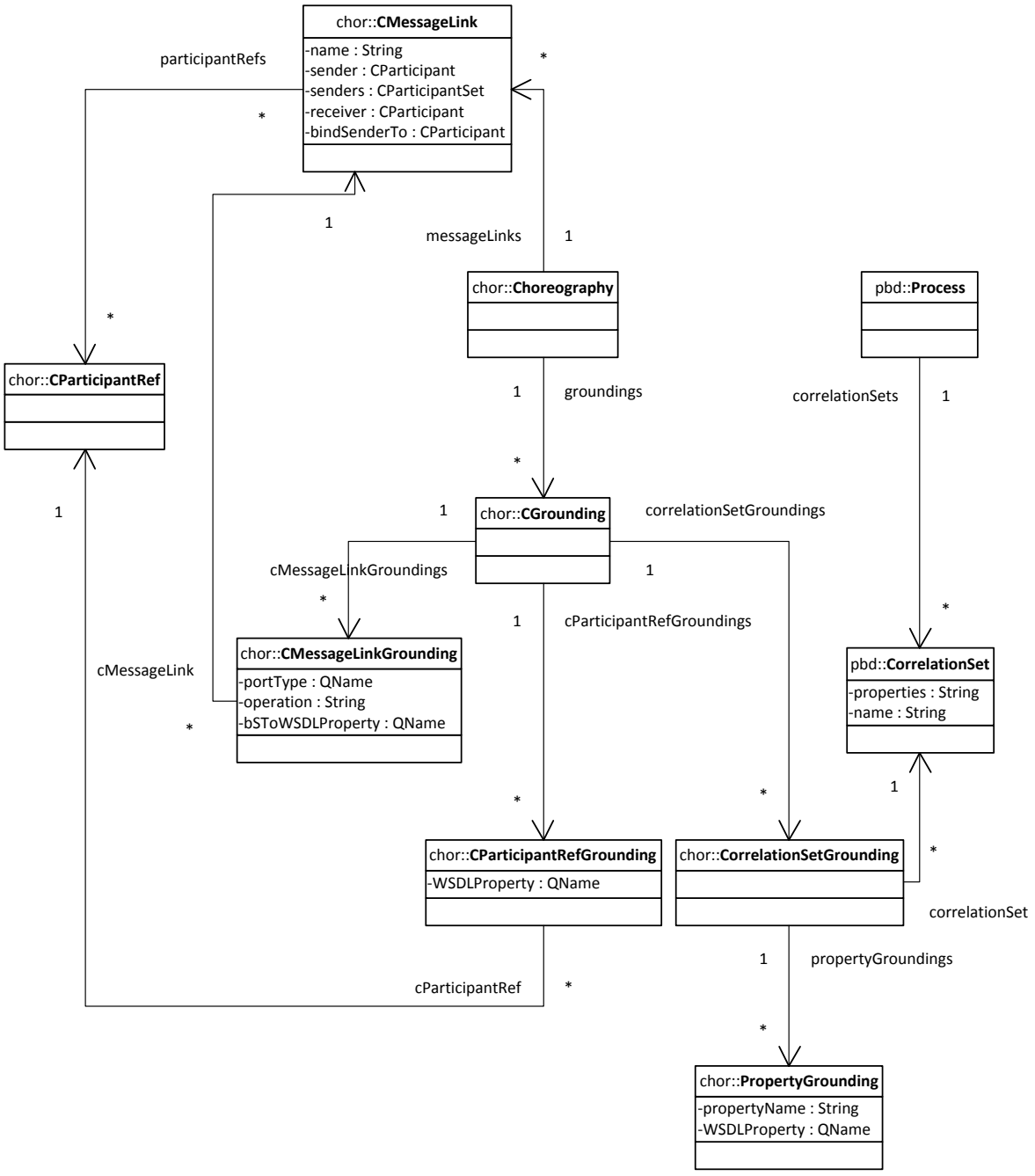

Abbildung 4.9: Chor Model mit `CGrounding`

4.5.2 Chor Model Transformation

Wir haben im Abschnitt 4.5.1 auf Seite 35 besprochen, wie das Datenmodell für den Editor aufgebaut ist. Wir können nun Choreographien modellieren und möchten jetzt die Möglichkeit haben, das Chor Model in ein anderes Modell zu transformieren. Da uns in dieser Arbeit nur BPEL4Chor und BPEL Prozesse interessieren, geben wir hier ein Konzept zur Generierung der BPEL4Chor Artefakte, sowie eine anschließende Transformation zu BPEL Prozessen an. Wir entscheiden uns hier, die Artefakte im Speicher als Document Object Model (DOM) [WLHa00] zu repräsentieren und serialisieren sie darauf in XML Dokumente zur Speicherung auf einem Datenträger. DOM Dokumente haben den Vorteil, dass die Daten in einer Baumstruktur repräsentiert werden und viele Methoden zur einfachen Navigation durch den Baum bereits durch das Framework gestellt werden. Zur Generierung von BPEL Prozessen benutzen wir zudem noch die bereits existierende Komponente BPEL4ChorToBPEL, welche in Abschnitt 5.1.6 auf Seite 105 beschrieben wird. Diese Komponente fordert als Eingaben ebenfalls DOM Dokumente, von daher scheint uns hier DOM die richtige und zugleich praktische Wahl zu sein.

Generierung der BPEL4Chor Artefakte

Durch das Modellieren mit dem Editor erzeugen wir das Chor Model, welches auch mehrere Instanzen des PBD Model über das `process` Attribut von `CParticipantCommon` referenziert. Die aktuelle Instanz des Chor Model dient also als Eingabe für die Transformer Komponente. An dieser Stelle definieren wir die Termini *Transformer* und *Builder*. *Transformer* Komponenten erzeugen DOM Objekte, *Builder* Komponenten erzeugen Modell Objekte. In Abbildung 4.10 ist der Dokumentenfluss zwischen den einzelnen Komponenten dargestellt. Da wir vom Chor Model ausgehen, müssen wir Topology Model und Grounding Model Instanzen zuerst erzeugen. PBD Model Instanzen sind vom Chor Model referenziert und werden durch Modellierung der *participants* mit seinen Aktivitäten erzeugt. Wie wir in Abschnitt 4.5.1 auf Seite 43 besprochen haben, müssen wir allerdings für die Links zwischen den Aktivitäten in einem ‹ `Flow` ›, das PBD Model mit den entsprechenden Daten anreichern. Das bedeutet, dass wir zusätzliche Instanzen von Modell Objekten bilden. Wir benötigen die *Builder* Komponenten *TopologyBuilder*, *GroundingBuilder* und *FlowBuilder*. Nachdem uns die Modell Instanzen vorliegen, transformieren wir diese zu DOM Dokumenten. Dafür benötigen wir die *Transformer* Komponenten *TopologyTransformer*, *GroundingTransformer*, *PBDTransformer* sowie den umfassenden *ChoreographyTransformer*, der den Transformationsprozess durchführt und die anderen Komponenten in der richtigen Reihenfolge, mit den passenden Eingaben, aufruft.

Wir stellen als nächstes die Algorithmen der *Builder* Komponenten vor. Als allgemeine Konvention für die Notation gilt, dass die tiefer gestellten Bezeichnungen *pbd*, *top*, *grnd* oder *chor* jeweils die Modellzugehörigkeit der Elemente kennzeichnen. Dabei gehören die mit *pbd*

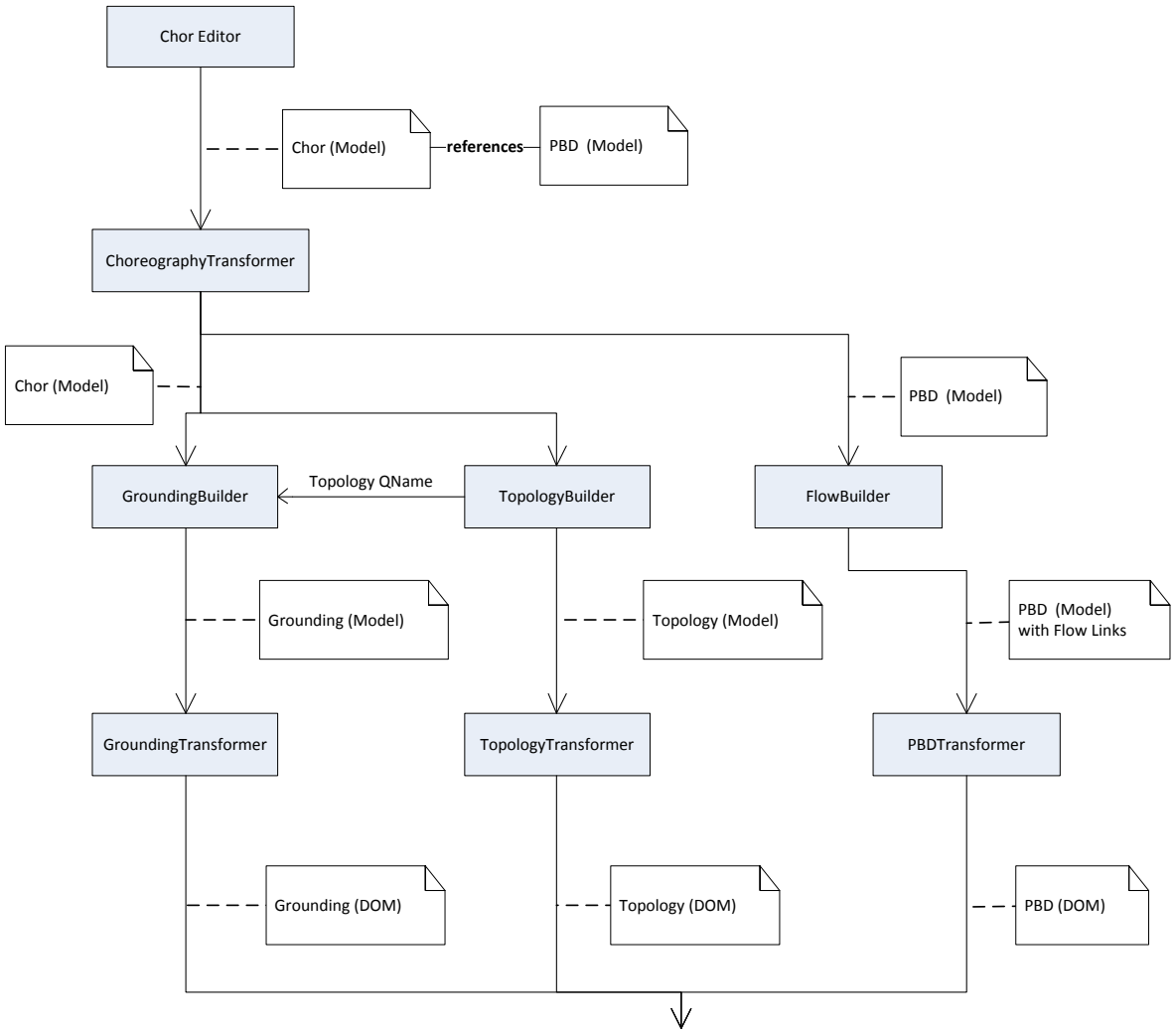

Abbildung 4.10: Dokumentenfluss zwischen den *Transformer* und *Builder* Komponenten

gekennzeichneten Elemente zum PBD Model, *top* zum Topology Model, *grnd* zum Grounding Model und *chor* zum Chor Model. Der Doppelpunkt wird zur Typisierung von Variablen verwendet, wie z. B. $e : Choreography_{chor}$ bedeutet: Variable e vom Typ *Choreography* aus dem $_{chor}$ Chor Model. Mehrwertige Attribute, Referenzen auf mehrere Elemente und Listen behandeln wir als Mengen wie z. B. $\mathcal{M} = \{cml | cml : CMessageLink_{chor}\}$ für die Menge aller `CMessageLink` Elemente des Chor Model. Die Variable *chormodel* ist global für alle *Builder* verfügbar und hält die Instanz der `Choreography` Klasse.

Gemeinsame Algorithmen

Die *Builder* Komponenten benötigen vor allem einheitliche Schemen zur Namensgenerierung von Elementen da Topology Model und Grounding Model entweder QNames oder NCNames verwenden um auf andere Elemente zu verweisen. Unser Quellmodell ist das Chor Model, in welchem wir stattdessen Referenzen benutzen daher brauchen wir eine Methode, die einerseits den Namen eines Chor Model Elements zurück gibt und andererseits müssen wir den Fall abfangen, wenn der Name nicht gesetzt wurde. Dazu führen wir die Hilfsmethode *getNormalizedName* ein. Der Algorithmus 4.1 zeigt, wie Namen für `CParticipantCommon`, `Choreography`, `CMessageLink` und `Process` erzeugt werden. Wir definieren zudem noch die Methode *classname*(), welche den Klassennamen des gegeben Elements zurück gibt wie z. B. bei Übergabe von `CMessageLink`, würde "cMessageLink" zurück gegeben werden. Wir hängen an den Namen noch zusätzlich einen Hash Code an welcher die Speicheradresse der Instanz zurück gibt. Dies sorgt dafür, dass die Elemente eindeutig benannt werden.

Da die Elemente vom PBD Model bzw. BPEL in einer Baumstruktur angeordnet sind, können wir davon ausgehen dass es Möglichkeiten gibt, zwischen den Knoten zu navigieren. Wir können also Kind- und Elternknoten bestimmen. Dafür definieren wir die Funktion $parentOf()$, welche vom übergebenen Element den Elternknoten zurück gibt oder *null*, falls es keinen gibt. Die Funktion $typeOf()$ gibt den Instanz Typ des gegebenen Elements zurück. Als Beispiel wäre der Wert von $typeOf(a : Activity_{pbd})$ gleich $Invoke_{pbd}$, wenn a eine Instanz von `Invoke` ist. In Algorithmus 4.3 bestimmen wir das `Process` Element von einem beliebigen Element aus PBD Model ($Element_{pbd}$) ausgehend.

Der Algorithmus 4.4 erzeugt einen QName für die gegebene Aktivität. Dieser QName setzt sich aus dem Namespace des zugehörigen `Process` Elements und dem `id` Attribut der Aktivität zusammen. Deshalb benötigen wir die `Process` Instanz, in welcher diese Aktivität definiert ist. Diese Instanz finden wir mit Algorithmus 4.3.

Algorithmus 4.1 Generierung von Namen, allgemein für Chor Model Elemente

procedure GETNORMALIZEDNAME($e : CParticipantCommon_{chor} \vee e : Process_{pbd} \vee e : Choreography_{chor} \vee e : CMessageLink_{chor}$)
  $name : String \leftarrow e.name$
  if $name = null$ **then**
    $name \leftarrow classname(e)$
    $name \leftarrow name + \text{'\_'} + e.hashCode$
  end if
  return *name*
end procedure

Algorithmus 4.2 Erzeugt den QName für eine *Participant Behavior Description*

```
procedure buildPBDQName(p : Process_pbd)
    pbdQName : QName ←new QName
    processName : String ←getNormalizedName(p)
    pbdQName.namespaceURI ← p.targetNamespace
    pbdQName.localPart ← processName
    pbdQName.prefix ← lowerCase(processName)
    return pbdQName
end procedure
```

Algorithmus 4.3 Findet das zugehörige `Process` Element, ausgehend von einem beliebigen Element aus dem PBD Model

```
procedure deduceProcess(e : Element_pbd)
    p : Element_pbd
    parentElement : Element_pbd ← parentOf(e)
    while parentElement ≠ null do
        if typeOf(parentElement) = Process_pbd then
            p ← parentElement
            break
        end if
        parentElement ← parentOf(parentElement)
    end while
    return p
end procedure
```

Algorithmus 4.4 Erzeugt einen QName für die gegebene Aktivität

```
procedure buildActivityQName(a : Activity_pbd)
    activityQName : QName ←new QName
    p : Process_pbd ←deduceProcess(a)
    pbdQName ←buildPBDQName(p)
    activityQName.namespaceURI ← pbdQName.namespaceURI
    activityQName.localPart ← a.id
    activityQName.prefix ← pbdQName.prefix
    return activityQName
end procedure
```

TopologyBuilder Algorithmen

Mit Algorithmus 4.5 erzeugen wir `ParticipantType` Elemente. Für die Erstellung des *name* Attributs benutzten wir den Algorithmus 4.6, da wir den Typ Namen auch bei `Participant` und `ParticipantSet` korrekt setzen müssen. Das *participantBehaviorDescription* Attribut ist ein QName, welchen wir mit Algorithmus 4.2 erzeugen, und verweist auf das entsprechende externe *Participant Behavior Description* Dokument. Für jeden `ParticipantType` erhöhen wir einen Zähler und hängen den Wert dem *name* Attribut an. Somit ist gewährleistet dass die Namen, selbst bei gleicher Benennung unterschiedlicher Typen, eindeutig sind.

Mit Algorithmus 4.7 erzeugen wir einen `Participant` im Topology Model. Das *forEach* Attribut ist ein QName und verweist auf die Aktivität der zugehörigen *Participant Behavior Description*. Das selbe gilt auch für *scope*. Wir erzeugen diesen QName mit Algorithmus 4.4. `ParticipantSet` und `MessageLink` Elemente erzeugen wir mit den Algorithmen 4.9 und 4.8. Der Startpunkt des *TopologyBuilders* ist in Algorithmus 4.10 realisiert. Er erzeugt aus dem Chor Model ein `Topology` Dokument.

Algorithmus 4.5 Erzeugt ein `ParticipantType` Element im Topology Model

procedure CREATEPARTICIPANTTYPE($p : Process_{pbd}$)
  $pt : ParticipantType_{top} \leftarrow$ new $ParticipantType_{top}$
  // $processId : Integer$ ist hier eine globale Variable
  $pt.name \leftarrow$CREATETYPENAME($p, processId$)
  $pt.participantBehaviorDescription \leftarrow$BUILDPBDQNAME(p)
  $processId \leftarrow processId + 1$
  return pt
end procedure

Algorithmus 4.6 Erzeugt den Namen eines `ParticipantType` Element im Topology Model

procedure CREATETYPENAME($p : Process_{pbd}, processId : Integer$)
  $processName : String \leftarrow$GETNORMALIZEDNAME(p)
  // $processId : Integer$ ist hier eine lokale Variable
  $processName \leftarrow processName + \text{'\_'} + processId + \text{'\_ type'}$
  return $processName$
end procedure

Algorithmus 4.7 Erzeugt ein `Participant` Element im Topology Model

procedure createParticipant($cp : CParticipant_{chor}$)
  $p : Participant_{top} \leftarrow$ new $Participant_{top}$
  $p.name \leftarrow$getNormalizedName(cp)
  // *process* ist *null*, wenn cp ein Kind Element eines Sets ist
  if $cp.process \neq null$ **then**
    $p.type \leftarrow$createTypeName($cp.process$)
  end if
  // $\mathcal{F} = \{f | f : ForEachIterationSpec_{chor}\}$
  $\mathcal{F} \leftarrow chormodel.forEachIterationSpecs$
  for all $f \in \mathcal{F}$ **do**
    if $f.iteratorValue \neq null \wedge f.iteratorValue = cp$ **then**
      $p.forEach \leftarrow$buildActivityQName($f.forEach$)
    end if
  end for
  if $cp.scope \neq null$ **then**
    $p.scope \leftarrow$buildActivityQName($cp.scope$)
  end if
  // $\mathcal{C} = \{c | c : CParticipantCommon_{chor}\}$
  $\mathcal{C} \leftarrow cp.selects$
  for all $c \in \mathcal{C}$ **do**
    $name : String \leftarrow$getNormalizedName(c)
    $p.selects \cup \{name\}$
  end for
  $p.containment \leftarrow cp.containment$
  return p
end procedure

Algorithmus 4.8 Erzeugt ein `ParticipantSet` Element im Topology Model

```
procedure createParticipantSet(cpSet : CParticipantSet_chor)
    pSet : ParticipantSet_top ← new ParticipantSet_top
    pSet.name ←getNormalizedName(cpSet)
                                  // process ist null, wenn cpSet ein Kind Element eines Sets ist
    if cpSet.process ≠ null then
        pSet.type ←createTypeName(cpSet.process)
    end if
                                                  // F = {f|f : ForEachIterationSpec_chor}
    F ← chormodel.forEachIterationSpecs
    for all f ∈ F do
        if f.iteratorSet ≠ null ∧ f.iteratorSet = cpSet then
            name : QName ←buildActivityQName(f.forEach)
            pSet.forEach ∪ {name}
        end if
    end for
    if cpSet.scope ≠ null then
        pSet.scope ←buildActivityQName(cpSet.scope)
    end if
                                                  // P = {child|child : CParticipant_chor}
    P ← cpSet.participants
    for all child ∈ P do
        p : Participant_top ←createParticipant(child)
        pSet.participants ∪ {p}
    end for
                                                  // S = {child|child : CParticipantSet_chor}
    S ← cpSet.participantSets
    for all child ∈ S do
                                                                  // rekursiver Aufruf
        s : ParticipantSet_top ←createParticipantSet(child)
        pSet.participantSets ∪ {s}
    end for
    return pSet
end procedure
```

Algorithmus 4.9 Erzeugt ein `MessageLink` Element im Topology Model

procedure createMessageLink($cml : CMessageLink_{chor}$)
  $ml : MessageLink_{top} \leftarrow$ new $MessageLink_{top}$
  $ml.name \leftarrow$getNormalizedName(cml)
  $ml.sendActivity \leftarrow cml.sendActivity.id$
  $ml.recieveActivity \leftarrow cml.receiveActivity.id$
  if $cml.sender \neq null$ **then**
    $ml.sender \leftarrow$getNormalizedName($cml.sender$)
  end if
  if $cml.senders \neq null$ **then**
    $ml.senders \cup \{$getNormalizedName($cml.senders$)$\}$
  end if
  $ml.receiver \leftarrow$getNormalizedName($cml.receiver$)
  $ml.messageName \leftarrow cml.messageName$
  // $\mathcal{R} = \{ref | ref : CParticipantRef_{chor}\}$
  $\mathcal{R} \leftarrow cml.participantRefs$
  for all $ref \in \mathcal{R}$ **do**
    if $ref.participant \neq null$ **then**
      $name : String \leftarrow$getNormalizedName($ref.participant$)
      $ml.participantRefs \cup \{name\}$
    end if
  end for
  if $cml.bindSenderTo \neq null$ **then**
    $ml.bindSenderTo \leftarrow$getNormalizedName($cml.bindSenderTo$)
  end if
  return ml
end procedure

Algorithmus 4.10 Erzeugt eine Topology Model Instanz aus dem gegebenen Chor Model

procedure BUILD
  $top : Topology_{top} \leftarrow$ new $Toplogy_{top}$
  $top.targetNamespace \leftarrow chormodel.targetNamespace$
  $name : String \leftarrow$ getNormalizedName($chomodel$)
  $top.name \leftarrow name+$'Topology'
  // $\mathcal{C} = \{c | c : CParticipantCommon_{chor}\}$
  $\mathcal{C} \leftarrow chormodel.participants$
  for all $c \in \mathcal{C}$ **do**
    $pType : ParticipantType_{top} \leftarrow$ createParticipantType($c.process$)
    $top.participantTypes \cup \{pType\}$
    if $typeOf(c) = CParticipant_{chor}$ **then**
      $p : Participant_{top} \leftarrow$ createParticipant(c)
      $top.participants \cup \{p\}$
    else if $typeOf(c) = CParticipantSet_{chor}$ **then**
      $pSet : ParticipantSet_{top} \leftarrow$ createParticipantSet(c)
      $top.participants \cup \{pSet\}$
    end if
  end for
  // $\mathcal{M} = \{cml | cml : CMessageLink_{chor}\}$
  $\mathcal{M} \leftarrow chormodel.messageLinks$
  for all $cml \in \mathcal{M}$ **do**
    $ml : MessageLink_{top} \leftarrow$ createMessageLink(cml)
    $top.messageLinks \cup \{ml\}$
  end for
  return top
end procedure

GroundingBuilder Algorithmen

Das Grounding Model referenziert das entsprechende Topology Model als QName, daher benötigen wir eine Methode um den QName von Topology zu erzeugen. In Algorithmus 4.11 ist die Vorgehensweise zu sehen. Mit den Algorithmus 4.12 erzeugen wir ein `MessageLink` Element im Grounding Model. Der Algorithmus 4.13 erzeugt ein `ParticipantRef` Element für einen Eintrag aus dem *`participantRefs`* Attribut eines `CMessageLink` und Algorithmus 4.14 erledigt das selbe für das *`bindSenderTo`* Attribut. Für alle *`properties`* Einträge eines `CorrelationSet`, müssen wir jeweils ein `Property` Element im Grounding Model erzeugen, was mit Algorithmus 4.15 durchgeführt wird. Der Startpunkt des *GroundingBuilders* ist in Algorithmus 4.16 realisiert. Er erzeugt für alle `CGrounding` Elemente des Chor Model jeweils ein eigenes `Grounding`.

Algorithmus 4.11 Erzeugt den QName für `Topology` aus dem Topology Model

```
procedure buildTopologyQName
    topQName : QName ←new QName
    chorName : String ←getNormalizedName(chormodel)
    topQName.namespaceURI ← chormodel.targetNamespace
    topQName.localPart ← chorName+'Topology'
    topQName.prefix ← +'top'
    return topQName
end procedure
```

Algorithmus 4.12 Erzeugt ein `MessageLink` Element im Grounding Model

```
procedure createMessageLink(cmlg : CMessageLinkGrounding_chor)
    ml : MessageLink_grnd ← new MessageLink_grnd
    mlQName : QName ← new QName
    topQName : QName ←buildTopologyQName
    mlQName.namespaceURI ← topQName.namespaceURI
    mlQName.localPart ←getNormalizedName(cmlg.cMessageLink)
    mlQName.prefix ← topQName.prefix
    ptQName ← cmlg.portType
    if ptQName ≠ null ∧ ptQName.localPart ≠ null then
        ml.portType ← ptQName
    end if
    ml.operation ← cmlg.operation
    return ml
end procedure
```

Algorithmus 4.13 Erzeugt ein `ParticipantRef` Element im Grounding Model für gesetzte *`participantRefs`* eines `CMessageLink`

```
procedure createParticipantRef(cPrefg : CParticipantRefGrounding_chor)
    pRef : ParticipantRef_grnd ← new ParticipantRef_grnd
    p : CParticipant_chor ← cPrefg.cParticipantRef.participant
    if p ≠ null then
        name : String ←getNormalizedName(p)
        pRef.name ← name
    end if
    wsdlPropQName : QName ← cPrefg.WSDLProperty
    if wsdlPropQName ≠ null ∧ wsdlPropQName.localPart ≠ null then
        pRef.WSDLProperty ← wsdlPropQName
    end if
    return pRef
end procedure
```

Algorithmus 4.14 Erzeugt ein `ParticipantRef` Element im Grounding Model für das gesetzte *`bindSenderTo`* eines `CMessageLink`

```
procedure createParticipantRef(cmlg : CMessageLinkGrounding_chor)
    pRef : ParticipantRef_grnd ← new ParticipantRef_grnd
    p : CParticipant_chor ← cmlg.cMessageLink.bindSenderTo
    if p ≠ null then
        pRef.name ←getNormalizedName(p)
    end if
    wsdlPropQName : QName ← cmlg.bSToWSDLProperty
    if wsdlPropQName ≠ null ∧ wsdlPropQName.localPart ≠ null then
        pRef.WSDLProperty ← wsdlPropQName
    end if
    return pRef
end procedure
```

Algorithmus 4.15 Erzeugt `Property` Elemente im Grounding Model für alle *property* Einträge des `CorrelationSet`

procedure createProperties($cSetg : CorrelationSetGrounding_{chor}$)
  $propertyList : List \leftarrow$ new $List$
  $corrSet : CorrelationSet_{pbd} \leftarrow cSetg.correlationSet$
  $process : Process_{pbd} \leftarrow$deduceProcess($corrSet$)
  $pbdQName : QName \leftarrow$buildPBDQName($process$)
  // $\mathcal{P} = \{pg | pg : PropertyGrounding_{chor}\}$
  $\mathcal{P} \leftarrow cSetg.propertyGroundings$
  for all $pg \in \mathcal{P}$ **do**
    $propQName.namespaceURI \leftarrow pbdQName.namespaceURI$
    $propQName.localPart \leftarrow pg.propertyName$
    $propQName.prefix \leftarrow pbdQName.prefix$
    $prop : Property_{grnd} \leftarrow$ new $Property_{grnd}$
    $prop.name \leftarrow propQName$
    $wsdlPropQName : QName \leftarrow pg.WSDLProperty$
    if $wsdlPropQName \neq null \wedge wsdlPropQName.localPart \neq null$ **then**
      $prop.WSDLProperty \leftarrow wsdlPropQName$
    end if
    $propertyList \cup \{prop\}$
  end for
  return $propertyList$
end procedure

Algorithmus 4.16 Erzeugt Grounding Model Instanzen aus den spezifizierten `CGrounding` Elementen im Chor Model

procedure BUILD
  $grndList : List \leftarrow$ new $List$
  // $\mathcal{G} = \{cg | cg : CGrounding_{chor}\}$
  $\mathcal{G} \leftarrow chormodel.groundings$
  for all $cg \in \mathcal{G}$ **do**
    $grnd : Grounding_{grnd} \leftarrow$ new $Grounding_{grnd}$
    $grnd.topology \leftarrow$BUILDTOPOLOGYQNAME
    // $\mathcal{M} = \{cmlg | cmlg : CMessageLinkGrounding_{chor}\}$
    $\mathcal{M} \leftarrow cg.cMessageLinkGroundings$
    for all $cmlg \in \mathcal{M}$ **do**
      $ml : MessageLink_{grnd} \leftarrow$CREATEMESSAGELINK($cmlg$)
      $grnd.messageLinks \cup \{ml\}$
      if $cmlg.cMessageLink.bindSenderTo \neq null$ **then**
        $pRef : ParticipantRef_{grnd} \leftarrow$CREATEPARTICIPANTREF($cmlg$)
        $grnd.participantRefs \cup \{pRef\}$
      end if
    end for
    // $\mathcal{P} = \{cPrefg | cPrefg : CParticipantRefGrounding_{chor}\}$
    $\mathcal{P} \leftarrow cg.cParticipantRefGroundings$
    for all $cPrefg \in \mathcal{P}$ **do**
      $pRef : ParticipantRef_{grnd} \leftarrow$CREATEPARTICIPANTREF($cPrefg$)
      $grnd.participantRefs \cup \{pRef\}$
    end for
    // $\mathcal{C} = \{cSetg | cSetg : CorrelationSetGrounding_{chor}\}$
    $\mathcal{C} \leftarrow cg.correlationSetGroundings$
    for all $cSetg \in \mathcal{C}$ **do**
      // $\mathcal{L} = \{prop | prop : Property_{grnd}\}$
      $\mathcal{L} \leftarrow$CREATEPROPERTIES($cSetG$)
      for all $prop \in \mathcal{L}$ **do**
        $grnd.properties \cup \{prop\}$
      end for
    end for
    $grndList \cup \{grnd\}$
  end for
  return $grndList$
end procedure

FlowBuilder Algorithmen

Für das `FlowActivityLink` Element aus Chor Model, generieren wir den Namen passend zum ‹ `Flow` › Kontext, da dieser Name für das `Link` Element gesetzt wird. In Algorithmus 4.17 ist die Vorgehensweise zu sehen.

Für jedes `FlowActvitiyLink` Element müssen einerseits die `Source` und `Target` Elemente der Quell- und Zielaktivität erzeugt werden und andererseits, ein `Link` Element im richtigen `Flow` Container erzeugt werden. Den `Flow` Container finden wir mit der *parentOf*() Funktion denn für jede Aktivität in einem ‹ `Flow` › gilt, dass der direkte Elternknoten der Container sein muss. Daher reicht die einmalige Anwendung von *parentOf*() auf die `sourceActivity` bzw. `targetActivity`.

Algorithmus 4.17 getNormalizedName speziell für das `FlowActivityLink` Element

```
procedure getNormalizedName(e : FlowActivityLink_chor)
    name : String ← e.name
    sourceId : String ← e.sourceActivity.id
    targetId : String ← e.targetActivity.id
    if name = null then
        name ←'link_'+sourceId+'_to_'+targetId+'_'+e.hashCode
    end if
    return name
end procedure
```

Algorithmus 4.18 Erzeugt `Link`, `Source` und `Target` Elemente im PBD Model

procedure BUILD
// $\mathcal{L} = \{clink | clink : FlowActivityLink_{chor}\}$
$\mathcal{L} \leftarrow chormodel.flowActivityLinks$
for all $clink \in \mathcal{L}$ **do**
$srcActivity : Activity_{pbd} \leftarrow clink.sourceActivity$
$tgtActivity : Activity_{pbd} \leftarrow clink.targetActivity$
$src : Source_{pbd} \leftarrow$ new $Source_{pbd}$
$src.linkName \leftarrow$getNormalizedName($clink$)
$src.transitionCondition \leftarrow clink.transitionCondition$
$srcActivity.sources \cup \{src\}$
$tgt : Target_{pbd} \leftarrow$ new $Target_{pbd}$
$tgt.linkName \leftarrow$getNormalizedName($clink$)
$tgtActivity.targets \cup \{tgt\}$
$srcFlow : Flow_{pbd} \leftarrow null$
$tgtFlow : Flow_{pbd} \leftarrow null$
$parentElement : Element_{pbd} \leftarrow parentOf(srcActivity)$
if $typeOf(parentElement) \neq null \wedge parentElement = Flow_{pbd}$ **then**
$srcFlow \leftarrow parentElement$
end if
$parentElement \leftarrow parentOf(tgtActivity)$
if $typeOf(parentElement) \neq null \wedge parentElement = Flow_{pbd}$ **then**
$tgtFlow \leftarrow parentElement$
end if
// $srcActivity$ und $tgtActivity$ beide im selben Flow Container
if $srcFlow \neq null \wedge tgtFlow \neq null \wedge srcFlow = tgtFlow$ **then**
$link : Link_{pbd} \leftarrow$ new $Link_{pbd}$
$link.name \leftarrow$getNormalizedName($clink$)
// $srcFlow$ und $tgtFlow$ sind die gleiche Instanz
$srcFlow.links \cup \{link\}$
end if
end for
end procedure

Transformer Algorithmen

Als Beispiel sehen wir in Algorithmus 4.19 das Vorgehen bei der Transformation vom Topology Model zum DOM Dokument. Diese Vorgehensweise ist für jedes unserer Modelle analog. Wir brauchen Zugriff auf die Metadaten aller Modellelemente und definieren daher die Methoden $attributesOf()$, $valueOf()$, $valuesOf()$, $nameOf()$ und $referencesOf()$.

attributesOf() gibt die Menge aller Attribute des übergebenen Modell Elements zurück. In unseren Modellen (Topology Model, Grounding Model und PBD Model) können Attribute einen bestimmten Datentyp haben oder auf andere Modell Elemente referenzieren. Daher benötigen wir die Methode *referencesOf*(), welche die Menge aller referenzierten Elemente zurückgibt. Der Wert eines Attributes kann entweder einfach oder mehrwertig sein. Die Methode *isMultiValued*() gibt *true* zurück, falls es sich um ein mehrwertiges Attribut handelt wie z. B. `selects` von `Participant`, sonst *false*. Mit *valueOf*() bekommen wir den gesetzten Wert eines Attributes zurück oder *null*, falls nicht gesetzt. Für den Fall eines mehrwertigen Attributs, benutzen wir *valuesOf*() und bekommen eine Menge zurück oder ebenfalls *null*, falls nicht gesetzt. Wenden wir *valueOf*() bzw. *valuesOf*() auf Referenzattribute an, ist das Verhalten analog nur dass die referenzierten Modell Elemente zurückgegeben werden. Mit der Methode *nameOf*() bekommen wir den Namen des gegeben Elements zurück. Übergeben wir z. B. das Attribut `selects`, gibt *nameOf*(`selects`) den Wert "selects" zurück. Die Methode *classname*() haben wir bereits in Abschnitt 4.5.2 auf Seite 52 definiert.

Algorithmus 4.19 Transformiert ein Topology Model Element mit all seinen Kind Elementen zu einem DOM Dokument

procedure MODELTODOM($doc : Document_{dom}, parentElement : Element_{dom}, modelElement : Element_{top}$)
  $elementName : String \leftarrow classname(modelElement)$
  $element : Element_{dom} \leftarrow doc.createElement(elementName)$
  // Wenn *parentElement* null ist, dann ist *element* das Wurzelelement
  if $parentElement = null$ **then**
    $doc.appendChild(element)$
  else
    $parentElement.appendChild(element)$
  end if
  HANDLEATTRIBUTES($doc, element, modelElement$)
  // $\mathcal{R}_{modelElement} = \{ref | ref$ ist Referenz von $modelElement\}$
  $\mathcal{R}_{modelElement} \leftarrow referencesOf(modelElement)$
  for all $ref \in \mathcal{R}_{modelElement}$ **do**
    if $valueOf(ref) \neq null \vee valuesOf(ref) \neq null$ **then**
      if $isMultiValued(ref)$ **then**
        $\mathcal{V}_{ref} \leftarrow valuesOf(ref)$
        for all $value \in \mathcal{V}_{ref}$ **do**
          MODELTODOM($doc, element, value$)
        end for
      else
        MODELTODOM($doc, element, valueOf(ref)$)
      end if
    end if
  end for
end procedure

Algorithmus 4.20 Transformiert die Attribute des gegebenen Topology Model Elements, zu Attributen des gegebenen DOM Elements

procedure HANDLEATTRIBUTES($doc : Document_{dom}, element : Element_{dom}, modelElement : Element_{top}$)
// $\mathcal{A}_{modelElement} = \{a | a$ ist Attribut von $modelElement\}$
$\mathcal{A}_{modelElement} \leftarrow attributesOf(modelElement)$
for all $a \in \mathcal{A}_{modelElement}$ **do**
if $valueOf(a) \neq null \vee valuesOf(a) \neq null$ **then**
$domAttribute : Attr_{dom} \leftarrow doc.createAttribute(nameOf(a))$
$attribValue : String$
if $isMultiValued(a)$ **then**
// $\mathcal{V}_a = \{value | value$ ist Wert des Attributs $a\}$
$\mathcal{V}_a \leftarrow valuesOf(a)$
for all $value \in \mathcal{V}_a$ **do**
$attribValue \leftarrow attribValue + value +$"
end for
$domAttribute.setValue(attribValue)$
else
$attribValue \leftarrow valueOf(a)$
$domAttribute.setValue(attribValue)$
end if
$element.setAttributeNode(domAttribute)$
end if
end for
end procedure

4.5.3 Generierung von BPEL Prozessen

Nachdem wir das Chor Model in die einzelnen DOM Dokumente, wie in Abschnitt 4.5.2 auf Seite 48 besprochen, transformiert haben, können wir diese Dokumente in BPEL Prozesse umwandeln. Dazu benutzen wir die bereits entwickelte Komponente *BPEL4ChorToBPEL* [Li10], auf welche wir in Abschnitt 5.1.6 auf Seite 105 genauer eingehen. Das Ergebnis dieser Komponente sind abstrakte BPEL Prozesse mit zugehörigen WSDL Definitionen, welche ebenfalls als DOM Dokumente vorliegen. Um ausführbare BPEL Prozesse als Endresultat zu erhalten, setzen wir noch einen weiteren Schritt dahinter und führen die Komponente *BasicExecutableCompletionTransformer* ein, welche die abstrakten BPEL Prozesse als Eingabe bekommt und eine „basic executable completion" durchführt. In Abbildung 4.11 sehen wir die Komponenten und deren Dokumentenfluss. Der *ChorToBPELTransformer* stellt die Rahmenkomponente dar, welche die DOM Dokumente aus dem Transformationsschritt an die *BPEL4ChorToBPEL* und *BasicExecutableCompletionTransformer* verteilt. Die von *BPEL4ChorToBPEL* ausgegebenen

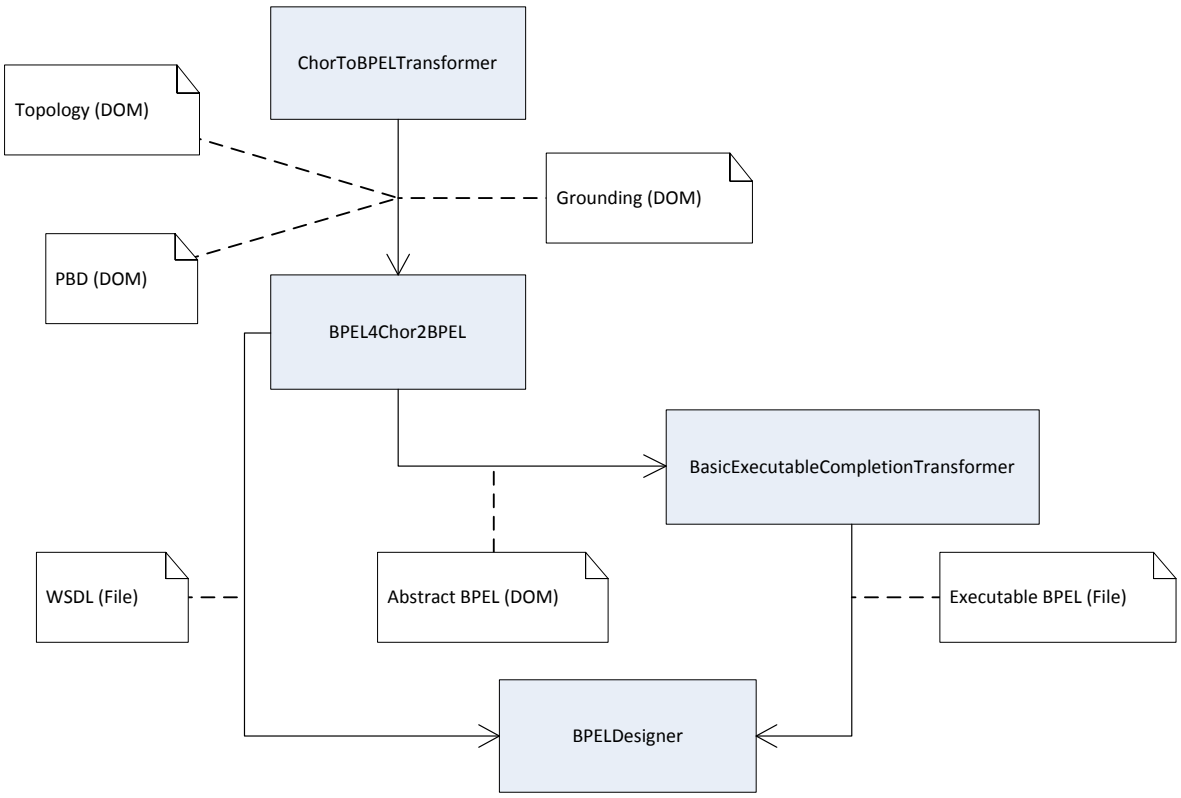

Abbildung 4.11: Dokumentenfluss zwischen Komponenten für die Umwandlung zu ausführbaren Prozessen

WSDL Dokumente schreiben wir auf den Datenträger in den passenden Projektordner. Die BPEL Dokumente senden wir an den *BasicExecutableCompletionTransformer*, welcher die Schritte, gelistet in Tabelle 4.1, durchführt. Zuletzt werden die modifizierten BPEL Dokumente ebenfalls auf den Datenträger geschrieben und können letztendlich mit dem *BPEL Designer* weiter bearbeitet werden.

4.5.4 Grafisches Konzept

In Abschnitt 4.5.1 auf Seite 35 haben wir das Datenmodell für den Editor vorgestellt, welches wir nun visualisieren. In Abbildung 4.12 ist die Editor Oberfläche zu sehen. Der mittlere Teil wird zur Darstellung der Elemente benutzt und in der *Palette* befinden sich die Werkzeuge dafür. Im unteren Bereich befindet sich die *Property View*, welche für das aktuell selektierte Element die Eigenschaften anzeigt. Über diese *Property View* lassen sich viele Einstellungen vornehmen, die nicht durch die Werkzeuge in der *Palette* eingestellt

werden können. Dabei sind die Einstellungen immer spezifisch für das gerade selektierte Element und werden durch *Tabs* in verschiedene Kategorien eingeteilt. In der *Toolbar* befinden sich Funktionen Laden, Speichern, Exportieren und Transformieren. Mit Hilfe der Werkzeuge können wir nun die Komponenten unserer Choreographie modellieren. Wir haben im Abschnitt 4.5.1 auf Seite 35 die Konvention getroffen dass alle grafischen Elemente, welche wir im Editor platzieren, eine eigene Entität bzw. Klasse im Chor Model sind was wir hier nun ausnutzen, denn für jede Entität gibt es ein Werkzeug in der *Palette*. In Abbildung 4.13 wurde ein `CParticipant` mit seinem zugehörigen `Process` Element platziert, welches zunächst keine Aktivitäten beinhaltet. In Abbildung 4.14 haben wir ein `CParticipantSet` Element dazu platziert sowie Aktivitäten mit den entsprechenden Werkzeugen modelliert. Die `CMessageLink` Elemente zwischen den Kommunikationsaktivitäten wurde ebenfalls gezogen und dazu noch ein `CParticipantRef` Element platziert, welches auf diesen `CMessageLink` zeigt, über welchen es ausgetauscht wird. Die Zugehörigkeit dieser Referenz zu seinem `CMessageLink` kann nur über die *Property View* eingestellt werden. Abbildung 4.15 zeigt einen `CParticipant` mit seinem `Process` und einer `Flow` Aktivität. Darin sind weitere Aktivitäten platziert, zwischen welchen `FlowActivityLink` Elemente gezogen wurden.

Als nächstes betrachten wir die grafischen Elemente der *Property View*. In Abbildung 4.16 sehen wir die Inhalte der *base* Kategorie von `CParticipant` und `CParticipantSet` sowie die aufrufbaren Dialoge durch Betätigung der Buttons. Im Textfeld "name:" lässt sich der Wert des *`name`* Attributes einstellen. Die Combobox "scope:" zeigt eine dropdown Liste aller verfügbaren `Scope` Aktivitäten dieser Choreographie. Generell gilt für alle folgenden Abbildungen von *Property View* Elementen, dass Felder mit einem Pfeil nach unten immer Comboboxen darstellen. Das "selects:" Feld ist eine Liste von selektierten `CParticipant` bzw. `CParticipantSet` Elementen. Einträge können über den "add" Button hinzugefügt werden. Dabei öffnet sich der Dialog, auf welchen der vom Button ausgehende Pfeil verweist. Dieser "Participants" Dialog zeigt eine Liste aller verfügbaren `CParticipant` und `CParticipantSet` Elemente dieser Choreographie mit Ausnahme des aktuell selektierten Elements. Durch Auswahl eines Eintrages und Klick des "select" Buttons, wird dieser Eintrag in die "selects:" Liste übernommen.

Abbildung 4.17 zeit die Inhalte der *participants* Kategorie von `CParticipantSet`. In die "participants:" Liste lassen sich über den "add" Button neue Elemente hinzufügen. Bei Klick auf diesen Button, öffnet sich der "Create new Participant" Dialog, über welchen sich das *`name`* Attribut einstellen lässt. Bei Klick auf den "save:" Button, wird ein neues `CParticipant` Element erstellt und der "participants:" Liste hinzugefügt. Die Eigenschaften dieses neuen Eintrages lassen sich über den "configure selection:" Button einstellen. Durch Klick darauf öffnet sich der "participant" Dialog. Die Inhalte und Funktionen dieses Dialogs sind identisch zu denen der *base* Kategorie von `CParticiapnt` nur dass sich zusätzlich noch der Wert des *`containment`* Attributes einstellen lässt.

Die Inhalte der *base* und *participants* Kategorien von `CMessageLink` sehen wir in Abbildung 4.18. In der *base* Kategorie ist im Feld "name:", der eindeutige Name für diesen Link einstellbar. Mit "messageName:" lässt sich ein zusätzlicher Name für diesen Link vergeben. Die *participants* Kategorie zeigt die Elemente zum Einstellen von "sender:", "senders:", "receiver:" und "bindSenderTo:" wobei dies alles Comboboxen sind. Die Einträge der dropdown Liste von "bindSenderTo:" zeigt entweder nur den `CParticipant`, den dieser `CMessageLink` als "sender:" hat oder, im Fall dass "sender:" ein `CParticipantSet` ist, alle `CParticipant` Kind Elemente. Der "ParticipantRef List:" können Eintrage über den "add" Button hinzugefügt werden wobei der zugehörige Dialog natürlich nur auswählbare Eintrage zeigt, wenn `CParticipantRef` Elemente in dieser Choreographie platziert wurden.

Abbildung 4.19 zeigt die Inhalte der *base* Kategorie von `CParticipantRef`. Die Combobox "participant:" enthält eine Liste von allen `CParticipant` Elementen dieser Choreographie womit die zu übertragende Referenz festgelegt werden kann. Zudem lässt sich für diese Referenz noch ein zusätzlicher Name vergeben.

Abbildung 4.20 zeigt die *base* Kategorie der `Process` Elemente, in welche sich die *`name`* und *`targetNamespace`* Attribute des Prozesses festlegen lassen. Unter *correlations* und *messageExchanges* können jeweils über den "add" Button neue `CorrelationSet` bzw. `MessageExchange` Elemente angelegt werden. Das *`properties`* Attribut eines `CorrelationSet` Elements lässt sich durch Auswahl eines Elementes in der "CorrelationSets:" Liste im Textfeld "properties:" editieren. Für die `Scope` Aktivität gibt es ebenfalls *correlations* und *messageExchanges* Kategorien die analog funktionieren.

Die Inhalte der *base* und *groundings* Kategorien von `Choreography` sehen wir in Abbildung 4.21. Die *base* Kategorie beinhaltet die selbe Funktionalität wie diese von `Process`. Unter der *groundings* Kategorie können mit dem "add" Button neue `CGrounding` Elemente für diese Choreographie angelegt werden. Wird ein Eintrag der "groundings:" Liste selektiert, können für die in dieser Choreographie definierten `CMessageLink`, `CorrelationSet` und `CParticipantRef` Elemente, groundings konfiguriert werden. Dafür öffnet sich beim Klick auf den jeweiligen "configure" Button der entsprechende Dialog. In Abbildung 4.22 können für alle modellieren `CMessageLink` Elemente, welche in der "MessageLinks:" Liste aufgeführt sind, `CMessageLinkGrounding` Elemente angelegt werden. Selektiert man einen Eintrag der "MessageLinks:" Liste können rechts, in der "Configure messageLink.name" Gruppe, die grounding spezifischen Angaben für diesen selektieren Link gesetzt, und mit dem "save" Button abgespeichert werden. Dabei wird entweder ein neues `CMessageLinkGrounding` Element angelegt oder, falls schon vorhanden, die neuen Werte für das bereits bestehende übernommen. Abbildung 4.23 zeigt die beiden Dialoge für `CorrelationSet` und `CParticipantRef` Elemente. Für letzteres ist die Funktionalität dem Dialog für `CMessageLink` Elemente äquivalent. Der Dialog für `CorrelationSet` beinhaltet die zusätzliche Liste "Properties of correlationSet.name", worin alle Einträge des *`properties`* Attributes untereinander gelistet werden. Dies ist notwendig, da für jeden Eintrag ein eigenes Grounding angegeben werden muss. Selektiert man also einen Eintrag

dieser Liste, kann das entsprechende `PropertyGrounding` Elemente dafür angelegt bzw. abgeändert werden.

Die Inhalte der *base* Kategorie sind für alle Aktivitäten gleich. Sie beinhalten "name:" und "id:" Felder zum editieren der entsprechenden Attribute. Je nach Aktivität kommen aber noch spezifische Elemente hinzu wie z. B. für `ForEach` eine Checkbox für parallele Ausführung. In Abbildung 4.24 sind die Elemente der *base* und *iteration* Kategorien von `ForEach` zu sehen. In der *iteration* Kategorie sind in der "Iterator Set:" Combobox alle `CParticipantSet` Elemente dieser Choreographie gelistet. Die "Iterator Value:" Combobox beinhaltet alle `CParticipant` Kind Elemente des ausgewählten `CParticipantSet` Elements der anderen Combobox. Für `Invoke`, `Receive`, `Reply` und `OnMessage` gibt es jeweils noch die *variables* Kategorie, in welcher sich eine Checkbox zum setzen der Opaque Variable befindet. Ebenfalls gemeinsam haben sie die *correlations* Kategorie, in welcher sich bereits definierte `CorrelationSet` Elemente dieser Aktivität zuweisen lassen. Für `Receive`, `Reply` und `OnMessage` existiert zusätzlich noch die *messageExchanges* Kategorie, in welcher sich eine Combobox mit allen in `Process` definierten `MessageExchange` Elementen befindet. Diese dropdown Liste enthält zusätzlich noch `MessageExchange` Elemente, welche im dieser Aktivität übergeordneten `Scope` definiert sind.

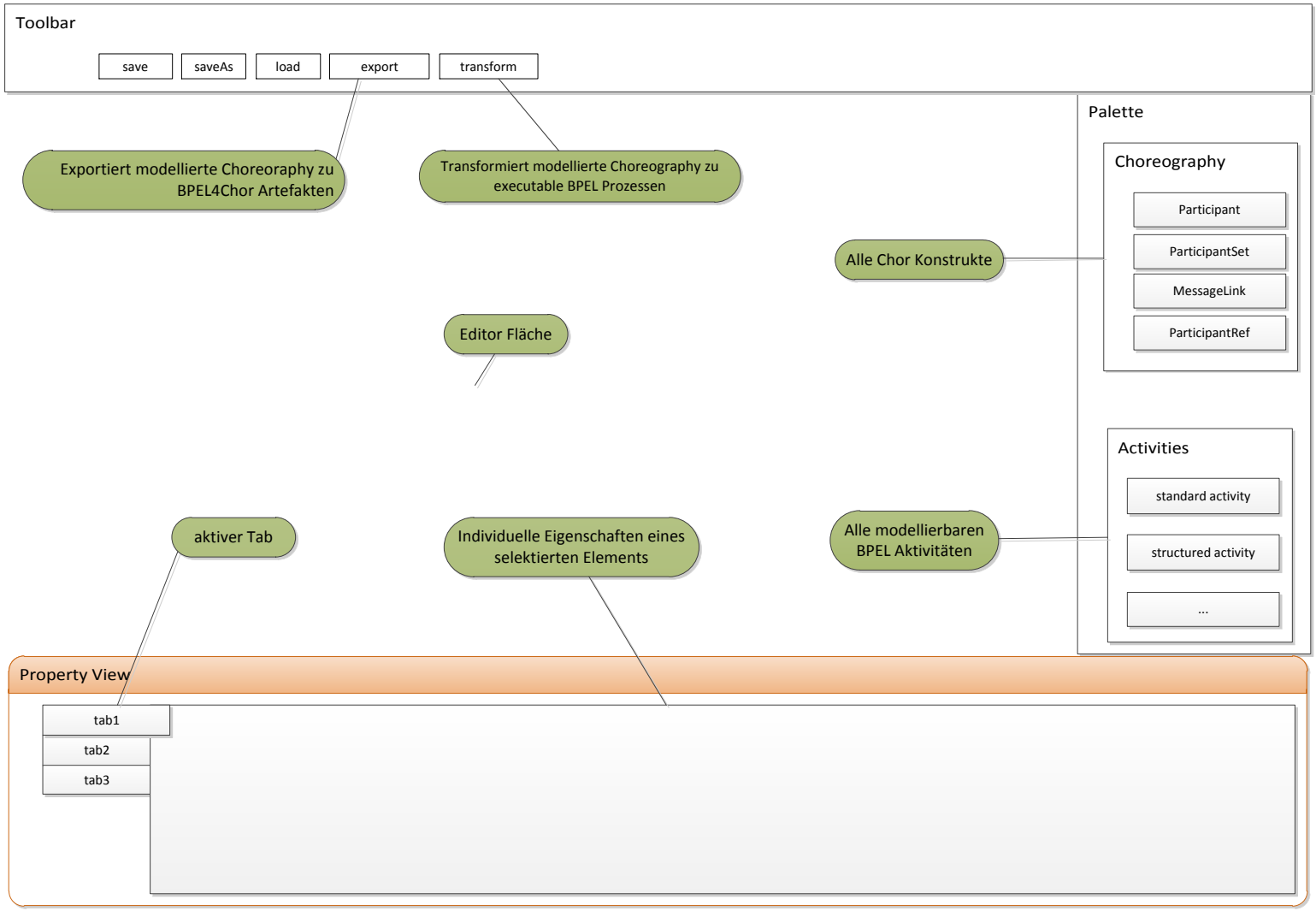

Abbildung 4.12: Konzept der Editor Oberfläche

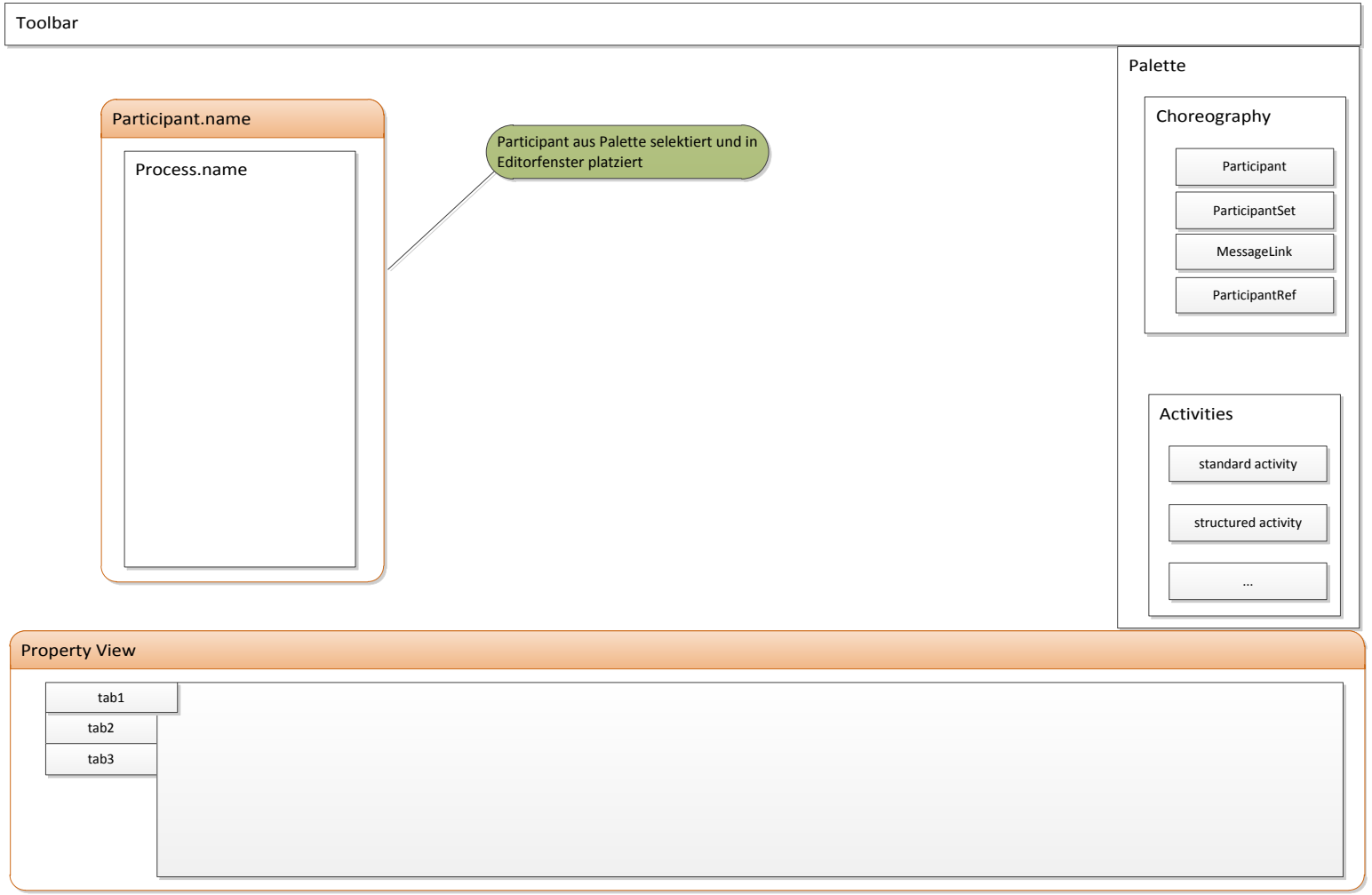

Abbildung 4.13: `CParticipant` mit leerem `Process`

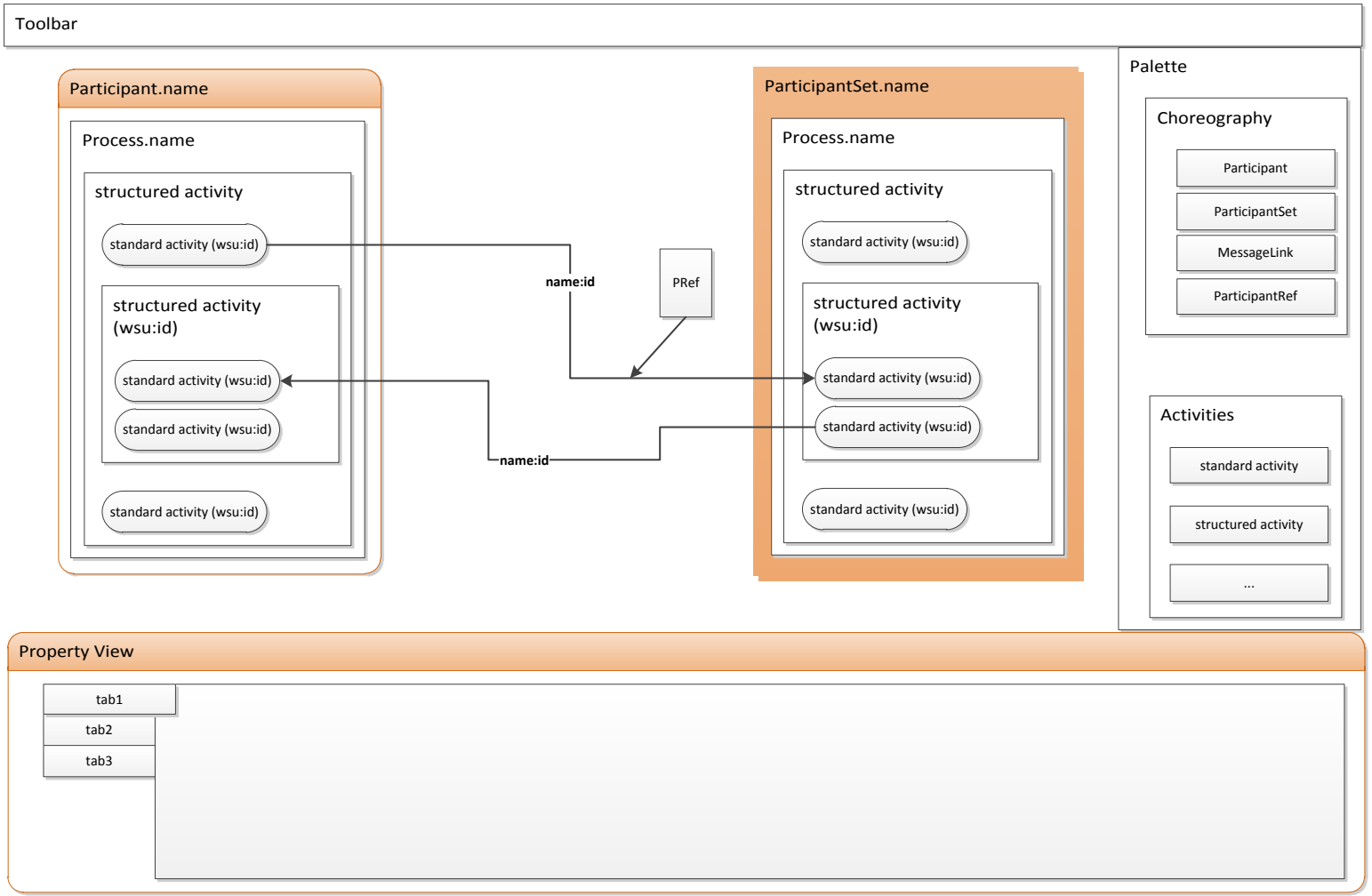

Abbildung 4.14: Participants mit Aktivitäten, Message Links und einer Participant Referenz

Abbildung 4.15: `Flow` mit Aktivitäten und `FlowActivityLink` Elementen

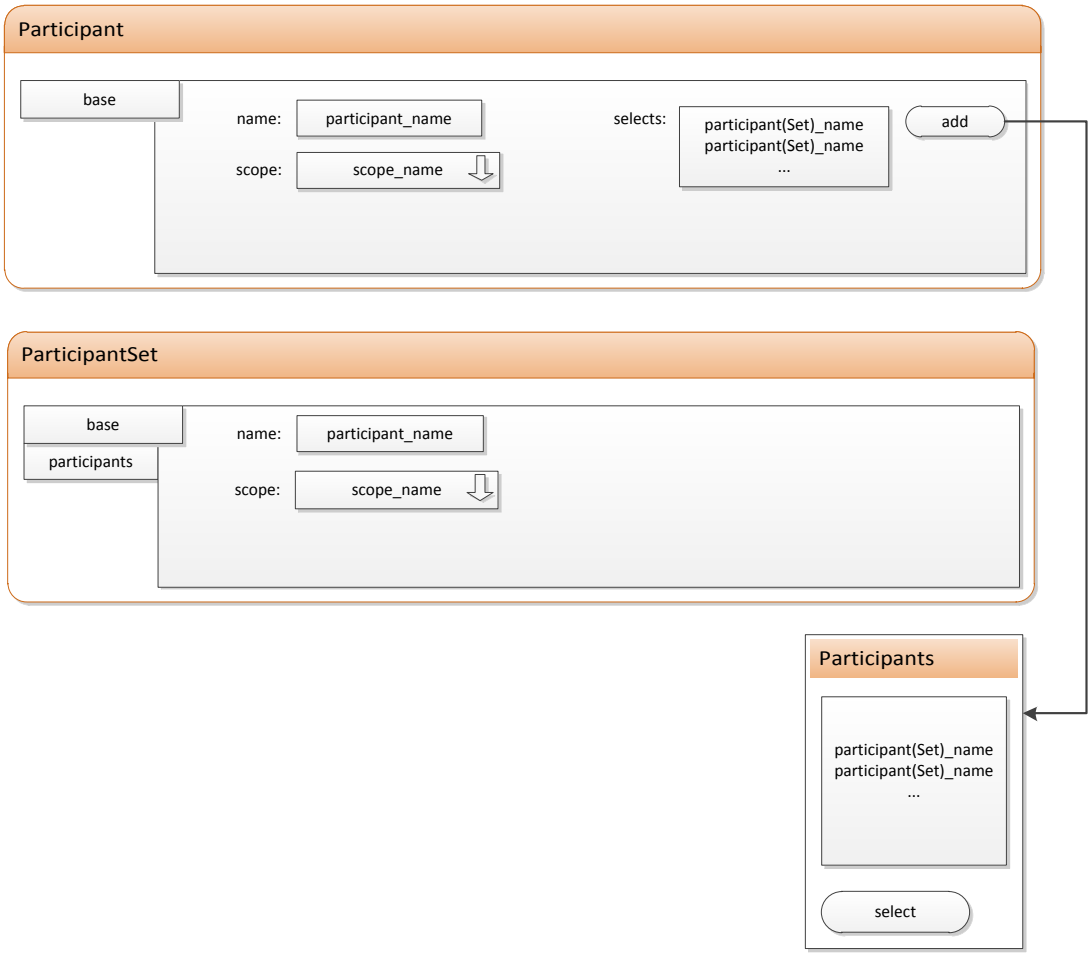

Abbildung 4.16: Elemente und Dialoge der *base* Kategorie von `CParticipant` und `CParticipantSet`

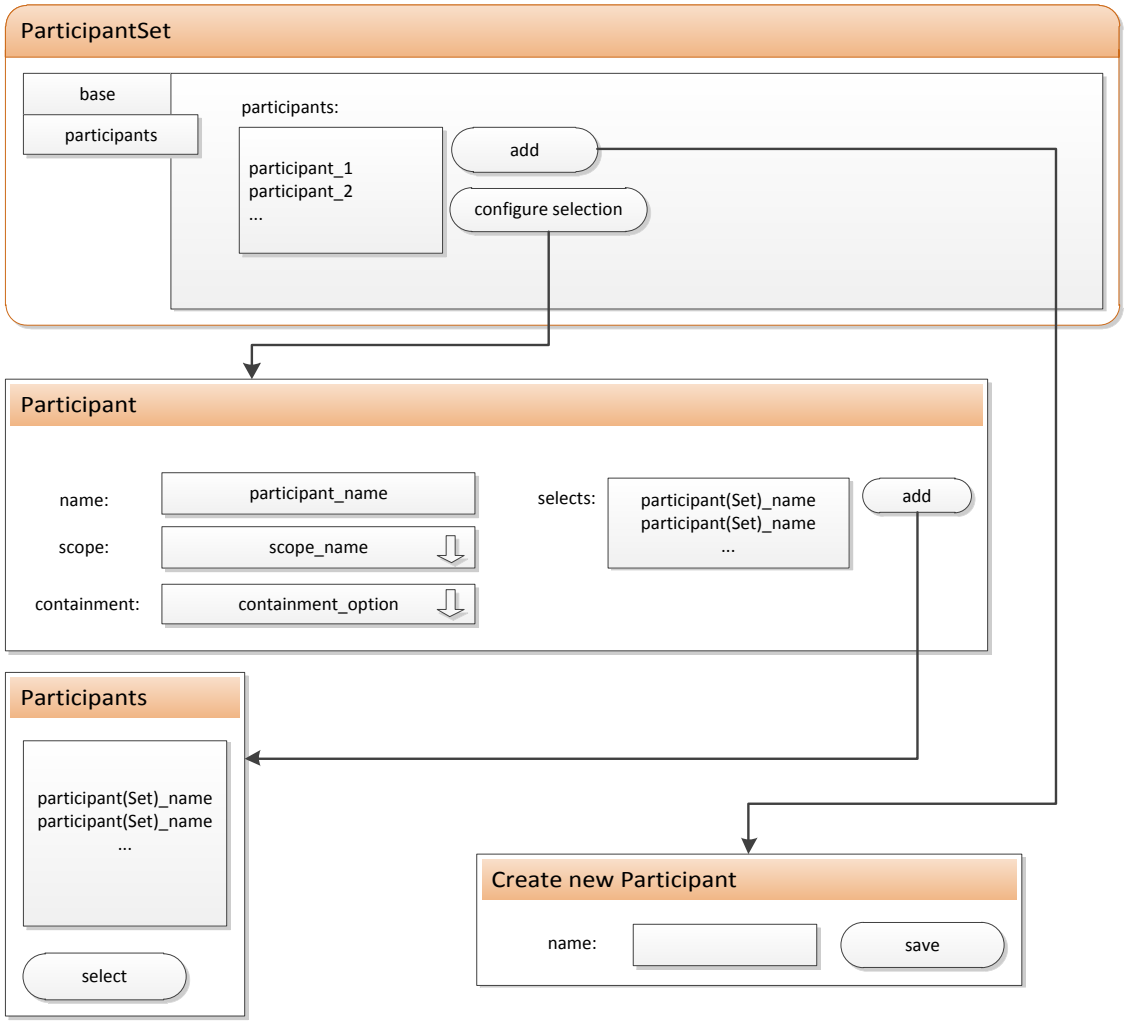

Abbildung 4.17: Elemente und Dialoge der *participants* Kategorie von `CParticipantSet`

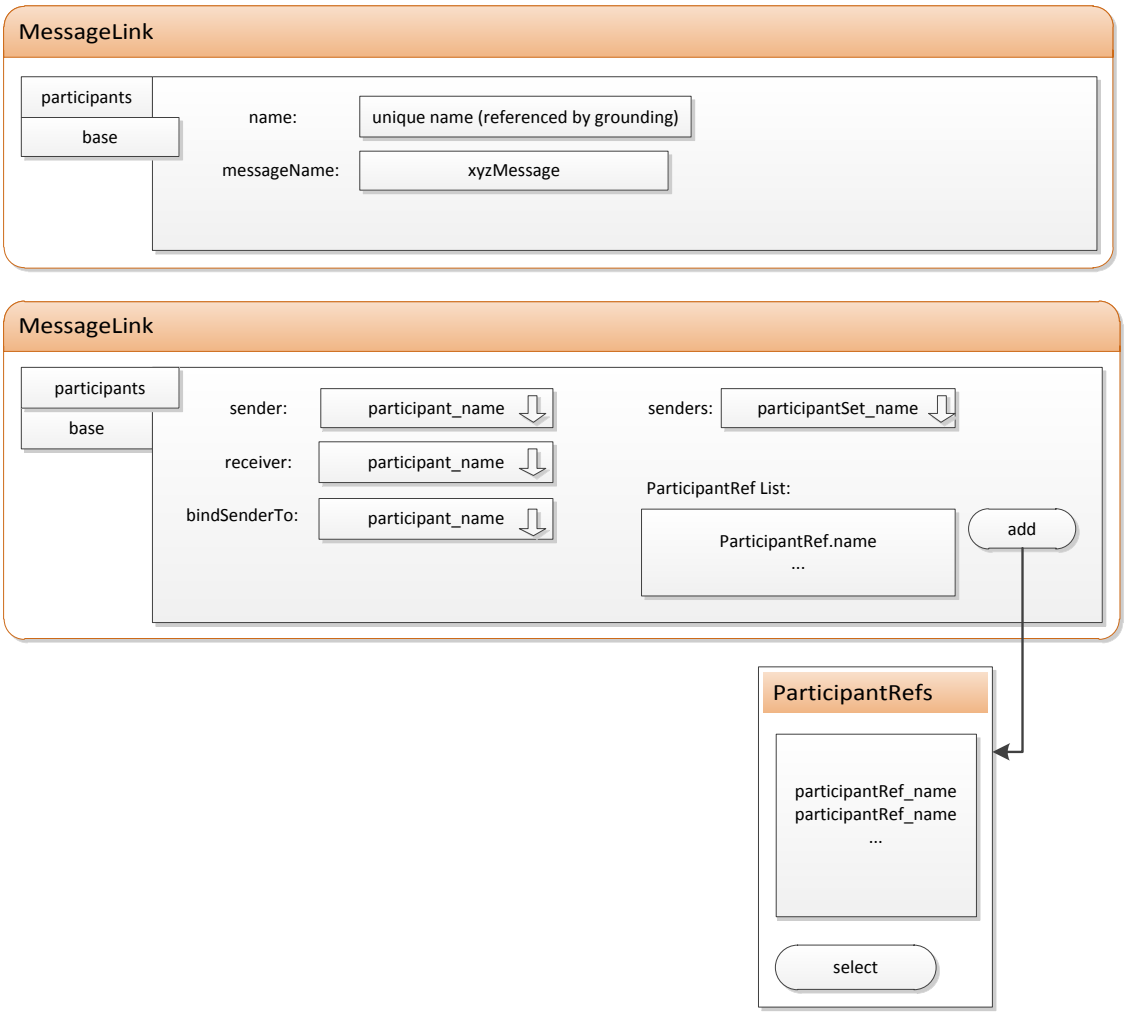

Abbildung 4.18: Elemente und Dialoge der *base* und *participants* Kategorien von `CMessageLink`

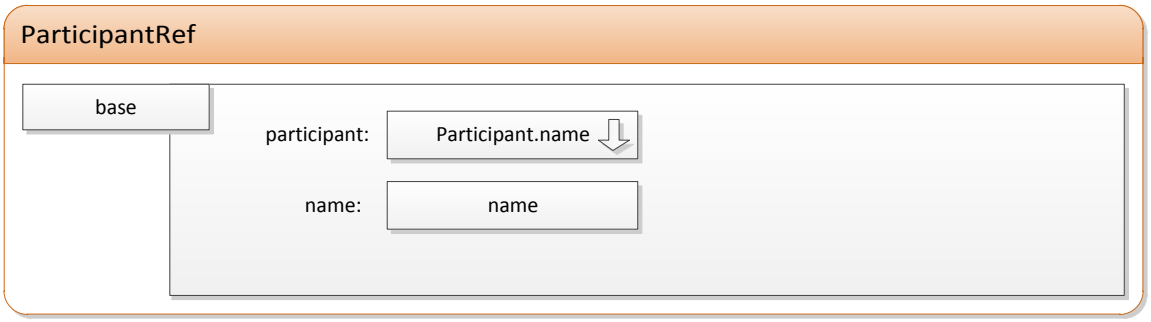

Abbildung 4.19: Elemente der *base* Kategorie von `CParticipantRef`

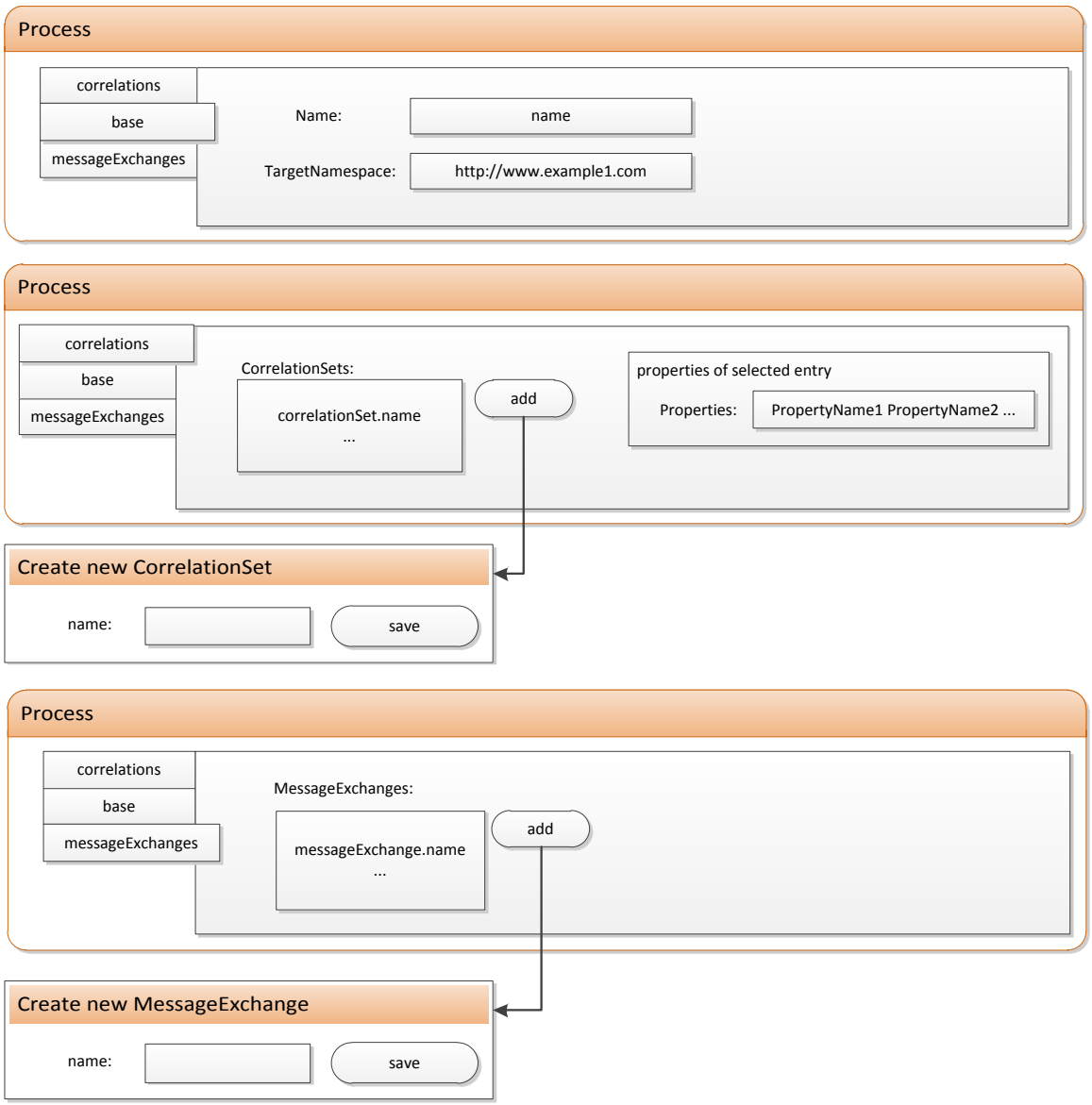

Abbildung 4.20: Elemente und Dialoge der *correlations*, *base* und *messageExchanges* Kategorien von `Process`

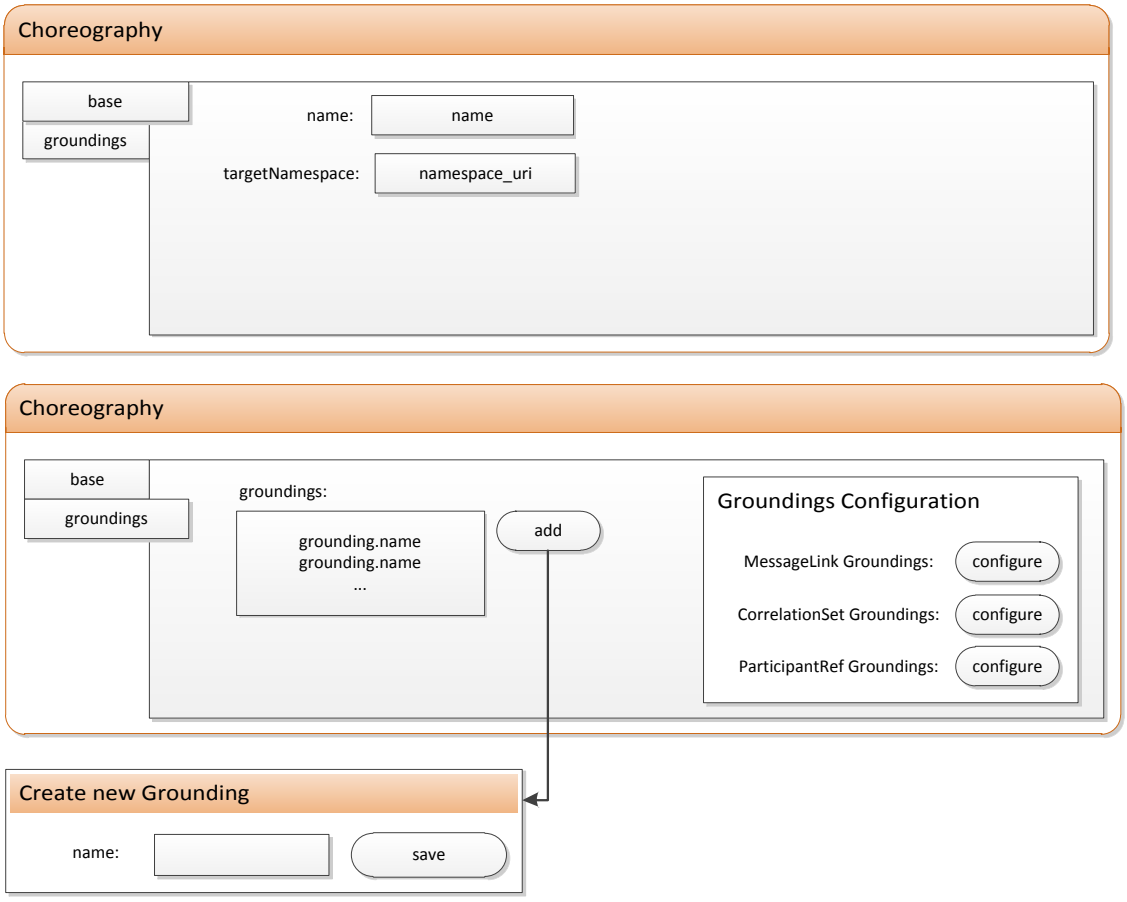

Abbildung 4.21: Elemente der *base* und *groundings* Kategorien von `Choreography`

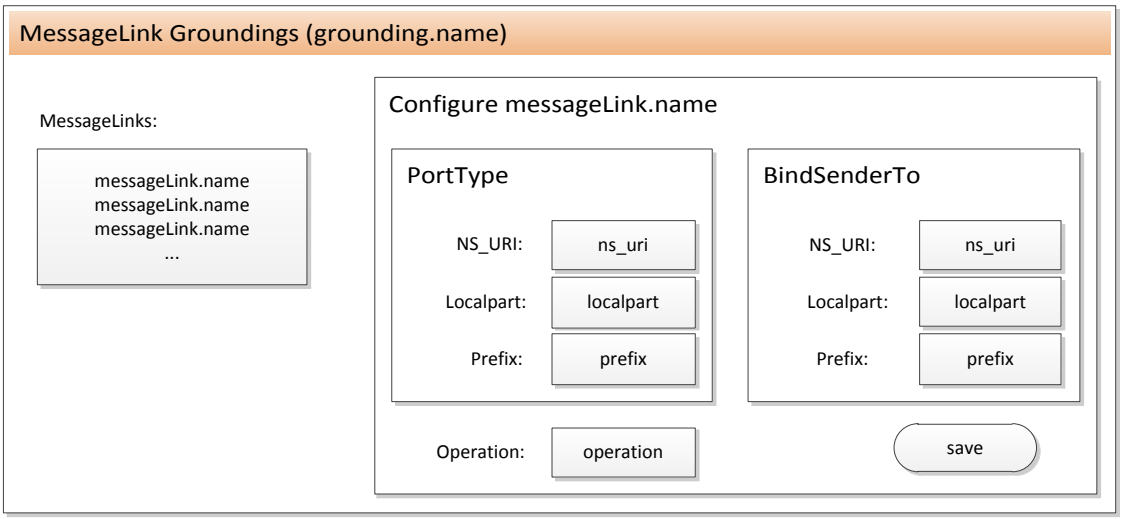

Abbildung 4.22: Dialog für die Konfiguration von `CMessageLinkGrounding` Elementen

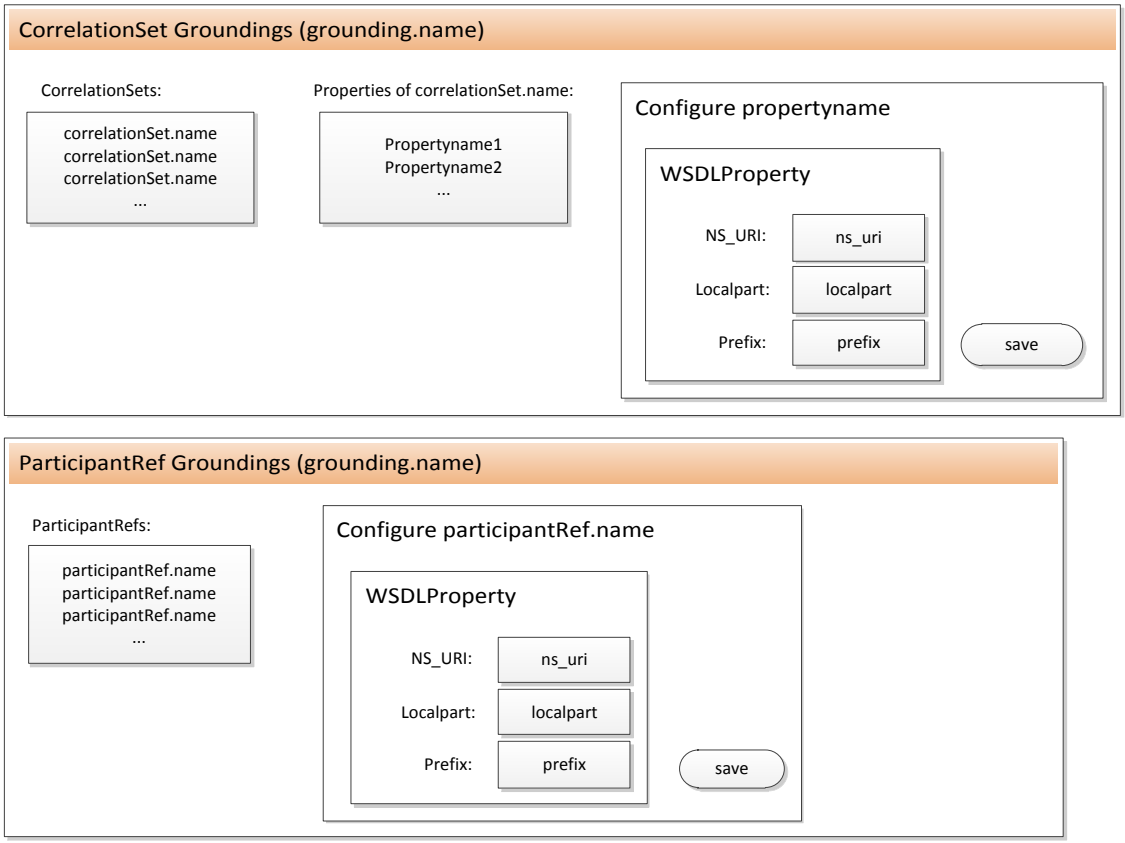

Abbildung 4.23: Dialog für die Konfiguration von `CorrelationSetGrounding` und `CParticipantRefGrounding` Elementen

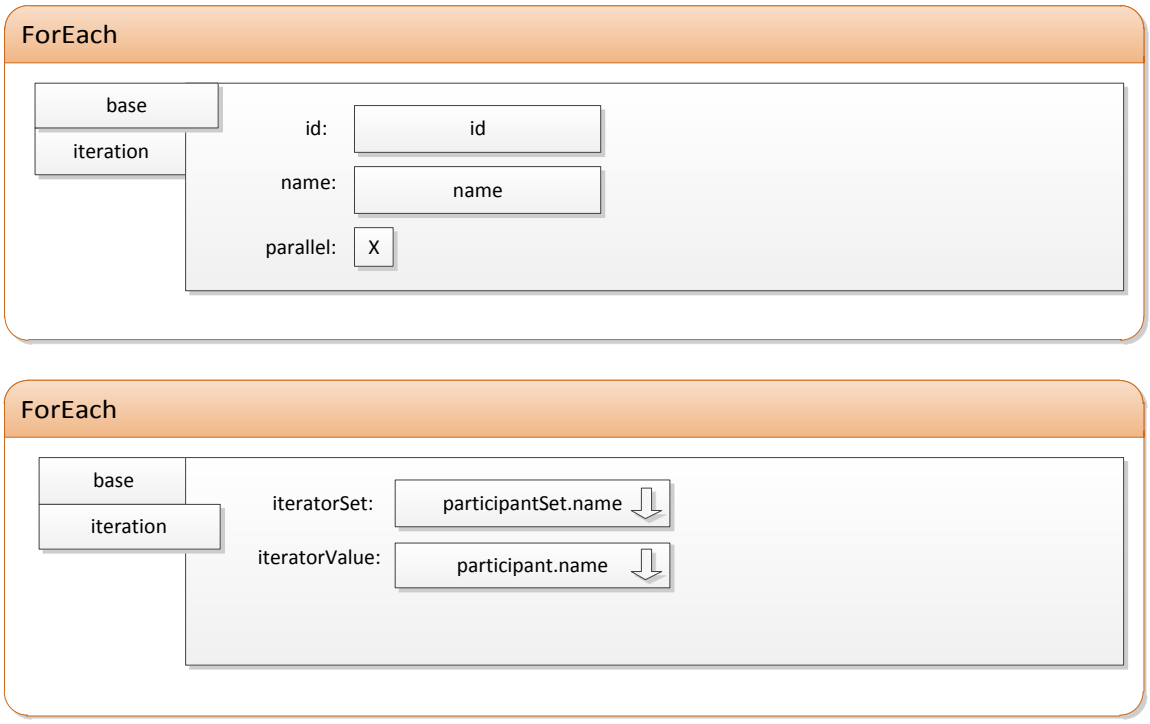

Abbildung 4.24: Elemente der *base* und *iteration* Kategorien von `ForEach`

5 Implementierung

Wir haben uns dafür entschieden, den Choreographie Editor als Eclipse Plugin zu realisieren. Dies liegt zum einen daran, dass der *BPEL Designer* ebenfalls als Eclipse Plugin realisiert wurde und dieser nach wie vor zum Bearbeiten der BPEL Prozesse verwendet wird – andererseits bietet die Eclipse Platform diverse Frameworks und vorgefertigte Komponenten um einen grafischen Editor schneller und einfacher zu entwickeln, als wenn alles von Grund auf neu implementiert werden müsste. Im Folgenden betrachten wir zuerst die Technologien, welche zur Implementierung benutzt werden und schauen uns dann einige Details an, wie diese zum Einsatz kommen.

5.1 Verwendete Technologien

Die Technologien werden im folgenden eher zusammenfassend, mit Schwerpunkt auf die in der Implementierung verwendeten Features, beschrieben.

5.1.1 Eclipse

Die Folgenden Beschreibungen zu Eclipse und seinen Komponenten basieren auf dem Buch [Ste11] von Steinberg und der Eclipse Dokumentation für Version 3.6 [ecl10], welche wir für die Implementierung benutzen.

Eclipse ist eine Open Source Integrationsplattform für Softwaretools sowie eine Entwicklungsumgebung für Java. Das Kernprojekt bietet ein generisches Framework für die Integration dieser Tools und andere Projekte erweitern dieses Framework um spezifische Tools und Entwicklungsumgebungen zu erstellen. Die Projekte in Eclipse sind in Java implementiert und laufen damit auf allen Betriebssystemen, zu welchen es Java Virtual Machines gibt. Eclipse besteht aus vielen Projekten wobei die Hauptbestandteile in den Projekten *Eclipse Project*, *Modeling Project*, *Tools Project* und *Technology Project* realisiert sind.

Das *Eclipse Project* unterstützt die Entwicklung einer Plattform oder eines Frameworks und ist in vier Unterprojekte aufgeteilt, die zusammen alle Features bieten um Eclipse basierte Tools zu entwickeln. *Equinox* ist das Projekt, welche das Komponentenmodell, auf welchem Eclipse

basiert, zur Verfügung stellt. Es implementiert die *OSGi R4 core Framework Spezifikation*[1]. Das *Platform* Projekt bietet Frameworks und Services für die Integration von Tools und die Entwicklung von Anwendungen und wurde auf Basis der *OSGi Service Platform* implementiert. Die *Java Development Tools* bieten eine umfangreiche Java Entwicklungsumgebung und werden selbst zur Entwicklung des *Eclipse Project* benutzt. Um die Entwicklung von Plugins in Eclipse zu unterstützen, bietet das *Plug-in Development Environment* Projekt unter anderem verschiedene Editoren und Mechanismen zu Registrierung von Plugin Erweiterungen.

Das *Modeling Project* bietet modellbasierte Entwicklungstechnologien die als Basis das Eclipse Modeling Framework (EMF, siehe 5.1.2 auf Seite 86) haben. Weitere Technologien, welche auf EMF aufbauen sind Model Transformationen, Datenbankintegration und die Generierung von grafischen Editoren. Somit bietet dieses Projekt für unsere Implementierung die zentralen Technologien, welche noch durch das Graphical Editing Framework (GEF, siehe Abschnitt 5.1.3 auf Seite 92) und das Graphical Modeling Framework (GMF, siehe Abschnitt 5.1.4 auf Seite 95) aus dem *Tools Project* ergänzt werden.

Im *Technology Project* landen Unterprojekte welche experimenteller Natur sind und noch am Anfang stehen. Sollten sie sich weiterentwickeln und reifen, werden sie in andere Projekte verschoben.

In Abbildung 5.1 sehen wir die Architektur der Eclipse Platform und ihre Strukturierung in Subsysteme. Das *Help* System definiert sogenannte *Extension points*, auf welche wir gleich eingehen werden, um Benutzerhilfen zu realisieren und *Team* bietet Verwaltung und Versionierung von Ressourcen für Programmierung im Team.

Plugins

Komponenten werden in Eclipse „Plugin“ (*Bundle* in OSGi) genannt und die *Platform* besteht selber aus mehreren Plugins. Sie ist verantwortlich für das Installieren, Entfernen, Auffinden ("discovering") und Ausführen von Plugins ohne das Eclipse neu gestartet werden muss. Diese Plugin Einheiten beinhalten alles was nötig ist, um die darin realisierte Komponente auszuführen. Neben dem obligatorischem Java Code sind unter anderem zwei Manifest Dateien enthalten, welche wir für unsere Implementierung anpassen müssen. Im Plugin Unterordner *META-INF* befindet sich die *MANIFEST.MF* Datei, welche unter anderem das Plugin identifiziert und Abhängigkeiten definiert. Dazu gehören die Abhängigkeiten von anderen Plugins und die exportierten Pakete sprich, diese Pakete, welche für andere Plugins sichtbar sind und damit auch von jenen importiert werden können. Dies ist für unsere Implementierung zentral, da wir die Editor Komponente aus mehreren Plugins realisieren werden. Die zweite Manifest Datei befindet sich im Plugin Wurzelverzeichnis und ist mit *plugin.xml* benannt. Hier werden zum einen *Extension points* Definiert welche

[1] http://www.osgi.org/Specifications/HomePage

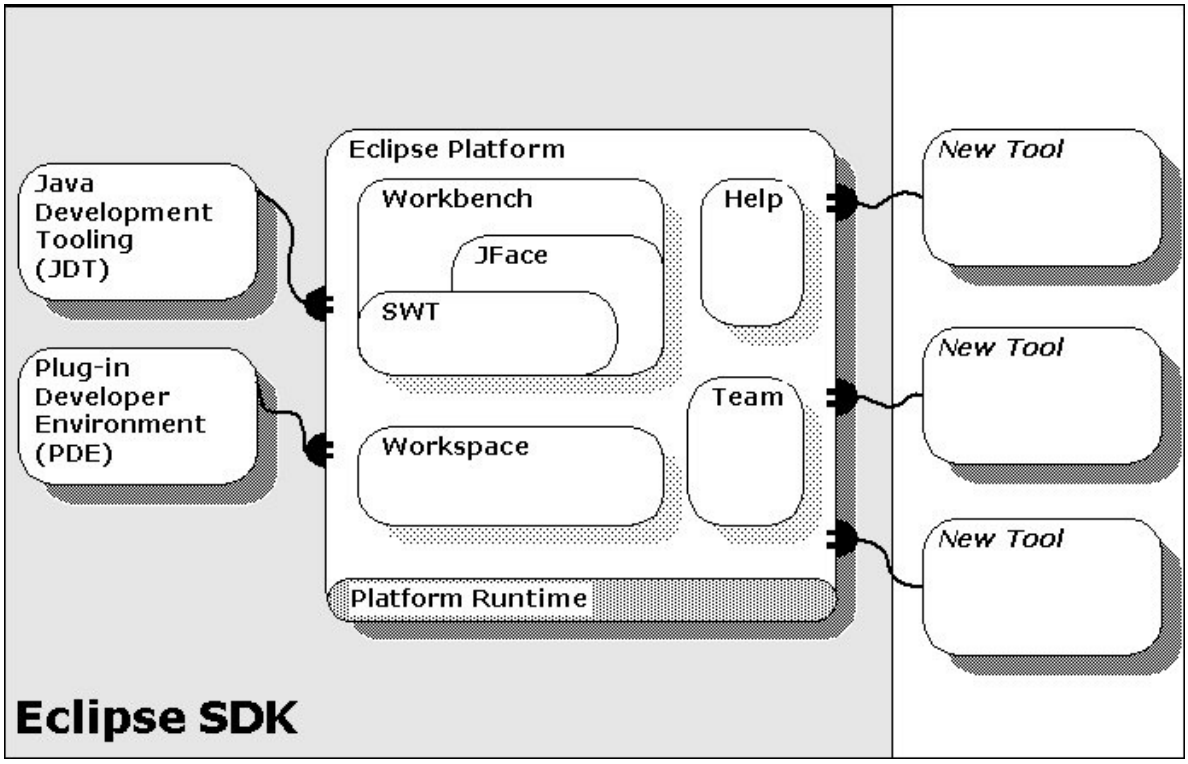

Abbildung 5.1: Eclipse Platform Architektur. Quelle: [ecl10]

beschreiben, was für eine Funktionalität dieses Plugin anderen Plugins zur Verfügung stellt und zum anderen *Extensions*, welche die von anderen Plugins zur Verfügung gestellten Funktionalitäten benutzen. *Extension points* und *Extensions* funktionieren sozusagen Hand in Hand und erlauben den Plugins sich untereinander beliebig zu kombinieren.

Workspace

Alle Eclipse Plugins arbeiten mit Dateien und Ordnern, jedoch werden diese zu *Ressourcen* durch die *Workspace* API abstrahiert. Eines dieser *Ressourcen* ist der Projekt Ordner, welcher der Top-Level Container jedes Eclipse Projekts ist und *Workspace* genannt wird. An jeder *Ressource* können *Listener* registriert werden, mit denen Änderungen an dieser *Ressource* überwacht werden. Es ist außerdem damit möglich beliebige Aktionen auszulösen, sollte sich eine *Ressource* verändert haben. Ein Beispiel dafür ist die standardmäßige Aktion, dass in Java Projekten bei Änderung und anschließender Speicherung einer Quellcode Datei, der Java Compiler neu aufgerufen wird [Sha04]. Wir machen uns die *Workspace* API für die Export- und Transformationsfunktion (siehe Abschnitt 4.5.2 auf Seite 48) zu nutze, wenn wir die transformierten Modelle in den Projektordner schreiben.

Workbench

Die *Workbench* ist die Benutzeroberfläche von Eclipse. In Abbildung 5.2 sehen wir die einzelnen Elemente eines *Workbench* Fensters, von welchen mehrere geöffnet werden können, wobei jedes dieser Fenster ein abgeschlossener Bereich aus *Editors* und *Views* ist. Innerhalb des *Workbench* Fensters befindet sich die *Page*, welche zur Gruppierung der einzelnen Teile dient. Möchte man visuelle Inhalte der *Workbench* hinzufügen, kann dies in Form von *Editors* und *Views* realisiert werden. *Views* werden üblicherweise zur Navigation durch hierarchische Daten verwendet wie z. B. der *Package Explorer* im linken Bereich der Abbildung 5.2 oder die *Properties View* im unteren Bereich, zum Anzeigen von Eigenschafen eines Objektes aus dem *Editor*. *Editors* werden dazu benutzt um Inhalte von Objekten anzuzeigen, verändern und abzuspeichern wie z. B. Java Dateien im Java Editor oder natürlich auch unser Chor Model im Choreographie Editor. *Menu Bar* und *Tool Bar* gehören zum *Workbench* Fenster und können beliebig mit Aktionen erweitert werden, die entweder nur für den gerade aktiven *Editor* oder auch global verfügbar sind.

Die grafischen Komponenten der *Views* und *Editors* – wie z. B. Textfelder, Buttons, Listen usw.– werden mit dem *Standard Widget Toolkit* (SWT) realisiert. SWT ist eine betriebssystemunabhängige Grafikbibliothek welche immer auf die nativen Widgets des Systems zurückgreift, außer wenn dieses keine Implementierung dafür bereit hält, in welchem Fall das betroffene Widget emuliert wird. Das *JFace* Toolkit erweitert SWT indem es, unter anderem für SWT Widgets wie Tabellen, spezielle *Viewer* Klassen bereit stellt, welche das Anzeigen und synchronisieren von Daten vereinfachen. In unserer Implementierung benutzen wir, für die Widgets der *Property View* und Dialoge, nur SWT.

5.1.2 Eclipse Modeling Framework

Die folgenden Beschreibungen basieren auf dem EMF Buch [Ste11] von Steinberg. Das *Eclipse Modeling Framework* (EMF) ist ein Framework und Programmcode Generator welches erlaubt, ein Modell – ausgehend von Java Interfaces, XML, UML Dateien oder direkt von Hand – zu erzeugen und davon eine passende Implementierung sowie einen Editor dafür zu generieren. Typischerweise werden für größere Anwendungen zuerst Modelle erstellt welche das was die Anwendung kann, in abstrakter Form darstellen. Diese Modelle müssen, für die Realisierung der Anwendung, angereichert und in Programmcode umgesetzt werden. Der EMF Codegenerator automatisiert den Schritt der Programmcode Erzeugung zu einem gewissen Grad, weshalb EMF als ein Schritt in Richtung *Model Driven Architecture*[2] gesehen werden kann. EMF Modelle sind im wesentlichen vergleichbar mit UML Klassendiagrammen indem sie ein vereinfachtes Modell der Klassen und Daten einer Anwendung

[2] http://www.omg.org/mda/

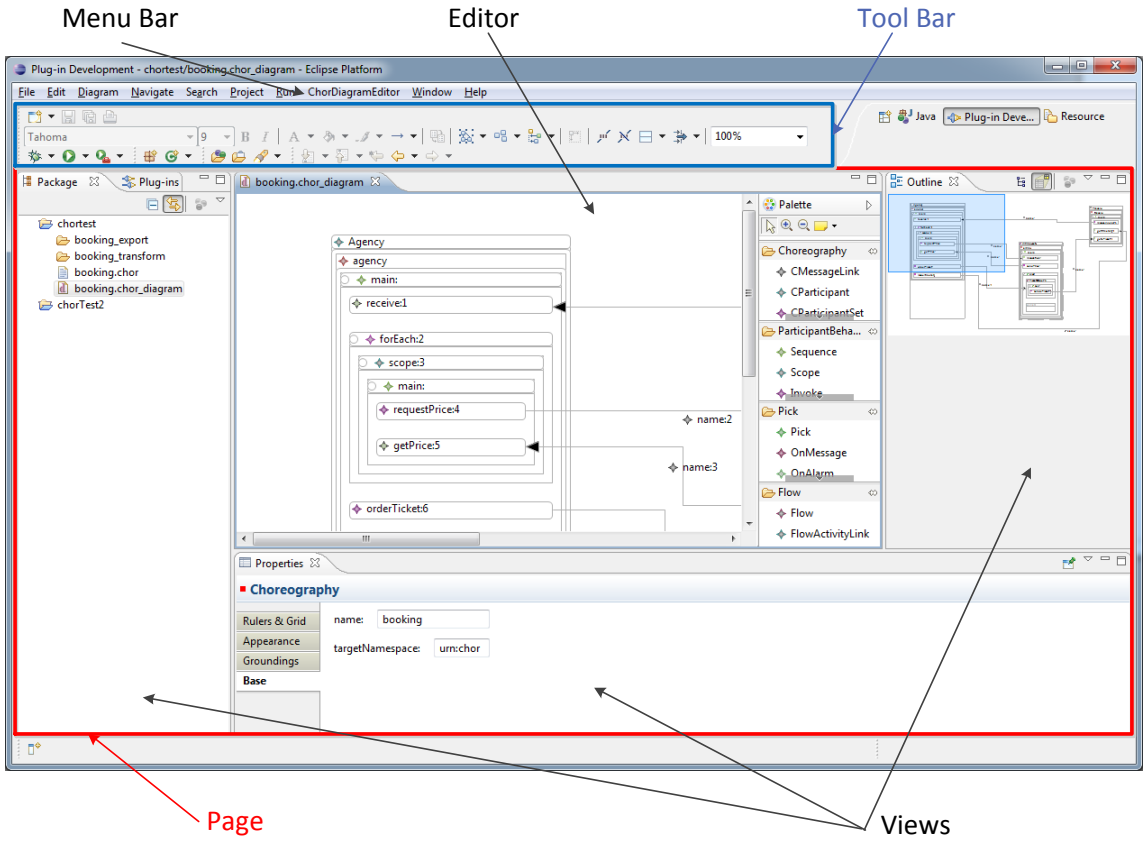

Abbildung 5.2: Eclipse Workbench. (Grafik basiert auf der Quelle: [ecl10]

repräsentieren. Um ein Modell in EMF zu beschreiben, wird das *Ecore* Meta-Modell benötigt welches, in einer vereinfachten Version zwecks Übersichtlichkeit, in Abbildung 5.3 zu sehen ist. `EClass` repräsentiert eine Klasse mit Namen und mehreren Attributen sowie Referenzen. `EAttribute` repräsentiert ein Attribut welches einen Namen und einen Typ hat. `EReference` repräsentiert ein Ende einer Assoziation zwischen Klassen, hat einen Namen und einen Referenztyp, welcher wiederum eine Klasse ist. `EAttribute` und `EReference` erben beide von `EStructualFeature` (in der Abbildung nicht dargestellt), was Attribute und Referenzen zu *Features* einer Klasse generalisiert. Für ein *Feature* kann eine Multiplizität angegeben werden um mehrwertige Attribute und Referenzen realisieren zu können. `EDataType` repräsentiert den Datentyp eines Attributes welcher entweder primitiv, wie z. B. `int` oder ein Objekttyp wie z. B. `java.util.Date`, sein kann. Wurde das gewünschte Modell in *Ecore* modelliert, kann Java Code daraus generiert werden. Für jede `EClass` Entität wird ein Java Interface generiert und dazu eine passende Klasse, welche dieses Interface implementiert. Dieser Ansatz ist eine Designentscheidung der EMF Entwickler und hat zu-

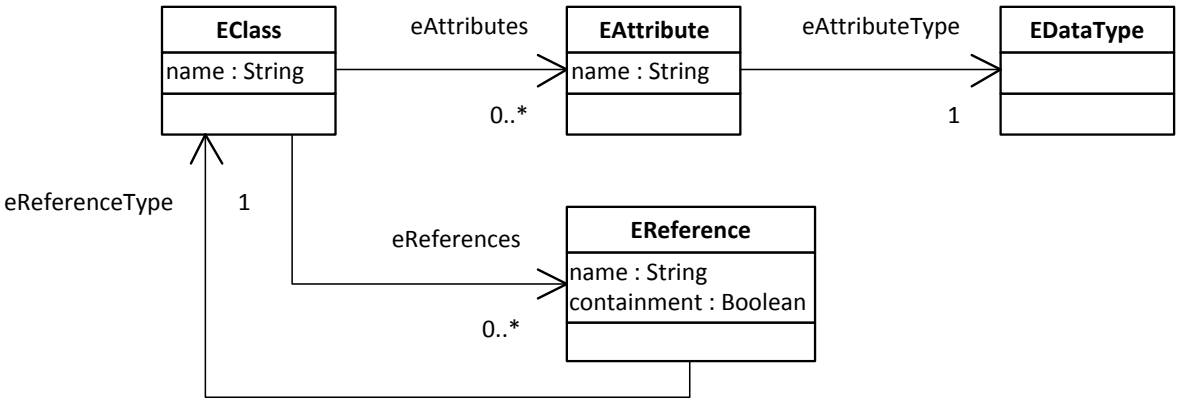

Abbildung 5.3: *Ecore* Meta-Modell. (Grafik basiert auf der Quelle: [Ste11]

dem den Vorteil, dass auch Mehrfachvererbung modelliert werden kann. Da eine Java Klasse immer nur von einer Klasse erben, jedoch mehrere Inferfaces implementieren kann, wird so das Problem der direkten Mehrfachvererbung (**`extends`**) gelöst. Jedes dieser generierten Inferfaces erbt von `EObject` und bringt dadurch Methoden mit, welche Zugriff auf das Meta-Modell gewähren, was vom Funktionsprinzip her der Java Reflection API[3] entspricht. In der folgenden Auflistung, sehen wir einige der wichtigen Methoden von `EObject` und deren Funktion.

eClass() gibt von der Objekt Instanz, die Instanz des Meta-Objekts `EClass` zurück.

eContainer() **und** *eRessource* geben von der Objekt Instanz, die Instanz des referenzierten Objekts zurück. *eContainer* gibt nur Referenz zurück, falls *containment* = *true* gesetzt wurde.

eGet() **und** *eSet*() bieten Zugriff auf Getter und Setter Methoden der Objekt Instanz.

eIsSet() überprüft, ob ein Wert in dieser Instanz gesetzt ist.

eUnset() entfernt den gesetzten Wert in dieser Instanz. Der Wert wird dabei entweder auf *null* oder, falls definiert, auf den Standardwert gesetzt.

Instanzen von EMF Modelle können einfach persistent gemacht werden, indem `Resource` und `ResourceSet` benutzt wird. Eine `Resource` repräsentiert einen physischen Speicherort wie z. B. eine Datei. `ResourceSet` ist eine Sammlung von `Resource` Instanzen, die alle zusammengehören oder sich untereinander referenzieren. `ResourceSet` unterstützt auch das Laden von Referenzen nach Bedarf, d.h. sie werden erst in den Speicher geladen, wenn sie vom Aufrufenden benötigt werden. Solange die Referenz nicht benötigt wird,

[3] http://docs.oracle.com/javase/6/docs/technotes/guides/reflection/index.html

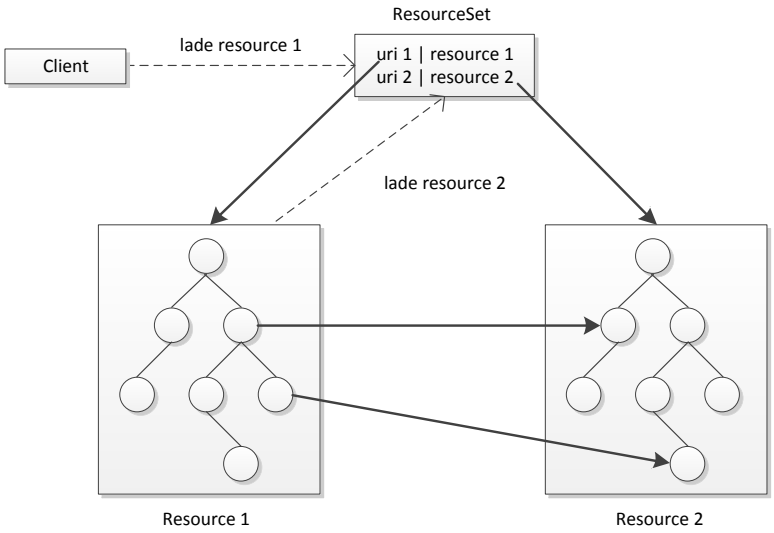

Abbildung 5.4: Laden von `Resource` Instanzen nach Bedarf. (Grafik basiert auf der Quelle: [Ste11]

werden sogenannte Stellvertreter (*proxies*) geladen und erst bei Bedarf das konkrete Objekt aufgelöst. In Abbildung 5.4 ist diese Funktionsweise dargestellt. Ein *Client* Programm lädt *Resource 1*, welche Referenzen – dargestellt durch Pfeile ausgehend von den Knoten – auf *Resource 2* beinhaltet. Sobald der *Client* diese Referenzen benötigt, veranlasst *Resource 1* beim zugehörigen *ResourceSet*, das Nachladen von *Resource 2*.

Eine weitere zentrale Eigenschaft von EMF ist, dass alle generierten Modell Klassen das *Observer* Entwurfsmuster[4] realisieren. Dies ist essentiell um einen Editor nach dem MVC Architekturmuster (siehe Abschnitt 3.4 auf Seite 27) zu realisieren. Das *Observer* Entwurfsmuster wird durch ein `Notifier` Interface realisiert, von welchem `EObject` erbt und somit die Registrierung von *Observern* sowie deren Benachrichtigung erlaubt. *Observer* werden in EMF *Adapter* genannt, da diese in EMF weitaus mehr als simple *Observer* sind, die nur auf Zustandsänderungen reagieren. *Adapter* können zusätzliche Funktionen für ihre beobachteten Objekte bereitstellen, wie wir im folgenden Abschnitt sehen werden, ohne von diesen erben zu müssen. Sie können sich an eine `EObject` Instanz mittels $eAdapters().add()$ Methode registrieren. Der generierte Code hält noch zwei weitere wichtige Klassen bereit. Zum einen eine `Factory` Klasse (realisiert das *Factory* Entwurfsmuster[5]) um die Modellobjekte zu erzeugen und zum anderen eine `Package` Klasse, welche einfachen Zugriff auf die *Ecore*

[4] http://www.oodesign.com/observer-pattern.html
[5] http://www.oodesign.com/factory-pattern.html

Metadaten ermöglicht. Der Generierte Code wird zudem als Eclipse Plugin erzeugt und kann somit leicht von anderen Plugins verwendet werden.

Neben dem *Ecore* Meta-Modell gibt es das Generator Modell, welches alle Informationen beinhaltet, wie genau der Code generiert werden soll. So kann unter anderem festgelegt werden, in welchem Ordner bzw. Paket die Klassen abgelegt werden. Diese Information ist nicht relevant für das Datenmodell, muss aber trotzdem irgendwo abgespeichert werden. Die Trennung in zwei Modelle erlaubt das erneute Generieren des Codes, ohne das *Ecore* Modell anpassen zu müssen.

EMF.Edit

EMF.Edit verbindet EMF mit der Eclipse Benutzeroberfläche (Eclipse UI Framework). Es hilft uns, einen grafischen Editor für unser Chor Model zu implementieren indem es Funktionen zur Darstellung und Bearbeitung des Modells bereitstellt. Es unterstützt außerdem das modifizieren von Modellobjekten nach dem *Command* Entwurfsmuster[6]. Der EMF.Edit Code lässt sich ebenfalls über das Generator Modell erzeugen. Um den EMF.Edit Code zu verstehen, erläutern wir zuerst die grundlegende Funktionsweise des Eclipse UI Frameworks. In Abbildung 5.5 sehen wir eine Instanz des Chor Model in einem *JFace* `TreeViewer` zusammen mit der *Property View* von `CParticipant`. Jede *JFace Viewer* Klasse hat einen *content provider*, der ein spezifisches Interface (`TreeViewer` benutzt `ITreeContentProvider`) implementiert, um die anzuzeigenden Objekte bereitzustellen sowie einen *label provider* um den Anzeigetext sowie ein Icon der Objekte zurückgibt. In Abbildung 5.5 wurde das Wurzelobjekt `Choreography` dem `TreeViewer` übergeben. Der *Viewer* ruft drauf *getText*() und *getImage*() des *label providers* auf, um den Text („Choreography booking“) und das Icon (`Choreography` hat hier keines, dessen Kind Elemente hingegen schon) anzuzeigen. Als nächstes wird *getChildren*() vom *Content Provider* aufgerufen um die Kind Elemente (im *Ecore* Modell *containment* = *true* gesetzt) zu bekommen. Der ganze Prozess wiederholt sich nun rekursiv, bis alle Objekte dargestellt sind.

Die *Property View* befüllt sich über den *Property Source Provider* des selektierten Objekts – in Abbildung 5.5 ist `CParticipant` selektiert – indem zuerst *getPropertySource*() aufgerufen wird. Darauf wird die *Property Source* von `CParticipant` zurück gegeben worauf mit *getPropertyDescriptors*() eine Liste aller *Property Descriptors* (hier: `Containment`, `Name`, `Scope` und `Selects`) zurückgegeben wird. Über diese *Property Descriptors* lassen sich schließlich die Werte editieren. EMF.Edit generiert eine Implementierung für *content provider*, *label provider*, *property source provider*, *property source* und *property descriptor* vereint im sogenannten *Item Provider*. Für jede *Ecore* Klasse wird ein *Item Provider* generiert. Für unser Chor Model wären dies `ChoreographyItemProvider`, `CParticipantItemProvider`,

[6] http://www.oodesign.com/command-pattern.html

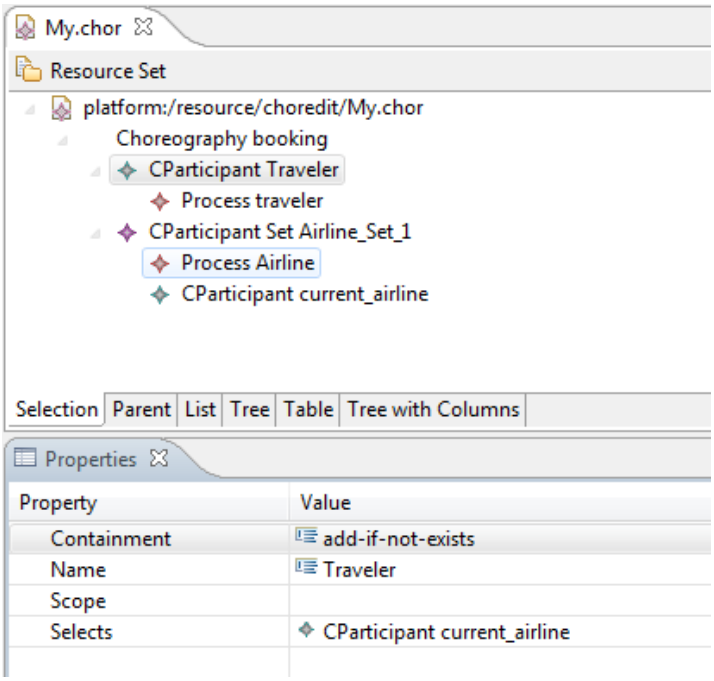

Abbildung 5.5: *JFace* `TreeViewer` zeigt eine Chor Model Instanz an

usw.. Diese *Item Provider* werden als *Adapter* realisiert um einerseits die *Observer* Funktion zu erfüllen und andererseits um beliebige Implementierungen für Interfaces von Editoren und *Views* bereit zu stellen. Es ist außerdem möglich, *Item Provider* zu erstellen, welche nicht *Adapter* für ein Modellobjekt sind. Somit können Sichten auf EMF Objekte erstellt werden, was etwas dem „Sichten“ Konzept aus dem Datenbank Bereich ähnelt. Die *Item Provider* bieten zusätzlich noch die Möglichkeit zur Erzeugung von *Commands* über eine *Command Factory* und erfüllen die Observer Funktion, indem sie Benachrichtigungen an *Viewer* Klassen weiterleiten.

Das *Command* basierte Editieren von EMF Objekten wird exzessiv von GMF (siehe 5.1.4 auf Seite 95) generierten Editoren benutzt und beinhaltet unter anderem ein automatisches rückgängig ("undo") machen und wiederherstellen ("redo"). EMF bietet ein *Command Framework*, welches sich aus zwei Bereichen zusammensetzt. Einerseits der allgemeine Teil (*Common Command Framework*), der unabhängig von EMF.Edit benutzt werden kann und andererseits der EMF.Edit spezifische Teil, basierend auf `EObject`. Wir werfen hier nur einen kurzen Blick auf das *Command Framework*, welches ausführlich in Kapitel 3, des Buches von Steinberg [Ste11] erläutert wird. `Command` ist das Basisinterface des *Common Command Frameworks* und alle *Commands* implementieren dieses. Die wichtigsten Methoden sind *execute*() zum Ausführen, *undo* und *redo* respektive zum rückgängig machen oder wiederherstellen. Bevor ein *Command* ausgeführt werden kann, wird es mit *canExecute*()

auf Ausführbarkeit getestet. Die Ausführbarkeit kann somit kontextabhängig erlaubt oder verboten werden. Komplexe *Commands* können Änderungen verursachen, die nicht durch ein simples *undo* rückgängig gemacht werden können weil zu viele Dinge verändert wurden. Deshalb kann mit *canUndo*() eine Prüfung davor geschoben werden. Für die Realisierung von zusammengesetzten *Commands* – dies sind *Commands*, welche andere *Commands* als Voraussetzung haben – kann mit *getResult*() das Ergebnis des Ausführung geholt werden und mit *getAffectedObjects*() eine Liste aller betroffenen Objekte des letzten *execute*(), *undo* oder *redo* Aufrufes. Der `CommandStack` ist das Interface für einen Stapel, auf welchem mehrere `Commands` hintereinander ausgeführt, rückgängig gemacht oder wiederhergestellt werden können. GMF Editoren machen auch davon intensiven Gebrauch, weshalb wir dieses Interface hier erwähnen. Im Folgenden listen wir die EMF.Edit spezifischen `Commands` auf, welche wir in der Implementierung benötigen.

- `AddCommand` fügt ein oder mehrere Objekte zu einem mehrwertigen *Feature* eines `EObject` hinzu.
- `SetCommand` setzt den Wert eines Attributes oder Referenz von einem `EObject`.
- `DeleteCommand` entfernt ein `EObject` von seinem Elterncontainer und löscht alle Referenzen, welche auf dieses Objekt zeigen.
- `RemoveCommand` entfernt ein oder mehrere Objekte vom einem mehrwertigen *Feature* eines `EObject`.

Wenn ein Editor *Commands* zur Manipulation der Modellobjekte benutzen möchte, muss eine *Editing Domain* benutzt werden, über welche die *Commands* erzeugt werden können. Die *Editing Domain* verwaltet den *Command* Stapel und bietet Zugriff auf das EMF `ResourceSet`, in welchem sich alle zu editierenden Objekte befinden. Um an die *Editing Domain* eines `EObject` zu kommen starten wir bei der von EMF bereitgestellten Klasse `AdapterFactoryEditingDomain`. Entweder wird ein passender Adapter – welcher den Typ `IEditingDomainProvider` unterstützt – gefunden oder, falls es einen solchen Adapter nicht gibt, kann über das `ResourceSet`, zu welchem das `EObject` gehört, ebenfalls die *Editing Domain* gefunden werden.

5.1.3 Graphical Editing Framework

Die folgenden Beschreibungen basieren auf dem Buch von Gronback [GG09]. Wir gehen hier nur oberflächlich auf dieses Framework ein, da wir es nicht explizit benutzen. Es besteht aus den zwei Plugins *Draw2d*, welches eine Erweiterung zu SWT darstellt indem es Zeichenfunktionen und Layout Funktionalitäten bietet und *GEF*, welches ein MVC Framework für grafische Editoren darstellt.

Draw2d Plugin

Den zentralen Teil von *Draw2d* bilden die *Figures*, was Komponenten zur grafischen Darstellung sind. Sie können aus weiteren Kind *Figures* zusammengesetzt sein, welche innerhalb der Begrenzung der Eltern *Figure* gezeichnet, und mit einem Layout Manager angeordnet werden. Weitere Möglichkeiten von *Figures* sind das Registrieren von *Listeners*, welche z. B. auf Mausklicks reagieren oder die Berechnung, ob ein Punkt innerhalb der *Figure* liegt oder nicht. Neben *Figures* gibt es noch die Möglichkeit Text darzustellen, was in Labels realisiert wird. Gezeichnet wird nach einer bestimmten Strategie. Zuerst werden die Eigenschaften der *Figure* und seinen Kind Elementen festgelegt wie z. B. Schriftart, Vordergrund- und Hintergrundfarbe. Diese Eigenschaften stellen den aktuellen Grafikzustand dar, welcher abgespeichert wird. Dann wird die *Figure* selbst gezeichnet wozu unter anderem die Umrandung ("bounding box") und die Hintergrundfarbe gehört. Als nächstes wird die "client area" gezeichnet. Dies ist die Fläche innerhalb der *Figure*, worauf dessen Kind Elemente gezeichnet werden. An dieser Stelle wird auch "clipping" durchgeführt was verhindert, dass Teile der Grafik auf nicht erlaubten Stellen erscheinen. Nun werden die Kind Elemente auf die "client area" gezeichnet und zuletzt alle Dekorationen, welche über den Kind Elementen erscheinen sollen. *Figures* sind in einer Baumstruktur zusammengesetzt. Dieser Baum wird mittels Tiefensuche traversiert, was eine Zeichnung in Reihenfolge der Z-Koordinate zur Folge hat. Das bedeutet, dass die *Figure*, welche am weitesten „unten“ liegt, zuerst gezeichnet wird. Die Anderen werden darüber gezeichnet. *Draw2d* bietet außerdem noch *Connections*. Dies sind Verbinder, bestehend aus einer Linie zwischen zwei Punkten. Sogenannte Anker (`ConnectionAnchor`) bestimmen, wie genau sich die Endpunkte der Linie mit Quell- und Zielobjekt verbinden sollen. In Abbildung 5.6 ist der `ChopboxAnchor` zu sehen, welcher die Verankerung an den Schnittpunkt von Linie und Umrandung der *Figure* setzt. Schließlich lässt sich noch der Verlauf der Linie zwischen Anfangs und Endpunkt über einen Router (`ConnectionRouter`) bestimmen. So kann der Verlauf zwischen den zwei Punkten z. B. grade sein oder auch mehrere Zwischenpunkte haben, an welchen der Verlauf seine Richtung um neunzig Grad ändert.

GEF Plugin

Dieses Plugin bietet die Funktionalität ein Datenmodell mit *Figures*, über Eingabegeräte wie z. B. Maus oder Tastatur, in einer Eclipse *Workbench* zu editieren. Es ist eine Implementierung der MVC Architektur in welcher der *View* Teil durch das *Draw2d* Plugin, der *Model* Teil durch *EMF* und der *Controller* Teil durch das GEF Plugin realisiert ist. In den `EditPart` Klassen werden die *Controller* für die Modell Elemente realisiert. Ihre Aufgabe ist einerseits, Eingaben des Benutzers entgegen zu nehmen und das *Model* zu aktualisieren und andererseits, die *View* über die Änderungen am Modell zu informieren. In Abbildung 5.7 sehen wir einen groben Überblick, wie GEF seine Modell Elemente grafisch darstellt. Für jedes dieser Modell Elemente muss ein `EditPart` erstellt werden. Die grafische Repräsentation

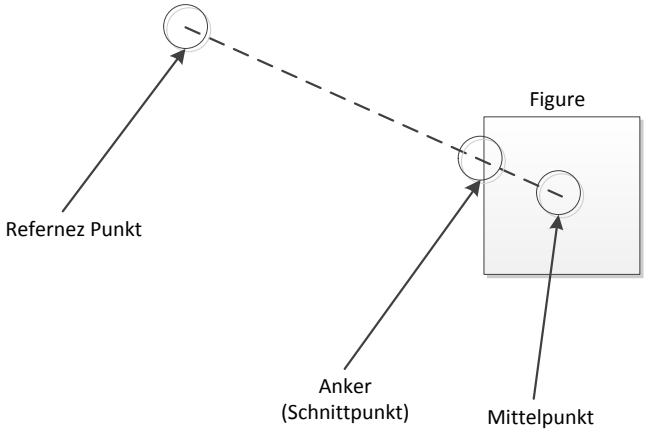

Abbildung 5.6: Funktionsweise des `ChopboxAnchor`. Basiert auf Quelle: [GG09]

des Elements wird über die `Figure` bestimmt, welche in diesem `EditPart` instanziiert wird. Die `EditPartFactory` ist zuständig für die Instanziierung des richtigen `EditPart` zum zugehörigen Modell Element und setzt auch die entsprechende Referenz, so dass jede `EditPart` Instanz sein Modell Element kennt. Diese `EditPartFactory` wird am `Viewer` registriert, so dass dieser die entsprechenden `Figure` Instanzen zeichnen kann.

Jeder `EditPart` Klasse referenziert eine Menge von `EditPolicy` Klassen, welche das Verhalten des *Controllers* implementieren. Die Idee ist, dass `EditPart` Klassen das Durchführen der Aufgaben, die anstehen wenn der Benutzer etwas editiert hat, an `EditPolicy` Klassen delegieren. Diese Aufteilung erlaubt es, dass sich `EditPart` Klassen verschiedene `EditPolicy` Klassen untereinander teilen können. So kann ein bestimmtes Verhalten von mehreren `EditPart` Klassen benutzt werden. `EditPolicy` Klassen werden, bei der Instanziierung einer `EditPart` Klasse, „installiert“. Dabei wird jeder Klasse eine Rolle zugewiesen. Rollen sind einfache Schlüsselwerte die es erlauben, `EditPolicy` Klassen auszuzeichnen bzw. zu identifizieren. So gibt es z. B. die Rolle `LAYOUT_ROLE` welche eine `EditPolicy` so auszeichnet, dass diese für bestimmte Layout Aufgaben wie z. B. Skalieren oder Positionieren zuständig ist. Durch diese Schlüsselwerte kann so von einem `EditPart` die gerade installierte `EditPolicy` für Layout Aufgaben abgefragt werden, ohne den konkreten Klassennamen zu kennen.

Wenn wir ein grafisches Modell in einem Editor erstellen, interagieren wir mit diesem über Eingabegeräte. GEF abstrahiert alle Interaktionen mit dem Editor durch eine `Request` Klasse. Eine `Request` Instanz kapselt alle Informationen, welche die `EditPart` Klassen benötigen, um ihre Funktionen auszuführen. Möchte der Benutzer z. B. ein neues Element erstellen, löst diese Aktion ein `Request` aus. Der `EditPart`, welcher den `Request` versteht und an der Erstellung des Elements beteiligt ist, wird aufgerufen und gibt das passende `Command`

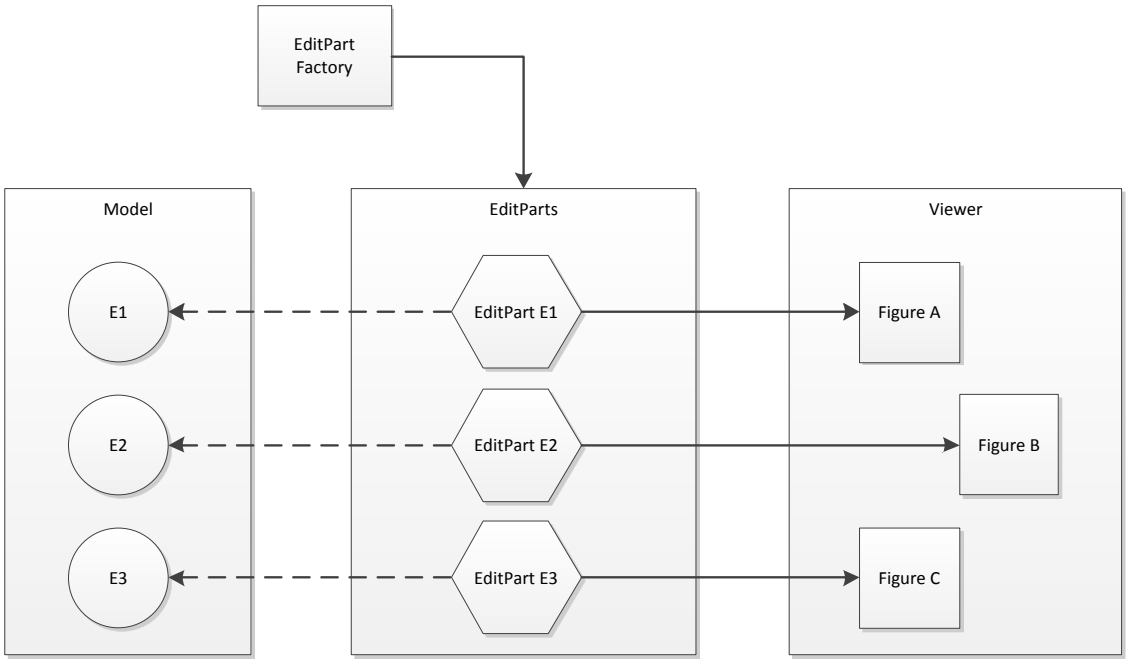

Abbildung 5.7: Grafische Darstellung der Modell Elemente in GEF durch `EditParts`

zurück, welches diesen `Request` durchführen kann. `Commands` sind dafür zuständig das, dem Editor zugrunde liegende Datenmodell abzuändern. GEF und EMF benutzen jeweils ihre eigenes `Command` Framework. GMF bietet dafür eine Vereinheitlichung.

5.1.4 Graphical Modeling Framework

Die folgenden Beschreibungen basieren auf dem Buch von Gronback [GG09]. GMF entstand aus dem bestreben einen grafischen Editor, basierend auf GEF, mit EMF zu verbinden. Es besteht aus den zwei Hauptkomponenten *Runntime* und *Tooling* Framework. Die *Runntime* verbindet EMF mit GEF indem sie *Services* und *API*s bereit stellt. Das *Tooling* Framework bietet einen modellbasierten Ansatz um grafische Elemente zu definieren, passende Werkzeuge zur Benutzung dieser Elemente zu erstellen, sowie ein "Mapping" um die grafischen Elemente auf Elemente eines zugrunde liegendes Datenmodells abzubilden. Das *Tooling* Framework ist eine Sammlung von Modellen, aus welchen ein grafischer Editor generiert werden kann, der die GMF *Runntime* benutzt.

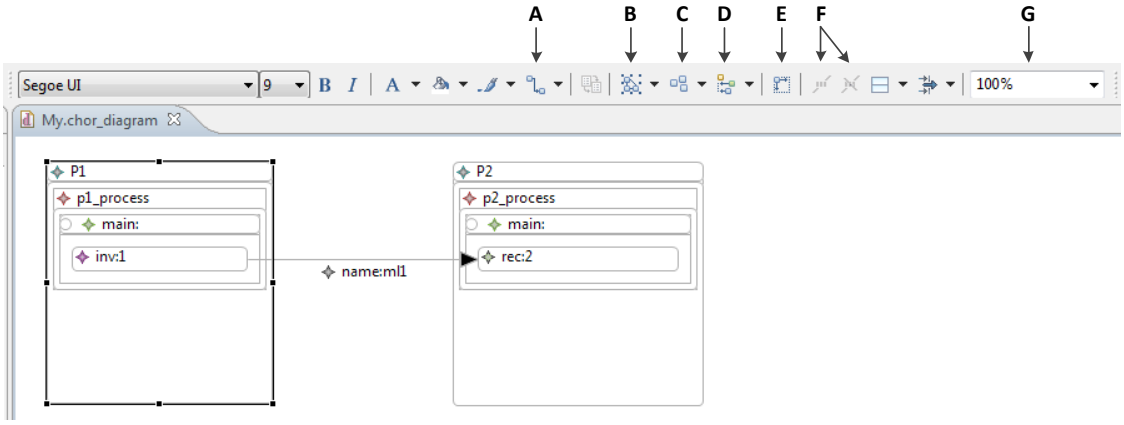

Abbildung 5.8: GMF Standard Werkzeuge in der Eclipse Toolbar

GMF Runntime

Die *Runntime* baut auf GEF auf und bietet eine Menge von wiederverwendbaren Komponenten für die Erstellung von grafischen Editoren. Dazu gehören unter anderem die Standardwerkzeuge in der Eclipse *Toolbar*, welche in Abbildung 5.8 zu sehen sind. Mit Werkzeug **A** lässt sich der Router Stiel eines selektieren Verbinders (in unseren Beipiel der Message Link "ml1") zwischen "rectilinear" und "oblique" hin und her wechseln. Im Modus "rectilinear", sind die Verbinder mit Punkten ausgestattet, an welchen die Linie eine neunzig Grad Wendung nimmt. Im Modus "oblique" sind die Verbinder gerade bzw. direkt zwischen Anfangs- und Endpunkt. Mit Werkzeug **B** können alle Elemente im Editor selektiert werden. Werkzeug **C** ordnet alle Elemente neu an, während Werkzeug **D** alle Elemente vertikal an der linken Kante ausrichtet. Werkzeug **E** skaliert alle selektierten Elemente auf eine passende Größe, wobei für jedes Element die minimale und bevorzugte Größe berücksichtigt wird. Die Werkzeuge **F** blenden die Labels an Verbindern ein oder aus. In unserem Beispiel wäre dies das Label "name:ml1". Werkzeug **G** erlaubt das Zoomen der Elemente. Die Ansicht kann vergrößert, verkleinert oder automatisch so angepasst werden, dass alle Elemente möglichst in die sichtbare Zeichenfläche passen.

Eine weite Komponente der *Runntime* sind die Standard Tabs "Rulers & Grid" sowie "Appereance" in der *Property View*, welche – je nach selektiertem Editor Element – angezeigt werden. In Abbildung 5.9 sehen wir die Inhalte vom "Rulers & Grid" Tab. Hierbei sind das Lineal ("Show Ruler"), sowie die Gitterpunkte ("Show Grid") aktiviert. Unter dem "Appereance" Tab finden wir Funktionen wie den Text der Editor Elemente Fett oder Kursiv darzustellen und Textfarbe sowie Flächenfüllung anzugeben.

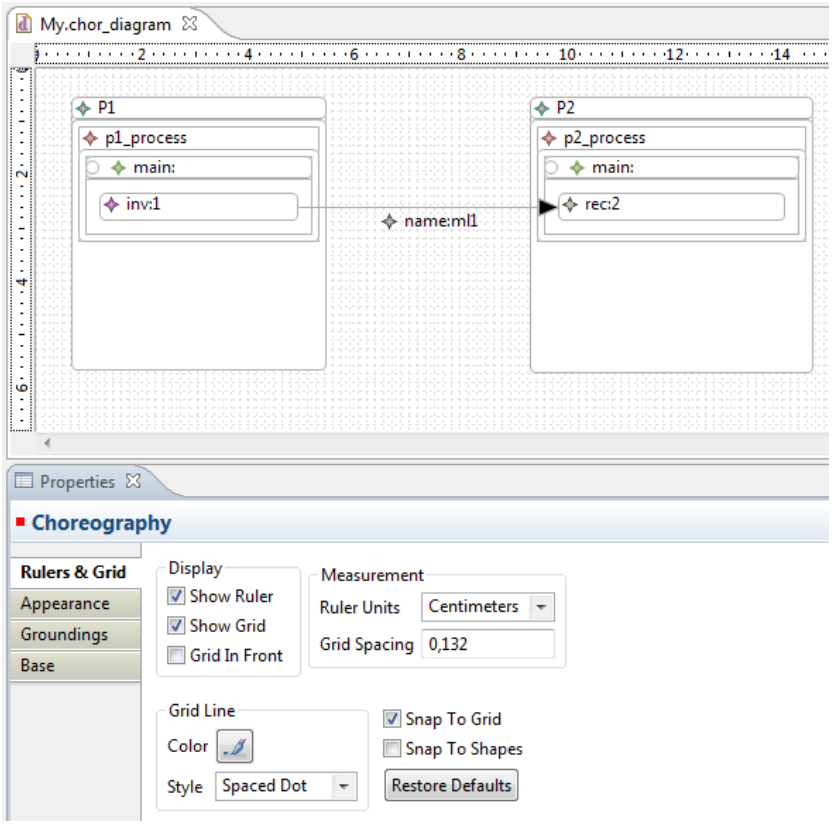

Abbildung 5.9: *Property View* mit den GMF Standard Tabs "Rulers & Grid" und "Appereance"

Die *Runntime* bietet zudem die Möglichkeit, das grafische Modell mit Informationen wie Position und Größe der Elemente, eingefärbte Bereiche, Schriftarten usw. abzuspeichern. Zusätzlich kann das grafische Modell noch als reines Datenmodell, ohne besagte Informationen, abgespeichert werden. Ein weiterer Teil der *Runntime* ist eine Service Schicht ("service layer"). Das Service Konzept sieht einen oder mehrere Anbieter vor, welche bestimmte Dienste der Anwendung anbieten. Einer dieser Services ist der *ViewService*, der für die Erstellung von *View* Klassen zuständig ist. *View* Klassen sind in GMF die übergeordneten Klassen aller grafischen Elemente, die auf der Zeichenfläche dargestellt werden. Sie halten ebenfalls eine Referenz auf ihr zugeordnetes EMF Modell Element. Die Menge aller *View* Klassen bilden ebenfalls ein EMF Modell. Wichtig ist hier die Unterscheidung. Der GMF Editor wird zum editieren eines Datenmodells benutzt, was in unserm Anwendungsfall dem Chor Model entspricht. Der Editor selbst hat aber sein eigenes, internes Editor Modell, zu welchem Informationen wie Position und Aussehen von grafischen Elementen gehören. Der *ViewService* hat einen *ViewProvider*, der über den `viewProviders`[7] *Extension point* registriert wird. Er

[7]org.eclipse.gmf.runtime.diagram.core.viewProviders

stellt die Methoden zur Erstellung der grafischen Elemente bereit. Diese Elemente können entweder vom Typ *diagram*, *node* oder *edge* sein. Das *diagram* Element ist das Wurzelelement sprich, die Zeichenfläche, in welcher die anderen Elementtypen platziert werden. Die *edge* Elemente sind Verbinder und *node* die übrigen Elemente, zwischen denen Verbinder gezogen werden können. Ein weiterer Service ist der *EditPartService* welcher benutzt wird, um die *EditParts* zu erstellen. Der zugehörige *EditPartProvider*, registriert im `editpartProviders`[8] *Extension point*, ersetzt die bereits besprochene `EditPartFactory` des GEF Plugins. GMF Editoren, welche über GMF Tooling (siehe nächsten Abschnitt) modelliert und generiert werden, benutzen immer *ViewProvider* und *EditPartProvider*.

GMF Tooling

In Abbildung 5.10 sehen wir das Vorgehen anhand eines BPMN Diagramms, welche Schritte notwendig sind, um einen GMF Editor mittels *Tooling* Framework zu erstellen. Wir starten mit der mit der Erstellung eines Domain Modells was hier unserer EMF Realisierung vom Chor Model entspricht. Dann erstellen wir das *Graphical Definition Model*, in welchem wir die grafische Repräsentation der darzustellenden Elemente festlegen. In Abbildung 5.11 ist eine vereinfachte Darstellung dieses Modells zu sehen. Als Konvention gilt, dass die gestrichelten Pfeile „referenziert“ bedeuten und die durchgehenden Pfeile „ist Elternknoten von“. Das Wurzelelement ist der *Canvas* Knoten, welchem wir einen Namen geben, da wir diesen später im *Mapping Model* referenzieren. Zuerst definieren wir einen oder mehrere *Figure Gallery* Knoten unter welchen wir zusammengehörende *Figure Descriptor* Knoten gruppieren. Diese Knoten legen das Aussehen eines Elementes fest. Das Aussehen definieren wir mit grafischen Komponenten wie hier in unserem Beispiel ein Rechteck, welches ein Label sowie ein weiteres Rechteck mit dem Namen "(Compartment)" beinhaltet. Für die Konten, auf welche von außerhalb des *Figure Descriptors* zugegriffen werden soll, müssen *Child Access* Knoten angelegt werden. In unserem Beispiel gewähren wir Zugriff auf das Label und das "(Compartment)" Rechteck. Haben wir alle *Figure Descriptors* festgelegt, definieren wir die Elemente welche unser Editor (*Diagram*) auf seiner Zeichenfläche (*Canvas*) darstellen soll. Ein Diagrammelement referenziert ein *Figure Descriptor* Knoten wobei auch mehrere Diagrammelemente den selben *Figure Descriptor* Knoten referenzieren können. *Nodes* sind die Diagrammelemente, mit denen wir modellieren und welche keine Verbinder sind. In unserem Beispiel referenzieren wir mit der *Node* den gewünschten *Figure Descriptor* und bestimmen somit das Aussehen dieser *Node*. *Compartments* sind Container Elemente, die andere *Nodes* aufnehmen. Sie sind ein Konstrukt, um eine verschachtelte Darstellung von *Nodes* zu erreichen. Wir müssen für jedes *Compartment* festlegen, an welcher Stelle genau die aufgenommenen *Nodes* auf der Zeichenfläche platziert werden sollen. In unserem Beispiel werden die *Nodes* im "(Compartment)" Rechteck des referenzierten *Figure Descriptors* platziert.

[8] org.eclipse.gmf.runtime.diagram.ui.editpartProviders

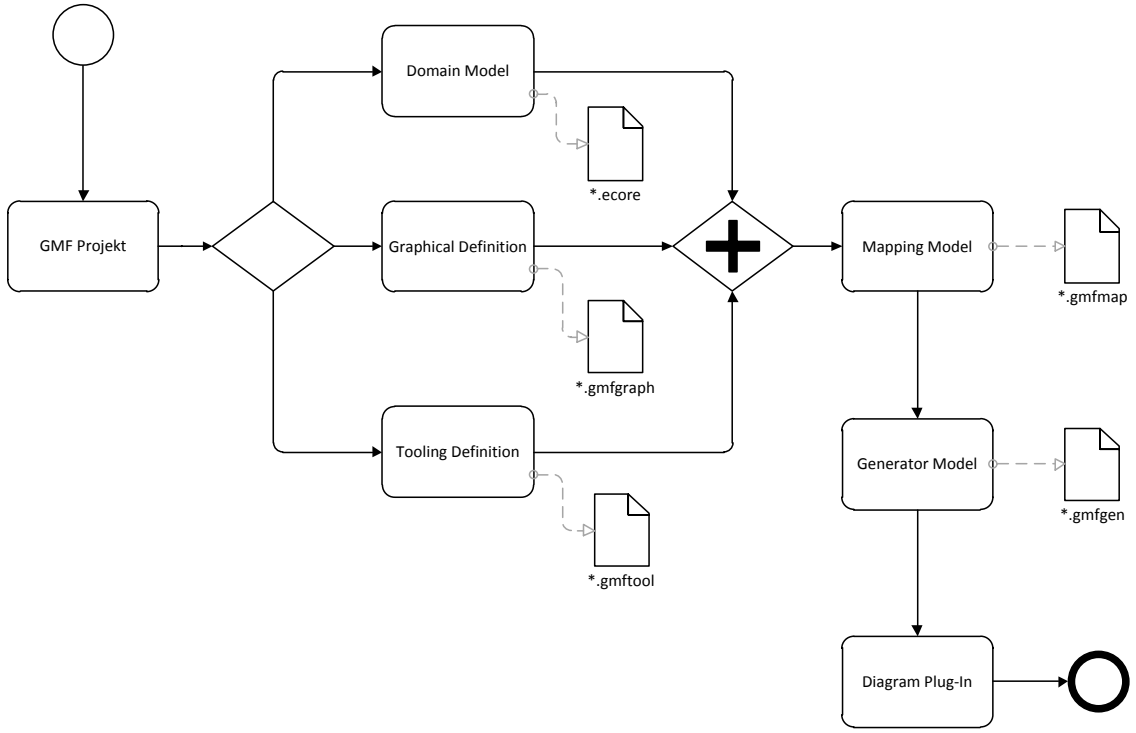

Abbildung 5.10: Vorgehensweise bei der Erstellung eines GMF Editors mittels *Tooling* Framework. Grafik basiert auf der Quelle: [ecld]

Hier sehen wir auch, warum wir zuvor ein *Child Access* Knoten für dieses Rechteck definiert haben, nämlich um es hier referenzieren zu können. Damit unser definiertes Label auch auf der Zeichenfläche erscheint, müssen wir ein *Diagram Label* Knoten definieren, welchen wir optional auch mit einem Icon dekorieren können. Dies ist in Abbildung 5.12 zu sehen. Wir möchten, dass unser *Diagram Label* das Label des referenzierten *Figures Descriptors* anzeigt und verweisen deshalb auf den entsprechenden *Child Access* Knoten. In Abbildung 5.13 sehen wir die nötigen Knoten für die Definition eines Verbinders. Zuerst muss wieder ein *Figure Descriptor* Knoten erstellt werden. Der Verbinder wird hier als Linie mit Ankerpunkten an jedem Ende (*Polyline Connection*) dargestellt und soll ein Label haben, für welches wir wieder einen *Child Access* Knoten definieren. Damit wir *Nodes* auf der Zeichenfläche auch Verbinden können, legen wir ein *Connection* Knoten an und bestimmen sein Aussehen durch die entsprechende *Figures Descriptor* Referenz. Das *Diagram Label* benötigen wir zur Anzeige des Labels und setzen die entsprechenden Referenzen.

Um die Diagrammelemente erstellen und auf der Zeichenfläche des Editors platzieren zu können, benötigen wir Werkzeuge, welche wir im *Tooling Definition Model* festlegen. In

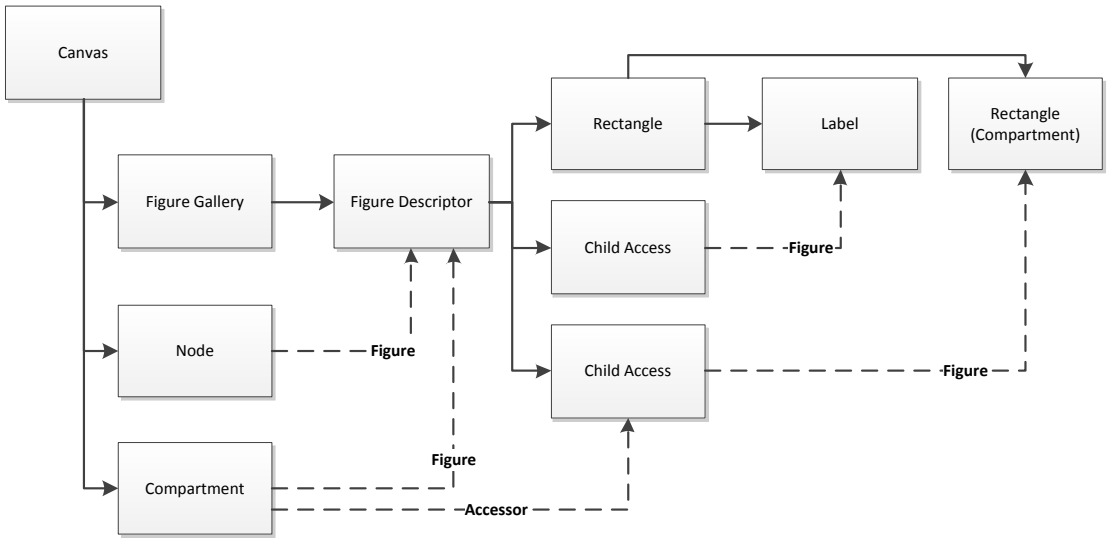

Abbildung 5.11: *Graphical Definition Model* für *Node* und *Compartment*

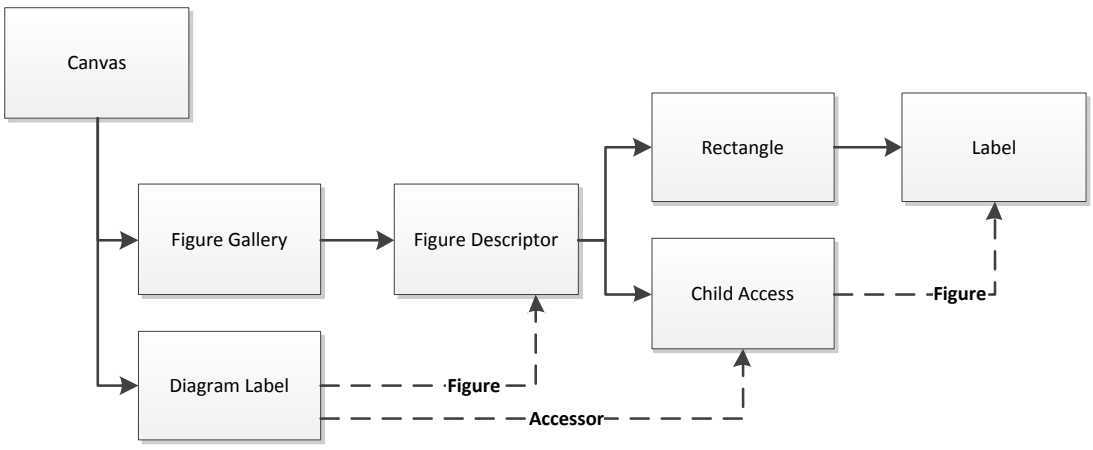

Abbildung 5.12: *Graphical Definition Model* für *Diagram Label*

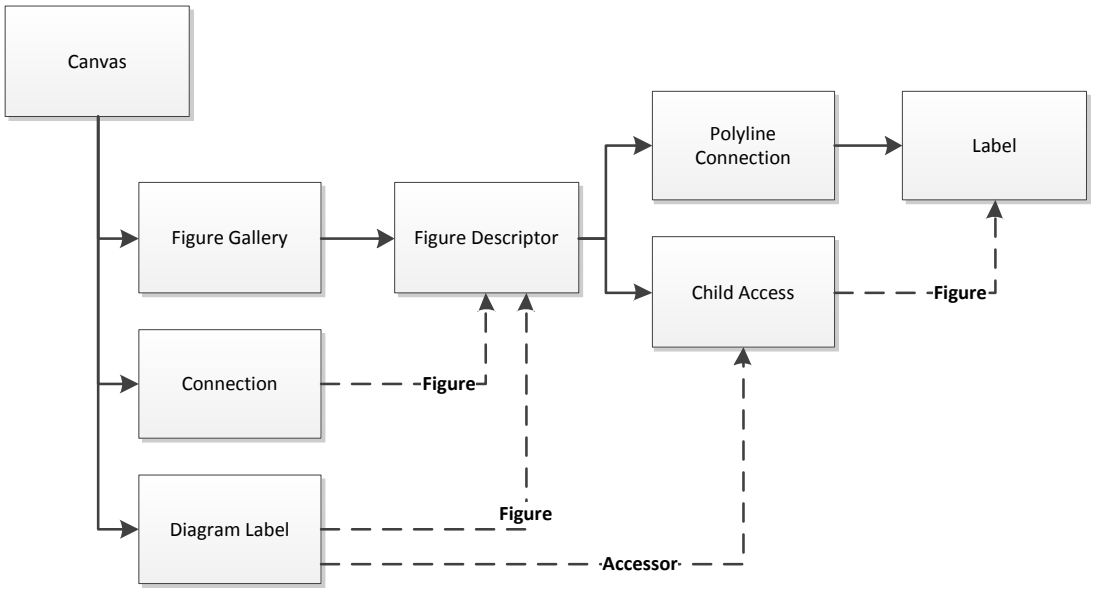

Abbildung 5.13: *Graphical Definition Model* für *Connection*

Abbildung 5.14 ist ein vereinfachtes Beispiel zu sehen. Das Wurzelelement ist der *Tool Registry* Knoten unter welchem wir einen *Palette* Knoten definieren und benennen, so dass wir im *Mapping Model* darauf referenzieren können. Die aktuelle GMF Version, zum Zeitpunkt dieser Arbeit, unterstützt nur Elemente für die *Palette*. Mit dem *Creation Tool* Knoten legen wir fest, dass es einen Eintrag in der *Palette* mit dem hier festgelegten Namen und Icon geben soll. Die eigentliche Funktionalität erhält das Werkzeug erst später durch das *Mapping Model*, auf welches wir gleich eingehen werden. Wir können diese Werkzeuge zusätzlich noch in Gruppen, durch Erstellung von *Tool Groups*, einteilen um zusammengehörige Werkzuge optisch voneinander abzugrenzen. Neben den *Creation Tools*, lassen sich noch vorgefertigte *Standard Tools* definieren die Funktionalitäten wie Selektion und Zoom mitbringen. In unserem Beispiel besteht die *Palette* aus einer *Tool Group* mit zwei *Creation Tools*.

Haben wir die bisher besprochenen Modelle angelegt, bringen wir sie alle im *Mapping Model* zusammen. Das *Mapping Model* ist das Herzstück vom *Tooling* Framework und erlaubt die Erstellung eines oder mehrerer *Generator Models*, aus welchem letztendlich der Editor Code generiert wird. Die Elemente des *Graphical Definition Model* werden hier mit den Elementen des Domain Modells verknüpft und die passenden Elemente des *Tooling Definition Model* zugewiesen. In Abbildung 5.15 sehen wir ein vereinfachtes Beispiel, wie die Zeichenfläche eines Editors definiert werden kann. Das Wurzelelement des *Mapping Model* ist der *Mapping* Knoten. Darunter definieren wir den *Canvas Mapping* Knoten und legen zuerst fest, wie wir

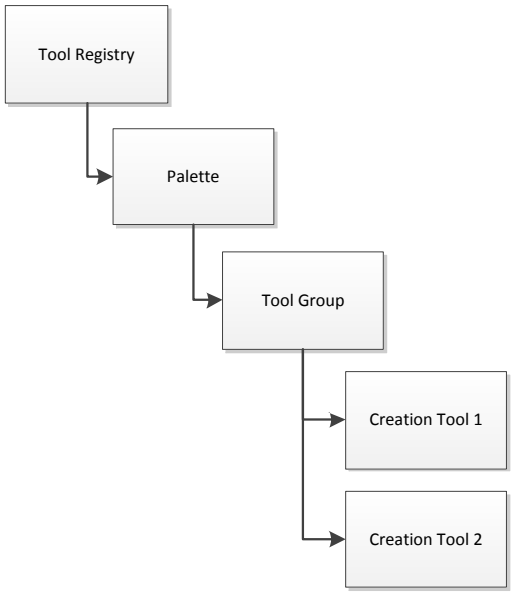

Abbildung 5.14: *Tooling Definition Model* mit zwei *Creation Tools*

unsere Elemente darstellen möchten indem wir auf den *Canvas* Knoten aus dem *Graphical Definition Model* verweisen. Dann müssen wir angeben, mit welchen Werkzeugen wir unsere Elemente erstellen möchten und verweisen dazu auf den *Palette* Knoten aus dem *Tooling Definition Model*. Zuletzt müssen wir das Domain Element bestimmen, welches alle modellierbaren Elemente beinhaltet. Im EMF Modell muss dazu das Wurzelelement für all seine ausgehenden Referenzen $containment = true$ setzen.

Im nächsten Schritt definieren wir die Elemente, welche direkt auf der Zeichenfläche platziert werden können. Dazu betrachten wir Abbildung 5.16. Der *Top Node Reference* Knoten verweist auf eine $containment = true$ Referenz des Wurzelelements aus dem Domain Modell, welches wir auch schon im *Canvas Mapping* Knoten festgelegt haben. Mit dem *Node Mapping* Knoten verweisen wir auf das entsprechende Domain Modell Element, welches diese Referenz vorgibt. In unserem Beispiel referenziert "Root Element"auf "Element 1". Dann legen wir aus Aussehen von "Element 1" fest, indem wir auf die gewünschte *Node* aus dem *Graphical Definition Model* verweisen. Zuletzt müssen wir noch angeben, mit welchem Werkzeug eine Instanz von "Element 1" erstellt werden soll. Dazu verweisen wir auf das gewünschte *Creation Tool* aus dem *Tooling Definition Model*.

Haben wir das *Mapping Model* komplett ausgearbeitet, können wir ein *Generator Model* daraus erstellen. Das *Generator Model* erlaubt uns auf die Code Generierung gewissen Einfluss zu nehmen wie z. B. an welchen Ort, unter welchen Namen der Code auf dem

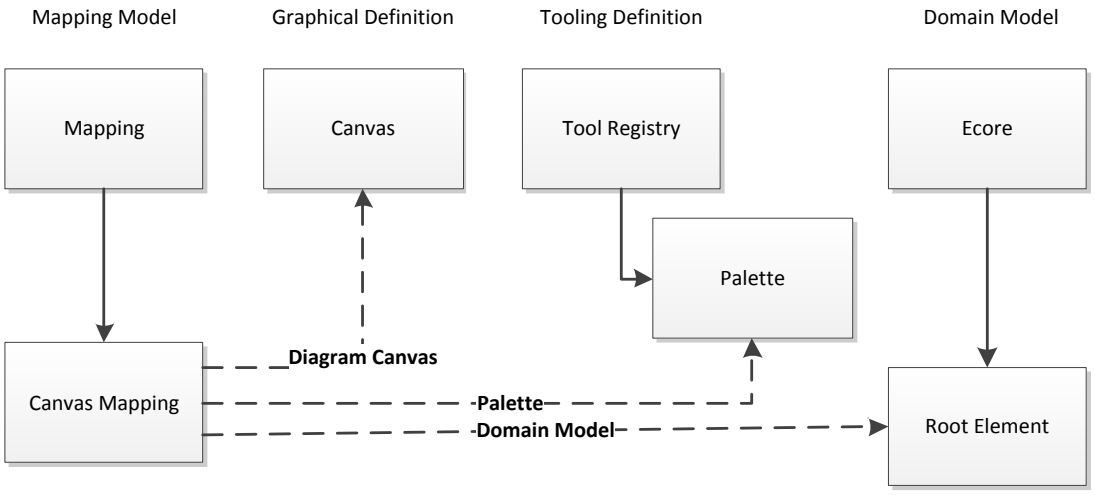

Abbildung 5.15: *Mapping Model* zur Definition der Zeichenfläche des Editors

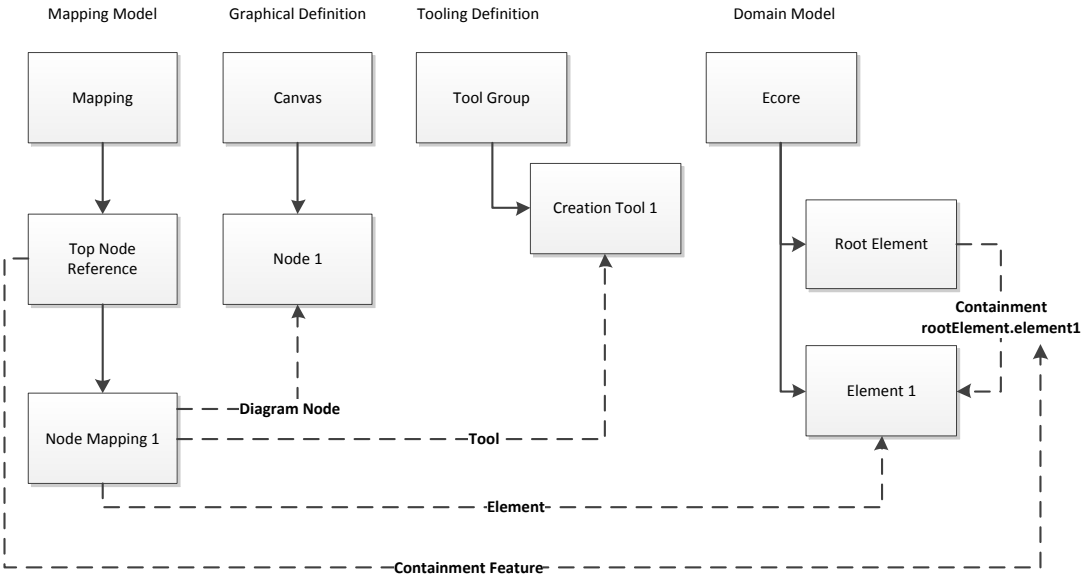

Abbildung 5.16: *Mapping Model* zur Definition eines, auf der Zeichenfläche des Editors, platzierbaren Elements

Datenträger abgelegt werden soll. Die Code Generierung basiert auf Code Schablonen (Templates), welche in der *Xpand Template Language* (siehe 5.1.5) definiert sind. Wir können, durch Anpassung der Schablonen, Einfluss auf den generierten Code nehmen. Dazu müssen wir die gewünschten Schablonen, welche wir anpassen wollen, aus dem GMF eigenen `org.eclipse.gmf.codegen` Plugin in unseren GMF Projektordner in die selbe Ordnerstruktur kopieren. Die Ordnerstruktur dient als Namespace der Schablonen. Wir betrachten dies genauer im nächsten Abschnitt.

5.1.5 Xpand Template Language

Die folgenden Beschreibungen basieren auf dem Buch von Gronback [GG09]. In der Modellbasierten Softwareentwicklung kommt es oft vor, dass wir Modelle definieren und daraus z. B. Programmcode, Datenbankschemen oder auch Dokumentation generieren wollen. Wir benötigen also *Model-to-Text* Transformationen und genau dafür existiert das *M2T* Eclipse Projekt, welches Technologie dafür bereitstellt. *Java Emitter Templates* (*JET*) und *Xpand* sind die zwei Hauptkomponenten diese Projekts. Auf *JET* gehen wir hier nicht näher ein, da wir es nicht direkt verwenden. *Xpand* wird exzessiv von GMF benutzt, um den Editor Code zu erzeugen. *Xpand Templates* sind einfache Textdateien. Programminstruktionen stehen zwischen «»Zeichen. Text, der nicht zwischen diesen Zeichen steht, wir direkt in die resultierende Textdatei kopiert. Das erste Element einer *Xpand Template* ist das «**IMPORT**» Element. Hiermit lassen sich gewünschten Meta Modelle importieren, auf wessen Elemente dann zugegriffen werden kann. Das Konzept von «**IMPORT**» ist dem **`import`** Statement aus Java gleich. In den *Xpand Templates* des GMF *Generator Models*, sieht das «**IMPORT**» Statement so aus: «**IMPORT** "http://www.eclipse.org/gmf/2009/GenModel"». Das nächste Hauptelement ist der «**DEFINE**» Block. Hier definieren wir ein Fragment, welches bei der Ausführung der Template ausgewertet und in der Ausgabe an eine bestimmten Stelle platziert wird. Ein «**DEFINE**» Block hat einen Namen, eine optionale Parameterliste und eine **FOR** Klausel, in welcher wir das Meta Modell Element angeben, auf welches «**DEFINE**» angewendet wird. Das «**EXPAND**» Statement stellt einen Methodenaufruf dar. Es verweist auf ein anderes «**DEFINE**» Statement, um der Kontrollfluss der Template Ausführung auf diesen Block umzulenken. *Xpand* bietet noch weitaus mehr Konstrukte, doch für unsere Implementierung ist nur das «**AROUND**» Statement interessant, denn es erlaubt uns die vorhandenen Templates nur an bestimmten Stellen zu erweitern, statt diese komplett überschreiben bzw. ersetzen zu müssen. Das «**AROUND**» Statement realisiert Aspekt orientierte Eingriffsmöglichkeiten in den Code. Wir können damit gezielt ein vordefiniertes «**DEFINE**» Statement durch eigene Logik ersetzen. In Abbildung 5.17 sehen wir ein Beispiel. Links ist die `templateX.xpt` aus dem `org.eclipse.gmf.codegen` Projekt mir ihrem Namespace `path/to/template` zu sehen. Diese Template hat drei «**DEFINE**» Statements. Wir möchten in unserem Beispielprojekt `my.gmf.project` nur «**DEFINE**» **B** durch eigene Logik ersetzen, **A** und **C** sollen unverändert bleiben. Dazu kopieren wir die Template in unseren Projektordner und ergänzen den Namespace mit "`aspects`", wie Rechts in der Abbildung dargestellt. Dann

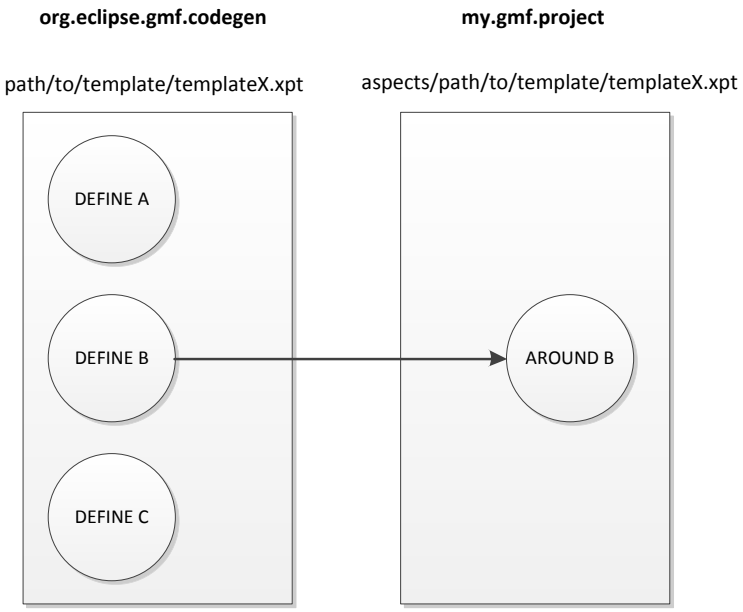

Abbildung 5.17: *Xpand* Template mit «**AROUND**» Erweiterung

löschen wir alles aus der Template raus, bis auf das «**DEFINE**» **B** Statement. Wir ersetzen dann «**DEFINE**» durch «**AROUND**» und schreiben eigene Instruktionen auf. Dies bewirkt schließlich, dass «**DEFINE**» **A**, «**AROUND**» **B** und «**DEFINE**» **C** ausgeführt werden.

5.1.6 BPEL4Chor2BPEL

Die Konzeptionelle Arbeit für diese Komponente wird in [Reio7] beschrieben. Sie bietet eine automatische Transformation von BPEL4Chor Artefakten zu abstrakten BPEL Prozessen. In Abbildung 5.18 sehen wir eine Übersicht. Wir beginnen mit einer Beschreibung eines Geschäftsprozesses. Dies kann z. B. eine Choreographie Beschreibung in BPMN sein. In Schritt **1** modellieren wir diese Choreographie in BPEL4Chor. Schritt **2** transformiert die BPEL4Chor Artefakte zu abstrakten BPEL Prozessen. In Schritt **3** werden diese Prozesse, durch eine manuell durchgeführte "executable completion", zu ausführbaren BPEL Prozessen angereichert und, in Schritt **4**, auf einem Workflowsystem ausgeführt. Schritt **1** führen wir mit unserem Choreographie Editor durch. Wie wir bereits in Kapitel 4 auf Seite 29 besprochen haben, kann unser Editor BPEL4Chor Artefakte aus dem grafischen Modell erzeugen. Für Schritt **2** benutzen wir die *BPEL4ChorToBPEL* Komponente. Schritt **3** wird zu einem sehr kleinen Teil in der *BasicExecutableCompletionTransformer* Komponente realisiert. Den übrigen

Teil müssen wir manuell, mit dem *BPEL Designer* durchführen, welcher uns dazu noch Schritt **4** ermöglicht.

Die *BPEL4ChorToBPEL* Komponente muss für alle Kommunikationsaktivitäten `partnerLink`, `portType` und `operation` Attribute sowie die zugehörigen Deklarationen der *Partner Link* Elemente erzeugen. Die *NCNames* im `properties` Attribut von `CorrelationSet`, müssen durch *QNames* ersetzt werden. In `Topology` kann definiert werden, dass eine ‹ `ForEach` › Aktivität über ein `ParticipantSet` iterieren soll. Dies ist in BPEL unzulässig. Für jede dieser ‹ `ForEach` › Aktivitäten, muss die Anzahl der Iterationen durch ‹ `startCounterValue` › und ‹ `finalCounterValue` › festgelegt werden. Für jeden BPEL Prozess müssen WSDLs mit *Partner Link Types* angelegt werden. Für die Umsetzung der "link passing mobility" (siehe Abschnitt 3.3.2 auf Seite 18) müsste auf Empfängerseite eine ‹ `assign` › Aktivität angelegt werden, welche die *endpoint reference* in das `partnerRole` Attribut des *Partner Links* kopiert. Diese Funktionalität ist konzeptionell angedacht, jedoch nicht in die Transformation integriert worden. In Abbildung 5.19 sehen wir die Ein- und Ausgaben der Komponente. Zuerst wird die *Topology* analysiert und dabei alle relevanten Daten zur Durchführung der Transformation gesammelt. Dann wird das *Grounding* analysiert und damit die gesammelten Daten erweitert. Schließlich werden alle *Participant Behavior Description* Dokumente zu abstrakten BPEL Prozessen umgewandelt. Die fehlenden Informationen werden aus den gesammelten Daten hergeleitet. Zusätzlich sieht die Komponente die Eingabe von WSDL Dokumenten vor, in welchen *Property* und *Property Alias* für *Correlation Sets* sowie *Endpoint References* für jedes *Participant Behavior Description* Dokument angegeben sind. In der Ausgabe werden die WSDLs mit *Partner Link Types* erweitert.

Eine Implementierung dieses Konzepts wurde in [Li10] erstellt. Diese Implementierung benutzen wir unverändert für diese Arbeit. Die Konzeption sieht zwar Eingaben von WSDL Dateien vor, doch wurde in der Implementierung darauf verzichtet. In der Ausgabe wird für jeden BPEL Prozess, ein WSDL Dokument mit *Partner Link Types* erzeugt. Die restlichen Elemente müssen danach ergänzt werden. Dazu kann der *BPEL Designer* unterstützend benutzt werden.

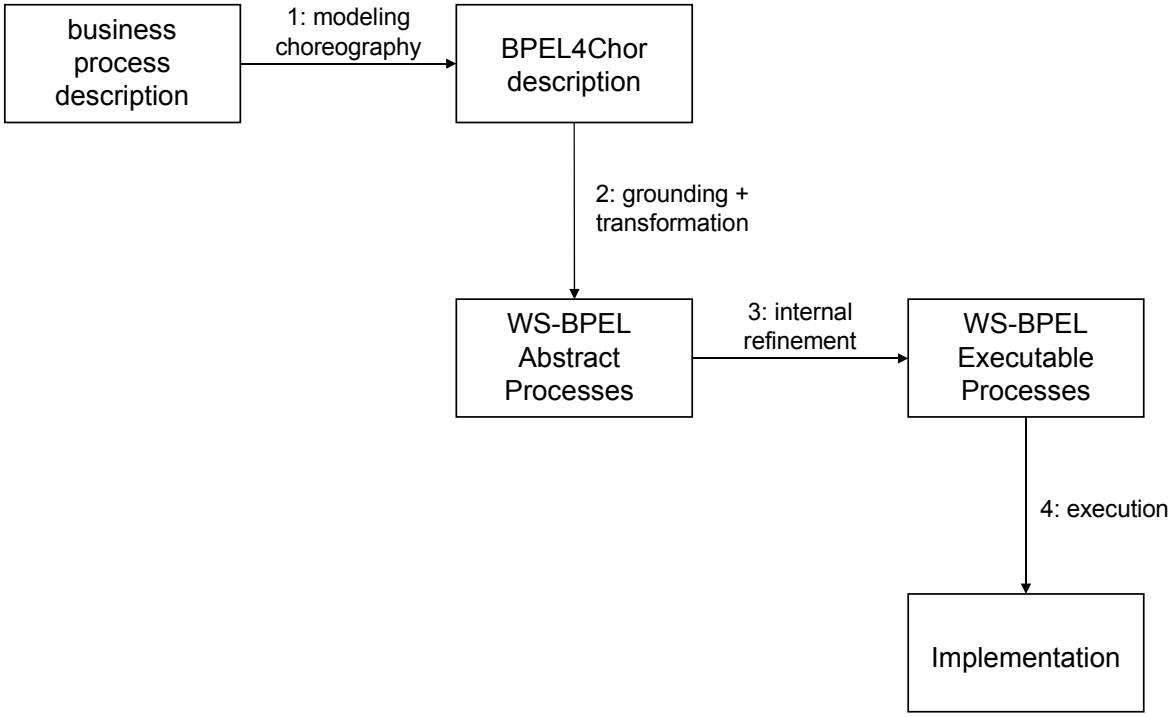

Abbildung 5.18: Von einer Beschreibung eines Geschäftsprozess zu ausführbaren BPEL Prozessen. Quelle: [Rei07]

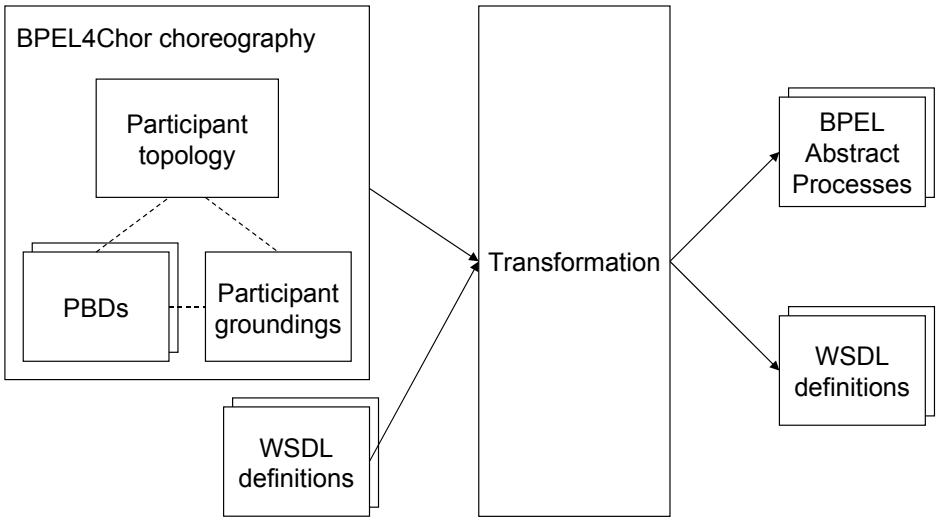

Abbildung 5.19: Ein- und Ausgaben der *BPEL4ChorToBPEL* Komponente. Quelle: [Rei07]

5.2 Chor Designer

In diesem Abschnitt beschreiben wir, wie die vorgestellten Technologien zum Einsatz kommen, um unseren Choreographie Editor umzusetzen. Wir gehen hier nur auf wesentliche Aspekte der Realisierung ein. In Anlehnung an den *BPEL Designer*, nennen wir unseren Choreographie Editor *Chor Designer*

5.2.1 EMF Modelle

Wie wir in der Konzeption in Abschnitt 4.5.1 auf Seite 35 besprochen haben, arbeiten wir mit den vier Datenmodellen Topology Model, Grounding Model, PBD Model und Chor Model. Wir realisieren diese Modelle in *EMF*. Dabei verwenden wir keine Import Funktion, sondern modellieren diese von Hand. Topology Model [DK12b] und Grounding Model [DK12a] stehen bereits als XSD Dateien zur Verfügung. Das PBD Model erstellen wir aus dem WS-BPEL Standard "abstract common base" [OAS07a] der als XSD vorliegt, und berücksichtigen dabei die in Abschnitt 3.3.1 auf Seite 17 besprochenen Einschränkungen, so dass das *Ecore* Modell zum *Abstract Process Profile for Participant Behavior Descriptions* passt. Als Beispiel beschreiben wir die Modellierung des Topology Model. Die Modellierung des Grounding Model ist analog. Für das PBD Model verfahren wir zum größten Teil gleich, bis auf einige Ausnahmen auf welche wir in Abschnitt 5.2.1 auf der nächsten Seite eingehen. Das Chor Model modellieren wir leicht abweichend vom der Konzeption und gehen darauf in Abschnitt 5.2.1 auf Seite 112 ein.

Topology Model

In Abbildung 5.20 sehen wir das Topology Model in *Ecore* modelliert. Wir erstellen die `topology.ecore` Datei und belegen die Attribute des `EPackage` Knoten mit den Werten *Name*="topology", *Ns Prefix*="top"und *Ns URI*="`urn:HPI_IAAS:choreography:schemas:choreography:topology:2006/12`". Dann definieren wir für die XSD Elemente mit Typ Attribut, welche in Listing 5.1 dargestellt sind, jeweils eine `EClass`. In Listing 5.2 sehen wir die XSD Schema Typ Definition vom `topology` Element. Aus Zeile **7** und **8** machen wir `EAttribute` Einträge wobei wir die XSD Attribute `type="xs:NCName"` und `type="xs:anyURI"` als `EString` Datentyp übernehmen. `use="required"` realisieren wir mit den EMF Properties *Upper Bound*=*Lower Bound*="1". Die Zeilen **3 - 5** werden als `EReference` angelegt und verweisen auf die entsprechenden `EClass` Einträge. Für die restlichen XSD Einträge verfahren wir analog, weisen hier aber noch auf drei Besonderheiten hin. In Listing 5.3 sehen wir die XSD Schema Typ Definition vom `Participant` Element. Wir realisieren den Typ `type="xs:QName"` als `EDataType`, womit wir die Java Klasse `javax.xml.namespace.QName` repräsentieren. Den Datentyp des Elements aus Zeile **7** realisieren wir als `EEnum` mit den entsprechenden Werten aus den Zeilen **10 - 12**.

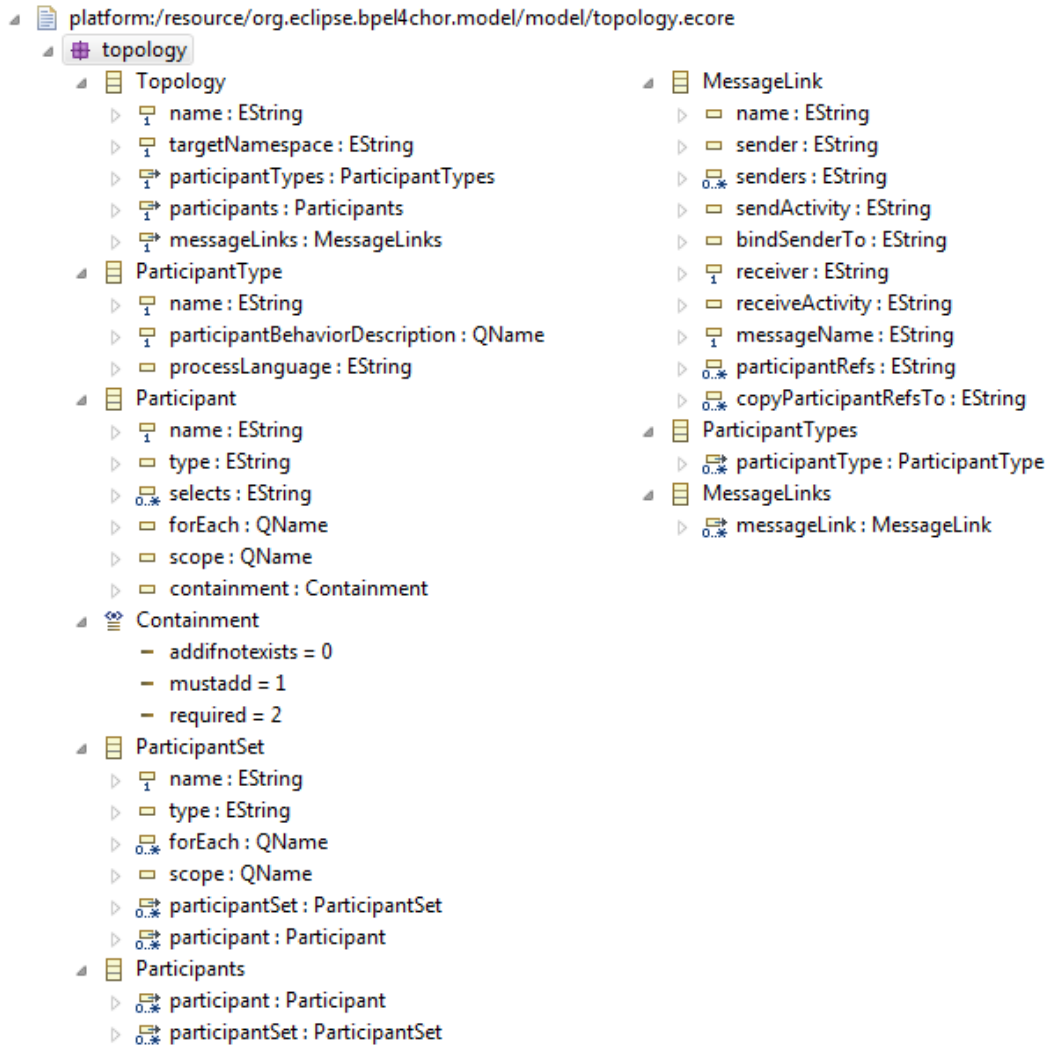

Abbildung 5.20: Topology Model als *Ecore* Modell

Mehrwertige Attribute, wie z. B. in Zeile **4**, realisieren wir durch setzen der EMF Property *Upper Bound*= **−1**.

PBD Model

Bei der Umsetzung vom PBD Model gibt es einige Besonderheiten, welche in der Ausdrucksweise von XSD liegen. Wir können in einem *Ecore* Modell nicht alles exakt so umsetzen, wie es in einer XSD vorgegeben ist. In Listing 5.4 sehen wir die XSD Schema Typ Definition vom `ExtensibleElements` Element. BPEL ist eine erweiterbare Sprache, was hier mit

Listing 5.1 XSD Schema Element Definitionen vom Topology Model. Quelle: [DK12b]

```
01 <xs:element name="topology" type="tTopology" />
02 <xs:element name="participantTypes" type="tParticipantTypes" />
03 <xs:element name="participantType" type="tParticipantType" />
04 <xs:element name="participants" type="tParticipants" />
05 <xs:element name="participant" type="tParticipant" />
06 <xs:element name="participantSet" type="tParticipantSet" />
07 <xs:element name="messageLinks" type="tMessageLinks" />
08 <xs:element name="messageLink" type="tMessageLink" />
```

Listing 5.2 XSD Schema Typ Definition von `Topology`. Quelle: [DK12b]

```
01 <xs:complexType name="tTopology">
02    <xs:sequence>
03       <xs:element ref="participantTypes" minOccurs="1" maxOccurs="1" />
04       <xs:element ref="participants" minOccurs="1" maxOccurs="1" />
05       <xs:element ref="messageLinks" minOccurs="1" maxOccurs="1" />
06    </xs:sequence>
07    <xs:attribute name="name" type="xs:NCName" use="required" />
08    <xs:attribute name="targetNamespace" type="xs:anyURI" use="required" />
09 </xs:complexType>
```

Listing 5.3 XSD Schema Typ Definition von `Participant`. Quelle: [DK12b]

```
01 <xs:complexType name="tParticipant">
02    <xs:attribute name="name" type="xs:NCName" use="required" />
03    <xs:attribute name="type" type="xs:NCName" use="optional" />
04    <xs:attribute name="selects" type="NCNames" />
05    <xs:attribute name="forEach" type="xs:QName" />
06    <xs:attribute name="scope" type="xs:QName" />
07    <xs:attribute name="containment" use="optional" default="add-if-not-exists">
08       <xs:simpleType>
09          <xs:restriction base="xs:string">
10             <xs:enumeration value="required" />
11             <xs:enumeration value="must-add" />
12             <xs:enumeration value="add-if-not-exists" />
13          </xs:restriction>
14       </xs:simpleType>
15    </xs:attribute>
16 </xs:complexType>
```

Listing 5.4 XSD Schema Typ Definition von `ExtensibleElements`. Quelle: [OAS07a]

```
01 <xsd:complexType name="tExtensibleElements">
02    ...
03    <xsd:sequence>
04       <xsd:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
05       <xsd:any namespace="##other" processContents="lax" minOccurs="0"
    maxOccurs="unbounded"/>
06    </xsd:sequence>
07    <xsd:anyAttribute namespace="##other" processContents="lax"/>
08 </xsd:complexType>
```

Listing 5.5 XSD Schema Typ Definition von `Process`. Quelle: [OAS07a]

```
01 <xsd:complexType name="tProcess">
02    <xsd:complexContent>
03       <xsd:extension base="tExtensibleElements">
04          <xsd:sequence>
05          ...
06          <xsd:group ref="activity" minOccurs="0"/>
07          ...
```

diesem Element realisiert wird. Fast alle BPEL Elemente erben von diesem Element. Zeile **5** macht sich das `any` Element[9] zu nutze, mit welchem sich beliebige Elemente hinzufügen lassen, die nicht im BPEL Schema deklariert sind. Genau die selbe Erweiterbarkeit wird für Attribute, in Zeile **7**, mit dem `anyAttribute` Element[10] erreicht. Wir setzen dies in *Ecore* so um, dass für `anyAttribute` ein mehrwertiges `EAttribute` mit dem Datentyp `EJavaObject`, und für `any`, eine mehrwertige `EReference` mit `EObject` als Typ erstellt wird. Bei der Serialisierung des *Ecore* Modelles muss für diese spezielle Umsetzung eine extra Behandlung entworfen werden, um syntaktisch korrektes XML zu erzeugen. Für unsere Implementierung verzichten wir auf die Erweiterbarkeit von BPEL.

Eine weitere Besonderheit ist in Listing 5.5, bei der Definition vom `Process` Element zu sehen. Wir wissen das BPEL Prozesse immer nur eine Aktivität haben können. Dies wird hier im XSD Schema, in Zeile **6**, durch Referenzierung eines `Group` Elements erreicht. Dazu sehen wir uns Listing 5.6 an. Hier sehen wir die Gruppendefinition von `Activity`. Was dies zum Ausdruck bringen soll ist, dass diese eine `Activity`, referenziert von `Process`, immer nur ein Eintrag aus dieser Gruppe sein darf. Das heißt entweder `Empty` oder `Flow` oder `Sequence` oder usw.. In *Ecore* setzen wir dies ganz simpel damit um, dass `Process` eine `EReference` auf `Activity` hat, sowie alle konkreten Aktivitäten von `Activity` erben.

Für die Umsetzung des `Expression` Elements betrachten wir Listing 5.7. Hier wird ebenfalls `anyAttribute` und `any` benutzt, jedoch gibt es hier einen Unterschied zum

[9] http://www.w3schools.com/schema/schema_complex_any.asp
[10] http://www.w3schools.com/schema/schema_complex_anyattribute.asp

Listing 5.6 XSD Schema Typ Definition der `Activity` Gruppe. Quelle: [OAS07a]

```
<xsd:group name="activity">
   ...
   <xsd:choice>
      <xsd:element ref="empty"/>
      <xsd:element ref="flow"/>
      <xsd:element ref="sequence"/>
      ...
```

Listing 5.7 XSD Schema Typ Definition von `Expression`. Quelle: [OAS07a]

```
01 <xsd:complexType name="tExpression" mixed="true">
02    <xsd:sequence>
03       <xsd:any minOccurs="0" maxOccurs="unbounded" processContents="lax"/>
04    </xsd:sequence>
05    <xsd:attribute name="expressionLanguage" type="xsd:anyURI"/>
06    <xsd:anyAttribute namespace="##other" processContents="lax"/>
07 </xsd:complexType>
```

`ExtensibleElements` Element welcher, in Zeile **1**, im *mixed*="true"[11] Attribut liegt. Dies bedeutet, dass das `Expression` Element weitere Attribute, Elemente und auch Text, in gemischter Form beinhalten kann. Wir realisieren dies in *Ecore* indem wir die `Query EClass` mit einem weiteren Attribut *body* vom Typ `EJavaObject` versehen. Bei der Serialisierung muss dafür eine extra Behandlung eingebaut werden.

Chor Model

Bei der Konzeption in Abschnitt 4.5.1 auf Seite 35 haben wir das Chor Model so definiert, dass das Wurzelelement `Choreography` in einer $1 : n$ Beziehung mit `CParticipantCommon` – der Generalisierung von `CParticipant` und `CParticipantSet` – steht (siehe dazu Abbildung 4.5). Dies ändern wir in unserer *Ecore* Umsetzung, welche in Abbildung 5.21 zu sehen ist, so ab, dass wir direkt von `Choreography` eine mehrwertige `EReference` auf jeweils `CParticipant` und `CParticipantSet` setzen. Der Gründe dafür sind ein vereinfachter Zugriff beim Lesen der Modellinstanz. Hier sparen wir uns die Prüfung mit dem Java Operator **`instanceof`**, ob `CParticipantCommon` eine Instanz von `CParticipant` oder `CParticipantSet` ist. Der andere Grund liegt in einer Feststellung, welche eventuell auf einen Fehler im generierten GMF Code zurück zu führen ist. Zieht man im generieren GMF Editor einen *Message Link* zwischen zwei Aktivitäten welche zu unterschiedlichen *Participants* gehören, wird die *Connection* doppelt dargestellt sprich zwei Pfeile, statt nur einem, erscheinen auf der Zeichenfläche. Die Ursache dafür ist unbekannt bzw. lies sich in

[11] http://www.w3schools.com/schema/schema_complex_mixed.asp

dieser Arbeit nicht herausfinden. Eine sichere Lösung besteht darin, von `Choreography` direkt auf `CParticipant` und `CParticipantSet` zu referenzieren. Die ursprüngliche Idee war sogar, alle ausgehenden Referenzen von `Choreography` auf eine abstrakte Klasse `ChoreographyElement` zu legen, von welcher alle anderen Klassen erben sollten. Dies erzeuge jedoch den selben Fehler.

Eine weitere Besonderheit liegt in der Standard Serialisierung von EMF. Der generierte GMF Editor benutzt diese um das Chor Model zu serialisieren und auf dem Datenträger abzuspeichern. Jedoch kennt EMF keine generische Serialisierung von `javax.xml.namespace.QName`[12], welche als Datentyp unter anderem für das *portType* Attribut von `CMessageLinkGrounding` in Frage kommt. Daher haben wir eine neue Klasse `CQname` eingeführt, welche die drei Attribute von QName `namespaceURI`, `localPart` und `prefix` übernimmt.

Serialisierung

EMF serialisiert *Ecore* Modelle standardmäßig in Form von XMI[13] , da wir aber beim Export des Chor Model syntaktisch korrekte BPEL4Chor Artefakte benötigen, müssen wir für eine XML Serialisierung sorgen. Um das Problem zu verdeutlichen, betrachten wir in Abbildung 5.22 einen BPEL Testprozess und in Listing 5.8 die zugehörige Standard Serialisierung der PBD Model *Ecore* Instanz. Der ‹ `process` › Wurzelknoten ist syntaktisch korrekt, nur bei den Aktivitäten liegt kein korrektes BPEL mehr vor, da hier ‹ `activity` › mit Referenz auf den konkreten Typ (`xsi:type=...`) angegeben wird. Des weiteren benötigen wir Ausnahmebehandlungen von Elementen wie `Expression`, da sonst die Seriealisierung das *body* Attribut auch tatsächlich als Attribut ausgibt. Ein Beispiel wäre die ‹ `for` › `Expression`, welche auf folgende Art serialisiert werden würde: ‹ `for body=...` ›. Korrekt wäre aber: ‹ `for` ›`...` ‹ `/for` ›. Daher implementieren wir ein eigenen Algorithmus zur Transformation des *Ecore* Modells in einen DOM Baum, wie wir es bereits in Abschnitt 4.5.2 auf Seite 50 besprochen haben. Das Resultat ist, nach XML Serialisierung des DOM Baumes, syntaktisch korrektes BPEL, wie es in Listing 5.9 zu sehen ist.

[12] http://www.eclipse.org/forums/index.php?t=msg&th=126150/
[13] XML Metadata Interchange

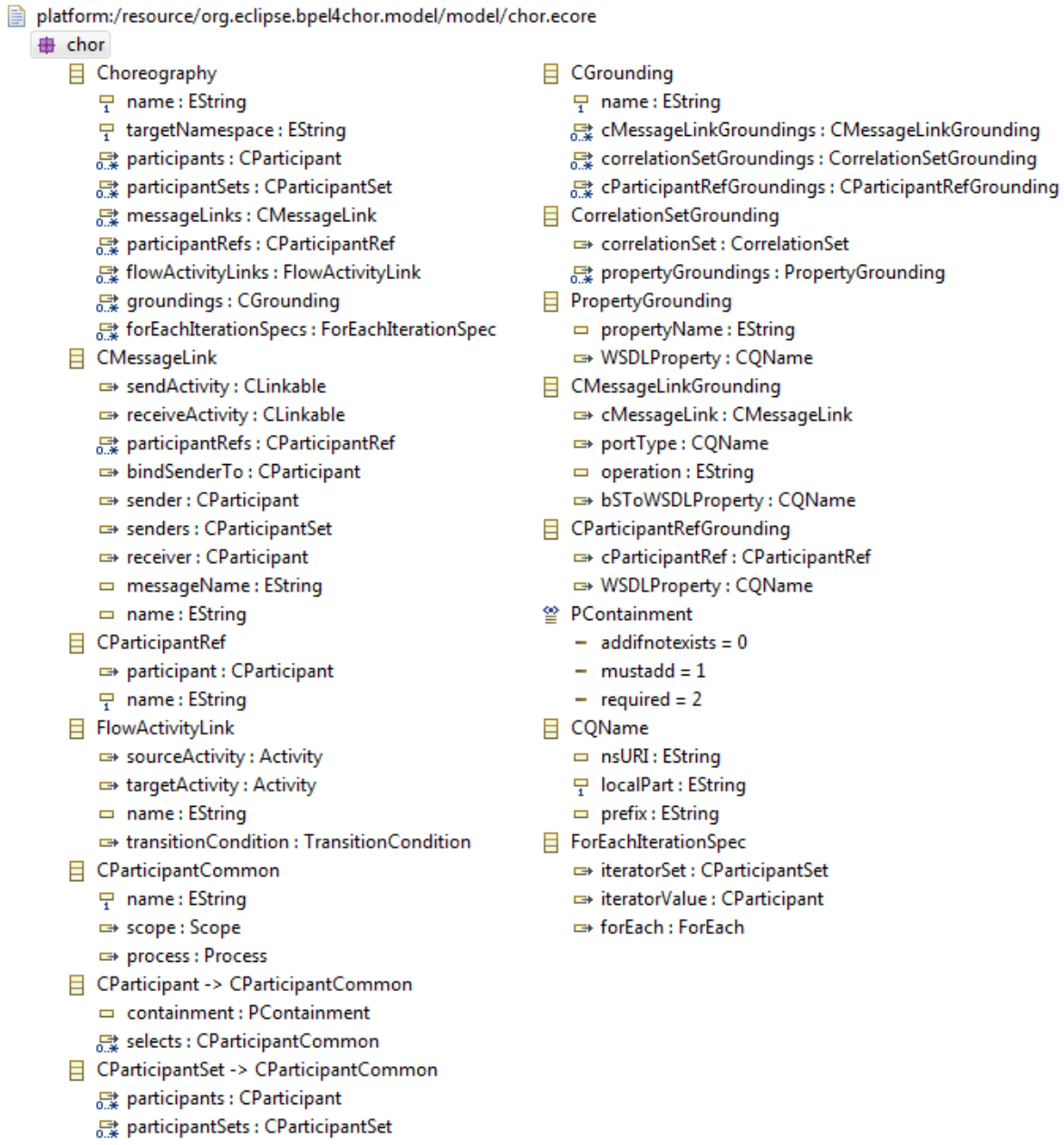

Abbildung 5.21: Chor Model als *Ecore* Modell

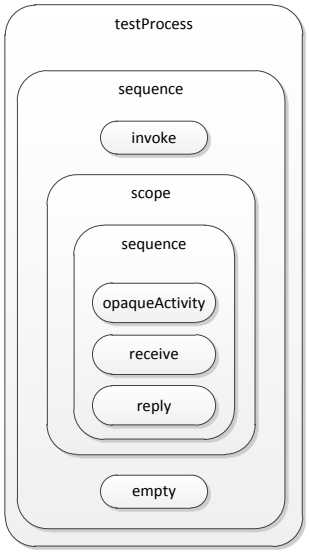

Abbildung 5.22: BPEL Testprozess

Listing 5.8 PBD Model *ECore* Standard XMI Serialisierung

```
01 <pbd:Process xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   name="testProcess" ...>
02    <pbd:activity xsi:type="pbd:Sequence">
03       <pbd:activity xsi:type="pbd:Invoke" outputVariable="##opaque"/>
04       <pbd:activity xsi:type="pbd:Scope" name="testScope">
05          <pbd:activity xsi:type="pbd:Sequence">
06             <pbd:activity xsi:type="pbd:OpaqueActivity" name="testOpaque"/>
07             <pbd:activity xsi:type="pbd:Receive" variable="##opaque"
   messageExchange="msgExA"/>
08             <pbd:activity xsi:type="pbd:Reply" variable="##opaque"
   messageExchange="msgExA"/>
09          </pbd:activity>
10       </pbd:activity>
11       <pbd:activity xsi:type="pbd:Empty" name="emptyActivity"/>
12    </pbd:activity>
13 </pbd:Process>
```

Listing 5.9 PBD Model *ECore* XML Serialisierung durch Transformation nach DOM

```
01 <process name="testProcess" ...>
02    <sequence>
03       <invoke outputVariable="##opaque"/>
04       <scope name="testScope">
05          <sequence>
06             <opaqueActivity name="testOpaque"/>
07             <receive messageExchange="msgExA" variable="##opaque"/>
08             <reply messageExchange="msgExA" variable="##opaque"/>
09          </sequence>
10       </scope>
11       <empty name="emptyActivity"/>
12    </sequence>
13 </process>
```

5.2.2 Graphical Definition Model

In diesem Modell definieren wir das Aussehen unserer Modellelemente. In Abbildung 5.23 sehen wir im linken Teil das Konzept, wie wir `CParticipant` Elemente darstellen. Ein äußeres Rechteck mit zwei inneren Rechtecken. Das obere Rechteck beinhaltet ein Label, welches den Namen des *Participants* darstellt. Das untere Rechteck stellt die Zeichenfläche dar, in welcher wir das zugehörige `Process` Element platzieren wollen. Im rechten Teil der Abbildung 5.23 sehen wir den *Figure Descriptor* des Graphical Definition Model. Für die Platzierung der inneren Rechtecke und des Labels verwenden wir Layout Manager. Das äußerste Rechteck wird mit einem `BorderLayout` versehen. Das innere Rechteck (`CParticipantNameFigure`), welches das Label beinhaltet , platzieren wir in Norden (`BEGINNING`) und das andere Rechteck (`CParticipantCompartmentFigure`) platzieren wir in der Mitte (`CENTER`). Das Label platzieren wir, mittels `FlowLayout`, im `CParticipantNameFigure` Rechteck. Die Nummern am rechten Rand der Abbildung verweisen auf die Code Zeilen in Listing 5.10, welches den resultierenden Java Code darstellt, der aus diesem *Figure Descriptor* generiert wird. Das Design der anderen Elemente ist diesem hier sehr ähnlich. Für die Elemente, welche keine verschachtelten Elemente darstellen wie z. B. `Invoke`, `Receive`, `Reply` usw. sprich, alle nicht strukturierten Aktivitäten, benötigen wir kein Rechteck für das *Compartment*. Lediglich Verbinder haben ein gänzlich anderes Design.

In Abbildung 5.24 sehen wir im linken Teil das Konzept, wie wir `CMessageLink` Elemente darstellen. Verbinder haben immer eine Quelle und ein Ziel und wir „dekorieren“ nur die Zielseite. Diese Dekoration wird als Pfeil dargestellt. Zudem soll es noch ein Label geben, welches den Namen des *Message Links* anzeigt. Im rechten Teil der Abbildung 5.24 ist der *Figure Descriptor* zu sehen, sowie die Definition des Pfeils in Form einer Polygon Beschreibung. Die Linie selbst ist als *Polyline Connection* definiert was zum einen spezielle Linie ist, welche durch mehrere Punkte – zwischen Anfangs und Endpunkt – gezogen wird

und zum anderen eine *Connection*, wie wir sie bereits in Abschnitt 5.1.3 auf Seite 92 vorgestellt haben. Über die Property `Target Decoration` können wir die Pfeildekoration zuweisen welche wir, separat von diesem *Figure Descriptor*, als Polygon Beschreibung definieren. Die Polygon Beschreibung besteht aus einer Punktliste und wird so gelesen, wie es schematisch Abbildung 5.24 dargestellt ist. Die Leserichtung ist gegen den Uhrzeigersinn und beginnt mit Punkt $(x, y) = (0, 0)$, geht dann zu $(-2, 2)$, über $(-2, -2)$ wieder zurück zu $(0, 0)$. Somit erhalten wir ein geschlossenes Dreieck mit schwarzer Hintergrundfarbe (`Background: black`). Am rechten Rand der Abbildung sind die Code Zeilen aus Listing 5.11 zu sehen, welche den zugehörigen Java Code zeigen.

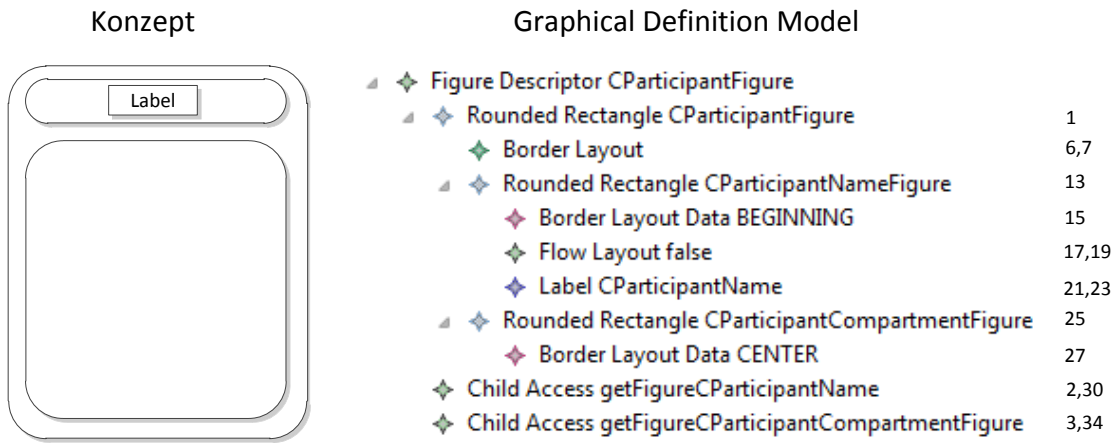

Abbildung 5.23: Grafische Repräsentation von `CParticipant`

Listing 5.10 Generierter Java Code aus dem *Figure Descriptor* von `CParticipant`

```
01 public class CParticipantFigure extends RoundedRectangle {
02    private WrappingLabel fFigureCParticipantName;
03    private RoundedRectangle fFigureCParticipantCompartmentFigure;
04
05    public CParticipantFigure() {
06       BorderLayout layoutThis = new BorderLayout();
07       this.setLayoutManager(layoutThis);
08       ...
09       createContents();
10    }
11
12    private void createContents() {
13       RoundedRectangle cParticipantNameFigure0 = new RoundedRectangle();
14       ...
15       this.add(cParticipantNameFigure0, BorderLayout.TOP);
16
17       FlowLayout layoutCParticipantNameFigure0 = new FlowLayout();
18       ...
19       cParticipantNameFigure0.setLayoutManager(layoutCParticipantNameFigure0);
20
21       fFigureCParticipantName = new WrappingLabel();
22       ...
23       cParticipantNameFigure0.add(fFigureCParticipantName);
24
25       fFigureCParticipantCompartmentFigure = new RoundedRectangle();
26       ...
27       this.add(fFigureCParticipantCompartmentFigure, BorderLayout.CENTER);
28    }
29
30    public WrappingLabel getFigureCParticipantName() {
31       return fFigureCParticipantName;
32    }
33
34    public RoundedRectangle getFigureCParticipantCompartmentFigure() {
35       return fFigureCParticipantCompartmentFigure;
36    }
37 }
```

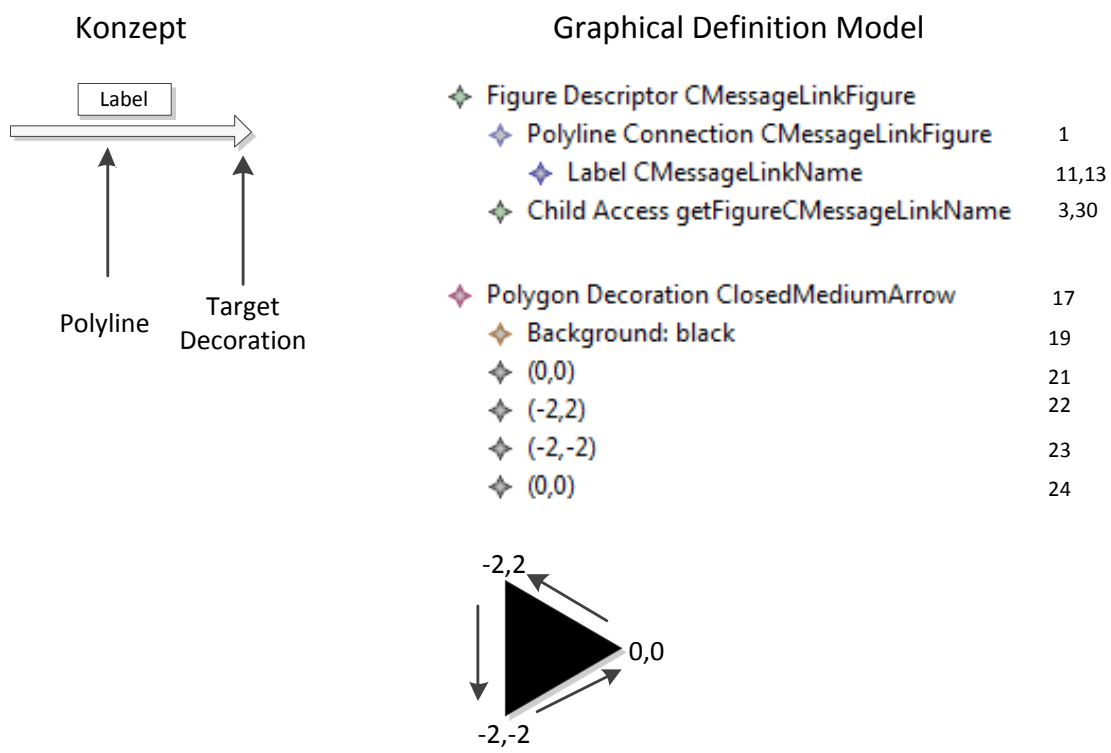

Abbildung 5.24: Grafische Repräsentation von `CMessageLink`

Listing 5.11 Generierter Java Code aus dem *Figure Descriptor* von `CMessageLink`

```
01 public class CMessageLinkFigure extends PolylineConnectionEx {
02
03    private WrappingLabel fFigureCMessageLinkName;
04
05    public CMessageLinkFigure() {
06       createContents();
07       setTargetDecoration(createTargetDecoration());
08    }
09
10    private void createContents() {
11       fFigureCMessageLinkName = new WrappingLabel();
12       ...
13       this.add(fFigureCMessageLinkName);
14    }
15
16    private RotatableDecoration createTargetDecoration() {
17       PolygonDecoration df = new PolygonDecoration();
18       df.setFill(true);
19       df.setBackgroundColor(ColorConstants.black);
20       PointList pl = new PointList();
21       pl.addPoint(getMapMode().DPtoLP(0), getMapMode().DPtoLP(0));
22       pl.addPoint(getMapMode().DPtoLP(-2), getMapMode().DPtoLP(2));
23       pl.addPoint(getMapMode().DPtoLP(-2), getMapMode().DPtoLP(-2));
24       pl.addPoint(getMapMode().DPtoLP(0), getMapMode().DPtoLP(0));
25       df.setTemplate(pl);
26       ...
27       return df;
28    }
29
30    public WrappingLabel getFigureCMessageLinkName() {
31       return fFigureCMessageLinkName;
32    }
33 }
```

5.2.3 Tooling Definition Model

In diesem Modell definieren wir für alle Elemente, welche wir auf der Zeichenfläche des Editors platzieren, unsere Werkzeuge. In Abbildung 5.25 ist im linken Teil das Modell zu sehen und im rechten Teil, das Resultat im generierten Editor. Wir definieren *Tool Groups* um optisch die zusammengehörenden Werkzeuge zu gruppieren. Jede Gruppe hat die Property `Collapsible`= *true*, so dass jede einzeln ein-, oder ausgeblendet werden kann. Für jedes *Creation Tool* lässt sich zudem ein Icon festlegen. In dieser Abbildung haben wir die Standard Icons verwendet. Im oberen Bereich der Palette sind Selektion und Zoom Tools zu sehen, welche standardmäßig von GMF generiert werden. Der generierte Java Code befindet sich in der `ChorPaletteFactory` (wir haben im Tooling Definition Model unsere Palette „chorPa-

lette“genannt) welche Instanzen von `org.eclipse.gef.palette.PaletteContainer` für jede *Tool Group* und Instanzen von `org.eclipse.gef.palette.ToolEntry` für jedes *Creation Tool* erzeugt.

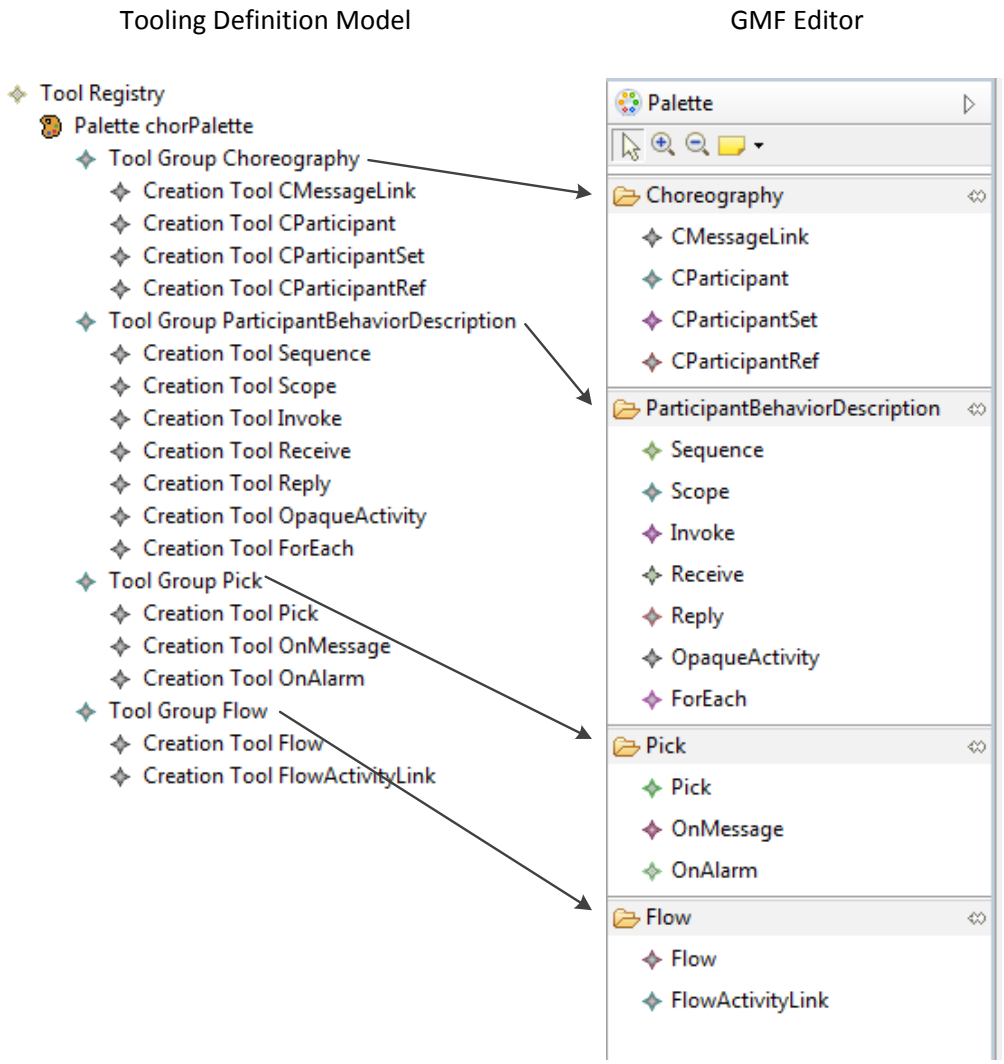

Abbildung 5.25: Tooling Definition Model und die Umsetzung im GEF Editor

5.2.4 Mapping Definition Model

In diesem Modell verbinden wir alle anderen Modelle miteinander und definieren somit unseren Editor. Die visuelle Repräsentation der Zeichenfläche ist das *Canvas* Element des

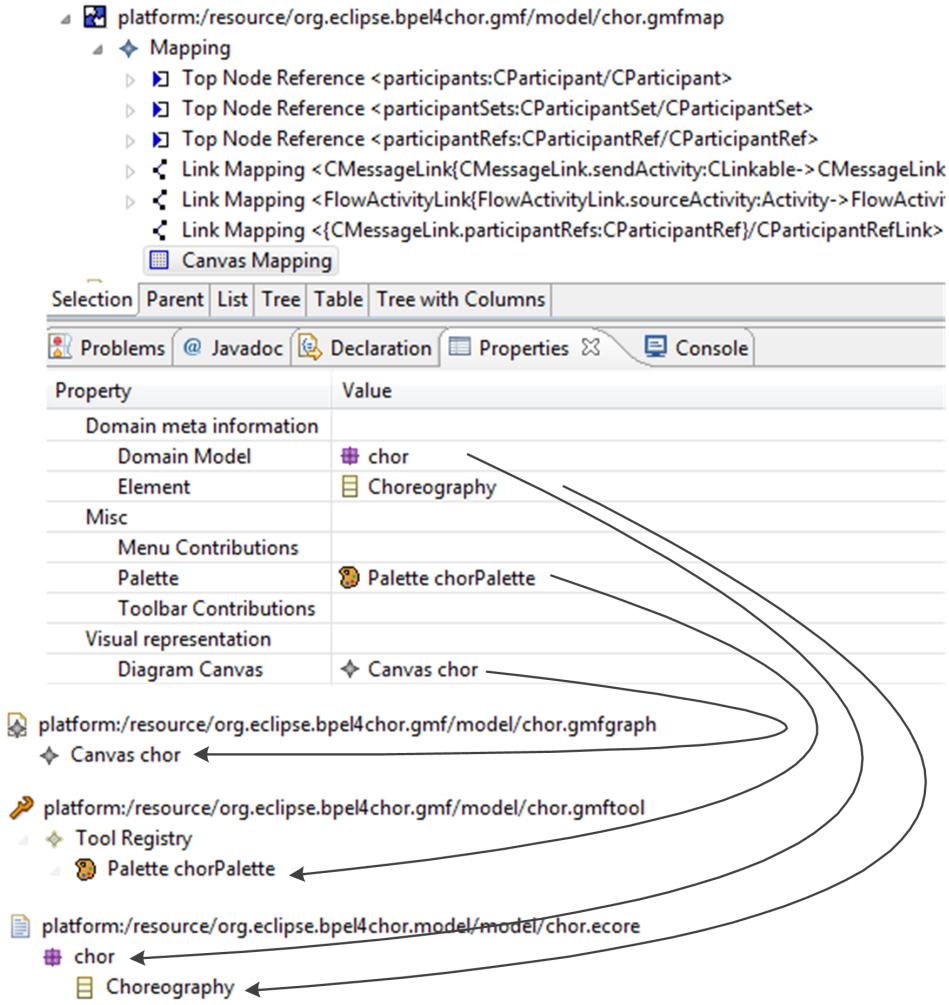

Abbildung 5.26: Definition der Zeichenfläche (*Canvas*) im Mapping Definition Model

Graphical Definition Model, unter welchem wir alle Repräsentationen unserer Elemente festgelegt haben. Das zugehörige Wurzelelement aus dem Chor Model ist die `Choreography` Klasse und unsere Werkzeuge haben wir in der *chorPalette* im Tooling Definition Model definiert. In Abbildung 5.26 sehen wir das *Canvas Mapping* Element mit seinen Properties und den Modellen, auf welche wir verweisen. Im folgenden beschreiben wir einen Teil der Mappings von `CParticipant`, `Process` und `CMessageLink`. Das Mapping der restlichen Elemente verläuft Analog.

`CParticipant` Mapping Definition

`CParticipant` Elemente platzieren wir direkt auf der Zeichenfläche. Das Wurzelelement der Zeichenfläche ist das `Choreography` Element. `CParticipant` Elemente werden von diesem referenziert. Im Mapping Definition Model werden Referenzen, vom Wurzelelement ausgehend, als *Top Node Reference* bezeichnet. In Abbildung 5.27 sehen wir die Definition der *Top Node Reference*. Mit dem Attribut *`Containment Feature`* verweisen wir auf die entsprechende `EReference` von `Choreography`. Mit dieser Definition haben wir festgelegt, dass alle Instanzen von `CParticipant` zur `Choreography` Zeichenflächeninstanz gehören und auch dort gesammelt werden. Nachdem die Referenz definiert ist, müssen wir festlegen, wie die Instanzen erstellt werden, wie sie aussehen und was für Eigenschaften diese haben sollen. Dies machen wir mit dem *Node Mapping* Element und betrachten dazu Abbildung 5.28. Mit dem *`Element`* Attribut legen wir das *Ecore* Element fest, welches wir hier modellieren wollen. Zudem muss es zur übergeordneten *Node Reference* passen, weshalb hier auch nur `CParticipant` angegeben werden kann. Mittels *Diagram Node* Attribut geben wir das Aussehen von `CParticipant` an, welches wir zuvor schon im Graphical Definition Model festgelegt haben. Wir möchten Instanzen von `CParticipant` über die Palette erstellen, daher verweisen wir auf unser bereits definiertes *Creation Tool* aus dem Tooling Definition Model. Das *`name`* Attribut von `CParticipant` soll auch auf der Zeichenfläche erscheinen und zudem editierbar sein. Um dies zu erreichen, definieren wir ein *Feature Label Mapping* wie in Abbildung 5.29 zu sehen ist. Mit dem Attribut *`Features to display`* geben wir das zugehörige *Ecore* Attribut an. In unserem Fall ist dies das *`name`* Attribut aus `CParticipantCommon`. Mittels *`Diagram Label`* Attribut geben wir noch das entsprechende Aussehen an und verweisen auf das passende *Diagram Label* Element aus dem Graphical Definition Model. Mit dem *`Edit Method`* Attribut lässt sich festlegen, wie der vom Benutzer eingegebene Text vom GMF Editor analysiert werden soll und mit dem *`View Method`* Attribut wird festgelegt, wie das Label auf der Zeichenfläche dargestellt wird [ecla]. `MESSAGE_FORMAT` bedeutet dabei, dass das Format von `java.text.MessageFormat` bestimmt wird [ecla]. Wir werden später, beim Mapping von Aktivitäten, genauer darauf eingehen. Mittels *Child Reference* Knoten können wir ausgehende Referenzen vom übergeordneten *Node Mapping* Element, in unserem Fall `CParticipant`, modellieren. `CParticipant` hat eine Referenz auf `Process` und wir möchten diesen mit all seinen verschachtelten Aktivitäten innerhalb von `CParticipant` darstellen. Grafisch haben wir schon für diese Möglichkeit gesorgt, indem wir ein *Compartment* im Graphical Definition Model dafür erstellt haben. Wir müssen im Mapping definieren, welche Instanzen in diesem *Compartment* platziert werden dürfen. Dazu definieren wir ein *Compartment Mapping* Element, wie in Abbildung 5.30 zu sehen ist. Mittels *`Children`* Attribut bestimmen wir alle Referenzen, welche in diesem *Compartment* aufgenommen werden. In unserem Fall ist dies nur die Referenz auf `Process`. Das passende Aussehen bzw. die Definition, wo genau innerhalb der `CParticipant` Repräsentation die `Process` Repräsentation dargestellt werden soll,

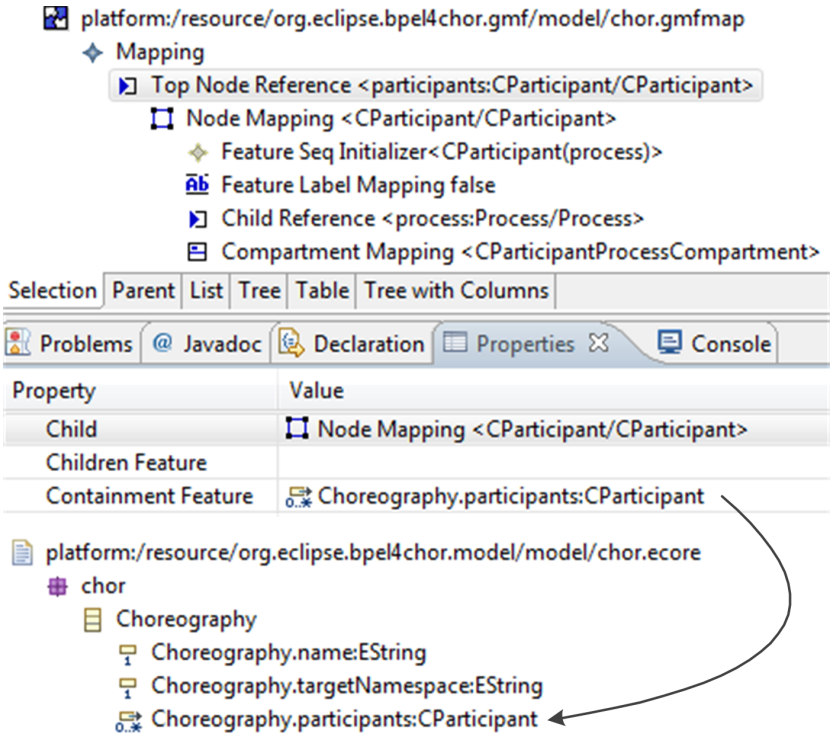

Abbildung 5.27: Definition der *Top Node Reference* von `CParticipant` im Mapping Definition Model

bestimmen wir mittels Verweis auf das *Compartment* Element aus dem Graphical Definition Model.

Wir haben in der Konzeption in Abschnitt 4.5.1 auf Seite 35 die Entscheidung getroffen, dass jeder *Participant*, bei seiner Erstellung, einen eigenen Prozess hat. Dies realisieren wir, indem wir bei jeder Instanz von `CParticipant` auch gleich eine neue Instanz von `Process` mit anlegen. Da es zudem seltenst der Fall ist, dass ein Prozess insgesamt nur aus einer einzelnen Aktivität besteht, legen wir auch eine neue Instanz von `Sequence` mit an. Dieses Verhalten können wir mittels *Feature Seq Initializer* Element, wie in Abbildung 5.31 zu sehen, modellieren. Wir beschreiben nacheinander die nummerierten Zeilen in dieser Abbildung. In Zeile **1** definieren wir, dass die neue `Process` Instanz von `CParticipant` aus referenziert und abgelegt wird. Zeile **2** erstellt eine neue Instanz von `Process`. In Zeile **8** definieren wir, dass bei dieser Instanziierung das Attribut *`abstractProcessProfile`* mittels Wert aus Zeile **9** initialisiert wird. In Zeile **4** geben wir an dass, im Zuge der Instanziierung von `Process`, die neue `Sequence` Instanz von `Process` als `Activity` referenziert wird. In Zeile **5** erstellen wir eine neue Instanz von `Sequence` und in Zeile **6** initialisieren wir das Attribut *`name`* mit dem Wert aus Zeile **7**.

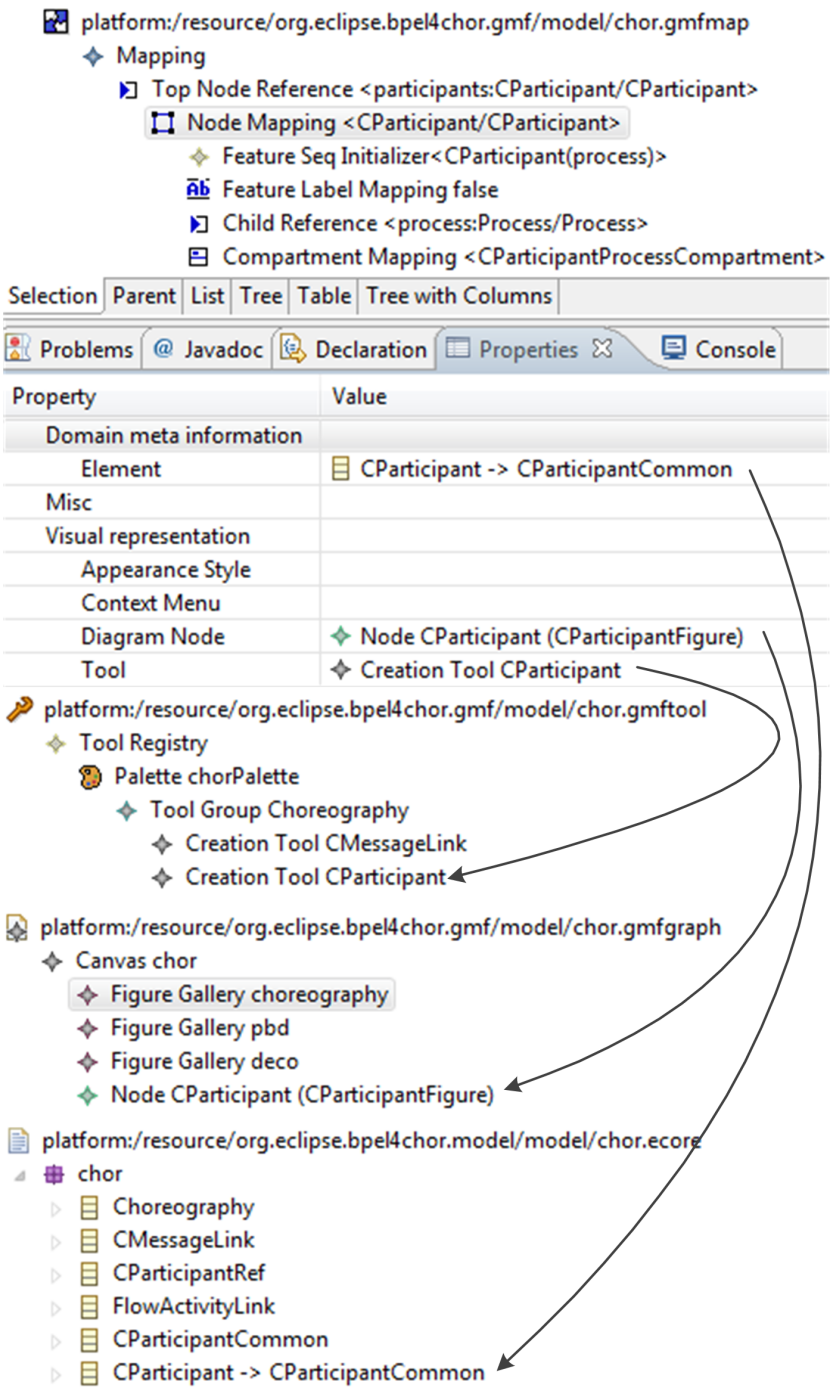

Abbildung 5.28: Definition des *Node Mapping* von `CParticipant` im Mapping Definition Model

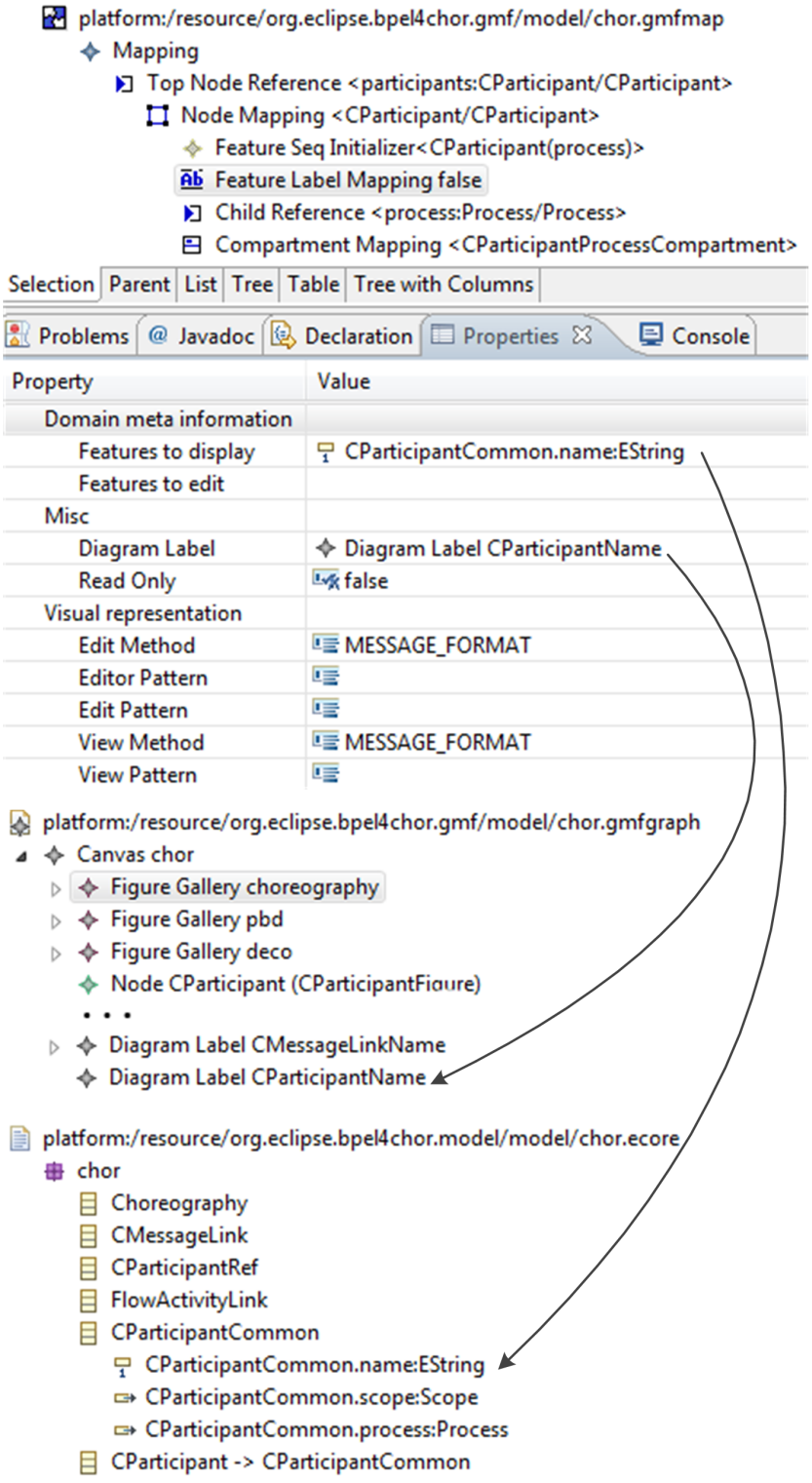

Abbildung 5.29: Definition des *Label Mapping* von `CParticipant` im Mapping Definition Model

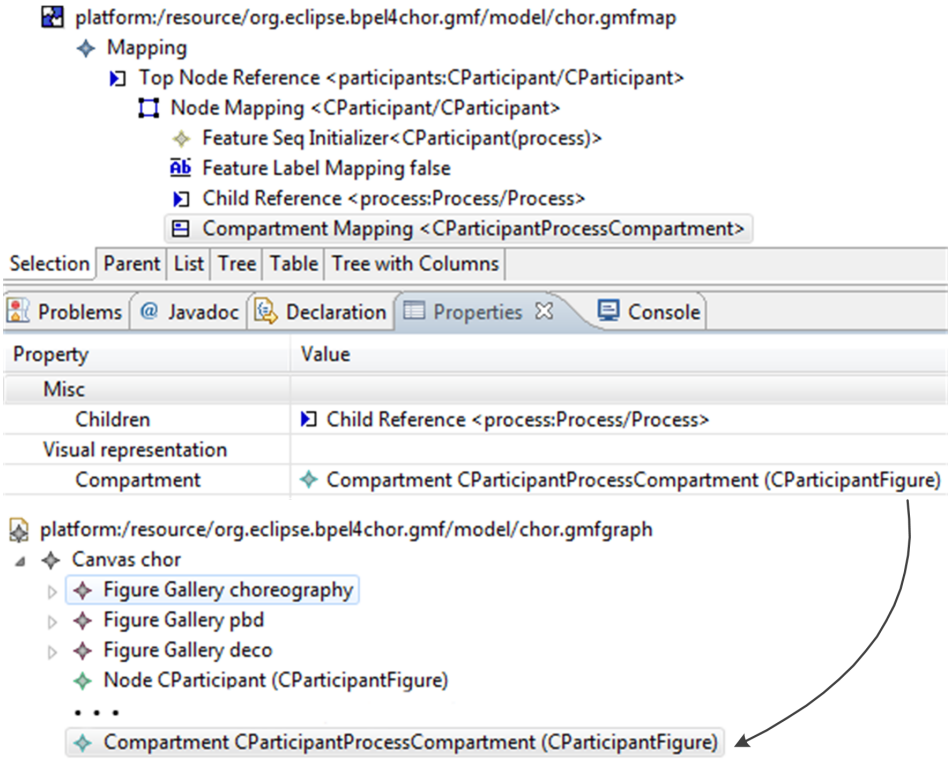

Abbildung 5.30: Definition des *Compartment Mapping* von `CParticipant` im Mapping Definition Model

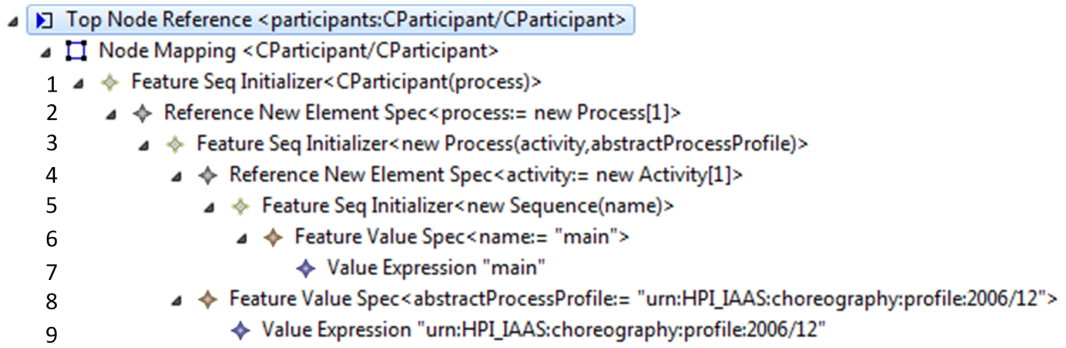

Abbildung 5.31: Erstellung zusätzlicher Instanzen beim Anlegen einer neuen `CParticipant` Instanz

`Process` Mapping Definition

Ein `Process` Element wird innerhalb des *Compartments* von `CParticipant` bzw. `CParticipantSet` platziert. Daher modellieren wir `Process` nicht als *Top Node Reference* sondern als *Child Reference* von `CParticipant`, wie in Abbildung 5.32 zu sehen ist. Mit dem *`Compartment`* Attribut bestimmen wir das *Compartment* von `CParticipant`, in welchem diese `Process` Referenz dargestellt werden soll. Die entsprechende Referenz von `CParticipantCommon` auf `Process`, legen wir im *`Containment Feature`* Attribut fest. Jetzt müssen wir das Aussehen und die zugehörige *Ecore* Klasse von `Process` im *Node Mapping* festlegen. Dies ist in Abbildung 5.33 zu sehen. Wir benötigen für die `Process` Instanzen allerdings kein *Creation Tool*, da der Benutzer nicht die Möglichkeit haben soll, welche zu erzeugen. Ebenfalls definieren wieder ein *Compartment Mapping*, in welchem die `Sequence` Aktivität platziert wird und legen eine *Child Reference* an, welche auf die von `Process` ausgehende `Activity` Referenz zeigt.

`Activity` Mapping Definition

In Abbildung 5.34 sehen wir das *Node Mapping* von `Sequence`. Wie zuvor auch legen wir Aussehen, *Ecore* Klasse und *Creation Tool* fest. Da eine `Sequence` beliebig viele Aktivitäten aufnehmen kann, müssen wir für jede Aktivität eine *Child Reference* und jeweils darunter das passende *Node Mapping* angeben. Wichtig ist hierbei, dass wir im ganzen Mapping Definition Model jede *Ecore* Klasse nur einmal in einem *Node Mapping* definieren, jedoch mit mehreren *Child Reference* Elementen vorhandene *Node Mapping* Elemente referenzieren können. So ist `Sequence` eine Aktivität und kann sich somit selber beliebig oft verschachtelt enthalten. Wir definieren folglich unterhalb des *Node Mapping* von `Sequence`, eine *Child Reference* auf sich selbst. Dies ist in Abbildung 5.35 zu sehen. Mittels *`Referenced Child`* Attribut verweisen wir, rekursiv, auf das bereits vorhandene *Node Mapping* Element von `Sequence` und haben damit eine beliebig tiefe Verschachtlung erreicht. Die anderen *Child Reference* Elemente von `Scope` bis `Flow` definieren wir wie gehabt, jeweils mit einem eigenen *Node Mapping* Element. Bei `Scope`, `Flow` und den übrigen verschachtelbaren Elementen, machen wir uns wieder das *`Referenced Child`* Attribut zu nutze. Haben wir auf diese Art `Process` komplett durch modelliert, müssen wir das ganze nicht nochmal neu für `CParticipantSet` definieren, sondern verweisen mit *`Referenced Child`* auf das bereits definierte *Node Mapping* Element von `Process`.

Wie bereits schon erwähnt, betrachten wir noch einmal ein *Feature Label Mapping*. Abbildung 5.36 zeigt das Mapping des Namen Labels von `Invoke`, welches für alle Aktivitäten analog definiert ist. Diesmal möchten wir im Label die zwei *Ecore* Attribute *`name`* und *`id`* anzeigen. Mittels *`View Pattern`* definieren wir, wie genau aus den Attributen, welche in *`Features to display`* eingetragen sind, der Label Text auf der Zeichenfläche zusammen gesetzt wird. {0} bezieht sich dabei auf das erste Attribut, in unserem Beispiel *`name`* und

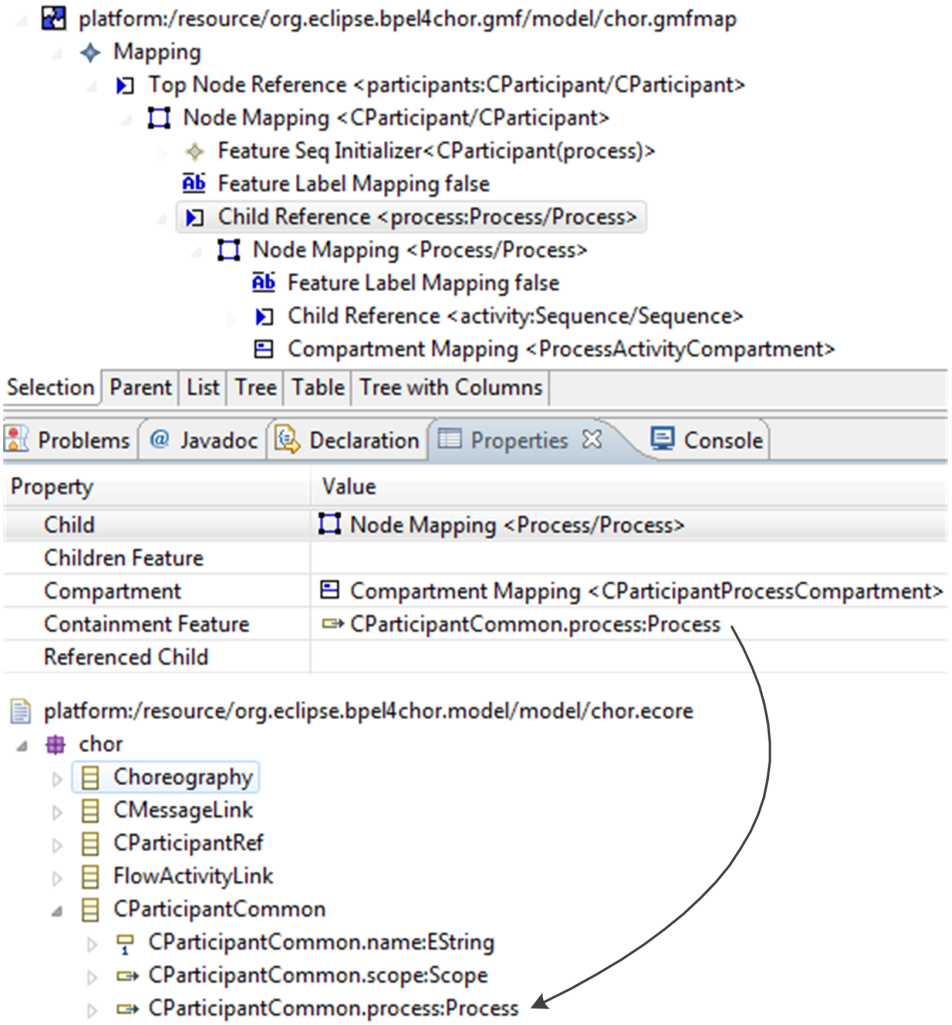

Abbildung 5.32: Definition von `Process` als *Child Reference* im Mapping Definition Model

{1} demnach auf das zweite, `id`. Diese beiden Attribute sollen durch ein ":" Zeichen getrennt sein. Der daraus resultierende Text sieht dann z. B. so aus: "xy : 1". Mittels `Edit Pattern` geben wir an, wie der vom Benutzer eingegebene Text analysiert werden soll und mittels `Editor Pattern` beschreiben wir, wie sich das Label vom Benutzer editieren lässt. In unserem Beispiel können wir `name` und `id` editieren, jedoch nicht ":" entfernen.

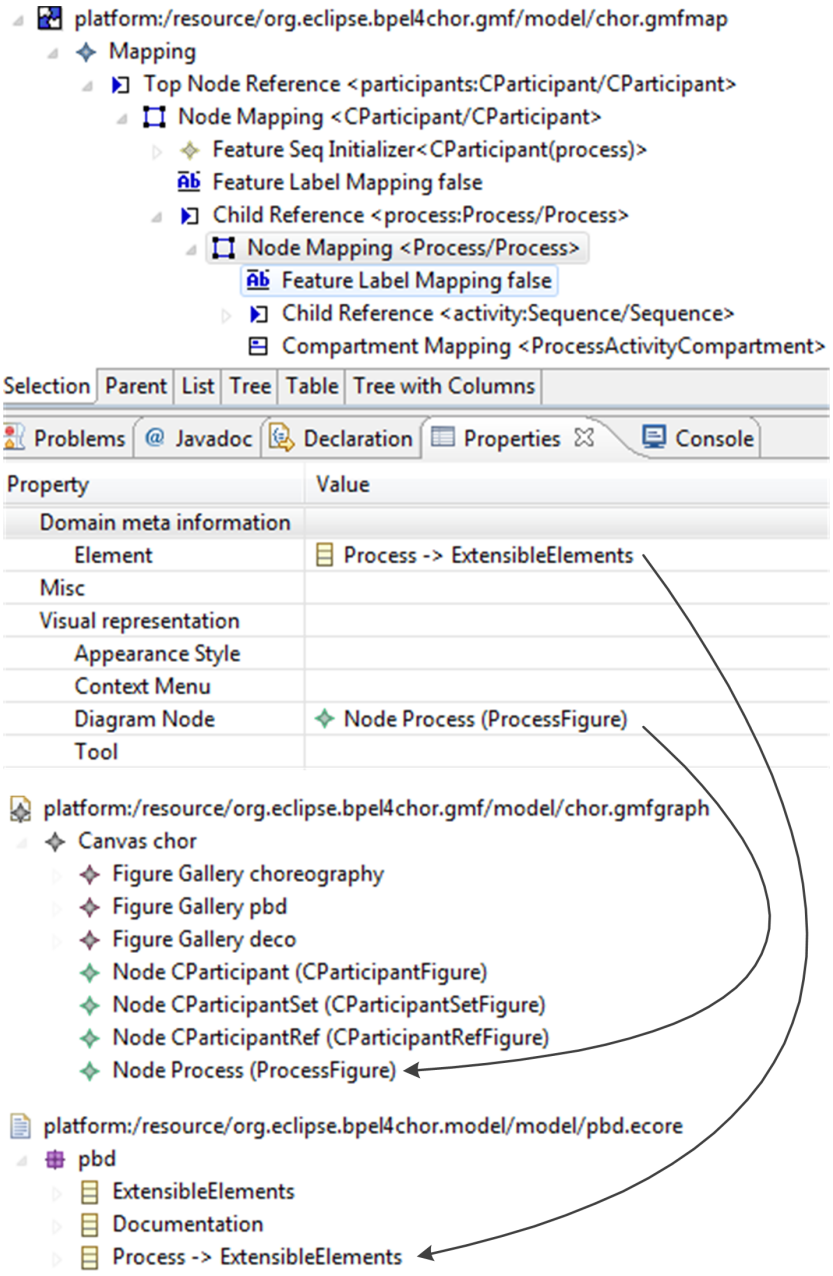

Abbildung 5.33: Definition des *Node Mapping* von `Process` im Mapping Definition Model

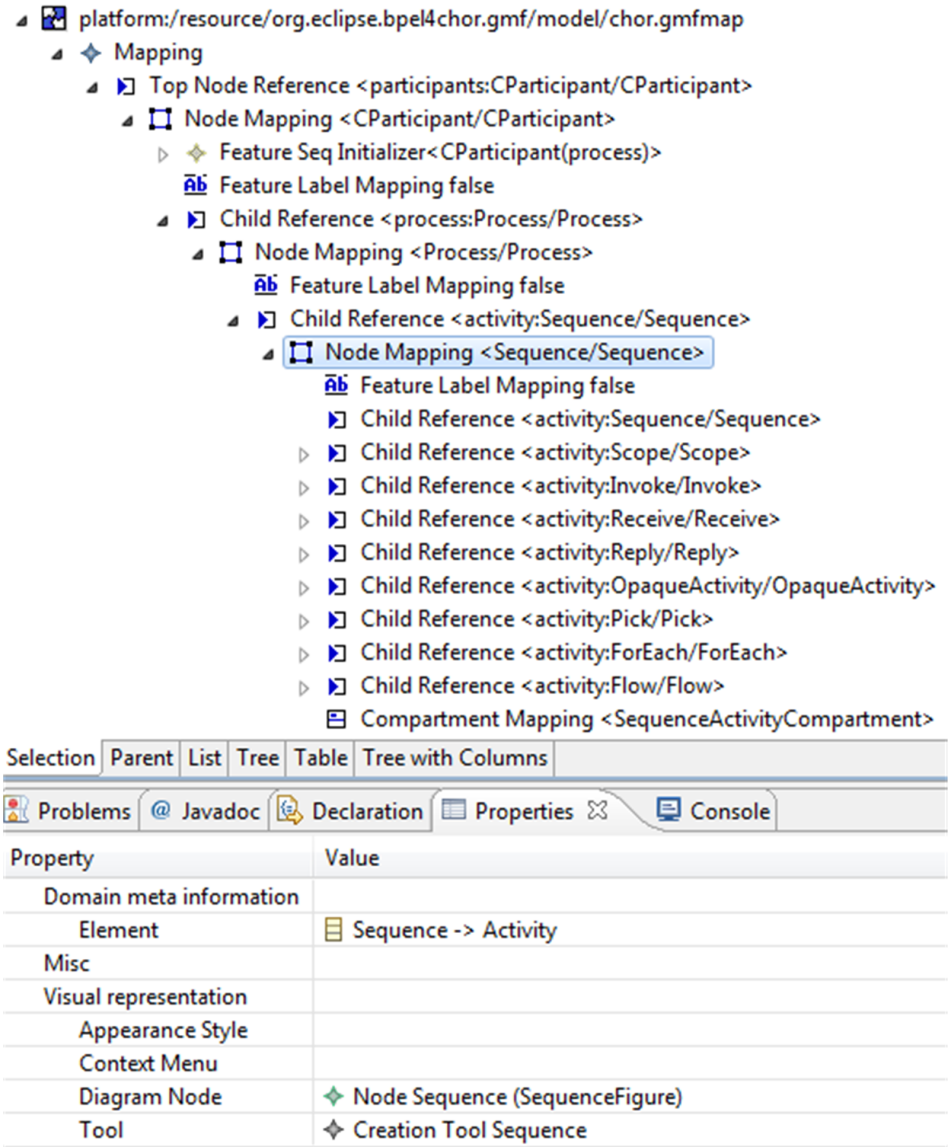

Abbildung 5.34: Definition des *Node Mapping* von `Sequence` im Mapping Definition Model

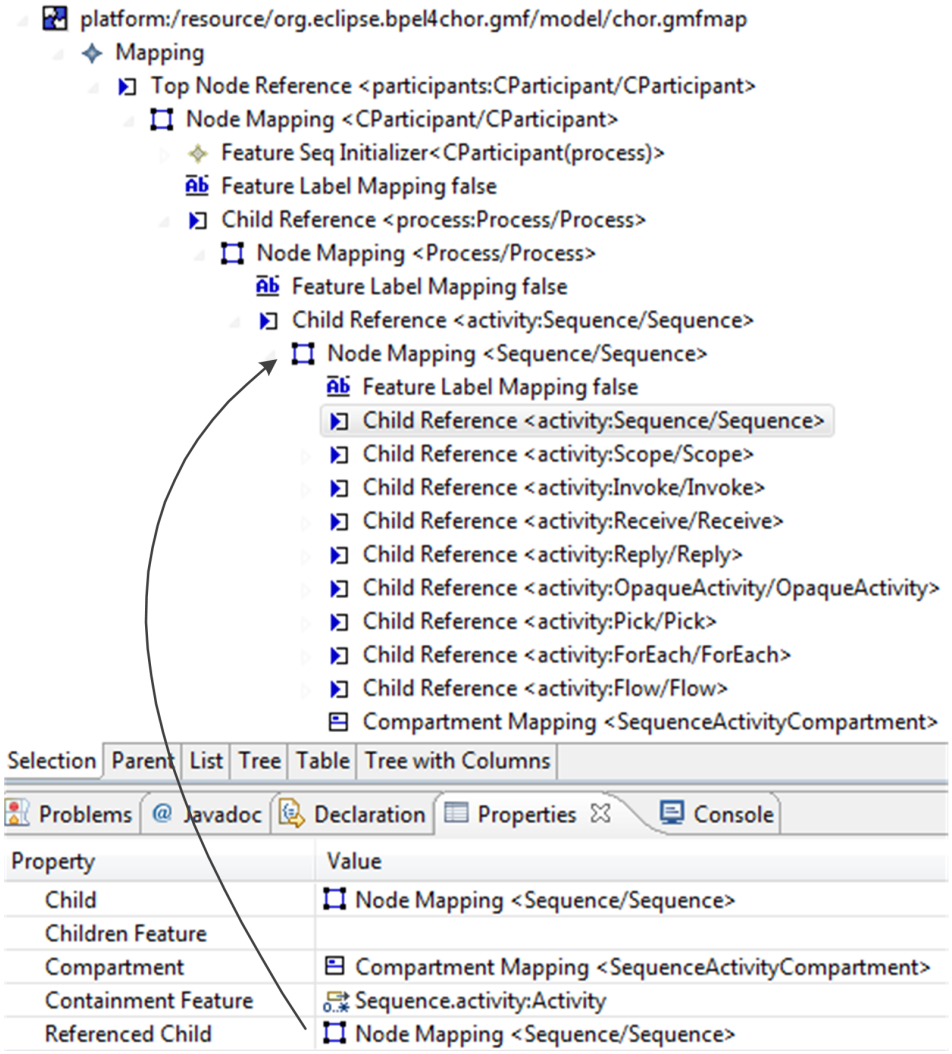

Abbildung 5.35: Definition der rekursiven *Child Reference* von `Sequence` im Mapping Definition Model

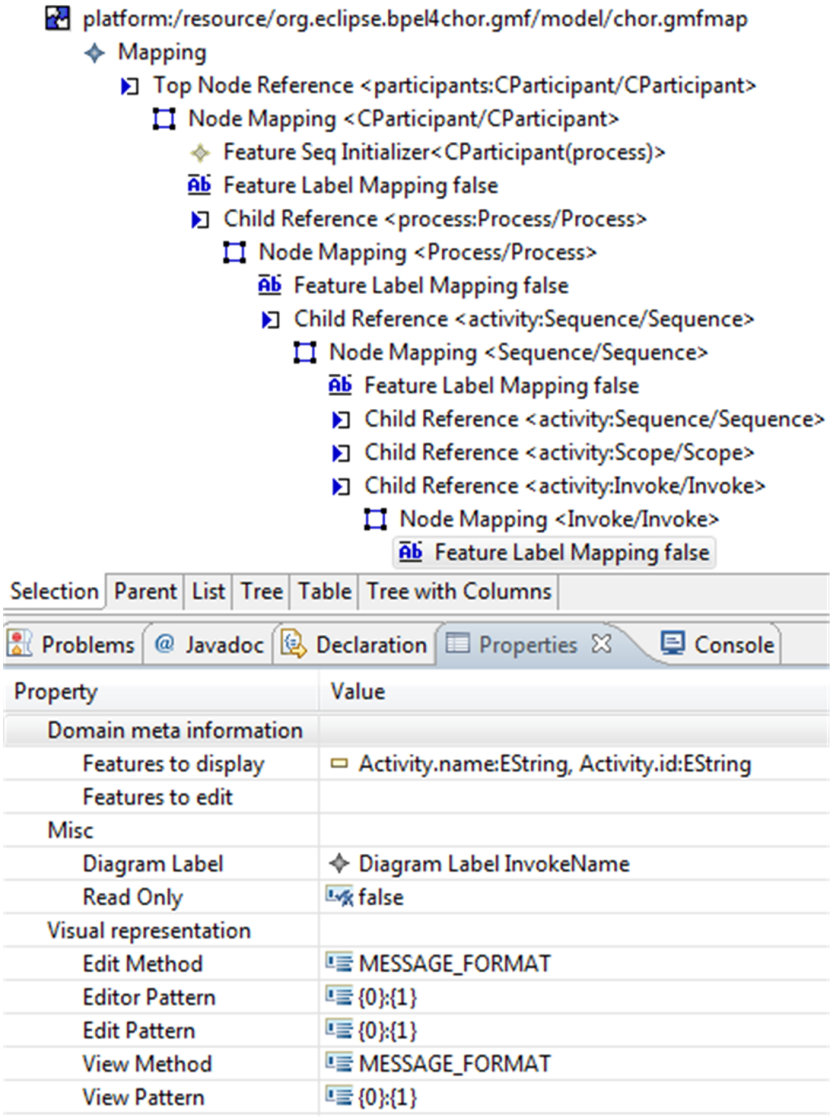

Abbildung 5.36: Definition des *Label Mapping* von `Sequence` im Mapping Definition Model

`CMessageLink` Mapping Definition

Zum modellieren von Verbindern benötigen wir *Link Mapping* Elemente. Diese können nur auf *Top Node Reference* Ebene definiert werden d. h., direkt auf der Zeichenfläche und nicht etwa in einem *Compartment*. In Abbildung 5.37 sehen wir das *Link Mapping* von `CMessageLink`. *`Containment Feature`* und *`Element`* werden wie gehabt definiert indem sie auf die entsprechenden Elemente aus dem *Ecore* Modell verweisen. *`Source Feature`* und *`Target Feature`* sind exklusive Attribute für *Link Mapping* Elemente in welchen sich festlegen lässt,

welche *Ecore* Referenzen Quelle und Ziel des Verbinders sein sollen. In unserem Fall sind das die `CMessageLink` Attribute *`sendActivity`* und *`receiveActivity`*, welche jeweils die Kommunikationsaktivitäten referenzieren. Über *`Diagram Link`* und *`Tool`* legen wir wieder Aussehen und das *Creation Tool* fest, mit welchen die Verbinder erstellt werden. Mit dieser Modellierung können wir allerdings alle Kommunikationsaktivitäten untereinander verbinden, was auch unzulässige bzw. unsinnige Verbindungen erlaubt wie z. B. `Receive` zu `Receive` oder `Receive` als Quelle und `Reply` als Ziel. Auch könnten wir zwei Kommunikationsaktivitäten im selben Prozess miteinander verbinden. Um dieses Problem zu lösen, gib es die Möglichkeit *Link Constraints* anzugeben. Wir definieren ein *Constraint* in Java Code, mit welchem wir prüfen ob ein Verbinder erstellt werden darf oder nicht. Der GMF Generator kopiert dann den hier angegeben Code in die entsprechende Prüfmethode, welche jedes mal ausgeführt wird, wenn der Benutzer einen Verbinder anlegt oder verändert. Die Code Logik ist so aufgebaut dass wir zuerst Prüfen, ob Quell- und Zielaktivität zu unterschiedlichen Prozessinstanzen gehören. Sollte dies der Fall sein, fahren wir fort und Prüfen, ob Quelle und Ziel in einer erlaubten Konstellation auftreten. Sind alle Prüfungen erfolgreich, kann der Benutzer den Verbinder erstellen.

5.2.5 GMF Generator Model

Haben wir unseren Editor im Mapping Definition Model spezifiziert, können wir das Generator Model daraus erzeugen. Diese Modell erlaubt uns einige Einstellung bezüglich der Editor Codegenerierung. Wir gehen hier nur auf ein paar Einstellungen ein welche wir, abweichend von der Standard Einstellung, definieren müssen. Die Platzierung von grafischen Elementen wird durch Layout Manager geregelt. Elemente innerhalb von *Compartments* können anhand von zwei verschiedenen Layout Managern verwaltet werden. Zum einen das `ListLayout`, welches eine feste Position der Elemente vorgibt und keine Verschiebung oder Skalierung erlaubt und, zum anderen, das `XYLayout`, welches freie Positionierung, Verschiebung und Skalierung ermöglicht [GG09]. Es gibt keine Möglichkeit, einen anderen Layout Manager über das Graphical Definition Model vorzugeben. Dies liegt an der Besonderheit, dass für *Compartments* und *Nodes* separate *EditParts* generiert werden. In Abbildung 5.38 stellen wir, am Beispiel vom `Process` Element, diesen Zusammenhang schematisch dar. Im Graphical Definition Model legen wir das Aussehen fest. Aktivitäten sollen im Rechteck `ProcessCompartmentFigure` dargestellt werden, welches Teil der grafischen Repräsentation von `Process` ist. Im Generator Modell werden aus *Node Mapping Process* und *Compartment Mapping* zwei verschiedene *EditParts* erstellt. `ProcessEditPart` generiert sein Aussehen anhand dem *Figure Descritor* Code. `ProcessActivtyCompartmentEditPart` erhält ein vom GMF vordefiniertes Aussehen durch `ResizableCompartmentFigure`. Würden wir im Graphical Definition Model am `ProcessCompartmentFigure` Rechteck z. B. einen `BorderLayout` Layout Manager festlegen, hätte dies keine Auswirkung auf die Platzierung von Aktivitäten im `ProcessActivityCompartment`.

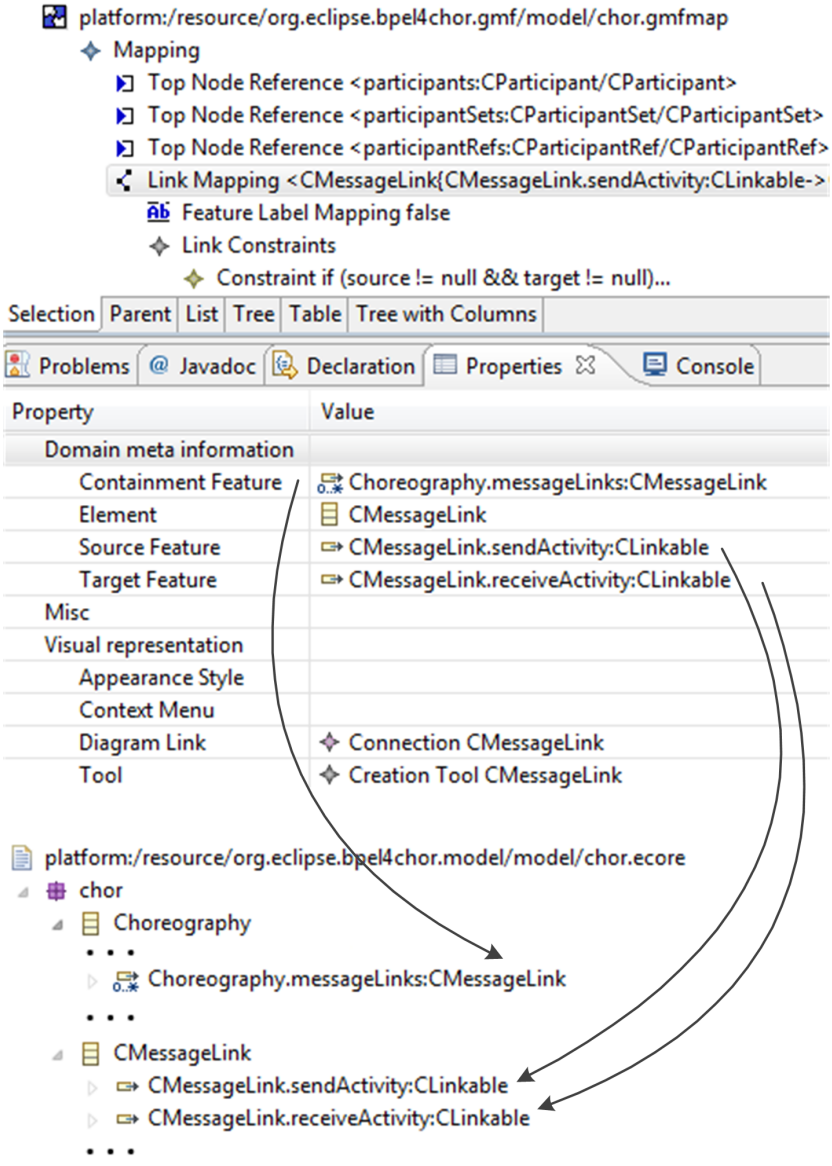

Abbildung 5.37: Definition des *Link Mapping* von `CMessageLink` im Mapping Definition Model

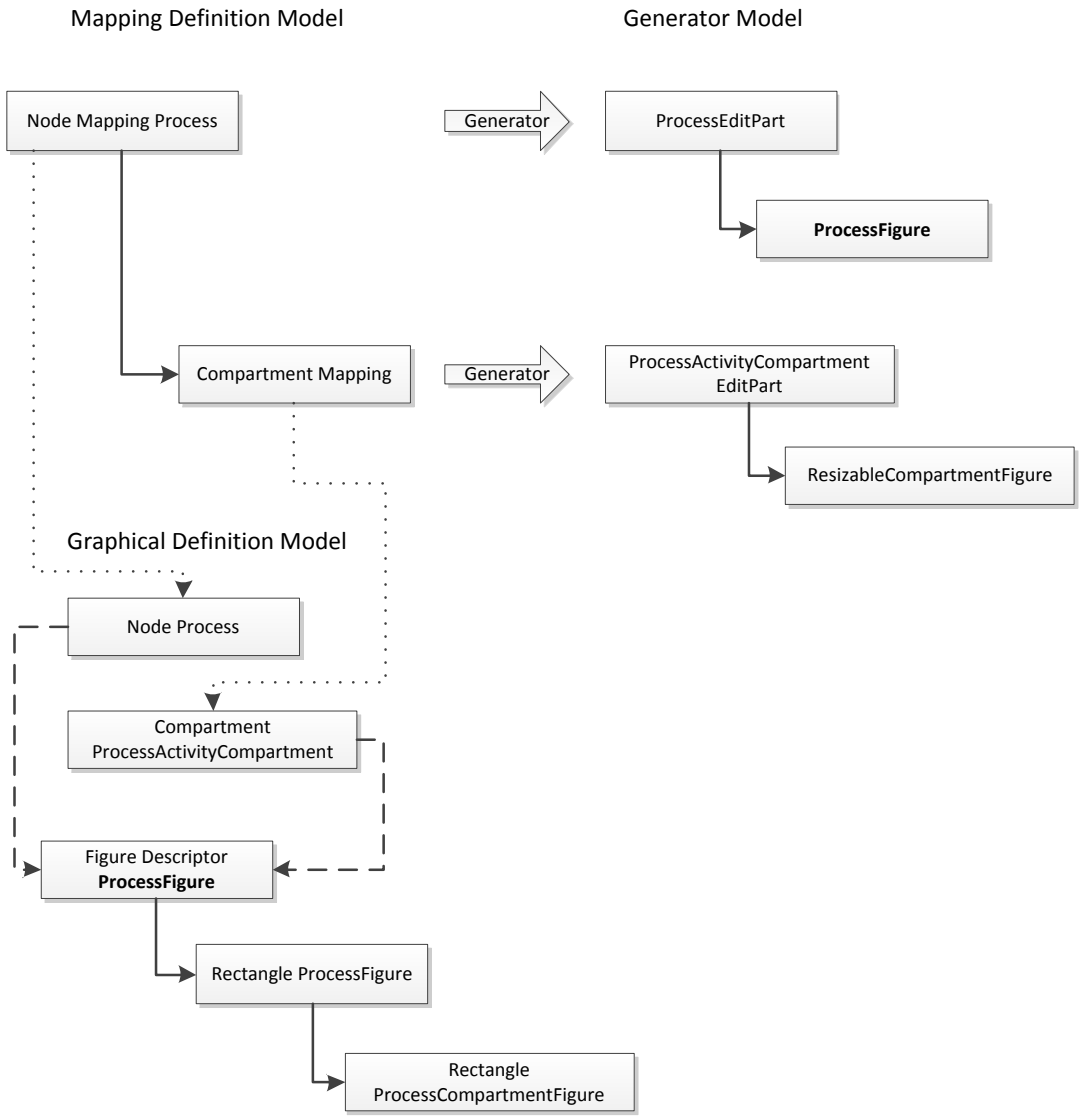

Abbildung 5.38: Generierung von separaten `EditParts` für *Compartments* und *Nodes*

Wir verwenden nur für Aktivitäten innerhalb von `Flow` das `XLLayout`, für die anderen grafischen Elemente, in welchen wir verschachtelte Darstellungen erlauben, benutzen wir das `ListLayout`. Dies ist in Abbildung 5.39 dargestellt. Wir setzen für alle `CompartmentEditPart` Klassen, das Property Attribut *`List Layout`* im Generator Model auf *true* außer für das `CompartmentEditPart` von `Flow`, da wir dort eine freie Positionierung der Aktivitäten erlauben wollen. Da jedoch `ListLayouts` standardmäßig die Elemente recht eng aneinander platzieren und wir dies nicht im Graphical Definition Model beeinflussen können, müssen wir in den generierten Code eingreifen. Dafür machen wir uns die, in Abschnitt 5.1.5 auf Seite 104 besprochenen, *Xpand* Templates zu nutze. Wir passen die Code Generierung für alle `CompartmentEditParts` an. Dazu kopieren wir die vorhandene *CompartmentEditPart.xpt* Template aus dem `org.eclipse.gmf.codegen` Plugin und passen nur das `createFigure()` «**DEFINE**» Statement mittels «**AROUND**» an. Mittels «**IF**» Statement können wir das `CompartmentEditPart` von `Flow` von der Anpassung ausschließen. In Abbildung 5.40 sehen wir die Auswirkungen der Anpassung. Wir haben Einschübe (`Insets`) an allen Seiten (`TOP`, `BOTTOM`, `LEFT`, `RIGHT`) des *Compartments* hinzugefügt und den Abstand zwischen den einzelnen Elementen (`Spacing`) vergrößert.

Eine weitere Verwendung einer *Xpand* Template haben wir für die `EditParts` von Verbindern. Verbinder können unterschiedlich zwischen Anfangs- und Endpunkt verlaufen. Wir haben bereits in Abschnitt 5.1.4 auf Seite 95 von Routern gesprochen. In Abbildung 5.41 ist zu sehen, dass wir unterschiedliche Routing Stiele für `CMessageLink` und `FlowActivityLink` verwenden. Der von GMF generierten Editoren globale Stiel für alle Verbinder ist "oblique", also eine gerade Linie zwischen Quelle und Ziel. Dies verwenden wir für `FlowActivityLinks`. Für `CMessageLink` möchten wir allerdings Linien mit neunzig Grad Wendepunkten benutzen, was dem Routing Stiel "rectilinear" entspricht. Dazu müssen wir den `EditPart` für `CMessageLink`, wie in [eclc] beschrieben, so anpassen dass bei jeder Erstellung eines Verbinders, der gewünschte Routing Stiel gesetzt wird. Wir kopieren dafür die *LinkEditPart.xpt* Template aus dem `org.eclipse.gmf.codegen` Plugin und passen nur das `createConnectionFigure()` «**DEFINE**» Statement mittels «**AROUND**» an.

Haben wir das Generator Model nach unseren wünschen eingestellt, können wir den eigentlich Java Code für den Editor generieren. GMF erstellt dazu ein neues Plugin Projekt und legt den Code darin in einer vordefinierten Struktur ab.

5.2.6 Diagram Extensions

Wir möchten den generierten Editor Code mit eigenen *Property Views* und Eclipse *Commands* erweitern. Dazu greifen wir nicht in den generierten Editor Code ein, oder reichern das Plugin mit neuen Klassen an, sondern erstellen ein weiteres Plugin Projekt, in welchem wir alle Erweiterungen definieren. Der Editor wird, in unserem Fall, im `org.eclipse.bpel4chor.diagram` Plugin abgelegt, die Erweiterungen erstellen wir im `org.eclipse.bpel4chor.diagram.extensions` Plugin.

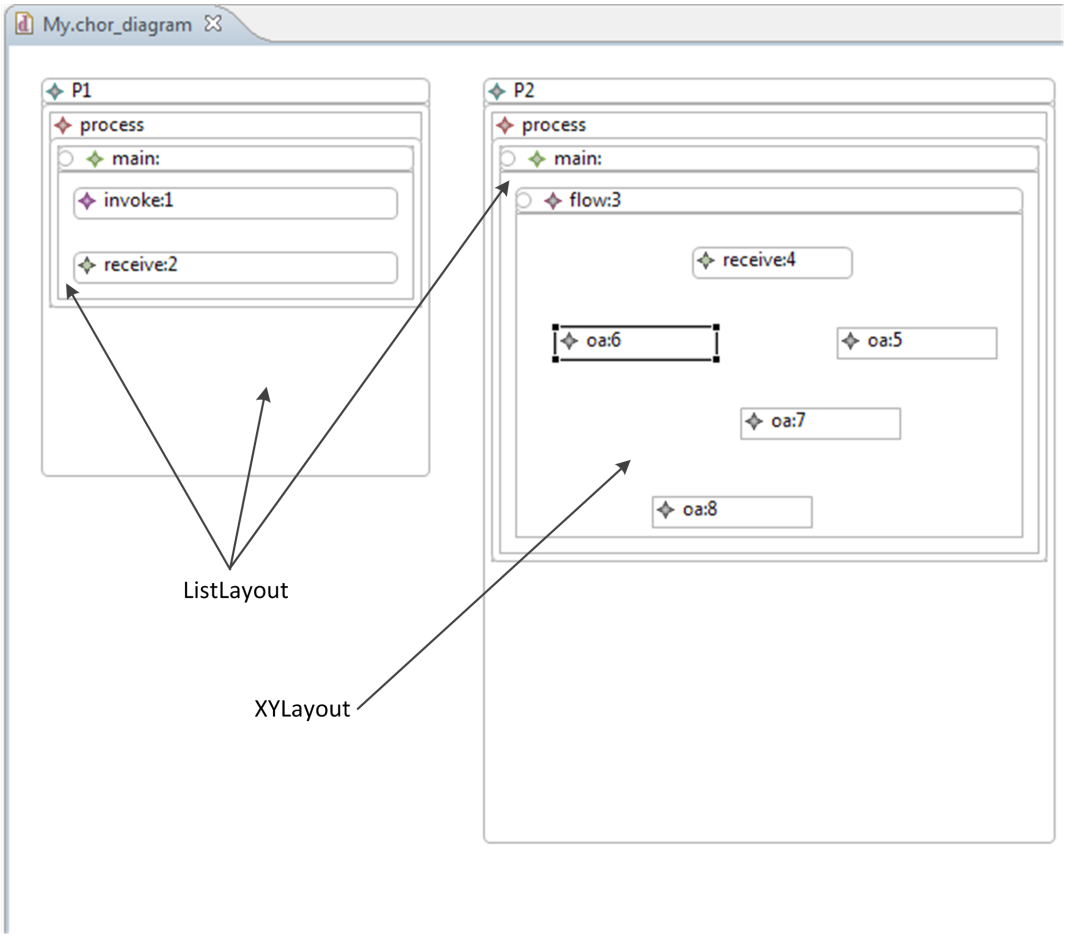

Abbildung 5.39: GMF Editor mit XYLayout und ListLayout

Property Views

Wir erstellen, auf Basis des grafischen Konzepts aus Abschnitt 4.5.4 auf Seite 66, unsere *Property View* Elemente. Dazu machen wir uns die Eclipse *Extension points* `propertyTabs`[14] und `propertySections`[15] nutze, welche wir im Manifest unseres Plugins einstellen. Die Funktionsweise dieser Plugins ist schematisch, anhand eines Beispiels, in Abbildung 5.42 dargestellt. Mittels `propertyTabs` definieren wir alle Tabs, welche unserm Editor zu Verfügung stehen sollen. In diesem Beispiel sind das **T1**, **T2** und **T3**. Der Inhalt eines Tabs, besteht aus einer oder wahlweise mehreren Sektionen, welche wir im `propertySections` *Extension point* definieren. In unserem Beispiel sind das **S1** - **S4**. Sektionen werden ihren

[14] org.eclipse.ui.views.properties.tabbed.propertyTabs
[15] org.eclipse.ui.views.properties.tabbed.propertySections

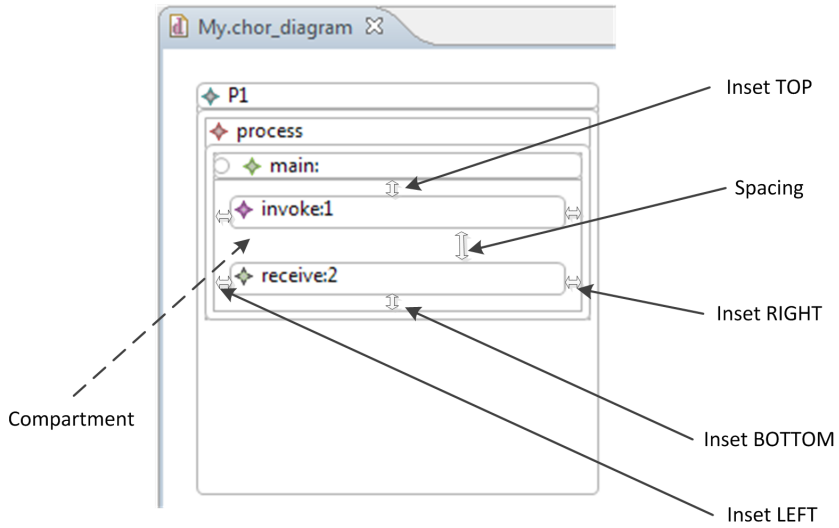

Abbildung 5.40: Abstände der Elemente in einem *Compartment* mittels `Inset` und `Spacing` angepasst

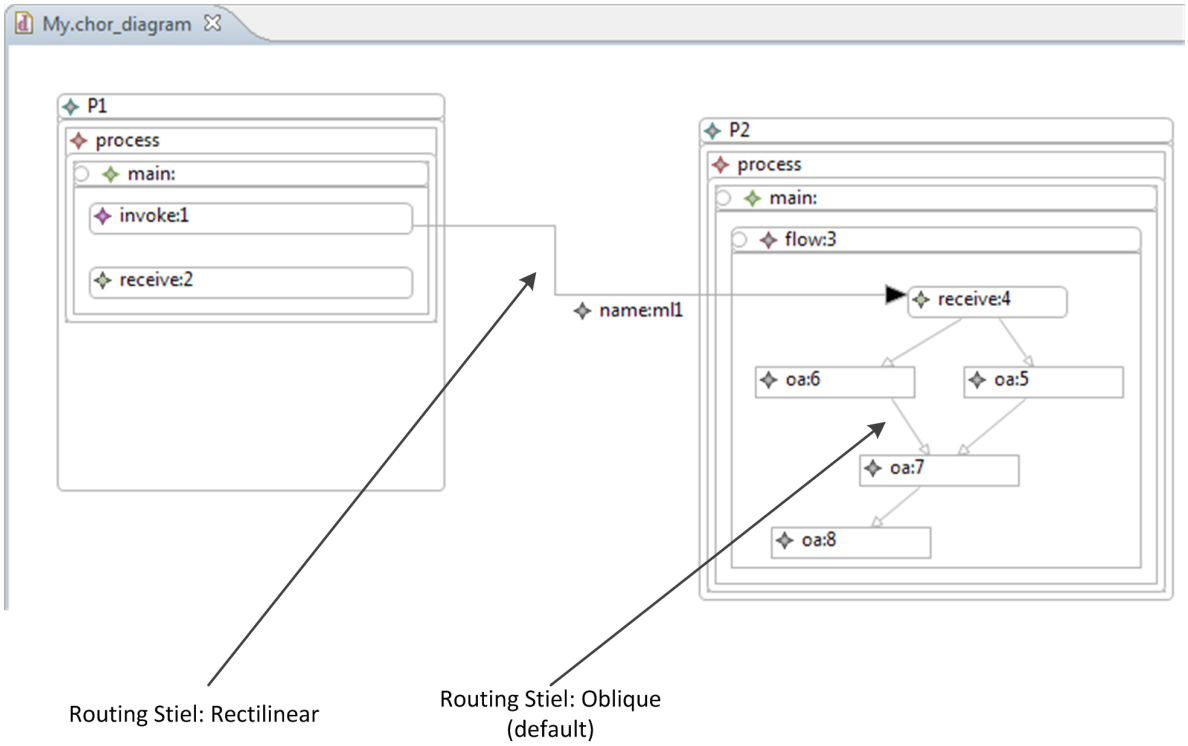

Abbildung 5.41: Verschiedene Routing Stiele für `CMessageLink` und `FlowActivityLink`

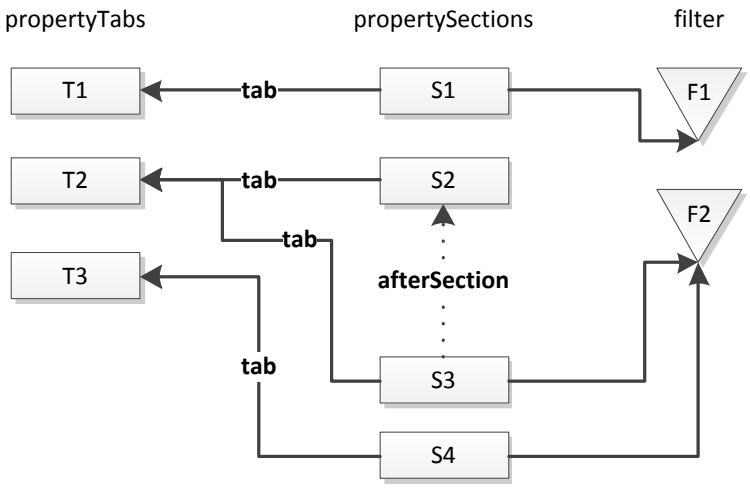

Abbildung 5.42: Konzept der *propertyTabs* und *propertySections Extension points*

Tabs, in welchen sie erscheinen sollen, zugewiesen. Im Beispiel erscheint **S1** in **T1**. **S2** und **S3** erscheinen in **T2**, wobei **S2** vor **S3** dargestellt wird. Diese visuelle Ordnung lässt sich mit dem `afterSection` Attribut definieren. Sektion **S4** erscheint schließlich in Tab **T3**. In grafischen Editoren selektieren wir Elemente auf der Zeichenfläche. Bei dieser Selektion wird bestimmt, welche Tabs und Sektionen zu diesem Element angezeigt werden. Dies können wir mit Filtern steuern, welches Java Klassen sind, die das `IFilter` Interface, bereitgestellt durch das *JFace* Framework, implementieren. Jede Sektion kann auf einen Filter verweisen. In unserem Beispiel wird Sektion **S1** und der zugehörige Tab **T1** nur dann angezeigt, wenn das selektiere Element den Filter **F1** passiert. Sektion **S2** und der zugehörige Tab **T2** werden immer bei jedem Element angezeigt, da kein Filter definiert wurde. Sektion **S3** erscheint nur auf Tab **T1**, wenn das selektierte Element Filter **F2** passiert. Sektion **S4** und Tab **T3** werden nur angezeigt, wenn das selektierte Element Filter **F2** passiert. Wir können also einen Filter, mehreren Sektionen zuweisen. Tabs erscheinen nur, wenn sie mindestens eine Sektion haben und, falls ein Filter dafür definiert wurde, das selektierte Element den Filter passiert.

In Abbildung 5.43 sehen wir die *Property View* eines selektierten `Receive` Elements. Der "Appereance" Tab kommt vom generierten Editor Code und ist ein Standardfeature von GMF. Die übrigen Tabs sind von uns definiert und entsprechen jenen, aus unserem grafischen Konzept in Abschnitt 4.5.4 auf Seite 66. Die Sektionen des gerade aktiven "Base" Tabs sind, zur Hervorhebung, eingerahmt. Die obere Sektion ist die `ActivityBaseSection`, welche für alle Aktivitäten erscheint. Die untere Sektion ist die `PickReceiveCreateInstanceSection`, welche nur für `Receive` und `Pick` erscheint. Alle Sektionen, die wir implementieren, erben

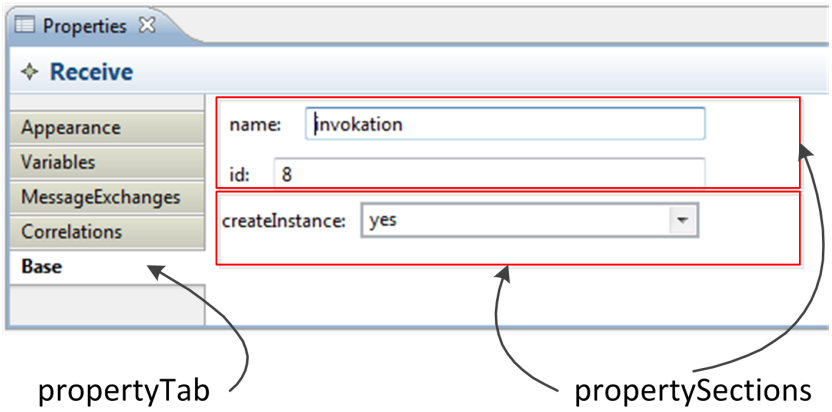

Abbildung 5.43: *Tabs* und *Sections* der *PropertyView* von `Invoke`

von der vorgegebenen Klasse `AbstractPropertySection`. Wie in [Hun09] beschrieben, müssen die Sektionen die drei Methoden *createControls*(), *setInput*() und *refresh*() implementieren. Diese Methode werden vom Eclipse Framework aufgerufen, wenn ein Element im Editor selektiert wurde. In *createControls*() erstellen wir die grafischen Komponenten, welche auf diese Sektion angezeigt werden. Durch *setInput*() erhalten wir das selektierte Element und *refresh*() wird vom Framework aufgerufen, falls der Inhalt dieser Sektion erneuert werden muss. Die ist z. B. der Fall, wenn der Benutzer am gerade selektierten Element etwas verändert.

Über die *Property View* führen wir Änderungen der aktuellen Instanz unseres Chor Model durch. GMF benutzt dazu im Hintergrund EMF.Edit und damit auch das *Command* basierte Editieren von EMF Objekten, wie wir bereits in Abschnitt 5.1.2 auf Seite 90 besprochen haben. Alle Elemente sind also Teil einer *Editing Domain* wobei GMF die transaktionsbasierte *Editing Domain* vom EMF verwendet. Wie in [IBM07] beschrieben, ist diese in der Klasse `TransactionalEditingDomain` realisiert und verwaltet mehrere, gleichzeitige Zugriffe auf ein Objekt. Wir verwenden daher die vom EMF Framework bereitgestellte Klasse `TransactionUtil` und dort, die statische Methode *getEditingDomain*() um für ein gegebenes Objekt, die passende *Editing Domain* zu finden. Haben wir die *Editing Domain*, können wir diverse *Command* Instanzen erstellen um damit Änderungen auf dem Chor Model durchführen. In Abbildung 5.44 sehen wir einen groben Überblick, wie der Editor mit *Property View*, die `propertyTabs` und *propertySections Extension points* sowie das zugrunde liegende *Ecore* Modell zusammen hängen. Wir veranschaulichen grob, anhand des `Receive` Elements, wie das `createInstance` Attribut über die *propertySection* vom Benutzer verändert werden kann. Im linken, oberen Bereich sehen wir dem GMF Editor. Es wurde ein `Process` Element mit einem `Receive` Element modelliert. Selektiert der Benutzer dieses, werden die definierten *Extension points* aktiv. Der `ReceivePickTypeFilter` stellt ein `Receive` Element fest und lässt es passieren. Darauf kann die `PickReceiveCreateInstanceSection` Sektion

im `base` Tab angezeigt werden. Der Benutzer verändert den Wert von `createInstance` über das *ComboBox* Widget. Darauf hin wird, für den Benutzer nicht sichtbar, die *Editing Doamin* von `Receive` bestimmt, ein neue *SetCommand* Instanz angelegt, der vom Benutzer veränderte Wert eingetragen und ausgeführt. Durch Ausführung wird das zugrunde liegende *Ecore* Modell am entsprechenden `createInstance` Attribut abgeändert.

Eclipse Commands

Wir haben im Abschnitt 4.5.2 auf Seite 48 besprochen, dass wir aus dem Chor Model einerseits BPEL4Chor Artefakte und andererseits BPEL Prozesse generieren möchten. Die dazu benötigen Komponenten rufen wir aus dem Editor mittels Eclipse *Commands* auf, welche nicht mit EMF/GMF *Commands* verwechselt werden sollten. Eclipse *Commands* werden dazu benutzt, dass der Benutzer Aktionen über die *Workbench* ausführen kann. Wie in [Vog12] beschrieben, benutzen wir für die *Commands* den `org.eclipse.ui.commands` *Extension point* sowie den `org.eclipse.ui.menus` *Extension point*, zur Platzierung der *Commands* in einem Menü der *Menu Bar* (als Referenz siehe Abbildung 5.2). Zuerst definieren wir die zwei *Commands* "Export" und "Transform". Diesen *Commands* weisen wir jeweils eine eindeutige **id** sowie *Command Hanlder* zu, welche die Logik enthalten, die beim Auslösen der *Commands* ausgeführt werden soll. *Command Handler* sind Java Klassen, welche das vom Eclipse Framework bereitgestellte `IHandler` Interface implementieren. Wir benutzen für unsere *Command Handler* die bereits definierte `AbstractHandler` Klasse, welche die wichtigsten Methoden von `IHandler` implementiert. Wir müssen lediglich die *execute*() Methode implementieren in welcher wir bestimmen, was beim Ausführen passiert. Haben wir beide *Commands* definiert, möchten wir diese dem Benutzer über die *Menu Bar* der *Workbench* zur Verfügung stellen. Dazu betrachten wir Abbildung 5.45. Im Manifest unseres generierten Editors finden wir die **id**, welche diesen Editor eindeutig auszeichnet. Dies ist notwendig, da in Eclipse mehrere Editoren gleichzeitig benutzt werden können. Die *Commands* und das neue Menü, welches wir "ChorDiagramEditor" nennen, definieren wir im Manifest unseres Diagram Extensions Plugins. Wir benutzen den `menus`[16] *Extension point*, definieren eine `menuContribution` und legen fest, dass das neue Menü in der *Menu Bar* – dem Hauptmenü der *Workbench* – angezeigt werden soll. Dies erreichen wir mit dem Wert "menu:org.eclipse.ui.main.menu". Jetzt können wir den Namen des Menüs festlegen, indem wir ein `menu` Knoten definieren und ihn mit "ChorDiagramEditor" belegen. Diesem `menu` Knoten fügen wir nun zwei *Commands* hinzu, indem wir weitere *Command* Knoten anlegen, sie entsprechend dem gewünschten Anzeigenamen benennen und auf die **id** unseres, bereits unter `org.eclipse.ui.commands`, definierten *Commands* verweisen. Führen wir nun Eclipse aus, ist das "ChorDiagramEditor" Menü immer sichtbar, selbst wenn unser Editor nicht geöffnet – oder sogar ein anderer Editor geladen ist. Dies müssen wir unterbinden, da

[16] org.eclipse.ui.menus

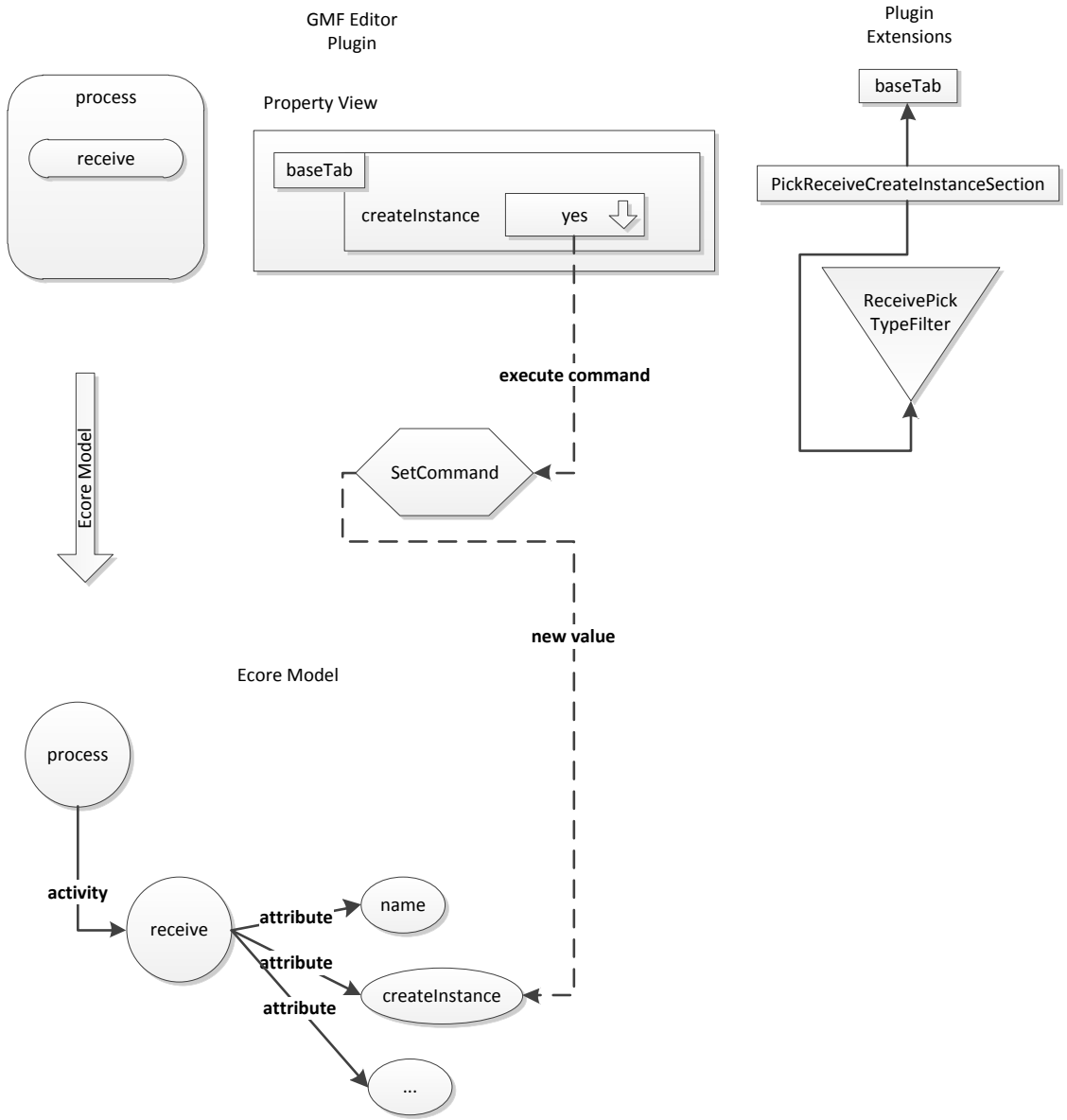

Abbildung 5.44: Das `createInstance` Attribut im *Ecore* Modell wird über die *propertySection* vom Benutzer verändert

dieses Menü nur zu unserem Editor gehört. Wir erreichen diese Einschränkung, indem wir die Funktionen des *Platform Expression Framework* Plugins nutzen, welche in [ecle] beschrieben sind. Wir schränken die Sichtbarkeit des "ChorDiagramEditor" Menüs ein, indem wir einen "visibleWhen" Knoten definieren. Dann benutzen wir die vom Framework bereitgestellte Variable "activeEditorId, welche die **id** des gerade Aktiven Editors der *Workbench* beinhaltet, und prüfen mit dem "equals"Knoten auf Übereinstimmung mit der **id** unseres Editors.

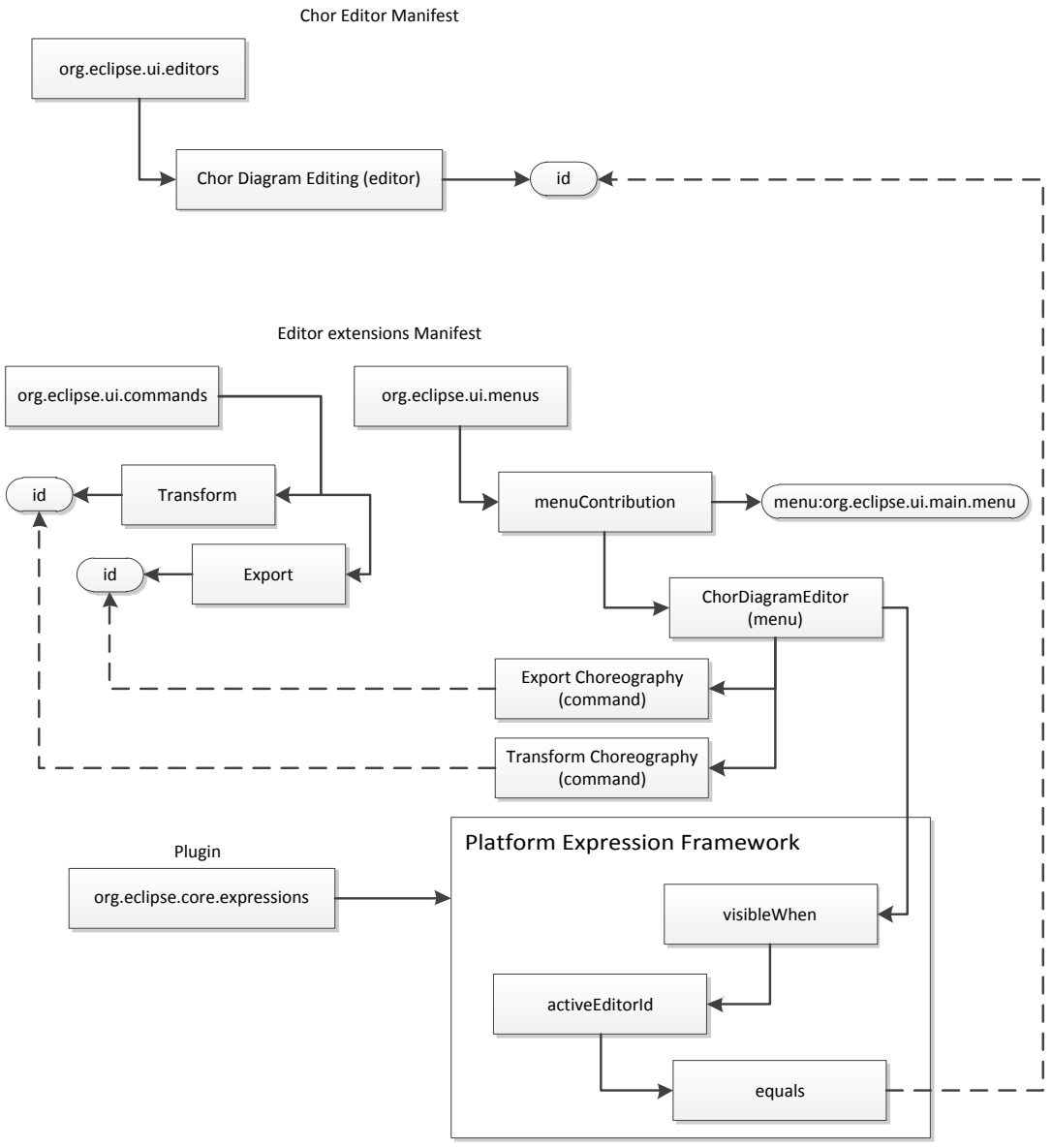

Abbildung 5.45: Einrichten eines neuen Menüs für die *Menu Bar* der *Workbench*

6 Zusammenfassung und Ausblick

Wir haben in Kapitel 4 ein Konzept für einen grafischen Editor zum Modellieren von Choreographien erarbeitet. Dabei haben wir mehrere Ansätze in Betracht gezogen und durch Vor- und Nachteile abgewägt, welchen Weg wir verfolgen wollen. Wir haben uns, beim Design des Editors, an dem bereits existierenden Prozesseditor *BPEL Designer* orientiert. Auch mussten wir Einschränkungen Treffen wie z. B. das Weglassen der von BPEL4Chor verwendeten Teilnehmertypisierung. Wir haben ein Modell für den Editor erarbeitet und Algorithmen zur Transformation in BPEL4Chor Artefakte angegeben. Über die bereits vorhandene Komponente *BPE4ChorToBPEL*, konnten wir aus den BPEL4Chor Artefakten abstrakte BPEL Prozesse generieren. Durch weitere kleine Anpassungen konnten wir einen kleinen Schritt in Richtung ausführbare Prozesse gehen. Doch mussten wir feststellen, dass die automatische Durchführung einer "executable completion" eine sehr komplexe Aufgabe ist.

Für die Implementierung des *Chor Designers* in Kapitel 5, haben wir einige Technologien des Eclipse Projekts benutzt. EMF zum erstellen der Modelle, GMF zum realisieren des Editors. Wir haben, durch die Benutzung von GMF, einen *Model Driven Architecture* Ansatz verwendet und damit Programmcode generiert. Den generierten Code haben wir durch von Hand erstellten Code ergänzt, um so unser Konzept exakt umsetzen zu können. Wir mussten den Funktionsumfang des Editors einschränken, um den zeitlich begrenzen Rahmen für diese Arbeit einzuhalten. Dies resultierte im Weglassen der Modellierungmöglichkeiten einiger BPEL Konstrukte, sowie vielen programmatischen Hilfestellungen, welche der *BPEL Designer* im Gegensatz leistet. Weitere Abstriche wurden in der Useability des Editors, sowie der grafischen Repräsentation der Elemente gemacht. Hier besteht zwar noch Verbesserungspotential, doch kann man abschließend sagen, dass die Ziele dieser Arbeit erreicht wurden. Wir geben im Folgenden einen kleinen Ausblick über möglichen Anpassungen und Erweiterungen des *Chor Designers*, die uns wichtig erscheinen.

Ausblick

Eine weiterführende Arbeit könnte sich mit der grafischen Darstellung befassen welche generell überarbeitet werden sollte. Es sollten z. B. die Repräsentation der strukturierten Aktivitäten wie ‹ `Sequence` › oder ‹ `Flow` › verbessert werden. Bei der ‹ `Flow` › Aktivität verhält sich das Layout im *Compartment*, bei der Platzierung von Aktivitäten, nicht ganz wie

gewünscht. Das *Compartment* sollte sich so vergrößern, dass die Platzierung der Aktivitäten leicht fällt. Zurzeit vergrößert sich das *Compartment* nur minimal, so dass bei der Platzierung einer Aktivität zu weit am Rand, eine Scrollbar eingeblendet wird. Dies kann den Benutzer verwirren. Aktivitäten in einer ‹ `Sequence` › sollten in Flussrichtung mit Pfeilen verbunden werden.

Zum Abschluss dieser Arbeit können die Basis Aktivitäten ‹ `Invoke` ›, ‹ `Receive` ›, ‹ `Reply` › und ‹ `OpaqueActivity` › modelliert werden. Von den strukturierten ist die Modellierung von ‹ `Sequence` ›, ‹ `Scope` ›, ‹ `Pick` › und ‹ `Flow` › möglich. Für ‹ `Process` › können ‹ `CorrelationSets` › und ‹ `MessageExchanges` › angegeben werden. Variablen von ‹ `Invoke` ›, ‹ `Receive` ›, ‹ `Reply` › und ‹ `OnMessage` › können nur Opaque gesetzt werden. Eine weiterführende Arbeit sollte alle restlichen BPEL Konstrukte für abstrakte Prozesse ergänzen. Dazu gehören unter anderem ‹ `Assign` ›, ‹ `While` ›, ‹ `If` › usw.. Im *Participant Behavior Description* EMF Modell sind bereits alle erlaubten Konstrukte modelliert.

Es ist zwar möglich BPEL4Chor Artefakte zu exportieren doch wäre auch vorstellbar, bereits vorhandene zu Importieren. Eine weiterführende Arbeit könnte einen Komponente entwerfen, welche BPEL4Chor Artefakte einliest und Instanzen der zugehörigem EMF Modelle erzeugt. Aus diesen EMF Modellen kann dann eine Instanz des Chor Model zusammengebaut werden. Um dieses Chor Model im Editor anzeigen zu können, muss allerdings erst erforscht werden, wie sich daraus die *GMF Runtime Notation* [eclb] generieren lässt. Diese Beinhaltet unter anderem die grafischen Elemente mit ihren Positionsdaten. Man könnte noch einen Schritt weiter gehen indem wir BPEL Prozesse als Ausgangsbasis nehmen. Mit der bereits vorhandenen Komponente *BPELToBPEL4Chor*, spezifiziert in [Ste07], können wir daraus BPEL4Chor Artefakte generieren. So könnte man einen Import von mehreren BPEL Prozessen in den Choreographie Editor realisieren.

Wir hatten in Kapitel 4 besprochen, dass wir die BPEL4Chor Teilnehmertypisierung weg lassen. Eine weiterführende Arbeit könnte jedoch ein Konzept dafür entwickeln und die Typisierung einbauen. Es wären z. B. Prozessschablonen, abrufbar aus einem zentralen Repository, denkbar. Wenn wir einen Blick auf den SimTech *BPEL Designer* und seine angebundenen Komponenten werfen, kommt eventuell die *Fragmento*[1] Komponente als Repository in Frage.

Wir können mit unserem Editor Choreographien modellieren, doch können wir diese nicht direkt aus dem Editor heraus Ausführen. Diese Einschränkung resultiert aus dem notwendigen Zwischenschritt, dass zuerst ausführbare BPEL Prozesse erzeugt werden müssen, dies aber nicht ohne weiteres bewerkstelligt werden kann. Eine weiterführende Arbeit könnte ein Konzept entwickeln, wie sich die "executable completion" von BPEL Prozessen, welche konform zum *Abstract Process Profile for Observable Behavior* Profil sind, automatisieren lässt.

[1] http://www.iaas.uni-stuttgart.de/forschung/projects/fragmento/source.htm

Literaturverzeichnis

[DB08] G. Decker, A. Barros. Interaction modeling using BPMN. In *Business Process Management Workshops*, S. 208–219. Springer, 2008. (Zitiert auf den Seiten 5 und 15)

[DK12a] G. Decker, O. Kopp. Grounding XSD Schema, 2012. URL https://github.com/IAAS/BPEL4Chor-model/blob/master/doc/BPEL4Chor%20schema/grounding.xsd. (Zitiert auf den Seiten 5, 23, 24 und 108)

[DK12b] G. Decker, O. Kopp. Topology XSD Schema, 2012. URL https://github.com/IAAS/BPEL4Chor-model/blob/master/doc/BPEL4Chor%20schema/topology.xsd. (Zitiert auf den Seiten 5, 9, 18, 19, 108 und 110)

[DKL+08] G. Decker, O. Kopp, F. Leymann, K. Pfitzner, M. Weske. Modeling service choreographies using BPMN and BPEL4Chor. In *Advanced Information Systems Engineering*, S. 79–93. Springer, 2008. (Zitiert auf Seite 13)

[DKLW07] G. Decker, O. Kopp, F. Leymann, M. Weske. BPEL4Chor: Extending BPEL for modeling choreographies. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, S. 296–303. IEEE, 2007. (Zitiert auf den Seiten 5, 16, 24, 34, 36 und 42)

[DKLW09] G. Decker, O. Kopp, F. Leymann, M. Weske. Interacting services: From specification to execution. *Data & Knowledge Engineering*, 68(10):946–972, 2009. (Zitiert auf den Seiten 5, 9, 16, 17, 18, 21, 22 und 27)

[ecla] eclipse.org. GMF Labels. Wiki. URL http://wiki.eclipse.org/GMF_Labels. (Zitiert auf Seite 123)

[eclb] eclipse.org. GMF Runtime API Specification. URL http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.gmf.doc/reference/api/runtime/org/eclipse/gmf/runtime/notation/package-summary.html. (Zitiert auf Seite 147)

[eclc] eclipse.org. Graphical Modeling Framework Tips. URL http://wiki.eclipse.org/GMF/Tips. (Zitiert auf Seite 137)

[ecld] eclipse.org. Graphical Modeling Framework Tutorial. URL http://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial/Part_1. (Zitiert auf den Seiten 6 und 99)

[ecle] eclipse.org. Platform Expression Framework. URL http://wiki.eclipse.org/Platform_Expression_Framework. (Zitiert auf Seite 144)

[ecl10] eclipse.org. Platform Plug-in Developer Guide, 2010. URL http://help.eclipse.org/helios/index.jsp. (Zitiert auf den Seiten 6, 83, 85 und 87)

[FUMK06] H. Foster, S. Uchitel, J. Magee, J. Kramer. LTSA-WS: a tool for model-based verification of web service compositions and choreography. In *Proceedings of the 28th international conference on Software engineering*, S. 771–774. 2006. (Zitiert auf Seite 13)

[GG09] R. C. Gronback, R. C. Gronback. *Eclipse modeling project: a domain-specific language toolkit*. Addison-Wesley, Upper Saddle River, NJ, 1. print. Auflage, 2009. (Zitiert auf den Seiten 6, 92, 94, 95, 104 und 134)

[Hun09] A. Hunter. The Eclipse Tabbed Properties View, 2009. URL http://www.eclipse.org/articles/Article-Tabbed-Properties/tabbed_properties_view.html. (Zitiert auf Seite 141)

[IBM07] IBM. Eclipse EMF Model Transaction Development Guide, 2007. URL http://www.linuxtopia.org/online_books/eclipse_documentation/eclipse_emf_model_transaction_developer_guide/topic/org.eclipse.emf.transaction.doc/references/overview/eclipse_emf_model_transaction_domains.html. (Zitiert auf Seite 141)

[KL08] O. Kopp, F. Leymann. Choreography design using WS-BPEL. *Data Engineering*, 31(2):31–34, 2008. (Zitiert auf den Seiten 11 und 14)

[Kop12] O. Kopp. BPEL4Chor - ohne Cross-Partner-Scopes, 2012. (Zitiert auf den Seiten 9, 22, 23 und 24)

[Ley10] F. Leymann. Architectural Diagrams and Styles, 2010. Vorlesungsunterlagen von Grundlagen der Architektur von Anwendungssystemen. (Zitiert auf Seite 14)

[Ley11] F. Leymann. BPEL: Web Service Business Process Execution Language, 2011. Vorlesungsunterlagen von Workflow Management 1. (Zitiert auf Seite 15)

[Li10] C. Li. *An Editing Environment for BPEL4Chor Cross-Partner Scopes*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2010. (Zitiert auf den Seiten 65 und 106)

[LL07] J. Ludewig, H. Lichter. *Software-Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt-Verl., Heidelberg, 1. Auflage, 2007. (Zitiert auf den Seiten 5, 27 und 28)

[OAS07a] OASIS. Abstract BPEL Common Base, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/process/abstract/ws-bpel_abstract_common_base.xsd. (Zitiert auf den Seiten 9, 16, 17, 108, 111 und 112)

[OAS07b] OASIS. Schema for Service Reference, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/serviceref/ws-bpel_serviceref.xsd. (Zitiert auf den Seiten 24 und 26)

[OAS07c] OASIS. Web Services Business Process Execution Language Version 2.0, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf. (Zitiert auf den Seiten 15, 16, 17, 18, 29 und 43)

[Rei07] P. Reimann. Generating BPEL Processes from a BPEL4Chor Description. Studienarbeit 2100, Institute of Architecture of Application Systems, 2007. (Zitiert auf den Seiten 6, 9, 27, 105 und 107)

[Sha04] S. Shavor. *Eclipse: Anwendungen und Plug-Ins mit Java entwickeln*. Addison-Wesley, München, 2004. (Zitiert auf Seite 85)

[Ste07] T. Steinmetz. Generierung einer BPEL4Chor-Beschreibung aus BPEL-Prozessen. Studienarbeit 2101, Institut für Architektur von Anwendungssystemen, 2007. (Zitiert auf den Seiten 35 und 147)

[Ste11] D. Steinberg. *EMF: Eclipse modeling framework*. Addison-Wesley, Upper Saddle River, NJ, 2. ed., rev. and updated, 2. printing Auflage, 2011. (Zitiert auf den Seiten 6, 83, 86, 88, 89 und 91)

[TDG06] I. J. Taylor, E. Deelman, D. B. Gannon. *Workflows for e-Science: scientific workflows for grids*. Springer, 2006. (Zitiert auf Seite 11)

[Vog12] L. Vogel. Eclipse Commands Tutorial, 2012. URL http://www.vogella.com/articles/EclipseCommands/article.html. (Zitiert auf Seite 142)

[Vuk09] K. Vukojevic. Architektur eines Workflow-Frameworks zur graphischen Erstellung und Ausführung von Simulationsexperimenten. Studienarbeit 2217, Institut für Architektur von Anwendungssystemen, 2009. (Zitiert auf Seite 31)

[WLHa00] L. Wood, A. Le Hors, et al. Document Object Model (DOM) Level 1 Specification (Second Edition). Technischer Bericht, W3C, 2000. URL http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/. (Zitiert auf Seite 48)

Alle URLs wurden zuletzt am 03. 05. 2013 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschift