Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 9

Konzept und Implementierung für Choreographiecontainer

Norman Wolter

Studiengang:	Softwaretechnik
Prüfer/in:	Jun.-Prof. Dr.-Ing Dimka Karastoyanova
Betreuer/in:	Dipl.-Inf. Michael Hahn, M.Sc. Wirt.-Inf. Andreas Weiß
Beginn am:	15. Juli 2014
Beendet am:	14. Januar 2015
CR-Nummer:	D.1.7, D.2.6, H.4.1

Kurzfassung

Diese Arbeit beschäftigt sich mit dem Konzept von Choreographiecontainern. Choreographiecontainer dienen dazu, einen Datenfluss außerhalb des regulären Daten– und Kontrollflusses einer Choreographie zu definieren. Außerdem erleichtern Choreographiecontainer das Modellieren und Pflegen der einzelnen Prozessmodelle einer Choreographie, da konkrete Daten zur Laufzeit und nicht bereits während der Modellierung der Prozessmodelle, bzw. der Choreographie, hinzugefügt werden können. Des Weiteren werden Choreographien um externe Nutzer erweitert. Diese können auf die Daten innerhalb eines Choreographiecontainers zugreifen und ermöglichen es, Daten die in der Choreographie erzeugt werden für definierte Teilnehmer, die kein Teil der eigentlichen Choreographie sind, zugänglich zu machen.

Außerdem beschreibt diese Arbeit, einen bereits an der Universität Stuttgart erstellten, Choreographieeditor der um die Möglichkeit der Darstellung einer Choreographie mit Choreographiecontainer erweitert wurde.

Inhaltsverzeichnis

1. Einleitung **9**

2. Grundlagen **11**
2.1. Grundbegriffe . . . 11
2.2. Extensible Markup Language . . . 12
2.3. XML Schema Definition . . . 13
2.4. Modellierung von Choreographien . . . 13
2.5. Business Process Execution Language . . . 15
2.6. BPEL4Chor . . . 16
2.7. Eclipse . . . 16

3. Verwandte Arbeiten **19**

4. Konzept **21**
4.1. Beispiel . . . 21
4.2. Definitionen . . . 22
4.3. Architektur einer Choreographie mit Choreographiecontainer . . . 23
4.4. Container Descriptor und External User Descriptor . . . 29
4.5. Graphische Darstellung einer Choreographie mit Choreographiecontainer . . . 38
4.6. Anwendungsfälle . . . 43
4.7. Entwurfsentscheidungen . . . 47
4.8. Softwarearchitektur eines Choreographiecontainers . . . 52
4.9. Choreographie mit Choreographiecontainer Editor . . . 52

5. Realisierung **59**
5.1. Übersicht des vorhandenen Editors . . . 59
5.2. EMF Modelle . . . 59
5.3. Tooling Definition Model . . . 61
5.4. Graphical Definition Model . . . 62
5.5. Mapping Definition Model . . . 63
5.6. Ergebnis . . . 68
5.7. Geplante Umsetzung für das Einfügen von Variablen . . . 68

6. Zusammenfassung und Ausblick **71**

A. Anhang **73**

Literaturverzeichnis **77**

Abbildungsverzeichnis

2.1. Darstellung einer Choreographie in zwei verschiedenen Versionen einer Choreographieumgebung . . . 12
2.2. Top–down (links) und bottom–up (rechts) Modellierungsansatz. Darstellung basierend auf [WK14] . . . 14
2.3. BPEL4Chor Artefakte. Darstellung basiert auf [DKLW07] . . . 17

4.1. Ein Beispiel für die Verwendung eines Choreographiecontainers . . . 23
4.2. Architekturübersicht aller Teilnehmer . . . 25
4.3. Datenmodell einer Choreographie mit Choreographiecontainer . . . 27
4.4. Zusammenhang zwischen Datenmodell und Choreographieartefakten . . . 28
4.5. Zusammenspiel der Prozessartefakte . . . 30
4.6. Beispiel für die Verwendung eines Choreographiecontainers . . . 31
4.7. Zwei Darstellungsoptionen für einen Prozess . . . 39
4.8. Darstellung der verschiedenen Verbindungsmöglichkeiten bei der Darstellung einer Choreographie mit Choreographiecontainer . . . 40
4.9. Visualisierung einer Choreographiecontainer mit Choreographievariablen . . . 41
4.10. Choreographievariablen mit detaillierten Schreibvorgängen . . . 42
4.11. Externe Nutzer mit Lese- und Schreibzugriffen . . . 43
4.12. Beispiel für eine Veröffentlichung von Zwischenergebnissen . . . 44
4.13. Beispiel für wechselnde Parameter . . . 45
4.14. Beispiel für ein zusammengefasstes Ergebnis . . . 46
4.15. Beispiel für einen schreibenden Prozess und viele Lesende . . . 47
4.16. Beispiel für den Umgang mit großen Datenmengen . . . 48
4.17. Beispiel für die Verwendung von Konstanten . . . 49
4.18. Synchronisation zwischen den verschiedenen Teilen einer Choreographie durch Abfragen . . . 50
4.19. Synchronisation zwischen den verschiedenen Teilen einer Choreographie durch eine Synchronisationsnachricht . . . 51
4.20. Softwarearchitektur eines Choreographiecontainers . . . 53
4.21. Oberfläche eines Designers für eine Choreographie mit Choreographiecontainer . . . 54
4.22. Eigenschaftsreiter des Choreographiecontainers . . . 55
4.23. Eigenschaftsreiter von Prozessen und Aktivitäten . . . 55
4.24. Eigenschaftsreiter eines externen Nutzers . . . 56
4.25. Eigenschaftsreiter einer zusammengesetzten Variable . . . 56
4.26. Eigenschaftsreiter einer atomaren Variablen . . . 57
4.27. Eigenschaftsreiter eines Kontrollflusses . . . 58

4.28. Eigenschaftsreiter einer Nachrichtenverbindung 58
4.29. Eigenschaftsreiter eines Datenflusses 58

5.1. Übersicht des Editors 60
5.2. Ecore Modell der PBD 61
5.3. Eigenschatfsfenster des CDataLinkable Elements 62
5.4. Ecore Modell der Choreographie 62
5.5. Das Tooling Definition Model 63
5.6. Figure Descriptor Abschnitt im Graphical Definition Model 64
5.7. Compartment Abschnitt im Graphical Definition Model 64
5.8. Labels Abschnitt im Graphical Definition Model 65
5.9. Nodes und Connections Abschnitt im Graphical Definition Model 65
5.10. Mapping Definition Model 66
5.11. Eigenschaftenfenster des CContainers 67
5.12. Eigenschaftenfenster der Child Reference einer CComplexVariable 67
5.13. Eigenschaftenfenster der Child Reference einer CComplexVariable innerhalb einer CComplexVariable zur Selbstbeinhaltumg 67
5.14. Eigenschaftenfenster eines CDataLink 68
5.15. Beispiel des Editors mit Choreographiecontainer 69
5.16. Beispiel einer Darstellung auf der Zeichenfläche 70

Verzeichnis der Listings

2.1. Beispiel für die Darstellung der Daten einer Person in XML 13
2.2. XSD des Beispiel für die Darstellung der Daten einer Person in XML 13
2.3. Definition einer BPEL Variable nach [OAS07] 15
2.4. Beispiel für eine Variable in BPEL 16

4.1. Schema des Wurzelelements 32
4.2. Aufbau des Wurzelelements 33
4.3. Schema einer atomaren Variablen 33
4.4. Darstellung der beiden atomaren Variablen aus Beispiel 4.6 34
4.5. Schema einer zusammengesetzten Variablen 34
4.6. Darstellung der zusammengesetzten Variablen aus Beispiel 4.6 35
4.7. Schema der schreibenden und lesenden Nutzer 35
4.8. Schema des Wurzelelements des External User Descriptors 36
4.9. Aufbau des Wurzelelements 36
4.10. Schema der Rollen der Nutzer 37
4.11. Schema Schreib- und Lesemöglichkeiten 37
4.12. Schema der Variablen 37

4.13. Darstellung der externen Nutzer Administrator und Forscher aus Beispiel 4.6 38

5.1. BPEL Serialisierung von Prozess1 70

A.1. Vollständiger Container Descriptor von Beispiel 4.6 73
A.2. Vollständiger External User Descriptor von Beispiel 4.6 74
A.3. XML-Schema des Container Descriptors 75
A.4. XML-Schema des External User Descriptors 76

Abkürzungsverzeichnis

Apache ODE	Apache Orchestration Director Engine
API	Application Programming Interface
BPEL	Business Process Execution Language
EMF	Eclipse Modeling Framework
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
PBD	Participant behavior description
REST	Representational State Transfer
WS-CDL	Web Services Choreography Description Language
WSDL	Web Services Description Language
XML	Extensible Markup Language
XSD	XML Schema Definition

1. Einleitung

Die Nutzung von Workflowtechnologien hat Einzug in eine Vielzahl von Anwendungsgebieten gehalten. Zum einen werden sie in der Industrie zur Darstellung und Automatisierung von Geschäftsprozessen, zum anderen in der Forschung zur Durchführung von Simulationen verwendet. Ein globales Modell in dem mehrere Prozesse bzw. Simulationen, als Teilnehmer miteinander interagieren wird als *Choreographie* bezeichnet. Das Verhalten jedes einzelnen Teilnehmers wird durch ein Prozessmodell definiert. Eine Choreographie beschreibt auch die Kommunikation zwischen den einzelnen Teilnehmern. In der Forschung können auf diese Art einzelne Simulationen zu einer großen Simulation, wie z. B. einer *Multi–Skalen Simulation*, zusammengefasst werden. Insbesondere bei den wissenschaftlichen Simulationen können dabei große Datenmengen anfallen. Diese Daten können z. B. eine große Menge an Einzelergebnissen oder auch ein vollständiger 3D–Scan eines menschlichen Körpers sein.

In bisherigen Choreographien können Daten nur statisch in den einzelnen Prozessmodellen hinterlegt werden oder dynamisch in Form von Nachrichten, zur Laufzeit, an eine Prozessinstanz geschickt werden. Wenn mehrere Prozessmodelle dieselben Daten benötigen, wie z. B. eine Konstante, muss diese in jedem Modell einzeln hinterlegt werden. Wenn eine Änderung der Konstante eintritt müssen alle Prozessmodelle entsprechend angepasst werden. Wenn Daten dynamisch zwischen Prozessinstanzen ausgetauscht werden, kann es passieren, dass ein Anwender die benötigten Daten über eine Nachricht an eine der Prozessinstanzen, welche auf diese Nachricht wartet, sendet und auf sie reagiert, in dem die Prozessinstanz die Nachricht an andere Teilnehmer weitersendet. Im schlimmsten Fall werden dabei die Daten über Prozessinstanzen geleitet, die die Daten gar nicht benötigen und nur zur Koordination anderer Prozessinstanzen verwendet werden. Durch dieses Verhalten entsteht ein höheres Datenaufkommen als eigentlich benötigt wird.

Daher wird im Rahmen dieser Arbeit ein Konzept für sogenannte *Choreographiecontainer* vorgestellt. Choreographiecontainer ermöglichen es, Datenflüsse zwischen den Teilnehmern einer Choreographie und Daten die außerhalb der Choreographieumgebung gespeichert sind zu definieren und können somit die oben beschriebenen Probleme reduzieren, indem alle Teilnehmer die Zugriff auf bestimmte Daten brauchen, diese gleichzeitig erhalten können. Ein weiterer Vorteil bei der Verwendung eines Choreographiecontainers ist es, dass auch Interessenten, die nicht Teil der eigentlichen Choreographie sind, Zugriff auf die erzeugten Daten gegeben werden kann.

Das primäre Ziel dieser Arbeit ist es, ein Konzept für die Darstellung, der Verwendung und des Aufbaus eines Choreographiecontainers zu entwickeln. Außerdem wird ein bestehender *Choreographieeditor* [Son13] erweitert um Choreographien mit Choreographiecontainern graphisch modellieren zu können.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen beschäftigt sich mit grundlegenden Konzepten und Technologien, deren Verständnis für diese Arbeit wichtig sind.

Kapitel 3 – Verwandte Arbeiten beschäftigt sich mit anderen Arbeiten, die sich mit einer ähnlichen Thematik befassen.

Kapitel 4 – Konzept beschreibt das Konzept von Choreographiecontainern und geht detailliert auf deren graphische Darstellung mit und ohne Hilfe eines graphischen Editors ein.

Kapitel 5 – Realisierung beschäftigt sich mit der geplanten und der durchgeführten Implementierung der Editor Komponente einer Choreographie mit Choreographiecontainer.

Kapitel 6 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und beschreibt Möglichkeiten für weiterführende Arbeiten zum Thema Choreographiecontainer.

2. Grundlagen

In diesem Kapitel werden die Grundlagen beschrieben, die für diese Arbeit verwendet werden. Zunächst werden die Grundbegriffe: *Orchestrierung*, Choreographie, *Orchestrierungsengine* und *Choreographieumgebung* erläutert. Im Anschluss werden die *Extensible Markup Language (XML)* und die *XML Schema Definition (XSD)* erläutert. Des Weiteren wird auf die Modellierung einer Choreographie, so wie die Orchestrierungssprache *WS–Business Process Execution Language (BPEL)* und die darauf aufbauende Choreographiesprache *BPEL4Chor*, eingegangen. Da für die Realisierung des Konzepts die Entwicklungsumgebung *Eclipse* und deren Erweiterungen *Modeling Framework (EMF)*, *Graphical Editing Framework (GEF)* und *Graphical Modeling Framework (GMF)* verwendet werden, werden diese näher beschrieben.

2.1. Grundbegriffe

Dieser Abschnitt erläutert einige Grundbegriffe die für das Verständnis dieser Arbeit besonders wichtig sind.

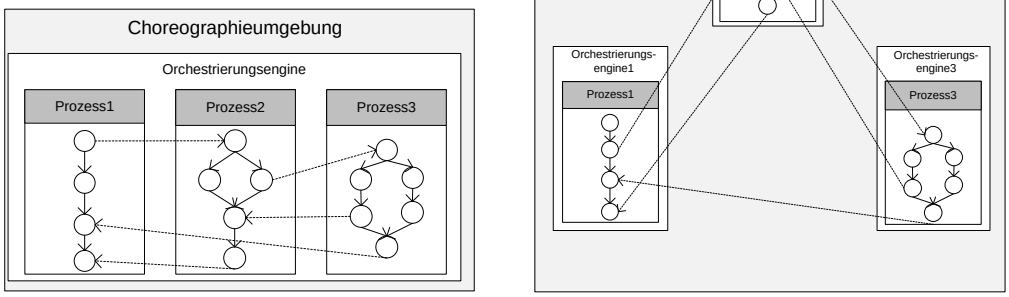
2.1.1. Orchestrierung und Choreographie

Eine Orchestrierung, im Kontext dieser Arbeit, beschreibt einen Prozess, dessen Aktivitäten als *Webservices* realisiert sind. Die Interaktion mit den Webservices wird dabei nur aus Sicht dieses Prozesses beschrieben [KL08].

Wenn mehrere Prozesse miteinander interagieren nennt man dies eine Choreographie. Die Choreographie beschreibt dabei jedoch nicht unbedingt den Ablauf der einzelnen Prozesse, sondern konzentriert sich hauptsächlich auf Nachrichtenaustausch zwischen den Prozessen. Eine Choreographie beschreibt somit das Zusammenspiel verschiedener Prozesse aus einer globalen Perspektive [KL08].

2.1.2. Orchestrierungsengine und Choreographieumgebung

Eine Orchestrierungsengine ist in der Lage, einen in einer Prozessbeschreibungssprache wie BPEL beschriebenen Prozess auszuführen. Die Orchestrierungsengine selbst dient dabei als eine Art Container in der die Prozess Beschreibung deployt werden kann. Die Orchestrierungsengine liest dabei die Prozessbeschreibung ein und führt die darin enthaltenen Befehle, wie den Aufruf eines Webservices, aus. Ein Beispiel einer solchen Orchestrierungsengine ist *Apache ODE (Orchestration Director Engine)* [Foua].



(a) Darstellung einer Choreographie mit einer Orchestrierungsengine

(b) Darstellung einer Choreographie mit mehreren Orchestrierungsengine

Abbildung 2.1.: Darstellung einer Choreographie in zwei verschiedenen Versionen einer Choreographieumgebung

Eine Choreographieumgebung ist ein Konstrukt welches, wie in Abbildung 2.1 dargestellt, die zu einer Choreographie gehörenden Prozesse und Orchestrierungsengines auf denen die Prozesse ausgeführt werden, umfasst. Da eine Orchestrierungsengine potentiell in der Lage ist mehrere Prozesse gleichzeitig auszuführen, kann die Choreographie Umgebung, wie in Abbildung 2.1a dargestellt, aus nur einer Orchestrierungsengine auf der sämtliche Prozesse einer Choreographie ausgeführt werden bestehen. Eine Choreographieumgebung kann aber auch, wie in Abbildung 2.1b dargestellt, aus mehreren Orchestrierungsengines bestehen auf denen, die jeweils zu dieser Choreographie gehörenden Prozesse ausgeführt werden.

2.2. Extensible Markup Language

XML ermöglicht es, Daten hierarchisch dargestellt, in einem Textdokument zu speichern [W3Ca]. Die für diese Arbeit wichtigsten Teile einer XML Datei sind Elemente und Attribute. Listing 2.1 zeigt ein Beispiel für die XML Darstellung der Daten einer Person. Das Wurzelelement dieser XML Darstellung ist das Element *person*. Dieses wird durch das Tag *<person>* begonnen und endet mit dem schließenden Tag *</person>*. In diesem Wurzelelement befinden sich drei weitere Elemente namens: *vorname*, *nachname* und *alter*. Das Element *person* enthält zusätzlich das Attribut *status*. Dieses stellt eine Metainformation für dieses Element dar.

Listing 2.1 Beispiel für die Darstellung der Daten einer Person in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<person status="wichtig">
      <vorname>Max</vorname>
      <nachname>Mustermann</nachname>
      <alter>42</alter>
</person>
```

Listing 2.2 XSD des Beispiel für die Darstellung der Daten einer Person in XML

```
<xs:element name="person">
      </xs:complexType>
            <xs:sequence>
                  <xs:element name="vorname" type="xs:string"/>
                  <xs:element name="nachname" type="xs:string"/>
                  <xs:element name="alter" type="xs:integer"/>
            </xs:sequence>
            <xs:attribute name="status" type="xs:string"/>
      </xs:complexType>
</xs:element>
```

2.3. XML Schema Definition

Eine XML Schema Definition (XSD) ermöglicht es, XML Dokumente strukturiert zu beschreiben [W3Cb]. Listing 2.2 zeigt die XSD für das XML Dokument aus Listing 2.1. Da das Element Person weitere Elemente beinhalten und über ein Attribut verfügen soll, wird es als *complexType* modelliert. Das Element *person* soll eine Reihe andere Elemente in festgelegter Reihenfolge enthalten. Dies wird durch das Tag *<xs:sequence>* realisiert. Die enthaltenen Elemente sind alle einfache Elemente und bestehen nur aus deren Namen und dem jeweiligen Datentyp. Zusätzlich soll das Element *person* über ein Attribut verfügen, die Attribute müssen nach den Elementen modelliert werden. Bis auf den Tagnamen Attribut, folgt ein Attribut dem selben Aufbau wie ein Element.

2.4. Modellierung von Choreographien

Die folgende Beschreibung basiert auf der Beschreibung in [WK14], wurde jedoch abstrahiert damit der darin beschriebene Ansatz sich nicht nur auf Multi–Skalen sondern allgemein anwenden lässt. Es werden zwei Ansätze für die Modellierung von Choreographien beschrieben.

Der erste Ansatz ist der *Top–down Ansatz*. Bei diesem Ansatz wird zunächst die Problembeschreibung bzw. die Aufgabe definiert die gelöst werden soll (1). Im nächsten Schritt wird mit Hilfe eines graphischen Editors die Choreographie modelliert(2). In dem darauf folgenden Schritt (3), werden aus der graphischen Darstellung der Choreographie abstrakte Prozesse erzeugt. Die abstrakten Prozesse

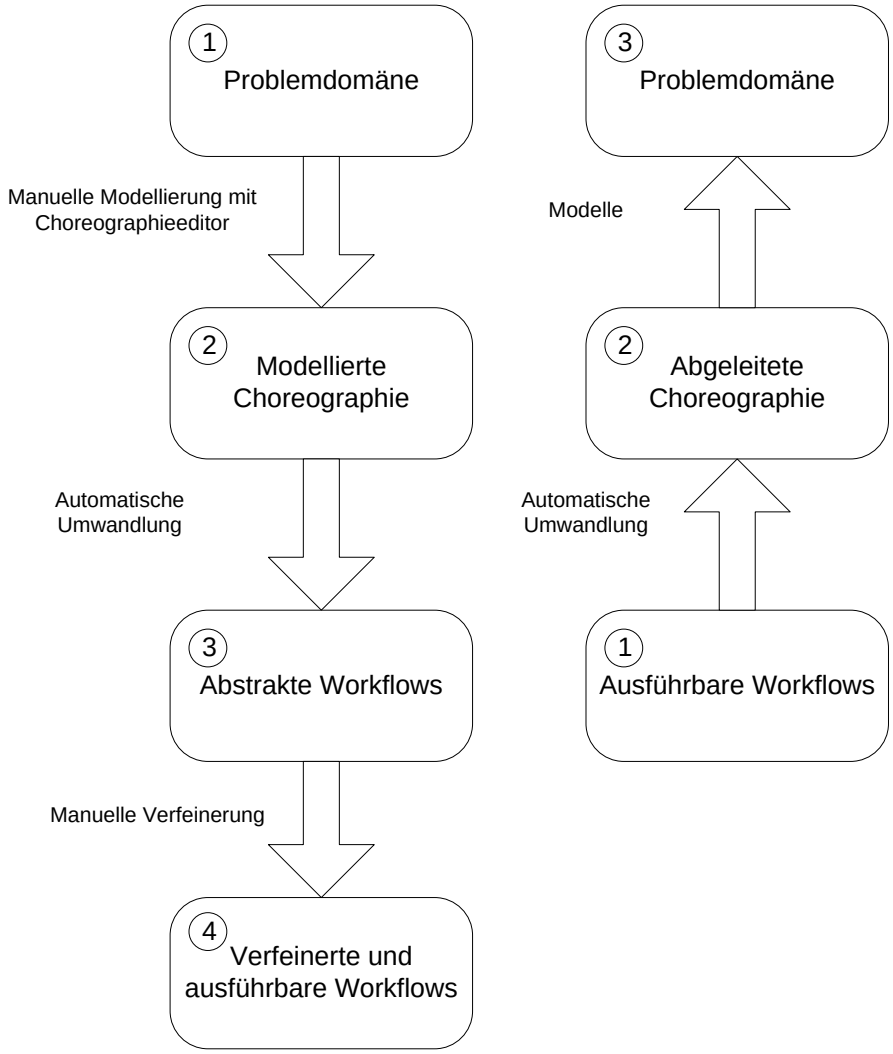


Abbildung 2.2.: Top–down (links) und bottom–up (rechts) Modellierungsansatz. Darstellung basierend auf [WK14]

enthalten dabei nur Aktivitäten welche für die Kommunikation zwischen den einzelnen Prozessen notwendig sind. Die abstrakten Prozesse werden anschließend (4) manuell verfeinert. Die Verfeinerung der einzelnen abstrakten Prozesse kann durch Spezialisten auf dem jeweiligen Gebiet erfolgen.

Bei dem *Bottom–Up* Ansatz wird auf bereits bestehende Prozesse, für die jedoch kein globales Modell vorhanden ist, zurück gegriffen (1). Diese werden in eine abgeleitete Choreographie umgewandelt (2). Aus der abgeleiteten Choreographie kann der Modellierer die Choreographie weiter anpassen, um z. B. nicht benötigte Prozesse zu entfernen, oder weitere hinzuzufügen. Die abgeleitete Choreographie repräsentiert ihrerseits eine Problembeschreibung (3).

Listing 2.3 Definition einer BPEL Variable nach [OAS07]

```
<variables>
  <variable name="BPELVariableName"
    messageType="QName"?
    type="QName"?
    element="QName"?>+
    from-spec?
  </variable>
</variables>
```

2.5. Business Process Execution Language

BPEL ist eine XML–basierte Sprache, mit der Orchestrierungen beschrieben werden können [OAS07]. Die für diese Arbeit verwendete Version BPEL 2.0 wurde 2007 von OASIS standardisiert. Es gibt keine standardisierte graphische Darstellung. Die Hauptbestandteile eines BPEL Dokumentes sind: *Partner Links*, *Correlation Sets*, *Handlers*, *Aktivitäten* und *Variablen*.

Zusätzlich zu den oben genannten Teilen gibt es das Prozesselement. Dieses ist das Wurzelelement welches alle anderen Bestandteile beinhalten kann. Es ist somit die BPEL Darstellung des modellierten Prozesses. Das Prozesselement kann in seiner Definition Variablen enthalten, auf die alle Aktivitäten innerhalb des Prozesses zugreifen können.

Partner Links dienen als Beschreibung für einen Kommunikationskanal zwischen dem Prozess und einem Partner. Correlation Sets ermöglichen es der Orchestrierungsengine Nachrichten zu der richtigen Prozessinstanz zu leiten. Handlers kümmern sich darum, was in bestimmten Situationen wie dem Auftreten eins Ausnahmezustandes oder eines bestimmten Ereignisses passieren soll.

Es gibt zwei Sorten von Aktivitäten: *Basisaktivitäten* und *strukturierte Aktivitäten*. Basisaktivitäten werden z. B. verwendet um Webservices aufzurufen oder Nachrichten zu empfangen. Strukturierte Aktivitäten realisieren den Kontrollfluss und können sowohl strukturierte Aktivitäten als auch Basisaktivitäten enthalten. Eine für diese Arbeit besonders wichtige strukturierte Aktivität ist die *Scope Activity* . Scope Activities können als einzige Aktivität Variablen enthalten. Wenn einer Scope Activity Variablen hinzugefügt werden, kann jede Aktivität innerhalb der Scope Activity auf die Variable zugreifen. Ein Zugriff von außerhalb der Scope Activity auf die Variable ist nicht möglich.

Variablen können Nachrichten oder Daten enthalten. Die Definition einer Variable ist ist Listing 2.3 dargestellt. Eine Variable hat das Attribut *name*, welches den Namen der Variablen wiedergibt. Eine Variable muss eines der Attribute *Web Services Description Language (WSDL) messageType* oder *XML Schema type* oder ein *XML Schema element* enthalten. Mit diesem Attribut kann man den Typ der Variablen definieren. Das Attribut *from–spec* kann optional verwendet werden um die Variable zu initialisieren. Listing 2.4 zeigt ein Beispiel für eine Variable. Diese Variable hat den Namen *UserName* und ist vom einfachen XSD Datentyp *String*.

Listing 2.4 Beispiel für eine Variable in BPEL

```
<variables>
      <variable   name="UserName"
            type="xsd:String" />
</variables>
```

2.6. BPEL4Chor

BPEL selbst fehlt eine Möglichkeit eine Choreographie zu modellieren. Aus diesem Grund wurde BPEL4Chor entwickelt [DKLW07]. Eine BPEL4Chor Choreographie besteht, wie in Abbildung 2.3 dargestellt, aus drei verschiedenen Artefakttypen: der *Participant topology*, den *Participant behavior descriptions* und den *Participant groundings*.

Die Participant topology beschreibt die strukturellen Aspekte einer Choreographie, in Form der verschiedenen Teilnehmer und deren Kommunikation untereinander und enthält die *Participant Declaration* und die *Message Links*. Sie enthält somit die drei neu hinzugefügten Elemente *Participant*, *Participant Set* und *Message Link*. Die Participants stellen die Teilnehmer der Choreographie dar und die Participant Sets eine Menge an Teilnehmern. Ein Beispiel für einen Teilnehmer kann ein bestimmtes Hotel sein dessen Prozess das Einchecken eines Kunden wiedergibt. Ein Participant Set in der selben Choreographie wären verschiedene Autovermietungen mit eigenen Prozessen. Die *Message Links* ermöglichen es die Kommunikation zwischen verschiedenen Teilnehmern darzustellen.

Für jeden Teilnehmer wird eine Participant behavior description (PBD) benötigt bzw. erzeugt. Dieses Artefakt beschreibt den Kontrollfluss innerhalb der einzelnen Teilnehmer. Bei den PBDs handelt es sich um abstrakte BPEL Prozesse. Abstrakte BPEL Prozesse, sind wie normale Prozesse jedoch enthalten sie nicht ausführbare *Opaque Activities* und nutzen den abstrakten Namespace. Diese Opaque Activities dienen als Platzhalter und können mit beliebigen BPEL Editoren verändert und angepasst werden. Die PBDs sind für diese Arbeit von besonderem Interesse, da in diese die BPEL Variablen und Scope Activities eingefügt werden können.

Die *Participant groundings* Artefakte enthalten genaue Informationen über technischen Details der Choreographie wie z. B. *port types*. Diese Informationen sind von den anderen Artefakten entkoppelt, um die Wiederverwendbarkeit der anderen Artefakte zu erhöhen.

2.7. Eclipse

Eclipse [Foue] ist eine auf *Java* basierende Entwicklungsumgebung. Die *Eclipse* Entwicklungsumgebung kann als Entwicklungsumgebung für diverse Programmiersprachen, wie z. B. Java [Ull10], genutzt werden. Außerdem ist der Funktionsumfang von Eclipse über *Plugins* erweiterbar. Bei EMF, GEF und GEF handelt es sich ebenso um Plugins wie bei dem an der Universität Stuttgart erstellten BPEL4Choreditor [Son13].

EMF wird genutzt um Datenmodelle, in Form von Metamodellen, für eine bestimmte Problem Domäne zu erzeugen. Das Metamodell kann entweder manuell von einem Entwickler erzeugt werden oder

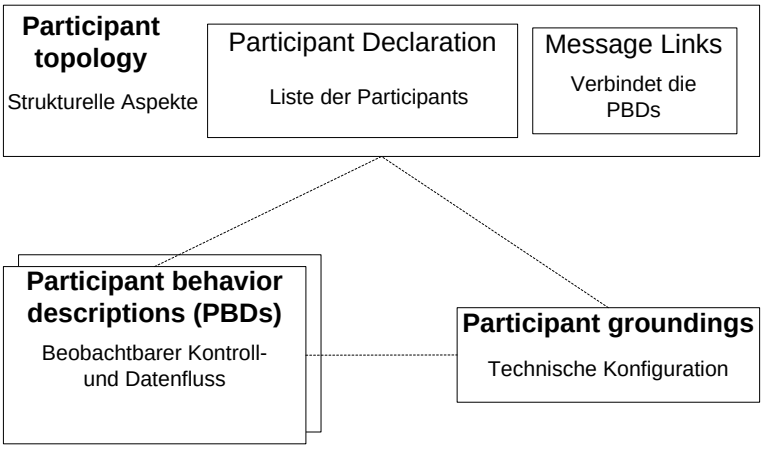


Abbildung 2.3.: BPEL4Chor Artefakte. Darstellung basiert auf [DKLW07]

aus z. B. bereits bestehenden XSDs oder auch Java *Annotationen* [Ull10] ausgelesen werden. Das Metamodell wird in einer *.ecore* Datei gespeichert und wird im weiteren Verlauf auch als *Ecore Modell* bezeichnet. Aus dem Metamodell kann automatisch Code erzeugt werden, mit dem sich Instanzen des Metamodells manipulieren lassen. Es nimmt somit dem Entwickler die Aufgabe ab, den Code der benötigt wird um die Instanzen z. B. zu ändern oder zu serialisieren, manuell erzeugen zu müssen [SBPM09].

GEF hilft dabei graphische Editoren zu erstellen. Es handelt sich hierbei um eine Implementierung der *Model-View-Controller Architektur* [Fow]. Ein Datenmodell entspricht hierbei dem *Model*, die *View* ist die graphische Repräsentation des Modells und der Editor übernimmt die Funktion des *Controllers* [Maj].

GMF wurde entwickelt um die Vorteile und Funktionen von EMF und GEF miteinander zu kombinieren [Gro09]. GMF ermöglicht es, mit Hilfe eines Editors, ein mit EMF erstelltes Modell mit GEF zu nutzen. Ein GMF Projekt besteht zunächst aus drei Teilen: dem Domain Model, dem Graphical Definition Model und dem Tooling Definition Model. Das *Domain Model* enthält die Beschreibung des Domänenmodells und entspricht somit einem Ecore Modell. Das *Graphical Definition Model* enthält die Informationen, wie die Teile des Ecore Modells, welche graphisch dargestellt werden, aussehen sollen, also welche Form, Farbe usw. die graphischen Darstellungen haben sollen. Das *Tooling Definition Model* gibt an, welche Werkzeuge mit denen die graphischen Darstellung erzeugt werden sollen, in der Palette zu sehen sind. Diese drei Modelle werden in einem *Mapping Definition Model* miteinander verknüpft. Aus dem Mapping Definition Model lässt sich ein *Generator Model* erstellen. Aus diesem Modell kann der Code für einen graphischen Editor in Eclipse automatisch erzeugt werden.

3. Verwandte Arbeiten

Die Choreographiesprache WS-CDL [W3C05] wurde von dem World Wide Web Consortium (W3C) entwickelt. Diese Sprache basiert auf XML und ermöglicht es, Choreographien auf eine vorgegebene Art zu beschreiben. Choreographien werden hierbei auf einer abstrakten Ebene beschrieben. In WS-CDL werden Variablen auf vier Arten genutzt:

- Information Exchange Capturing Variables, werden verwendet um Nachrichten mit einem Inhalt zu füllen oder sie werden durch eine erhaltene Nachricht gefüllt.
- State Capturing Variables, werden verwendet um den Zustand der einzelnen Teilnehmer darzustellen.
- Channel Capturing Variables, werden verwendet um Informationen über die Adressen an welche Daten gesendet werden sollen, deren Policies usw. zu speichern.
- Exception Capturing Variables, werden verwendet um Informationen über Exceptions zu speichern.

Alle Choreographieteilnehmer, die in der entsprechenden Variablendefinition vermerkt sind, können diese lesen oder je nach Definition auch deren Wert verändern. Alle anderen Variablen Typen werden auch global definiert, die einzelnen Teilnehmer verwenden diese allerdings lokal. Lokal bedeutet hier, dass die Variablen zwar denselben Namen haben können, jedoch unterschiedliche Werte enthalten können. Es existiert keine offizielle Graphische Notation für WS-CDL. Die Hauptgemeinsamkeit zu dieser Arbeit liegt darin, dass Variablen außerhalb von einzelnen Teilnehmern auf einer Choreographieebene definiert werden und die Aufteilung in verschiedene Variablentypen.

In [BWH08b] wird beschrieben, wie Kontroll- und Datenfluss in einem Workflow getrennt werden können. Bei diesem Ansatz handelt es sich um einen Hybridansatz bestehend aus Choreographie und Orchestrierung. Der Kontrollfluss wird zentral durch eine Workflow Engine gesteuert, während der Datenfluss auch zwischen einzelnen Webservices ermöglicht wird. Zu diesem Zweck werden sogenannte Proxies eingeführt. Proxies dienen als zwischen Station zwischen der Orchestrierung und den einzelnen Webservices. Durch entsprechende API-Befehle [BWH08a] ist es der Orchestrierung möglich, durch den Proxy, Daten direkt von einem oder mehreren Webservices zu erhalten oder zu befehlen, dass Daten, über den Proxy direkt zwischen Webservices ausgetauscht werden. Die Idee Daten über einen Proxy auszutauschen wird in dieser Arbeit aufgegriffen.

In [WGSL09] wird beschrieben, wie größere Datenmengen mittels einer Referenz ausgetauscht werden können. Dieser Ansatz basiert auf BPEL und ist deshalb ebenfalls auf Orchestrierungen und nicht direkt auf Choreographien bezogen. Jeder Webservice erhält ein Reference Resolution System (RRS), dieses ist in der Lage, die Referenzen, die es von einem Workflow oder einem anderen RRS erhält, aufzulösen. Auf diese Art ist es möglich, Daten zwischen den RRS auszutauschen, ohne

dass dabei große Dateimengen durch die Orchestrierungsengine geleitet werden müssen. Für diesen Ansatz werden keine neuen BPEL Konstrukte benötigt, sondern lediglich bestehende erweitert. Der Ansatz ist somit BPEL konform. Dies wäre eine Realisierungsmöglichkeit wie der Austausch großer Datenmengen mittels Referenzen von statten gehen kann.

Die Orchestrierungsengine Apache ODE [Foua] hat bereits ein Konzept implementiert, mit dem auf Daten außerhalb des eigentlichen Workflows zugegriffen werden kann [Fouc]. Bei diesem Konzept werden Variablen, wie in [OAS07] beschrieben angelegt, jedoch um das Attribut *xvar:id* erweitert. Durch dieses zusätzliche Attribut erkennt ODE, dass es sich um eine externe Variable handelt, deren Daten außerhalb der Orchestrierungsengine verwaltet werden und ermittelt aus dem Deployment Descriptor [Foub] wie auf die Variable zugegriffen werden soll. Derzeit wird nur der Zugriff auf eine Datenbank mittels eines Java Database Connectivity(JDBC) Mappings ermöglicht. Der Deployment Descriptor enthält dabei die Information auf welche Tabelle, welche Spalte usw. zugegriffen werden soll.

In [RHEA05] werden verschieden Data Patterns beschrieben und klassifiziert. Das Pattern Enviromental Data (Pattern 8) ist das Pattern, das bei der hier erstellen Arbeit hauptsächlich verwendet wird. Dieses Pattern beschreibt, dass die Daten, die von einem Workflow verwendet werden, außerhalb des Workflows gespeichert werden.

4. Konzept

Dieses Kapitel beschreibt das Konzept eines, im folgenden auch nur *Container* genannten, *Choreogrpahiecontainers*. Dieser ermöglicht es, Daten außerhalb der Choreographieumgebung zu definieren und diese für die Teilnehmer der Choreographie zur Verfügung zu stellen. Im ersten Teil des Kapitels wird die Verwendung eines Choreographiecontainers anhand eines Beispiels erläutert. Im Anschluss werden die Definitionen von Choreographiecontainern und *Choreographievariablen* gegeben. Die folgenden Abschnitte beschreiben die Architektur eines Systems mit Choreographiecontainer, die erzeugten Choreographieartefakte *Container Descriptor* und *External User Descriptor*, die graphische Darstellung einer Choreographie mit Choreographiecontainer, Anwendungsfälle für einen Choreographiecontainer, Entwurfsentscheidungen die bei der Entwicklung eines Choreographiecontainer getroffen werden müssen, die Softwarearchitektur eines Choreographiecontainers und den Aufbau eines Editors zur Erstellung einer Choreographie mit Choreographiecontainer.

Im Rahmen dieser Arbeit wird davon davon ausgegangen, dass alle Choreographievariablen zur Laufzeit der Choreographie nur einmal einen Wert zugewiesen bekommen können. Diese Restriktion wird eingeführt, um Probleme bezüglich konkurrierenden Zugriffen zu vermeiden. Zum Beispiel kann das Problem, dass ein Prozess veraltete Daten, im Kontext der Choreographie, aus einer Choreographievariablen liest nicht auftreten, da diese im Kontext der aktuellen Choreographie immer als aktuell zu betrachten sind.

4.1. Beispiel

Das Konzept der Choreographiecontainer erweitert die Nutzung von Variablen dahingehend, dass sie nicht nur lokal innerhalb der Orchestrierungen, sondern in einem globalen Kontext einer Choreographie, verwendet werden können. Abbildung 4.1 zeigt ein Beispiel für einen Choreographiecontainer, in welchem ein Kunde eines Reiseveranstalters seine Reise organisiert. Die Choreographie besteht aus einem externen Nutzer, einem Choreographiecontainer und vier Teilnehmern, die durch Prozessmodelle repräsentiert werden.

Der Kunde tritt in der Rolle des externen Nutzers *Kunde* auf. Dieser gibt, z. B. über eine Internetseite seine *Kundeninformationen*, wie Name, Alter, Wunschziel, Preisspanne, Mietwagenwunsch, usw. an. Durch die Eingabe der Daten wird, nach deren Bestätigung, die Choreographie angestoßen.

Der Choreographiecontainer trägt den Namen *Reisedaten* und beinhaltet die Choreographievariablen: *ReiseAngebot*, die bereits weiter oben beschriebene Choreographievariable *Kundeninformation* und *AbgeschlosseneSchritte*. *ReiseAngebot* enthält am Ende der Choreographie das erstellte Reiseangebot. *AbgeschlosseneSchritte* enthält Informationen darüber, welche wichtigen Schritte der Choreographie bereits abgeschlossen wurden.

Die Choreographie hat vier Prozesse: *Reisebüro*, *Flug*, *Hotel* und *Mietwagen*. Der Prozess *Reisebüro* wird, nach der Eingabe der *Kundeinformationen* gestartet. Nach der Startaktivität *R1* werden, durch die beiden Aktivitäten *R2* und *R4*, die Prozesse *Flug* und *Hotel* gestartet. Danach wartet der *Reisebüro* Prozess bis die Ergebnisse der beiden anderen Prozesse gesendet werden.

Der Prozess *Flug* wird durch seine Aktivität *F1* gestartet und erarbeitet in den folgenden Aktivitäten den Flug, der die größte Übereinstimmung mit den angegeben Kundeninformationen und den besten Preis hat. Das Ergebnis wird im Anschluss durch die Aktivität *F4* an den Prozess *Reisebüro* gesendet. Außerdem wird in der Choreographievariablen *AbgeschlosseneSchritte* hinterlegt, dass ein Flugangebot ausgewählt und somit der Prozess abgeschlossen ist.

Der Prozess Hotel wird durch den Empfang der Nachricht von *R4* durch die Aktivität *H1* gestartet. *H2* sucht im Anschluss das Hotel, welches sich am besten mit den Kundeninformationen deckt und hinterlegt nachdem dieses gefunden wurde, den entsprechenden Wert in *AbgeschlosseneSchritte*. Die Aktivität *H3* sendet die Daten des gefunden Hotels an den Prozess Mietwagen.

Der Prozess *Mietwagen* sucht, nachdem er in *M1* gestartet wurde in *M2* den besten Mietwagen für die Gegend des Hotels. Dieser Prozess benötigt Daten des *Kunden*, wie das Alter, da Angebote eventuell von Parametern wie dem Alter des *Kunden* abhängig sein können. Diese Daten werden direkt aus dem Choreographiecontainer gelesen. Nachdem das beste Angebot gefunden wurde, wird in der Choreographievariablen der entsprechende Wert gesetzt und das Ergebnis zurück an den *Hotel* Prozess gesendet. Der Prozess *Hotel* sendet nach Erhalt der Information über das beste Mietwagenangebot, dieses und das beste Hotelangebot an den *Reisebüro* Prozess.

Nachdem die beiden Aktivitäten *R3* und *R4* die Nachrichten über den besten Flug, das beste Hotel und den besten Mietwagen erhalten haben, wird das beste gesamt Angebot erstellt und in die Choreographievariable *ReiseAngebot* geschrieben. Diese kann von dem *Kunden* eingesehen werden.

Dieses Beispiel zeigt den Vorteil des Choreographiecontainers, indem alle Prozesse, die Daten aus *Kundeninformationen* benötigen auf diese zugreifen können. Ohne Choreographiecontainer müsste der Prozess *Reisebüro* die *Kundeninformationen* an die Prozesse *Flug* und *Hotel* senden und der Prozess *Hotel* müsste die Daten anschließend noch an den Prozess *Mietwagen* weiterleiten. Statt Nachrichten mit potentiell großen Nutzerdaten werden nur leere oder Steuerungsnachrichten gesendet um die Prozesse zu starten. Ein weiterer Vorteil ist, dass der *Kunde* über die Einsicht auf die Choreographievariable *AbgschlosseneSchritte* eine Möglichkeit erhält den Status seines Auftrags einzusehen, ohne Einblicke in die Choreographie erhalten zu müssen. Außerdem kann der Kunde, eine Referenz auf die Daten im *ReiseAngebot*, z. B. in Form eines Links, an Freunde weitersenden.

4.2. Definitionen

Definition 4.2.1 (Choreographiecontainer)
Ein Choreographiecontainer ist eine Komponente welche es ermöglicht prozessübergreifend Daten in einer Choreographie zu nutzen. Er dient dabei als Container für einzelne Choreographievariablen. Ein Choreographiecontainer ermöglicht es, Daten außerhalb des Nachrichtenflusses einer Choreographie zu definieren und somit dass die Daten auch außerhalb des Nachrichtenflusses ausgetauscht werden können.

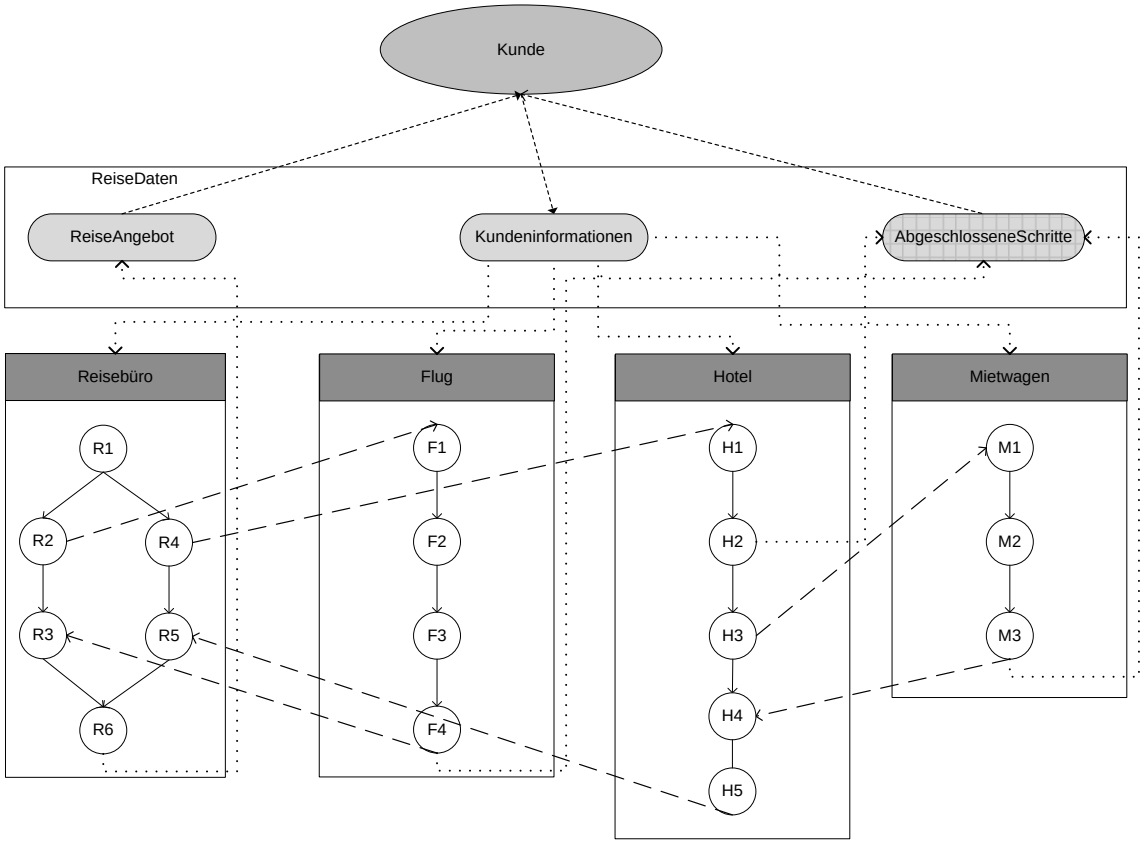


Abbildung 4.1.: Ein Beispiel für die Verwendung eines Choreographiecontainers

Definition 4.2.2 (Choreographievariable)
Als Choreographievariable wird eine konkrete Entität innerhalb eines Choreographiecontainers bezeichnet in der Daten gespeichert werden können. Dies können z. B. einfache Datentypen wie eine Gleitkommazahl oder auch binäre Daten sein. Es ist außerdem möglich, dass die Variablen auch komplexe Datenstrukturen enthalten können.

4.3. Architektur einer Choreographie mit Choreographiecontainer

In diesem Abschnitt wird ein Überblick über die Komponenten gegeben, aus denen eine Choreographie mit Choreographiecontainer besteht. Im weiteren Verlauf des Abschnitts wird das Datenmodell einer solchen Choreographie und das Zusammenspiel der erzeugten Choreographieartefakte erläutert.

4.3.1. Architekturübersicht aller Komponenten

Wie in Abbildung 4.2 dargestellt, besteht eine Choreographie mit Choreographiecontainer aus den folgenden vier Elementen: der Choreographieumgebung, dem Choreographiecontainer, einem Datenserver und externen Nutzern.

Eine Choreographieumgebung besteht, wie in 2.1.2 näher beschrieben, aus einer oder mehreren Orchestrierungsengines, welche die infrastrukturelle Grundlage für das Ausführen der einzelnen Prozessmodelle der Teilnehmer der Choreographie, bilden. Jede dieser Orchestrierungsengines muss um einen Choreographiecontainer zu unterstützen entsprechend erweitert werden. Im Rahmen dieser Arbeit nicht weiter auf die Entwicklung einer *Containererweiterung* für Orchestrierungsengines eingegangen, da sie sich auf die Modellierung von Choreographien mit Choreographiecontainer fokussiert. Sobald die Orchestrierungsengine auf einen Befehl stößt für den Daten aus dem Choreographiecontainer benötigt werden, sendet die Containererweiterung eine Anfrage an den Choreographiecontainer.

Der Choreographiecontainer enthält ein *Mapping* welches es der Choreographiecontainer Komponente ermöglicht die benötigten Daten, von einem *Datenserver* abzufragen. Das Mapping wandelt die Anfrage der Orchestrierungsengine in eine standardisierte Anfrage an den entsprechenden Datenserver um. Der Choreographiecontainer liest bei seinem Start eine, in 4.4.2 detailliert beschriebene, Konfigurationsdatei ein. Diese Datei enthält die Informationen, welche Variablen angelegt werden, wie diese aufgebaut sind und welche Teilnehmer Zugriff auf die Daten erhalten. Nach Erhalt einer Anfrage prüft der Choreographiecontainer zunächst ob der entsprechende Teilnehmer auf die Daten zugreifen darf und weist die Anfrage bei negativer Prüfung mit einer entsprechenden Nachricht ab. Falls dem Teilnehmer der Zugriff erlaubt wird, sendet der Choreographiecontainer Daten an den Datenserver oder liest Daten von diesem. Der Choreographiecontainer kann die Datenanfragen durch *Caching* oft verwendeter Daten im internen Speicher beschleunigen. Der Choreographiecontainer kann Anfragen von externen Nutzern entgegen nehmen und gibt ihnen entweder die Daten selbst weiter oder leitet diese direkt auf den Datenserver weiter.

Der Datenserver ist eine Komponente die Daten verwalten kann. Der Datenserver kann z. B. ein Representational State Transfer (REST)–Server [Fie00], eine Datenbank oder auch ein Dateiserver sein. Für diese Komponente wird keine spezielle Anpassungen an eine Choreographie benötigt.

Die *externen Nutzer* sind beliebige Anwendungen oder Benutzer, die sich außerhalb der Choreographieumgebung befinden. Diese können entweder auf den Choreographiecontainer oder direkt auf den Datenserver zugreifen um Daten zu speichern oder anzufragen.

4.3.2. Datenmodell

Abbildung 4.3 zeigt das zu der Darstellung von Choreographien mit Choreographiecontainer gehörende Datenmodell. Alle Attribute die in diesem Modell nicht als optional beschrieben werden sind verpflichtend. Das Element Choreographie entspricht der Zeichenfläche der gesamten Choreographie. Es gibt drei Elemente, die direkt zu einer Choreographie gehören: der Choreographiecontainer, die externen Nutzer und die Prozesse. Diese Elemente sind, mit Ausnahme von Datenfluss– und Nachrichtenaustausch–Verbindungen, die einzigen Elemente die direkt auf der Zeichenfläche platziert werden können.

Eine Choreographie kann beliebig viele Elemente vom Typ externer Nutzer und vom Typ Prozess enthalten. Beide Elemente haben das Attribut *Name*, welches den eindeutigen Namen des jeweiligen Elements enthält.

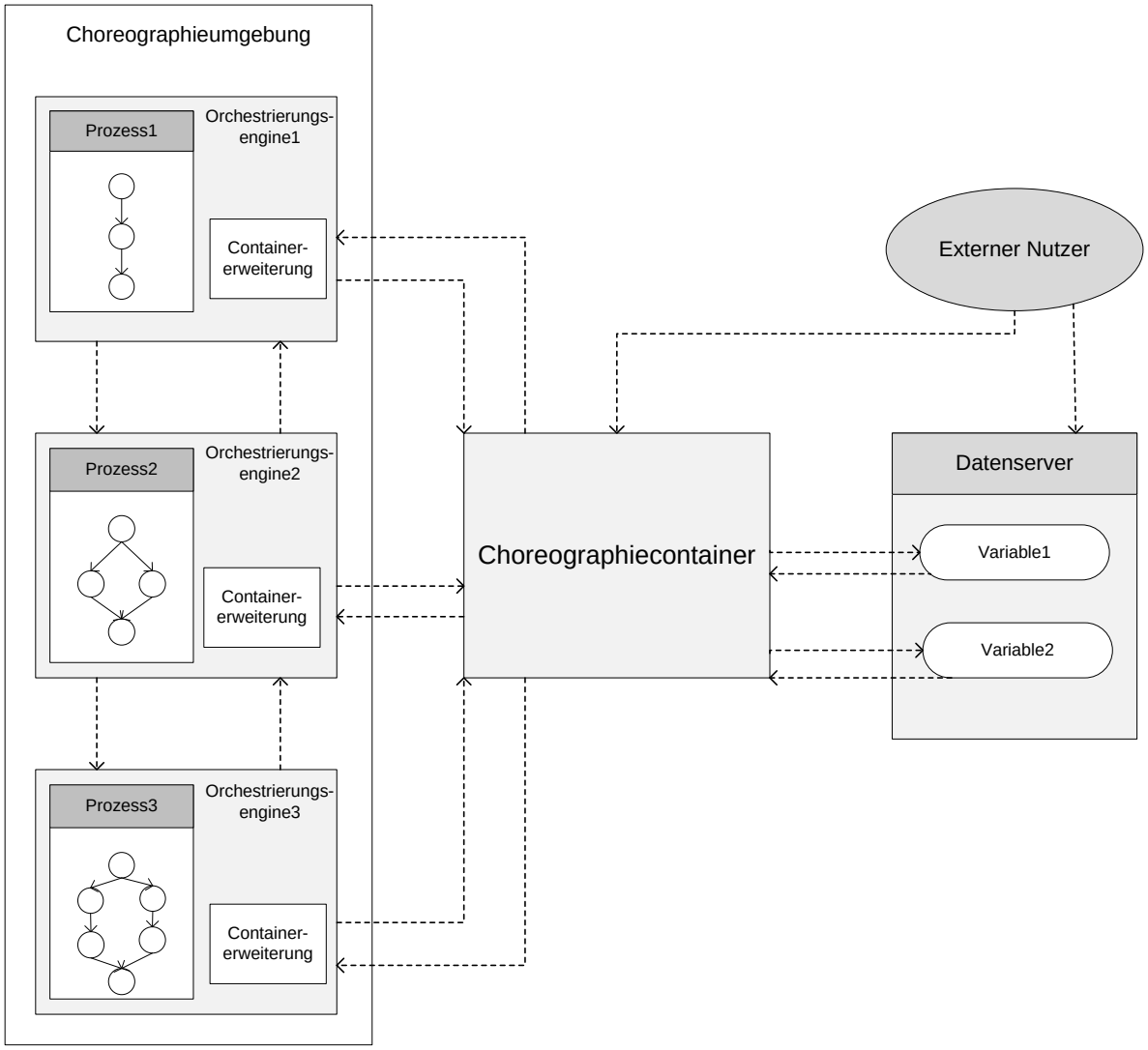


Abbildung 4.2.: Architekturübersicht aller Teilnehmer

Jede Choreographie kann nur maximal ein Element vom Typ Choreographiecontainer enthalten. Diese Einschränkung wird im Rahmen dieser Arbeit getroffen, um Synchronisations– und Abhängigkeitsprobleme zwischen mehreren Choreographiecontainern zu vermeiden. Jeder Choreographiecontainer hat die Attribute *Name* und *Adresse*. Das Attribut *Name* enthält den Namen des *Choreographiecontainers* und das Attribut *Adresse* die Adresse unter der der *Choreographiecontainer* erreichbar ist. Ein *Choreographiecontainer* kann beliebig viele Elemente vom Typ *atomare Variable* oder vom Typ *zusammengesetzte Variable* enthalten. Er muss jedoch mindestens ein Element von einem der beiden Typen beinhalten, da der *Choreographiecontainer* ansonsten überflüssig wäre.

Atomare Variablen und zusammengesetzte Variablen können nur innerhalb eines Choreographiecontainer oder einer zusammengesetzten Variablen platziert werden. Zusammengesetzte Variablen können eine beliebige Anzahl an zusammengesetzten Variablen oder atomaren Variablen beinhalten.

Das unterste Element einer zusammengesetzten Variablen muss immer eine atomare Variable sein, da nur diese Daten enthalten können. Beide Arten von Variablen haben das Attribut *Name* und das optionale Attribut *Permanent*. *Name* enthält den einzigartigen Namen der jeweiligen Variablen. Das Attribut *Permanent* signalisiert der Choreographiecontainer Komponente, dass die Daten dieser Variablen persistent gespeichert werden. Sobald einer Variablen, bei der dieses Attribut verwendet wird, ein Wert zugewiesen wird, wird dieser auf dem Datenserver serialisiert. Wenn das Attribut bei einer zusammengesetzten Variablen verwendet wird, gilt der Wert dieses Attributs für alle atomaren und zusammengesetzten Variablen die sich darin befinden. Atomare Variablen enthalten zusätzlich die Attribute: *Datentyp*, *Konstante* und *Referenz*. Das Attribut *Datentyp* enthält den Datentyp der für dies Variable verwendet wird. Das optionale Attribut *Konstante*, enthält die Information darüber, ob es sich bei der Variablen um eine *Konstante* handelt. Der Wert einer Konstanten muss gesetzt sein bevor eine Choreographie gestartet wurde und darf sich in deren Verlauf nicht mehr ändern. Das optionale Attribut *Referenz* zeigt an, ob die Variable einen direkten Wert oder die Referenz auf einen Wert oder eine Datei enthält. Dieses Attribut wird genutzt, wenn der Choreographiecontainer in einem System verwendet wird welches *Reference Passing* [WGSL09] unterstützt.

Prozesse können eine beliebige Anzahl von Aktivitäten enthalten. Da es möglich ist einen Prozess als Blackbox zu modellieren, muss ein Prozess nicht zwangsläufig Aktivitäten enthalten. Prozesse und Aktivitäten enthalten das Attribut *Name*, welches den einzigartigen Namen des jeweiligen Elements enthält.

Aktivitäten in dem selben Prozess können mit einer Kontrollfluss–Verbindung verbunden werden. Dabei gilt, dass es sich immer um eine 1:1 Verbindung und eine gerichtete Kante im Graphen handelt. Falls eine Aktivität mehrere Nachfolger hat, so sind Verbindungen als einzelne Elemente zu betrachten. Kontrollflüsse haben das Attribut *Bedingung*. Die Verwendung dieses Attributs ist optional und gibt an welche Bedingungen erfüllt sein müssen damit der Kontrollfluss ausgeführt wird.

Aktivitäten in zwei verschiedenen Prozessen können mit einer Nachrichtenaustausch–Verbindung, verbunden werden. Hierbei handelt es sich ebenfalls um eine 1:1 Verbindung. Dieses Element hat das Attribut *Name*, welches den Namen des Datenaustauschs und das Attribut *Datentyp*, welches den Datentyp der übertragenen Nachricht enthält.

Prozesse und strukturierte Aktivitäten können eigene lokale Variablen enthalten. Diese sind nur lokal im jeweiligen Element Prozess oder Aktivität vorhanden und auf diese kann nicht von außerhalb des jeweiligen Elements zugegriffen werden. Der Name der für das Attribut *Name* verwendet wird, muss einzigartig für den jeweiligen Prozess bzw. die jeweilige Aktivität in der sie definiert ist, sein. Das Attribut *Datentyp* gibt den Datentyp der betreffenden Variablen an. Diese Variablen werden in der folgenden Arbeit nicht explizit modelliert, weder in der graphischen Darstellung noch im dazu gehörenden Editor. Sie werden automatisch von dem Editor in den, im Abschnitt 4.3.3 beschriebenen, Prozessartefakten eingefügt.

Das Element, welches die meisten anderen Elemente verbindet, ist die Datenverbindung. Folgende Verbindungen sind möglich: externer Nutzer zu Choreographiecontainer, externer Nutzer zu zusammengesetzter Variable, externer Nutzer zu atomarer Variable, Prozess zu Choreographiecontainer, Prozess zu zusammengesetzter Variable, Prozess Teilnehmer zu atomarer Variable, Aktivität zu Choreographiecontainer, Aktivität zu zusammengesetzter Variable und Aktivität zu atomarer Variable.

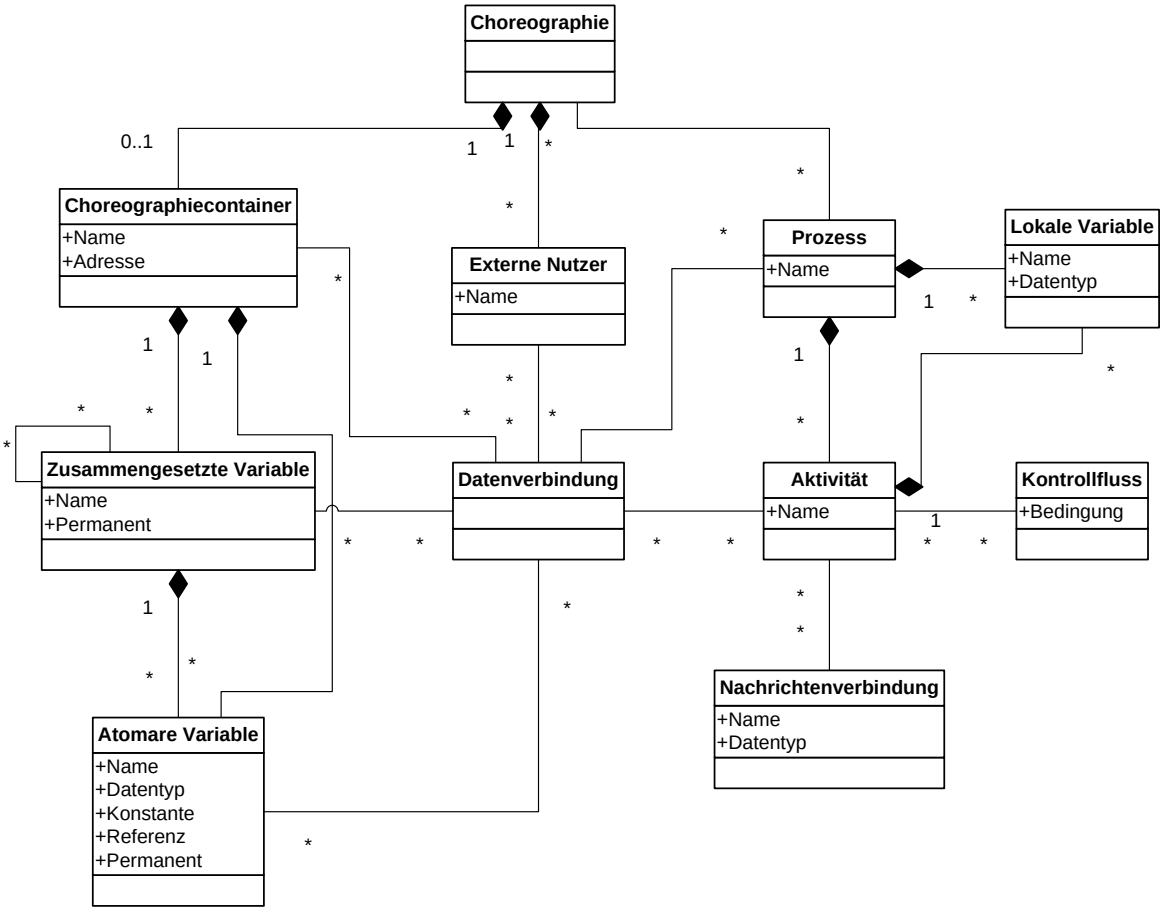


Abbildung 4.3.: Datenmodell einer Choreographie mit Choreographiecontainer

4.3.3. Übersicht und Zusammenspiel der Choreographieartefakte

Als Choreographieartefakte werden sämtliche Beschreibungen bezeichnet, die verwendet werden um eine Choreographie zu spezifizieren und darzustellen. Die für dieses Konzept wichtigsten Artefakte sind: die graphische Darstellung der Choreographie und die Modelldaten, die Prozessartefakte, der Container Descriptor und der External User Descriptor. Diese Artefakte können von Hand oder durch einen in 4.9 beschriebenen Editor zunächst modelliert und dann automatisch generiert werden.

Abbildung 4.4 zeigt den Zusammenhang zwischen dem Datenmodell und den Choreographieartefakten. Die rot umrandeten Elemente sind solche, die für die Erstellung des Container Descriptors benötigt werden. Für dieses Artefakt werden der Choreographiecontainer, die Choreographievariablen in Form der atomaren und zusammengesetzten Variablen, die externen Nutzer und die Datenverbindungen benötigt.

Die blau umrandeten Elemente werden benötigt, um den External User Descriptor zu erstellen. Für den External User Descriptor werden die Elemente: atomare Variable, Datenverbindung und

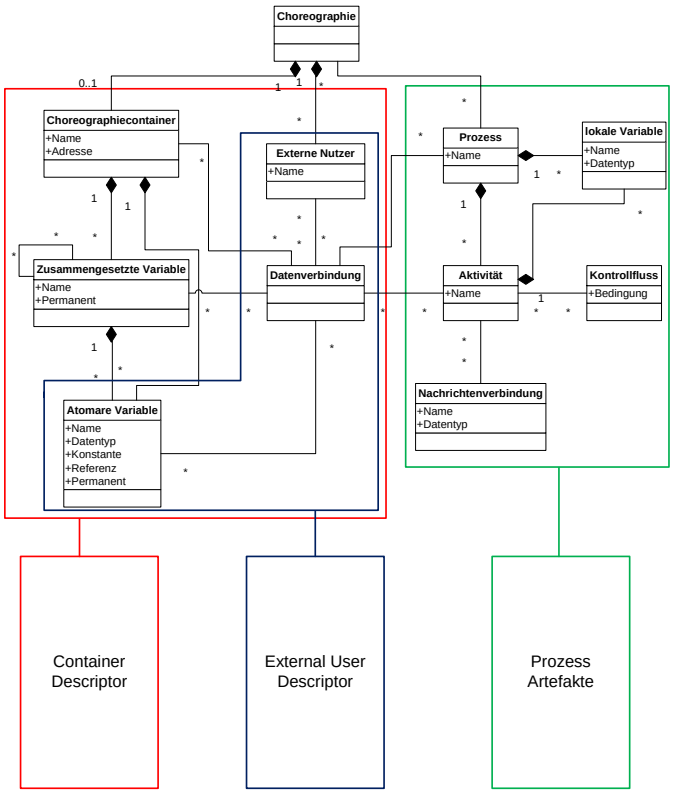


Abbildung 4.4.: Zusammenhang zwischen Datenmodell und Choreographieartefakten

externer Nutzer benötigt. Aus den Datenverbindungen zu den Elementen Choreographiecontainer und zusammengesetzte Variable wird abgeleitet auf welche atomaren Variablen der externe Nutzer Zugriff erhält. Ist er mit dem Choreographiecontainer verbunden, sind dies alle Choreographievariablen, wenn eine Verbindung zu einer zusammengesetzten Variablen besteht, erhält der externe Nutzer Zugriff auf alle Variablen innerhalb der zusammengesetzten Variablen.

Die grün umrandeten Bereiche enthalten die Elemente, die für die Erstellung der Prozessartefakte nötig sind. Diese Elemente sind: Prozesse, Aktivitäten, Variablen, Nachrichtenverbindungen und Kontrollfluss.

Wie in Abbildung 4.5 dargestellt, werden die graphische Darstellung und die Modelldaten der Choreographie von dem Editor gespeichert und geladen. Beides wird in einer Datei gespeichert, diese enthält die Informationen darüber, welche Elemente sich an welcher Stelle der Zeichenfläche befinden und welche Werte das in 4.3.2 beschriebene Datenmodell enthält. Diese Datei wird in dieser Arbeit nicht näher beschrieben, da dafür bereits vollständige Ansätze, wie Eclipse z. B. GMF [Gro09] existieren.

Aus der graphischen Darstellung und den Modelldaten werden außerdem abstrakte Prozessartefakte erzeugt. Bei abstrakten Prozessartefakten handelt es sich um Dateien, die in einer Choreographieumgebung ausgeführt werden können, da diese zwar syntaktisch korrekt sind, jedoch nur kommu-

nikationsrelevante Aktivitäten enthalten. Diese können anschließend entweder manuell oder mit entsprechenden Werkzeugen in ausführbare Prozessartefakte überführt werden. Das Erstellen und Umwandeln von Prozessartefakten sind nicht Teil dieser Arbeit.

Der Choreographie Descriptor, welcher in 4.4.2 näher beschrieben wird, ist ein weiteres durch den Editor erzeugtes Artefakt. In diesem Artefakt wird der Aufbau des Choreographiecontainers mit allen Choreographievariablen und deren Optionen festgehalten. Er dient als Konfigurationsdatei einer erweiterten Choreographieumgebung, welche in der Lage sein muss, diese Datei zu lesen und deren Semantik zu verstehen. Durch diese Konfiguration ist es der Umgebung möglich, die angegebenen Daten auf einem eigenen Datenspeicher zu speichern und bei Bedarf auch wieder zu laden. Der Datenspeicher befindet sich auf einem eigenen Datenserver, welcher abhängig von der Implementierung der Choreographieumgebung, z. B. ein REST-Server [Fie00] oder eine Datenbank sein kann.

Falls externe Nutzer für die Choreographie zugelassen werden, wird der, in 4.4.3 näher beschriebene, External User Descriptor erstellt. Dieser enthält die Information welche Rollen auf welche Choreographievariablen in der Choreographie zugreifen dürfen und kann somit als Konfiguration, oder als Ausgangspunkt für eine komplexere Konfigurationsdatei, für den Zugriff auf den Datenserver verwendet werden.

4.4. Container Descriptor und External User Descriptor

In diesem Abschnitt wird zunächst ein Beispiel einer Choreographie mit mehreren externen Nutzern und Prozessen gegeben. Anhand dieses Beispiels wird der Aufbau eines Container Descriptors und eines External User Descriptors erläutert.

4.4.1. Beispiel

Das, in Abbildung 4.6, gezeigte Beispiel hat drei externe Teilenehmer. Diese drei Teilnehmer sind: *Administrator*, *Forscher* und *Interessenten*. Die Rolle *Administrator* kann den Inhalt aller Choreographievariablen im Choreographiecontainer lesen und auf die Variable *Konfiguration* schreiben. Die Rolle *Forscher* kann Daten aus der zusammengesetzten Choreographievariablen *Zwischenergebnis Gesamt* und der atomaren Variablen *Endergebnis* lesen. Die letzte Rolle, die des *Interessenten*, kann nur auf die Variable *Endergebnis* lesend zugreifen.

Prozess 1 benötigt den Inhalt der Variablen Konfiguration und kann deshalb lesend auf diese zugreifen. Nachdem *Prozess 1* gestartet wurde, werden mehrere komplexe Berechnungen durchgeführt, und deren Ergebnisse von der Aktivität *C1* in *Teil B* und *E1* in *Teil C* der zusammengesetzten Variablen *Zwischenergebnis Prozess 1* geschrieben. Die zusammengesetzte Variable *Zwischenergebnis Prozess 1* ist selbst ein Teil der zusammengesetzten Variablen *Zwischenergebnis Gesamt*. *Prozess 2* kann unabhängig von *Prozess 1* gestartet werden. Die letzte Aktivität in *Prozess 2*, *D2* schreibt das Ergebnis des Prozesses in *Teil A* der Variablen *Zwischenergebnis Gesamt*. Für den Zugriff auf die Variable *Zwischenergebnis Gesamt* wurde nur definiert, dass *Prozess 3* Zugriff auf diese hat und nicht genau welche Aktivität. Aus

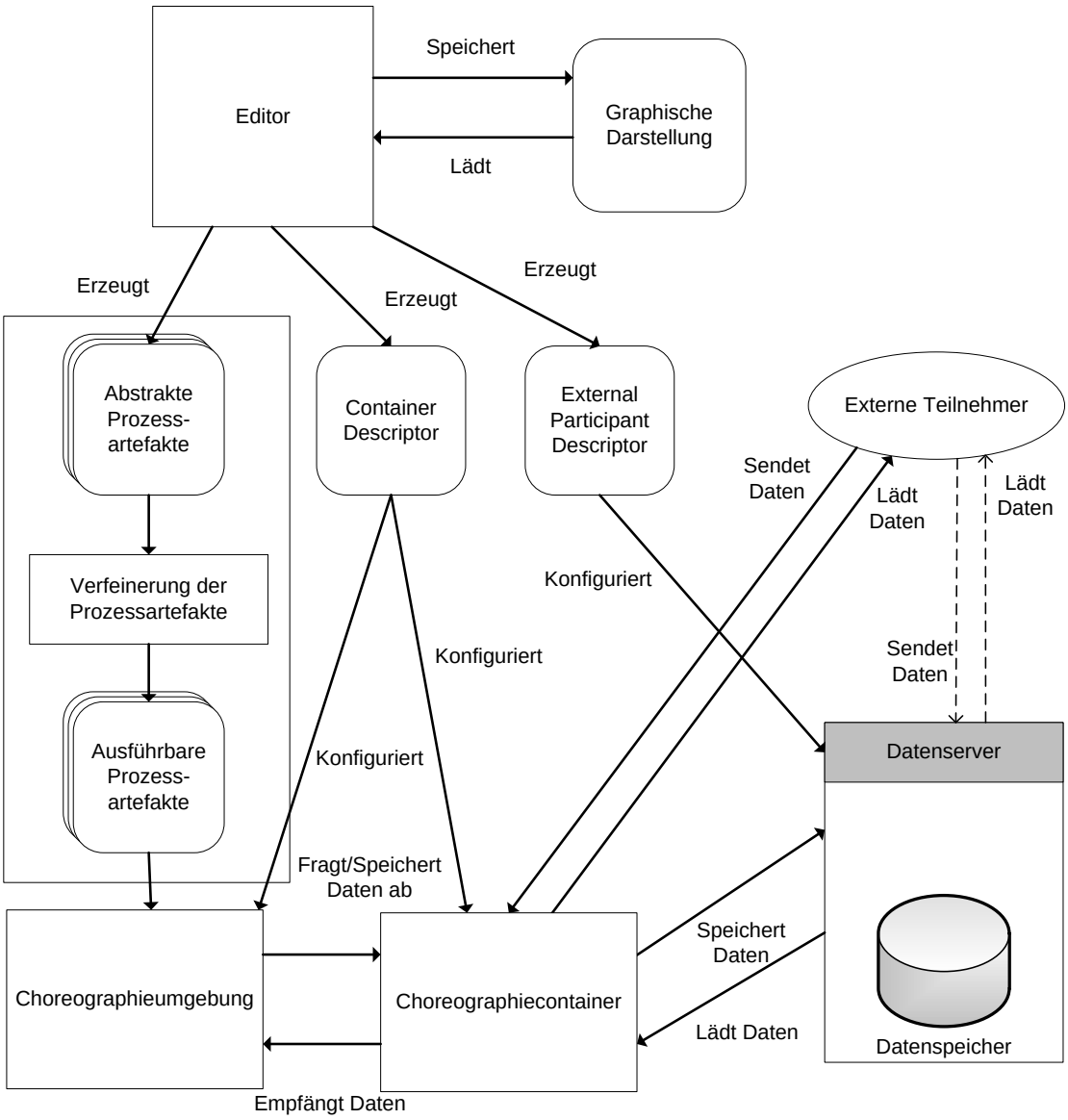


Abbildung 4.5.: Zusammenspiel der Prozessartefakte

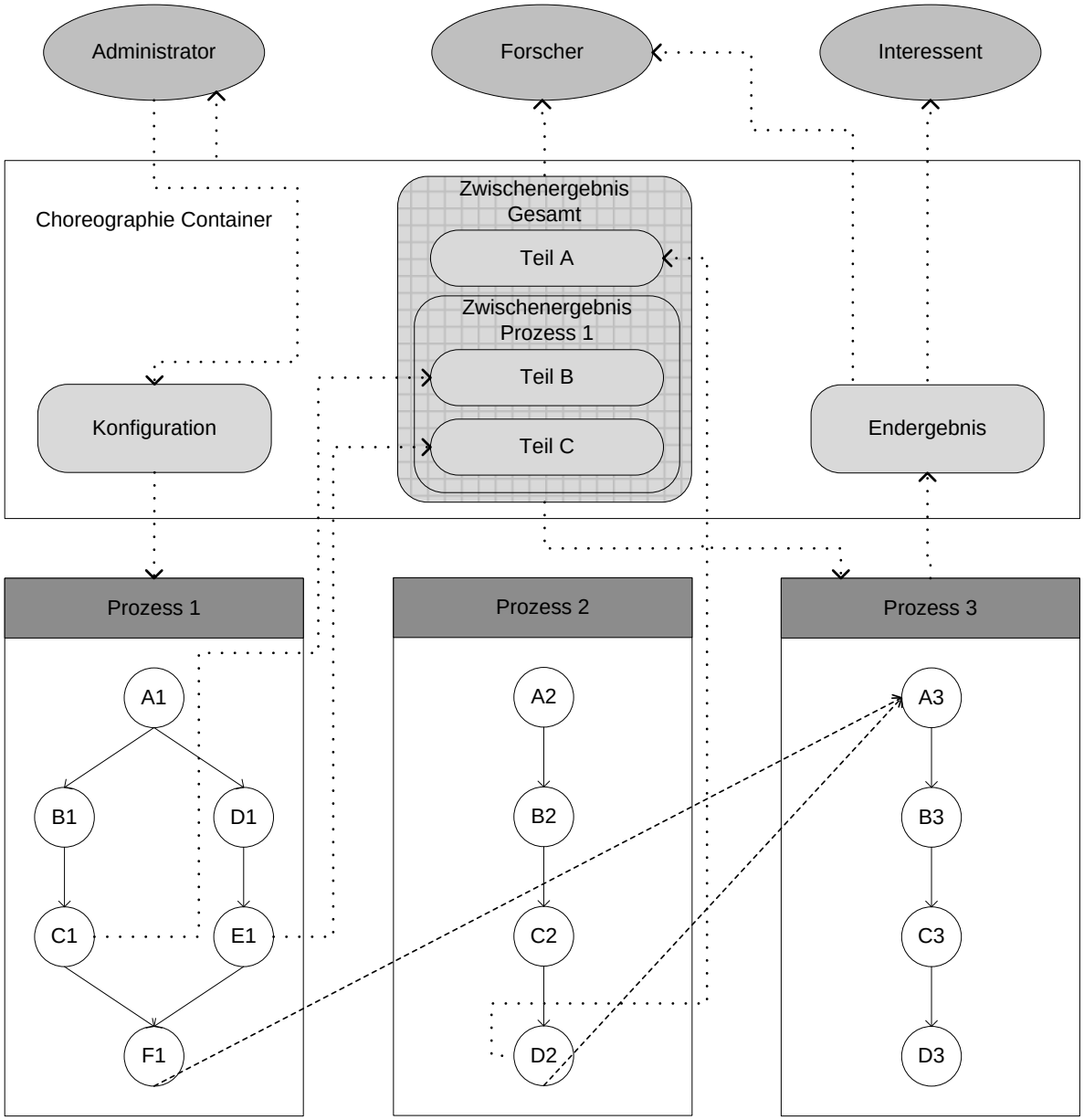


Abbildung 4.6.: Beispiel für die Verwendung eines Choreographiecontainers

diesem Grund kann *Prozess 3* erst gestartet werden wenn alle atomaren Variablen von *Zwischenergebnis Gesamt* beschrieben wurden.

4.4.2. Container Descriptor

Der Container Descriptor ist ein Choreographieartefakt, welches zusätzlich zu bestehenden Artefakten wie z. B. der Beschreibung eines Workflows mittels BPEL Datei, erzeugt wird. Der Container Descriptor

Listing 4.1 Schema des Wurzelelements

```
<xs:element name="choreographyContainer" type="ChoreographyContainerType"/>

<xs:complexType name="ChoreographyContainerType">
      <xs:sequence>
            <xs:choice maxOccurs="unbounded">
                  <xs:element ref="atomicVariable"/>
                  <xs:element ref="complexVariable"/>
            </xs:choice>
            <xs:element ref="reader" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:NCName" use="required"/>
</xs:complexType>
```

enthält die Beschreibungen des gesamten Choreographiecontainers und der Choreographievariablen einer Choreographie. Dieses zusätzliche Artefakt kann einer entsprechend modifizierten Choreographieumgebung zur Verfügung gestellt werden, um es dieser zu ermöglichen Choreographievariablen zu erkennen. Dadurch, dass es sich um eine eigenständige Datei handelt, kann der Workflow in jeder beliebigen Choreographiesprache definiert werden. Damit Variablen und Choreographie Teilnehmer eindeutig zugeordnet werden können, muss jedes *name* Attribut einzigartig in dem Container Descriptor sein. Der vollständige Container Descriptor für das in Abbildung 4.6 dargestellte Beispiel und das vollständige Schema sind im Anhang bei A.1 bzw. A.3 zu finden.

In Anlehnung an bereits am Institut erstellte Arbeiten (z. B. [DK14]), wird der Namespace urn:IAAS:choreography:schemas:choreography:choreographycontainer:2014 verwendet. Wie Listing 4.1 zeigt, ist jeder Choreographiecontainer vom Typ *ChoreographieContainerType*. Ein Choreographiecontainer besteht aus einer beliebigen Menge von atomaren Variablen, hier *atomicVariable* und zusammengesetzten Variablen, hier *complexVariable* genannt. Zusätzlich zu den Variablen kann ein Choreographiecontainer auch beliebig viele Elemente vom Typ *readerType* enthalten. Dies ermöglicht dem angegebenen Leser Zugriff auf sämtliche Choreographievariablen im Container. Vollen Zugriff wird explizit nur Lesern eingeräumt, da die Möglichkeit auf alle Variablen schreiben zu können ein zu hohes Sicherheitsrisiko birgt. Außerdem verfügt jeder Choreographiecontainer über das Pflichtattribut *name*, welches den Namen des Choreographiecontainer enthalten muss. Der Name sollte hierbei dem Namen der Choreographie selbst entsprechen.

Listing 4.2 zeigt den äußeren Aufbau für das Wurzelelement für das in Abbildung 4.6 dargestellte Beispiel. Dem Beispiel entsprechend hat der *Administrator* vollen Lesezugriff auf die Variablen: *Konfiguration*, *Zwischenergebnis* und *Endergebnis*.

Atomare Variablen, in Listing 4.3 *atomicVariable* genannt, ist vom Typ *atomicVariableType*. Dieser Typ besteht aus einer beliebigen Menge von Lesern *reader* und Schreibern *writer*, die in beliebiger Reihenfolge auftreten können. Es ist zwar möglich mehr als einen Schreiber zu verwenden, dies widerspricht jedoch der Definition in 4.2.1 und wird nur als Möglichkeit beibehalten, für den Fall, dass die Definition sich ändert und mehrere Schreiber erlaubt werden. Jede atomare Variable hat die Pflichtattribute *name* und *dataType*. Das Attribut *name* ist vom Typ NCName und gibt entsprechend den Namen der Variablen an. Das Attribut *dataType* ist vom Typ String und gibt den Datentyp der

Listing 4.2 Aufbau des Wurzelelements

```
<choreographyContainer
  xmlns="urn:IAAS:choreography:schemas:choreography:choreographycontainer:2014"
  name="Beispiel">
       :
       :
 <reader name="Administrator"/>
</choreographyContainer>
```

Listing 4.3 Schema einer atomaren Variablen

```
<xs:element name="atomicVariable" type="atomicVariableType"/>

<xs:complexType name="atomicVariableType">
      <xs:choice maxOccurs="unbounded">
            <xs:element ref="reader"/>
            <xs:element ref="writer"/>
      </xs:choice>

      <xs:attribute name="name" type="xs:NCName" use="required"/>
      <xs:attribute name="dataType" type="xs:string" use="required"/>
      <xs:attribute name="constant" type="xs:boolean"/>
      <xs:attribute name="reference" type="xs:boolean"/>
      <xs:attribute name="permanent" type="xs:boolean"/>
</xs:complexType>
```

Variablen wieder. Dies ermöglicht es, dass der Datentyp unabhängig von einer Programmiersprache definiert werden kann. Nur den einzelnen Orchestrieungsengines müssen zur Laufzeit bekannt sein um welchen Datentyp es sich konkret handelt. Somit könnten die Datentypen völlig unabhängig von der Choreographie definiert werden und bei einer eventuellen Änderungen müssen nur Anpassungen in den entsprechenden Artefakten und nicht an der gesamten Choreographie durchgeführt werden. Die folgenden drei Attribute: *constant*, *reference* und *permanent* sind vom Typ Boolean. Das Attribut *constant* gibt an, ob es sich um eine Konstante handelt. Der Wert einer Konstante muss bereits vor der Laufzeit definiert sein und darf sich in deren Verlauf nicht mehr ändern. Das Attribut *reference* gibt an, dass es sich bei der Variablen um eine Referenz auf Daten außerhalb des Choreographieumgebung handelt und deshalb von den einzelnen Orchestrierungsengines entsprechend verwendet werden muss. Das letzte Attribut, *permanent*, gibt an, dass die Daten dieser Variablen zum Zeitpunkt des Schreibens persistent gespeichert werden müssen.

Listing 4.4 zeigt die erstellten Instanzen für die beiden Variablen *Konfiguration* und *Endergebnis*. *Konfiguration* wird von *Administrator* beschrieben und von *Prozess1* gelesen. Die Variable *Endergebnis* wird von *Prozess3* beschrieben und von *Forscher* und *Interessent* gelesen.

Eine zusammengesetzte Variable, in Listing 4.5 *complexVariable* genannt, ist vom Typ *complexVariableType*. Ein *complexVariable* Element hat die Elemente *atomicVariable* und *complexVariable*. Diese können in beliebiger Menge und Reihenfolge vorkommen. Da zusammengesetzte Variablen beliebig geschachtelt werden können, referenziert sich das Element *complexVariable* selbst. Außerdem verfügt

Listing 4.4 Darstellung der beiden atomaren Variablen aus Beispiel 4.6

```
<atomicVariable name="Konfiguration" dataType="KonfigurationTyp">
      <writer name="Administrator"/>
      <reader name="Prozess1"/>
</atomicVariable>

<atomicVariable name="Endergebnis" dataType="EndergebnisTyp" permanent="true">
      <writer name="Prozess3"/>
      <reader name="Forscher"/>
      <reader name="Interessent"/>
</atomicVariable>
```

Listing 4.5 Schema einer zusammengesetzten Variablen

```
<xs:element name="complexVariable" type="complexVariableType"/>
<xs:complexType name="complexVariableType">
      <xs:sequence>
            <xs:choice maxOccurs="unbounded">
                  <xs:element ref="atomicVariable"/>
                  <xs:element ref="complexVariable"/>
            </xs:choice>
      <xs:element ref="reader" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:NCName" use="required"/>
      <xs:attribute name="permanent" type="xs:boolean"/>
</xs:complexType>
```

jede zusammengesetzte Variable über eine beliebige Anzahl von *reader* Elementen. Jeder Leser hat vollständigen Zugriff auf sämtlichen Untervariablen der zusammengesetzten Variablen bei der er als Leser hinterlegt ist. Jede zusammengesetzte Variable hat die Attribute *name* und *permanent*, welche die selbe Bedeutung wie bei einer atomaren Variablen haben. Zusammengesetzte Variablen können selbst keine Daten enthalten. Deshalb muss das unterste Element immer eine atomare Variable sein.

Listing 4.6 zeigt den Aufbau der Variablen *ZwischenergebnisGesamt* aus Beispiel 4.6. Die zusammengesetzte Variable, hat als direktes Unterelement eine atomare Variable namens *TeilA*. Diese wird von Prozess2 beschrieben. Außerdem enthält die Variable *ZwischenergebnisGesamt* eine weitere zusammengesetzt Variable namens *ZwischenergebnissProzess*. Diese enthält zwei weitere atomare Variablen *TeilB* und *TeilC*. Die atomaren Variablen *TeilB* und *TeilC* werden von Aktivität *C1* und *E1* beschrieben. Die beiden Leser *Forscher* und *Prozess3* können sämtliche Elemente der Variablen *ZwischenergebnisGesamt* lesen.

Die, in Listing 4.7, gezeigten Elemente *reader* und *writer* sind vom Typ *readerType* bzw. *writerType*. Sie haben jeweils das Attribut *name*, welches den einzigartigen Namen des jeweiligen Teilnehmers wiedergibt. Beispiele für die Verwendung dieser Elemente wurden bereits in den vorhergehenden Listings gezeigt.

Listing 4.6 Darstellung der zusammengesetzten Variablen aus Beispiel 4.6

```
<complexVariable name="ZwischenergebnisGesamt">

      <atomicVariable name="TeilA" dataType="TeilATyp">
            <writer name="Prozess2"/>
      </atomicVariable>

      <complexVariable name="ZwischenergebnisProzess1">

            <atomicVariable name="TeilB" dataType="TeilBTyp">
                  <writer name="C1"/>
            </atomicVariable>

            <atomicVariable name="TeilC" dataType="TeilCTyp">
                  <writer name="E1"/>
            </atomicVariable>
      </complexVariable>

      <reader name="Forscher"/>
      <reader name="Prozess3"/>

</complexVariable>
```

Listing 4.7 Schema der schreibenden und lesenden Nutzer

```
<xs:element name="reader" type="readerType"/>

<xs:complexType name="readerType">
      <xs:attribute name="name" type="xs:NCName" use="required"/>
</xs:complexType>

<xs:element name="writer" type="writerType"/>

<xs:complexType name="writerType">
      <xs:attribute name="name" type="xs:NCName" use="required"/>
</xs:complexType>
```

4.4.3. External User Descriptor

Der External User Descriptor ist ein weiteres Choreographieartefakt, welches zusätzlich zu der Choreographiebeschreibung selbst und dem Container Descriptor verwendet werden kann. Dieses Artefakt beschreibt die externen Nutzer einer Choreographie und auf welche Choreographievariablen diese Zugriff erhalten. Es kann verwendet werden um daraus Policies, welche z. B. festlegen auf welche Server ein Nutzer zugreifen darf und welche nicht, für den Zugriff auf die Choreographievariablen zu erstellen. So könnte die Standardrolle in Beispiel 4.6 *Interessent* sein. Die Daten aus der Variablen *Endergebnis* könnten öffentlich zugänglich sein. Jedem der auf diese Variable zugreift würde somit automatisch die Rolle *Interessent* zugewiesen bekommen. Durch die Abfrage einer Kombination aus Nutzername und Passwort können die anderen Rollen geschützt werden bzw. nur für entsprechende

Listing 4.8 Schema des Wurzelelements des External User Descriptors

```
<xs:element name="externalUserRoles" type="externalUsersRolesType"/>

<xs:complexType name="externalUsersRolesType">
      <xs:sequence>
            <xs:element ref="role" maxOccurs="unbounded"/>
      </xs:sequence>

      <xs:attribute name="name" type="xs:NCName" use="required"/>
</xs:complexType>
```

Listing 4.9 Aufbau des Wurzelelements

```
<externalUsersRoles
  xmlns="urn:IAAS:choreography:schemas:choreography:externalUsers:2014"
  name="Beispiel">
      :
      :
</externalUsersRoles>
```

Personen zugänglich gemacht werden. Der vollständige External User Descriptor für 4.6 und das vollständige Schema sind im Anhang bei A.2 bzw. A.4 zu finden.

Wie bei dem External User Descriptor wurde der Namespace urn:IAAS:choreography:schemas: choreography:externalPaticipants:2014, des External User Descriptors, an bereits bestehende Arbeiten [DK14] angelehnt. Das in Listing 4.8 dargestellte Element *externalUserRoles*, vom Typ *externalPaticipantsRolesType*, dient als Wurzelelement. Es kann eine unbegrenzte Menge an *role* Elementen enthalten. Außerdem hat es das Attribut *name*, dieses bezieht sich auf den Namen der gesamten Choreographie und ist zwingend erforderlich.

Listing 4.9 zeigt das Wurzelelement aus Beispiel 4.6. Der Name entspricht dem gewählten Namen der Choreographie *Beispiel*, welche für die gesamte Choreographie festgelegt wurde.

Das in Listing 4.10 gezeigte Element *role*, vom Typ *roleType*, beschreibt eine Rolle in der Choreographie. Es enthält eine beliebige Menge von *readsFrom* und *writesTo* Elementen, welche in beliebiger Reihenfolge auftreten können. Jedes *readsFrom* Element ist, wie in Listing 4.11 gezeigt, vom Typ *readsFromType* und jedes *writesTo* Element vom Typ *writesToType*. Beide Elemente können beliebig viele *variable* und *complexVariable* in beliebiger Reihenfolge enthalten. Eine atomare Variable vom Typ *variableTyp* hat die Attribute *name* und *dataType*, die den Namen und den Datentyp der Variablen wiedergeben. Beide Attribute müssen verpflichtend angegeben werden. Eine zusammengesetzte Variable vom Typ *complexVariable* kann beliebig viele *variable* Elemente enthalten. Eine Schachtelung ist hier nicht notwendig, da alle atomaren Variablen die sich innerhalb der zusammengesetzten Variablen befindet aufgelistet werden. Das Attribut *name* gibt den Namen der zusammengesetzten Variablen wieder. Hierbei kann der Name der obersten zusammengesetzten Variablen oder auch tiefer geschalteten zusammengesetzten Variablen entsprechen. Die Schemas der Variablen werden in Listing 4.12 dargestellt.

Listing 4.10 Schema der Rollen der Nutzer

```
<xs:element name="role" type="roleType"/>

<xs:complexType name="roleType">
      <xs:choice maxOccurs="unbounded">
            <xs:element ref="readsFrom"/>
            <xs:element ref="writesTo"/>
      </xs:choice>

      <xs:attribute name="name" type="xs:NCName" use="required"/>
</xs:complexType>
```

Listing 4.11 Schema Schreib- und Lesemöglichkeiten

```
<xs:element name="readsFrom" type="readsFromType"/>
<xs:complexType name="readsFromType">
      <xs:choice maxOccurs="unbounded">
            <xs:element ref="variable"/>
            <xs:element ref="complexVariable"/>
      </xs:choice>
</xs:complexType>

<xs:element name="writesTo" type="writesToType"/>
      <xs:sequence maxOccurs="unbounded">
            <xs:element ref="variable"/>
            <xs:element ref="complexVariable"/>
      </xs:sequence>
</xs:complexType>
```

Listing 4.12 Schema der Variablen

```
<xs:element name="variable" type="variableType"/>
<xs:complexType name="variableType">
      <xs:attribute name="name" type="xs:NCName" use="required"/>
      <xs:attribute name="dataType" type="xs:string" use="required"/>
</xs:complexType>

<xs:element name="complexVariable" type="complexVariableType"/>
<xs:complexType name="complexVariableType">
      <xs:sequence maxOccurs="unbounded">
            <xs:element ref="variable"/>
      </xs:sequence>

      <xs:attribute name="name" type="xs:NCName" use="required"/>
</xs:complexType>
```

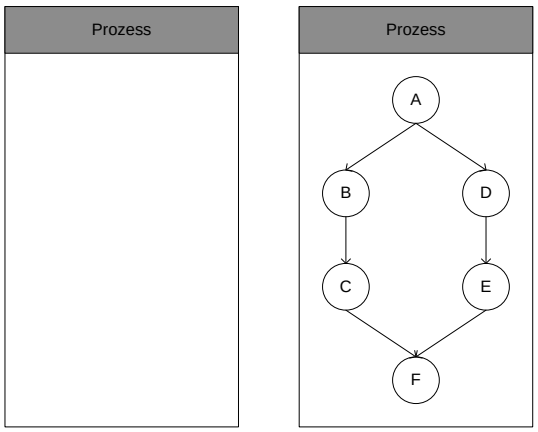
Listing 4.13 Darstellung der externen Nutzer Administrator und Forscher aus Beispiel 4.6

```
<role name="Administrator">
      <writesTo>
            <variable name="Konfiguration" dataType="KonfigurationTyp"/>
      </writesTo>

      <readsFrom>
            <complexVariable name="ChoreographieContainerBeispiel">
                  <variable name="Konfiguration" dataType="KonfigurationTyp">
                  <variable name="TeilA" dataType="TeilATyp">
                  <variable name="TeilB" dataType="TeilBTyp">
                  <variable name="TeilC" dataType="TeilCTyp">
                  <variable name="Endergebnis" dataType="EndegebnisTyp">
            </complexVariable>
      </readsFrom>
</role>

<role name="Forscher">
      <readsFrom>
            <variable name="Endergebnis" dataType="EndegebnisTyp">
            <complexVariable name="Zwischenergebnis">
                  <variable name="TeilA" dataType="TeilATyp">
                  <variable name="TeilB" dataType="TeilBTyp">
                  <variable name="TeilC" dataType="TeilCTyp">
            </complexVariable>
      </readsFrom>
</role>
```

Listing 4.13 zeigt zwei Instanzen von *role* für das Beispiel 4.6. Die Rolle *Administrator* schreibt auf die Variable *Konfiguration* und kann alle Variablen innerhalb des Choreographiecontainers *ChoreographieContainerBeispiel* lesen. Die Datentypen der Variablen *Konfiguration* und *Endergebnis* sind dieselben wie bereits im Container Descriptor. Die Rolle Administrator hat die Möglichkeit auf alle Variablen im Choreographiecontainer zuzugreifen. Dies ist ist ein Sonderfall und wird so behandelt als wäre der Container eine zusammengesetzte Variable, mit sämtlichen möglichen Variablen und deren Datentyp als Inhalt. Die Rolle Forscher hat Lesezugriff auf die atomare Variable Endergebnis und vollständigen Lesezugriff auf die zusammengesetzte Variable Zwischenergebnis und somit auch auf deren atomare Variablen.

4.5. Graphische Darstellung einer Choreographie mit Choreographiecontainer

Im folgenden wird eine Notation eingeführt, die dazu dient, die Nutzung eines Choreographiecontainers bzw. von Choreographievariablen in eine Choreographie darzustellen. Diese Notation kann verwendet werden um bereits bestehende Choregraphien abstrakter darzustellen und um Choreographiecontainer zu erweitern. Die Darstellung der Choreographie mit Choreographiecontainer kann somit als ein weiteres Choreographieartefakt betrachtet werden.



(a) Ein Blackbox Prozess

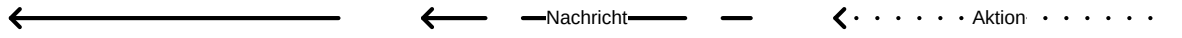
(b) Ein detaillierter Prozess

Abbildung 4.7.: Zwei Darstellungsoptionen für einen Prozess

4.5.1. Prozesse

Als Prozesse werden die Teilnehmer einer Choreographie bezeichnet. Die Prozesse können entweder als Blackbox , wie in Abbildung 4.7a dargestellt, ohne Aktivitäten oder detaillierter mit Aktivitäten dargestellt werden. Diese Darstellungsform ist besonders sinnvoll wenn Top-Down, beginnend mit der abstraktesten Form der Problemmodellierung, wie in 2.4 beschrieben, modelliert werden soll, da zu beginn noch nicht bekannt ist wie die Interaktionen innerhalb oder auch zwischen den einzelnen Teilnehmern aussehen. Es sollte jedoch bereits zu Beginn einer Modellierungsphase überlegt werden welche Teilnehmer es geben wird und bei welchen Daten es Sinn macht diese prozessübergreifend verfügbar zu machen. Beide Prozess Typen werden nachfolgend als weiße rechteckige Zeichenfläche, auf der sich ein farblich hervorgehobenes kleineres Rechteck befindet, dargestellt. Der Name des jeweiligen Prozesses wird in den farblich hervorgehobenen, oberen Teil des Symbols geschrieben. Prozesse können durch Aktivitäten präzisiert werden.

Aktivitäten sind Arbeitsschritte innerhalb einer Orchestrierung. Sie entsprechen nicht zwangsläufig einer Basisaktivität in BPEL [OAS07], es können auch zusammengefasste Abläufe wie in einem Scope in BPEL oder einer *Workunit* in WS-CDL [W3C05] sein. Durch diese detailliertere Abstufung wird es ermöglicht noch während der Ausführung eines Prozess nicht mehr benötigte Daten zu beseitigen, ähnlich der *Garbage Collection* in Java [Ora]. Aktivitäten werden durch einen Kreis innerhalb eines Prozesses dargestellt. Der Kreis enthält entweder einen Namen oder eine Identifikationsnummer durch die eine Aktivität bei der Modellierung der Orchestrierung mit dem entsprechenden Konstrukt der zugrunde liegenden Choreographiesprache assoziiert werden kann. Aktivitäten werden mit einer durchgezogenen Linie wie in 4.5.2 beschrieben verbunden.



(a) Kontrollfluss zwischen 2 zwei Aktivitäten

(b) Nachrichtenaustausch zwischen Prozessen

(c) Daten Verbindung zwischen Choreographiecontainer und Prozessen

Abbildung 4.8.: Darstellung der verschiedenen Verbindungsmöglichkeiten bei der Darstellung einer Choreographie mit Choreographiecontainer

4.5.2. Verbindungen

Es gibt bei dieser Darstellung drei Arten von Verbindungen: die Verbindung zwischen Aktivitäten (Kontrollfluss), dem Nachrichtenaustausch zwischen Prozessen und dem Datenaustausch zwischen dem Choreographiecontainer und Prozessen. Alle drei werden durch Pfeile mit offener Spitze dargestellt. Das Objekt von dem der Pfeil ausgeht, sendet bzw. schreibt Daten während das Objekt, auf das der Pfeil zeigt empfängt bzw. Daten liest.

Der Kontrollfluss zwischen Aktivitäten wird durch einen Pfeil mit durchgehender Linie dargestellt. Mit diesem Pfeil können nur Aktivitäten innerhalb des selben Prozesses verbunden werden. Es ist somit nicht möglich eine solche Verbindung außerhalb eines Prozesses zu zeichnen. Diese Verbindung dient dazu, dem Modellierer eine bessere Übersicht der Abhängigkeiten der Aktivitäten zu ermöglichen.

Der Nachrichtenaustausch zwischen zwei Prozessen wird durch einen Pfeil mit gestrichelter Linie dargestellt. Der Pfeil kann den Namen der Nachricht enthalten. Mit diesem Pfeil können zwei Aktivitäten verbunden werden, die sich nicht im gleichen Prozess finden. Dieser Pfeil stellt das senden bzw. empfangen einer Nachricht dar. Er wird verwendet um zu zeigen, dass Daten nur einmal ausgetauscht werden. Wenn z. B. ein Prozess ein Ergebnis erzeugt, welches nur für einen beliebigen Prozess interessant ist, ist es nicht notwendig im Choreographiecontainer eine extra Variable zu erstellen und damit Speicher zu belegen. Außerdem kann der Verbinder genutzt werden um den Austausch privater Daten, welche möglicherweise nicht für alle Teilnehmer sichtbar sein sollten, zu symbolisieren.

Der Austausch von Daten zwischen dem Choreographiecontainer bzw. dessen Choreographievariablen und den Prozessen oder externen Nutzern wird durch einen Pfeil mit gepunkteter Linie dargestellt. Die Linie kann den Namen einer Aktion, wie Lesen oder Schreiben, enthalten. Dieser Pfeil kann Aktivitäten oder Prozesse mit Choreographievariablen verbinden. Er steht für das lesende bzw. schreibende Zugreifen auf eine Choreographievariable. Falls eine Choreographievariable mit einem Prozess verbunden ist, kann jede Aktivität innerhalb dieses Prozesses auf die Daten in der Variablen zugreifen. Wenn eine Aktivität mit einer Choreographievariablen verbunden wird ermöglicht dies eine feinere Abstufung der Zugriffsmöglichkeit auf die Variablen.

4.5.3. Choreographiecontainer und Variablen

Choreographiecontainer werden als weißes Rechteck mit einem Namen darin, wie in Abbildung 4.9 gezeigt, dargestellt. Der Container dient als Zeichenfläche für Choreographievariablen. Der Choreographiecontainer kann überall auf der Zeichenfläche platziert werden, die Stelle sollte jedoch so gewählt werden, dass es ohne Überkreuzungen möglich sein sollte Verbindungen zwischen ihm und

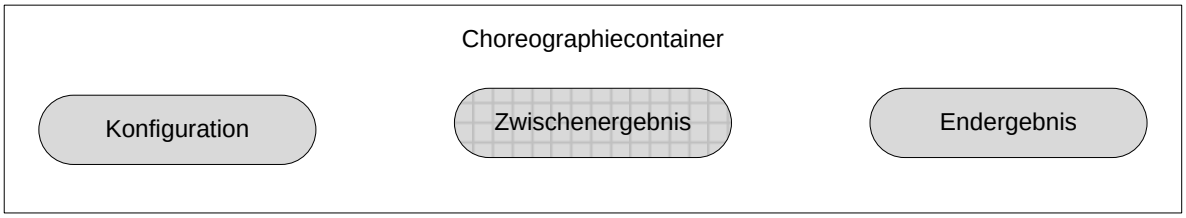


Abbildung 4.9.: Visualisierung einer Choreographiecontainer mit Choreographievariablen

Prozessen zu zeichnen. Wenn der Choreographiecontainer oberhalb oder unterhalb der Prozesse gezeichnet wird, besteht zusätzlich eine visuelle Abgrenzung zu eventuellen externen Nutzern. Es darf nur ein Choreographiecontainer pro Choreographie definiert werden. Dies dient dazu, dass der Modellierer sich keine Gedanken darüber machen muss, in welchen Container er die Choreographievariablen hinzufügen muss. Jeder Teilnehmer ist potentiell in der Lage auf den Choreographiecontainer zuzugreifen.

Es gibt zwei Arten von Choreographievariablen, die atomaren Choreographievariablen und die zusammengesetzten Choreographievariablen. Beide Arten werden als Rechtecke mit abgerundeten Ecken und dem jeweiligen Namen darin dargestellt und unterscheiden sich durch das Hintergrundmuster. Zusammengesetzte Choreographievariablen wie z. B. Zwischenergebnis in Abbildung 4.9 haben ein quadratisches Hintergrundmuster, während atomare Choreographievariablen, wie Konfiguration oder Endergebnis, ohne Hintergrundmuster dargestellt werden. Beide Arten von Variablen müssen eine andere Hintergrundfarbe als der Choreographiecontainer haben um leichter erkennbar zu sein und dürfen nur innerhalb des Choreographiecontainers platziert werden.

Atomare Choreographievariablen enthalten beliebige Daten. Sie können sowohl simple Datentypen, wie z. B. eine Ganzzahl enthalten, als auch komplexere Daten wie ein Array aus Ganzzahlen. Wie bereits in der Definition 4.2.2 erläutert, darf nur ein Prozess schreibend auf eine atomare Variable zugreifen. Lesend dürfen beliebig viele Prozesse auf eine atomare Choreographievariable zugreifen.

Zusammengesetzte Choreographievariablen dürfen ebenfalls beliebige Datentypen enthalten und können beliebig tief geschachtelt werden. Die Abgrenzung zu einer atomaren Choreographievariablen liegt darin, dass mehrere Prozesse schreibend, auf sie zugreifen dürfen. Dies scheint auf den ersten Blick der Definition einer Choreographievariablen zu widersprechen. Dieser Widerspruch wird dadurch aufgelöst, dass jeder Prozess nur auf einen einzelnen Teil der Variablen schreiben darf. Wenn eine Verbindung zu einer zusammengesetzten Variablen gezogen wird, hat der verbundene Prozess oder die verbundene Aktivität Zugriff auf alle zusammengesetzten und atomaren Variablen innerhalb der zusammengesetzten Variablen.

Ein Beispiel für dieses Konzept wird in Abbildung 4.10 dargestellt. In diesem Beispiel gibt es drei Prozesse: den *LeihwagenProzess*, den *FlugProzess* und den *HotelProzess*. Jeder dieser Prozesse sucht das jeweils beste Angebot.

Die zusammengesetzte Variable *Reise* besteht aus der atomaren Variablen *Leihwagen* und der zusammengesetzten Variablen *ReisebüroInfo*. Die zusammengesetzte Variable *ReisebüroInfo* besteht aus den atomaren Variablen *Flug* und *Hotel*.

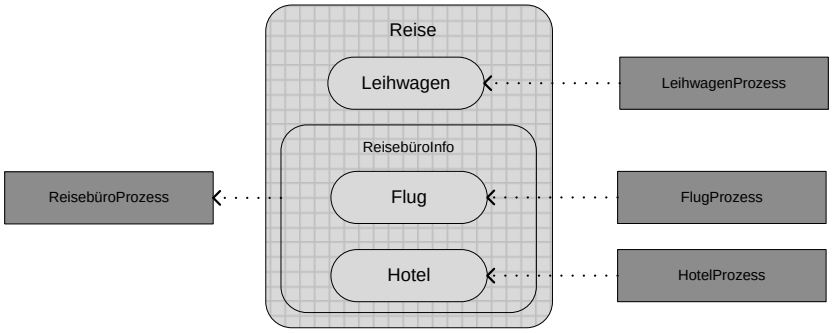


Abbildung 4.10.: Choreographievariablen mit detaillierten Schreibvorgängen

Die Prozesse *Leihwagenprozess*, *FlugProzess* und *HotelProzess* schreiben auf die entsprechenden Variablen. Da jeder der Prozesse eine direkte Verbindung aufweist, können die Prozesse nur auf die jeweilige Variable zugreifen. Der *ReisebüroProzess* hat eine Verbindung zu der zusammengesetzten Variablen *ReisebüroInfo* und kann somit auf alle Variablen zugreifen, die sich innerhalb dieser zusammengesetzten Variablen befinden.

4.5.4. Externe Nutzer

Externe Nutzer werden wie in Abbildung 4.11 gezeigt, als Ellipse dargestellt. Die Rolle, die der Nutzer in der Choreographie spielt, wie z. B. Administrator oder Nutzer muss angegeben werden. Die Rolle kann dabei auch für eine ganze Gruppe von Nutzern stehen. So könnten hinter der Rolle Nutzer zum Beispiel in Wirklichkeit 100 Nutzer sein, die alle in ihrer Rolle Nutzer den durch Verbindungen definierten Zugriff auf bestimmte Variablen haben. Externe Nutzer können sowohl schreibend als auch lesend auf den Choreographiecontainer bzw. auf die Choreographievariablen zugreifen.

Rolle1 kann in diesem Beispiel nur schreibend auf die Variable *Konfiguration* zugreifen. Es sind wie im Fall von *Rolle2* auch mehrfach Verbindungen möglich. Falls ein externer Nutzer nur mit dem Choreographiecontainer und nicht mit einer konkreten Choreographievariablen verbunden wird, bedeutet dies, dass der Nutzer, je nach Pfeilrichtung des Verbinders, vollständigen lesenden oder schreibenden Zugriff auf sämtliche Choreographievariablen der Choreographie hat. In Abbildung 4.11 ist es *Rolle3* möglich auf *Konfiguration*, *Zwischenergebnis* und *Endergebnis* lesend zuzugreifen. Bei lesenden Zugriffen können in diesem Fall, falls die Variablen private Daten enthalten, Datenschutzprobleme auftreten. Wenn wie im Fall von *Rolle4* vollständiger schreibender Zugriff auf sämtliche Choreographievariablen besteht, könnten massive Störungen im Ablauf oder falsche Ergebnisse erzeugt werden. *Rolle4* könnte z. B. ein falsches Zwischenergebnis erzeugen bevor die Prozesse überhaupt an dem Punkt angekommen sind, an dem diese geschrieben werden. Diese würden entweder zum Anhalten der Choreographie oder dem Verwenden von falschen Zwischenergebnissen führen. *Rolle4* könnte auch direkt die Variable *Endergebnis* mit einem Wert füllen und somit das publizieren eines falschen oder geschönten Ergebnisses ermöglichen. Aus diesen Gründen sollten externe Nutzer wenn möglich nur direkt mit Choreographievariablen verbunden werden und nicht mit dem Choreographiecontainer.

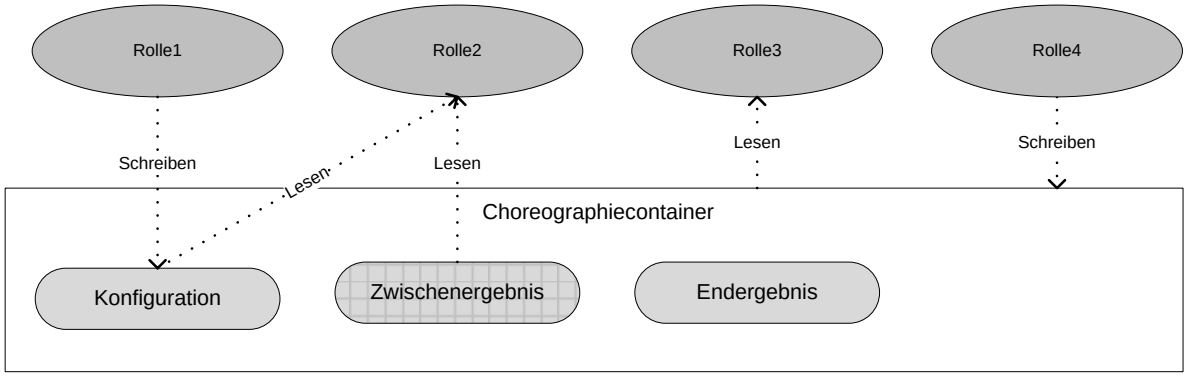


Abbildung 4.11.: Externe Nutzer mit Lese- und Schreibzugriffen

4.6. Anwendungsfälle

In diesem Abschnitt werden Anwendungsfälle beschrieben bei denen Vorteile durch die Nutzung eines Choreographiecontainers entstehen. Die Fälle lassen sich auch kombinieren, für diesen Abschnitt werden sie soweit wie möglich vereinfacht um das Konzept welches hinter ihnen steht deutlich hervor zu heben.

4.6.1. Veröffentlichung von Zwischenergebnissen

Die Grundvoraussetzung für den, in Abbildung 4.12 dargestellten Anwendungsfall ist, dass komplexe Berechnungen oder Verarbeitungsschritte in mehrere Prozesse oder Aktivitäten aufgeteilt werden können. In dem Choreographiecontainer kann eine beliebige Menge von Choreographievariablen, welche Zwischenergebnisse, z. B. aus komplexen Berechnungen oder auch Zwischenschritte bei der Verarbeitung von großen Bildermengen [JKP+04], erstellt werden. Die Prozesse und Aktivitäten stehen für einzelne Berechnungs– oder Verarbeitungsschritte. Bei diesem Anwendungsfall wird abgewogen, welche Zwischenergebnisse, über Choreographievariablen öffentlich gemacht werden sollen. Dies stellt eine Erweiterung des in [BWH08b] beschriebenen sequentiellen Musters dar, da so beliebig viele externe Nutzer die Zwischenergebnisse einsehen können. Externe Nutzer sind hier in der Rolle Interessenten vertreten. Interessenten können in diesem Beispiel nur lesend auf die Choreographievariablen zugreifen und stellen Personen dar, die an Zwischenergebnissen interessiert sind wie z. B. Wissenschaftler. Wenn größere Datenmengen entstehen, kann dieser Fall auch mit dem Fall große Datenmengen in 4.6.5 kombiniert werden. Der Vorteil der Nutzung eines Choreographiecontainers liegt bei diesem Fall darin, dass bereits bei der ersten Konzeption erwogen werden kann, welche Rollen außerhalb der Choreographie Zugriff auf welche Daten erhalten. So ist es möglich zu definieren, dass nur wenige privilegierte Nutzer, z. B. Wissenschaftler Zugriff auf alle Choreographievariablen haben, während für normale Interessenten nur der Zugriff auf bestimmte Daten, wie Meilensteine, haben.

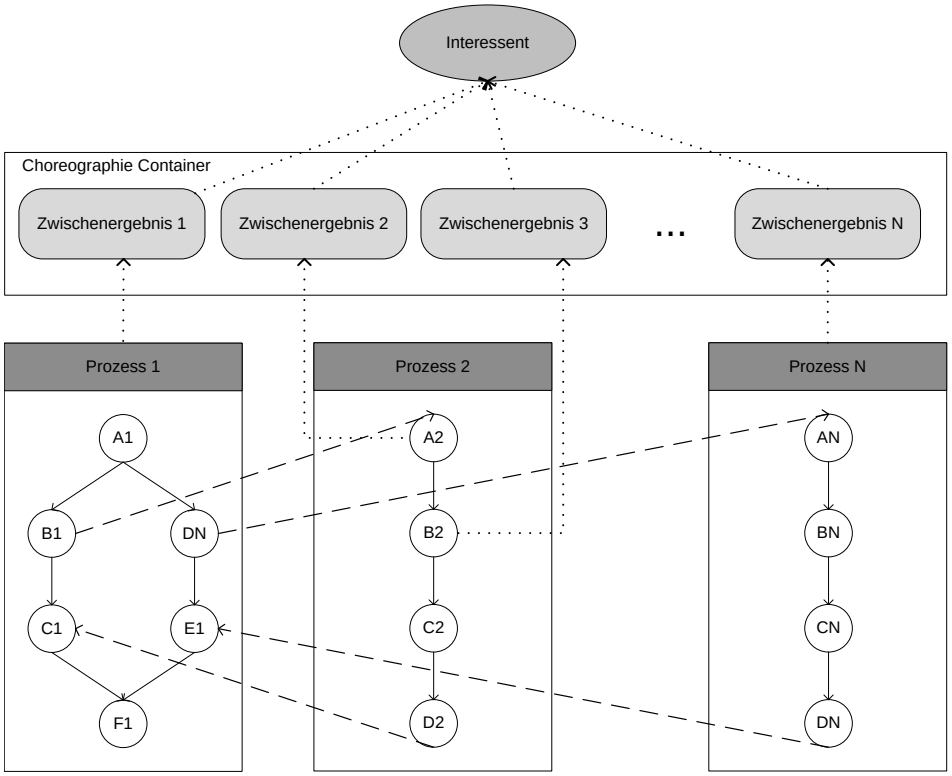


Abbildung 4.12.: Beispiel für eine Veröffentlichung von Zwischenergebnissen

4.6.2. Wechselnde Parameter

Bei diesem Anwendungsfall werden Daten außerhalb der Choreographieumgebung erzeugt und über Choreographievariablen diesem zur Verfügung gestellt. In dem, in Abbildung 4.13 dargestellten, Beispiel wird eine Simulation mit verschiedenen, häufig wechselnden, Parametern, gestartet. Die Parameter werden in einer Choreographievariablen namens Parameter durch die Rolle Forscher gespeichert. Die Choreographie besteht aus mehreren Prozessen die gleichzeitig gestartet und Zugriff auf die Parameter benötigen. Bei *Prozess 1* benötigt die Aktivität *A1* Zugriff und bei *Prozess 2* und *3* die gesamten Prozesse. Alle drei führen Berechnungen durch und *Prozess 2* und *3* senden mit ihrer letzten Aktivität ihre Ergebnisse in Form einer Nachricht an *Prozess 1*, der die Ergebnisse verwendet um ein Endergebnis zu erstellen.

Dieses Vorgehen wird in [SK13], als *Parameterstudie* bezeichnet. Die entsprechenden Parameter werden dabei durch einen Dialog des Editors abgefragt und in Form einer Nachricht an die Choreographie gesendet. Mit einem Choreographiecontainer bietet sich die Möglichkeit bereits mehrere dieser Parametersätze vorzubereiten.

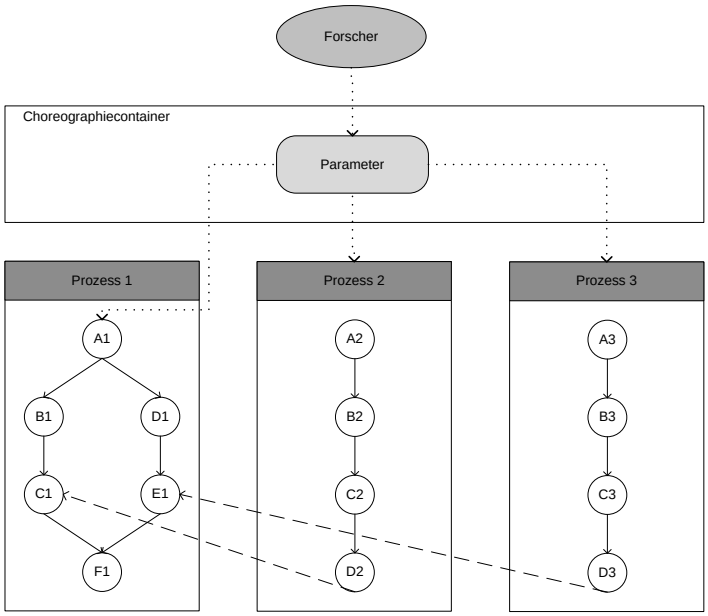


Abbildung 4.13.: Beispiel für wechselnde Parameter

4.6.3. Zusammengefasste Ergebnisse

Dieser Anwendungsfall basiert auf der Annahme, dass es einen Prozess gibt, der Daten von mehreren anderen Prozessen benötigt. Dieser Fall entspricht, aus Sicht des Choreographiecontainers bzw. des *Prozesses X*, dem in [BWH08b] beschriebenen Muster, Fan-In. Abbildung 4.14 zeigt ein Beispiel für diesen Fall. Die *Prozesse 1–N*, welche aus Gründen der Übersicht gestapelt dargestellt werden, starten gleichzeitig, stellen Berechnungen an und speichern diese in der zusammengesetzten Choreographievariablen *Zwischenergebnisse*. Außerdem senden die letzten Aktivitäten der Prozesse eine Nachricht an *Prozess X* um zu signalisieren, dass diese fertig sind. Dieser Prozess nimmt alle Nachrichten entgegen und sendet nachdem die *Prozesse 1–N* ihre Beendigung signalisiert haben eine entsprechende Nachricht an *Prozess Y* der die Daten aus der Choreographievariablen liest und die abschließenden Berechnungen durchführt.

4.6.4. Viele Leser

Dieser Anwendungsfall entspricht dem umgekehrten Fall von zusammengefassten Ergebnissen und entspricht somit, dem in [BWH08b] beschriebenen Muster Fan-Out. Wie in Abbildung 4.15 dargestellt, gibt es bei diesem einen *Prozess 1* von dessen Ergebnis alle folgenden Prozesse *2–N* abhängig sind. Deshalb sendet *Prozess 1* mit seiner letzten Aktivität Startnachrichten an die *Prozesse 2–N*. Diese erhalten die benötigten Daten aus der Choreographievariablen *Zwischenergebnis*.

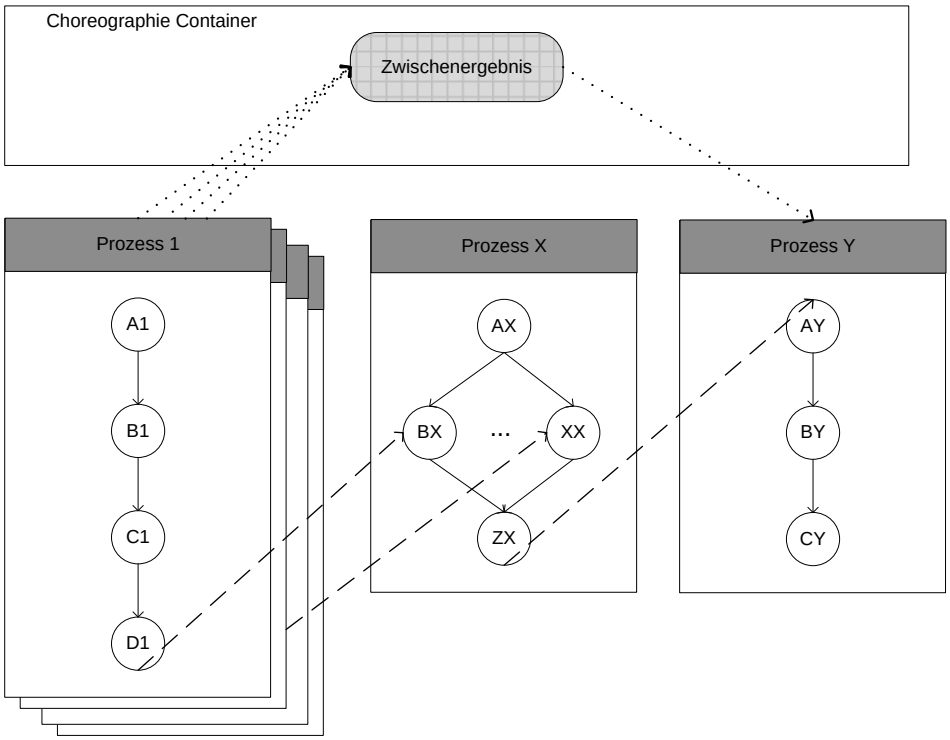


Abbildung 4.14.: Beispiel für ein zusammengefasstes Ergebnis

4.6.5. Große Datenmengen

Bei diesem Anwendungsfall sollen große Datenmengen zwischen verschiedenen Prozessen ausgetauscht werden. Dies können sowohl atomare Choreographievariablen, wie z. B. binäre Daten in Form eines Bildes, oder zusammengesetzte Choreographievariablen, wie eine große Serie von Messdaten, sein. Das Rechteck in Abbildung 4.16 mit der Beschriftung große Datei steht für eine große Datei die an einem beliebigen Ort, innerhalb des Choreographieumgebung oder völlig außerhalb des Systems, gespeichert werden kann. Prozess 1 erstellt als Ergebnis eine große Datei. Die Datei wird auf einem Server, auf den mittels einer REST–API zugegriffen werden kann, außerhalb des Choreographieumgebung gespeichert. Die Referenz auf die Datei, in diesem Fall eine URL, wird in der Choreographievariablen Speicherort abgespeichert. Prozess 2 kann, sobald die entsprechende Datei von einem Webservice benötigt wird, die Referenz direkt an die benötigten Webservices, wie z. B. in [BWH08b] beschrieben, gesendet werden. Der Webservice kann in diesem Fall dank der Referenz die Datei von ihrem Speicherort laden [WGSL09], ohne das sie durch die Orchestrierungsengines geschickt werden müssen. Die Möglichkeiten Daten zu speichern und auszutauschen, werden im Abschnitt 4.7.2 näher beschrieben.

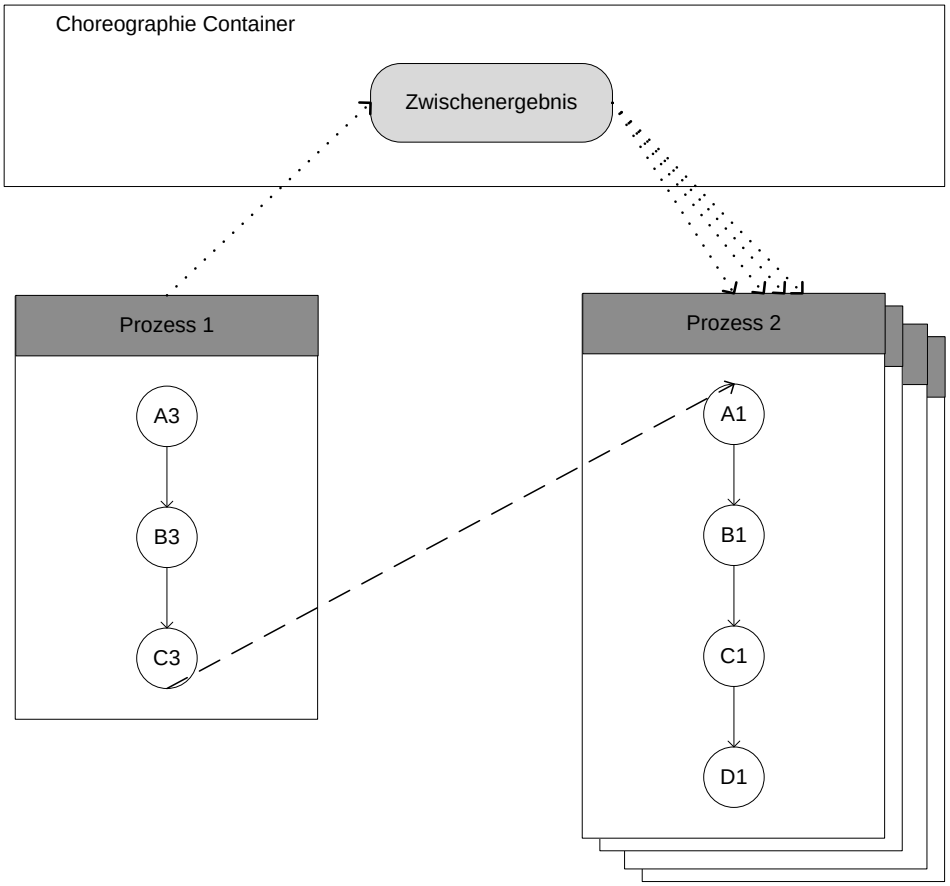


Abbildung 4.15.: Beispiel für einen schreibenden Prozess und viele Lesende

4.6.6. Konstanten

Für diesen Fall werden bestimmte Konstanten für die gesamte Choreographie definiert. Im Beispiel von Abbildung 4.17 sind dies die Konstanten *Pi* und *Grenzwert*. *Pi* steht stellvertretend für Mathematische Konstanten, welche eventuell von den zugrunde liegenden Systemen unterschiedlich präzise definiert werden könnten. *Grenzwert* steht für einen Schwellwert ab wann eine Abweichung zu groß wird. Der Vorteil für diesen Fall liegt in der Vereinfachung der Prozessartefakte, da die entsprechenden Variablen nicht für jeden Prozess einzeln deklariert werden müssen. Außerdem müssen im Falle einer Änderung eines Grenzwertes, nicht alle Einträge für die Variable in jedem einzelnen Prozessartefakt verändert werden.

4.7. Entwurfsentscheidungen

Dieser Abschnitt beschäftigt sich mit Entwurfsentscheidungen, die bei der Umsetzung eines Editors und der Choreographieumgebung abgewogen werden müssen.

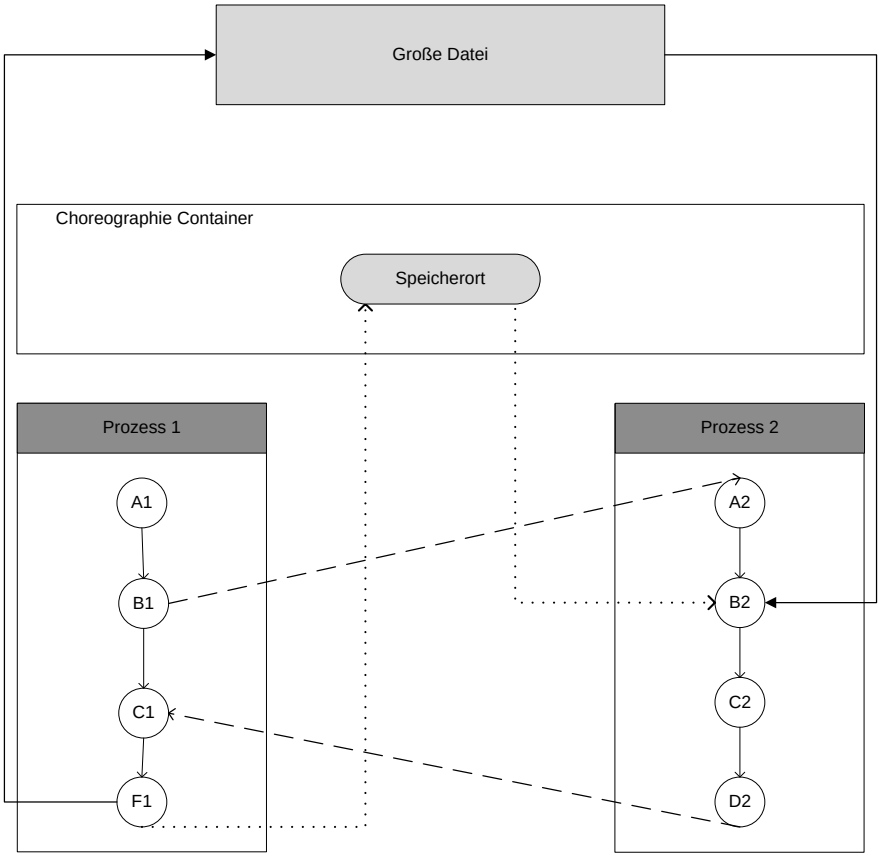


Abbildung 4.16.: Beispiel für den Umgang mit großen Datenmengen

4.7.1. Synchronisation

Es gibt drei Möglichkeiten die Synchronisation zwischen dem Choreographiecontainer und den einzelnen Teilnehmern der Choreographie zu ermöglichen: Synchronisation durch Pullen, durch Nachrichtenaustausch und eine zentrale Synchronisation durch die Choreographieumgebung.

Synchronisation durch Abfragen

Bei dem Abfragen einer Variablen werden, wie in Abbildung 4.18 dargestellt, von einer Aktivität aus einer Orchestrierung wiederholt Anfragen an den Choreographiecontainer gesendet. Der Container liefert hierbei solange negative Antworten, bis die angefragte Ressource tatsächlich für den Prozess verfügbar ist. Um diese Vorgehensweise umzusetzen ist es nötig, dass die zugrunde liegende Choreographiesprache zwei Aktivitäten zu Verfügung stellt, um auf eine externe Variable zuzugreifen: Die Möglichkeit Anfragen an den Choreographiecontainer zu senden und eine Schleife um diese Anfragen zu wiederholen. Die Anzahl von Anfragen sollten begrenzt werden, um im Fehlerfall, z. B. bei einem Absturz des Choreographiecontainers oder der Nichtverfügbarkeit der Infrastruktur, nicht endlos ausgeführt zu werden und somit Ressourcen verbrauchen. Des Weiteren müssen Zeitabstände

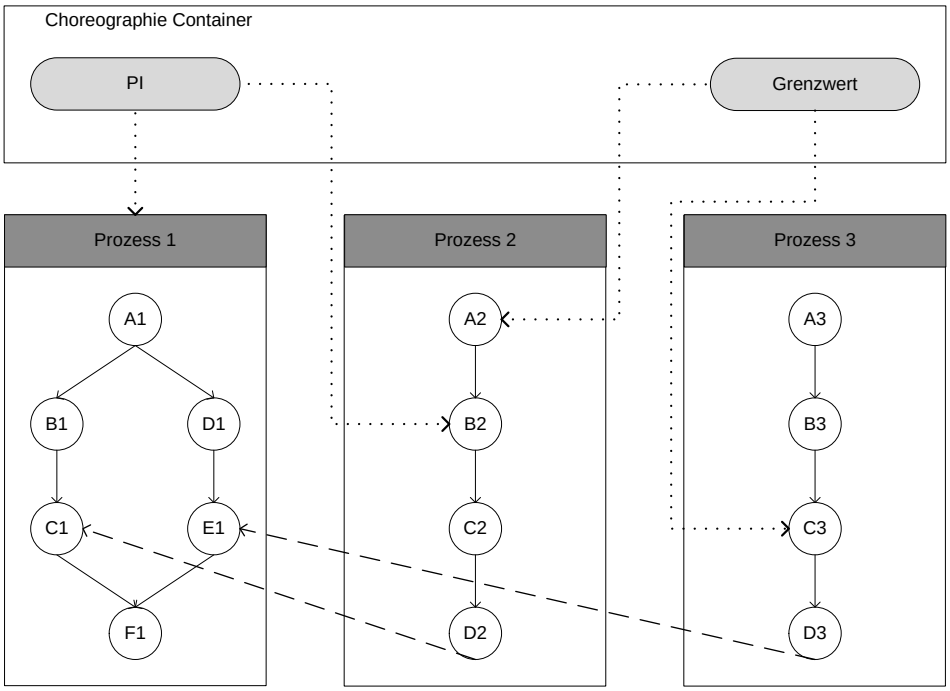


Abbildung 4.17.: Beispiel für die Verwendung von Konstanten

definiert werden um nicht zu viel Bandbreite der zugrunde liegenden Infrastruktur zu verbrauchen. Dieser Ansatz kann auf der Ausführungsebene oder auf Choreographieebene realisiert werden. Auf Choreographieebene ist dies nur möglich, falls der Choreographiecontainer in Form eines Proxy realisiert wird.

Synchronisation durch Nachrichtenaustausch

Abbildung 4.19 zeigt den Ablauf einer Synchronisierung durch Nachrichtenaustausch. Die Aktivität *D1* in *Prozess 1* erzeugt in diesem Fall Daten die anschließend in der Choreographievariablen Zwischenergebnis gespeichert werden. Im Anschluss an diese Aktivität sendet die Aktivität *E1*, welche durch die dickere Umrandung gekennzeichnet ist, eine Nachricht an *Prozess 2*. Da die Daten aus der Variablen Zwischenergebnis von der Aktivität *C2* benötigt werden, blockiert die Aktivität *B2*, die ebenfalls durch den dickeren Rand gekennzeichnet wurde, den Prozess bis die Nachricht von Aktivität *E1* eintrifft. Diese Form der Synchronisation hat den Vorteil, dass die Synchronisationsaktivitäten auf Ebene der Choreographie definiert werden. Um diese Variante des Zugriffs umzusetzen, benötigt die zugrunde liegende Chorgeographiesprache zwei Aktivitäten: Eine Sendeaktivität und eine blockierende Empfangsaktivität. Das Senden der Synchronisationsnachricht kann synchron oder asynchron erfolgen. Der Empfang der Nachricht muss blockierend sein, da sonst eine Aktivität versuchen könnte Daten aus einer Choreographievariablen zu lesen, obwohl diese nicht existiert oder leer ist. Dieser Ansatz kann, falls der Choreographiecontainer in Form eines Webservices realisiert wird, vollständig auf Choreographie Ebene umgesetzt werden.

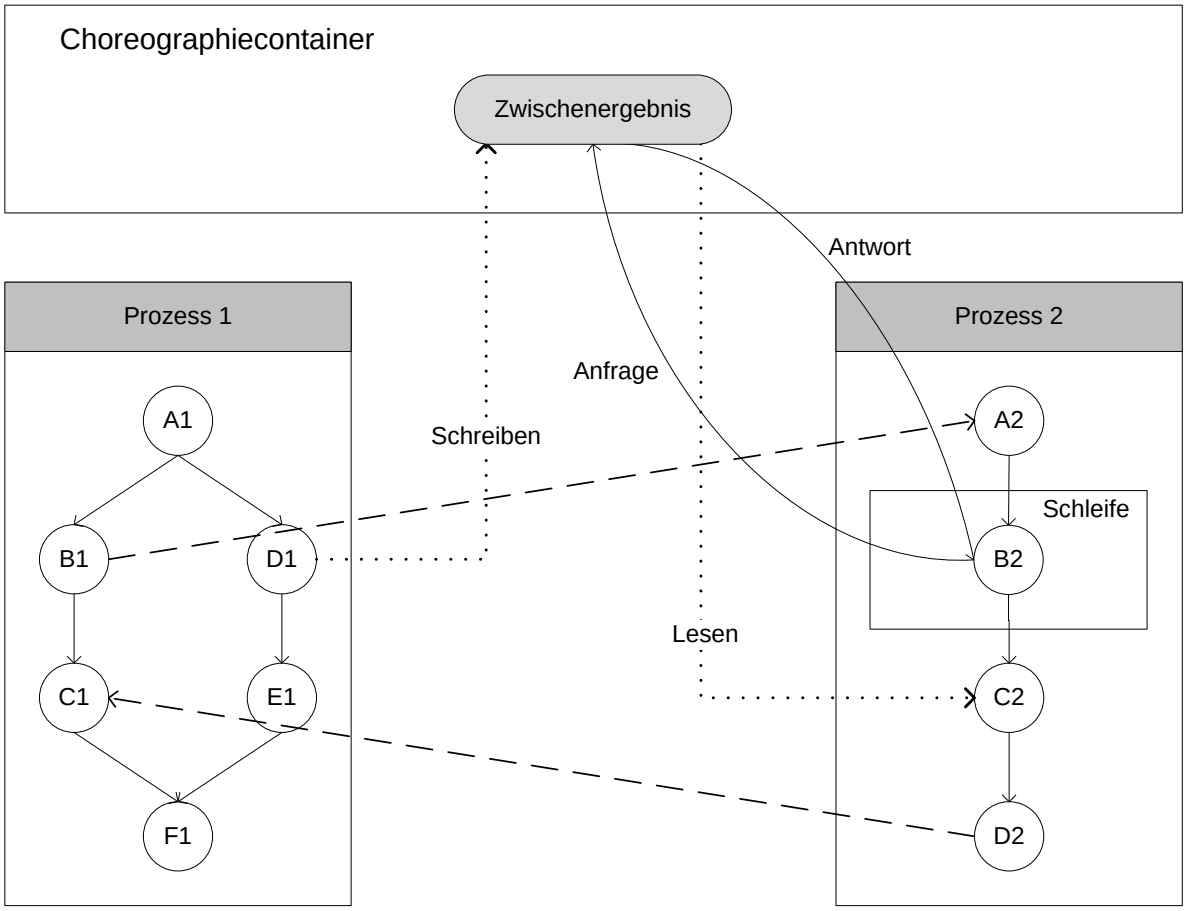


Abbildung 4.18.: Synchronisation zwischen den verschiedenen Teilen einer Choreographie durch Abfragen

Synchronisation durch die Choreographieumgebung

Bei diesem Ansatz übernimmt die jeweils ausführende Orchestrierungsengine in Verbindung mit dem Choreographiecontainer die Synchronisation, ohne dass eine explizite Synchronisation in den Prozessartefakten festgelegt werden muss. Dies wäre möglich wenn ein synchrones Protokoll für die Abfrage der Daten aus dem Choreographiecontainer verwendet wird. Die jeweilige Orchestrierungsengine wird bei diesem Ansatz durch Funktionsaufrufe blockiert. Diese Funktionsaufrufe blockieren den Programmablauf der Choreographieumgebung so lange, bis eine Antwort von dem Choreographiecontainer erfolgt. Bei diesem Ansatz wird die Orchestrierungsengine eng an den Choreographiecontainer gebunden, da eigene Funktionen für den Abruf von Daten aus einem Datenserver implementiert werden müssen. Dafür muss keine explizite Synchronisation durch den Modellierer der Choreographie erfolgen.

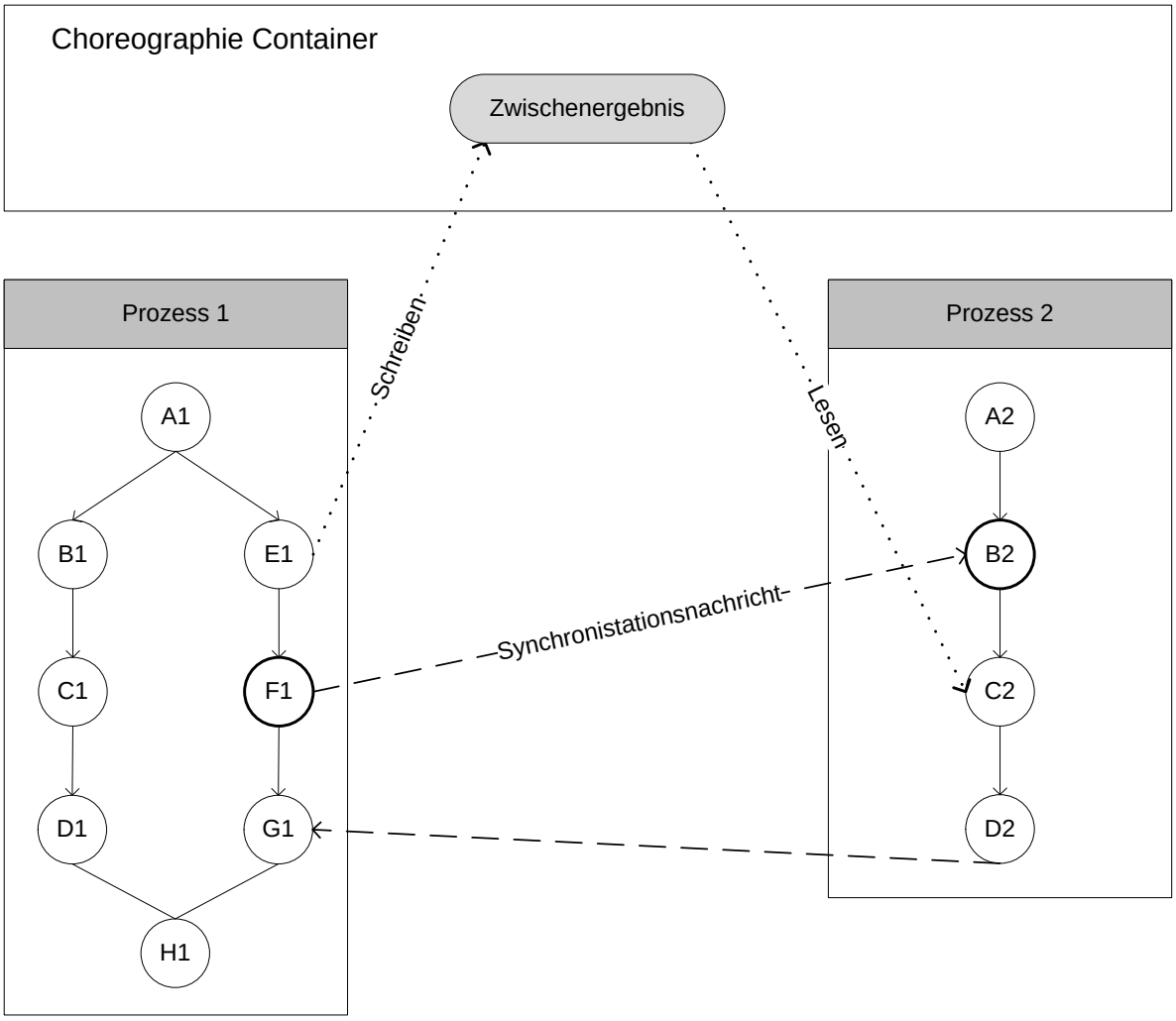


Abbildung 4.19.: Synchronisation zwischen den verschiedenen Teilen einer Choreographie durch eine Synchronisationsnachricht

4.7.2. Datenhaltung

Grundsätzlich gibt es zwei Möglichkeiten der Datenhaltung bei diesem Konzept. Die erste Möglichkeit besteht darin, dass die Daten der Choreographievariablen in der Choreographieumgebung selbst gespeichert werden. Choreographievariablen werden bei diesem Ansatz wie Variablen, oder dem Variablen entsprechenden Konstrukt der gewählten Choreographiesprache, behandelt. Der Nachteil dieses Ansatzes besteht darin, dass der Zugriff von externen Nutzern, mit sämtlichen Zugriffs- und Sicherheitsmechanismen hinzugefügt werden müsste.

Der andere Ansatz besteht darin, dass die Daten auf einem eigenen Datenserver gespeichert werden. Die Choreographieumgebung wird bei diesem Ansatz dahingehend erweitert, dass bei dem Aufruf einer Choreographievariablen, die Daten von dem Datenserver abgerufen werden können. Der

Datenserver kann z. B. in Form eines REST-Servers oder einer Datenbank realisiert werden. Der Vorteil dieses Ansatzes besteht darin, dass der Zugriff von externen Nutzern mit der Zugriffskontrolle des Datenservers realisiert werden kann. Der Datenserver kann bei diesem Ansatz getrennt von der Choreographieumgebung existieren. Je nach Art des Datenservers können auf diesem auch große Dateien wie z. B. hochauflösende Bilder gespeichert sein.

4.8. Softwarearchitektur eines Choreographiecontainers

Die Softwarearchitektur eines Choreographiecontainers folgt, wie in Abbildung 4.20 gezeigt, einer Drei-Schichten-Architektur [Mic].

Die oberste Schicht ist die *Präsentationsschicht*. Diese Schicht ist das User-Interface über welches mit dem Choreographiecontainer interagiert werden kann. Sie enthält die Komponenten *WSDL–* und *REST–Interface*. Da eine große Zahl von Webservice mit WSDL verwendet werden und Orchestrierungen Webservices verwenden, kann davon ausgegangen werden, dass die meisten Orchestrierungsengines über eine eingebaute Möglichkeit verfügen, mittels WSDL mit einem Webservice zu kommunizieren. Der Choreographiecontainer verfügt über ein WSDL–Interface, um die Kommunikation mit einer Orchestrierungsengine, mit möglichst wenig Änderungen, durchführen zu können. Außerdem verfügt die Präsentationsschicht über ein REST–Interface, um das Abfragen von Daten durch externe Teilnehmer zu ermöglichen und ein Service–Interface um die Funktionsfähigkeit des Choreographiecontainer zu überwachen. Teil der Präsentationsschicht ist die Zugriffskontrollschicht. Diese prüft ob Anfragen von externen Nutzern oder Prozessen, wie im Container Descriptor definiert, erlaubt sind und weist gegebenenfalls unberechtigte Anfragen mit einer entsprechenden Nachricht ab.

Die *Logikschicht* steuert den Ablauf und koordiniert die Funktionen des Choreographiecontainers. In dieser Schicht befindet sich die *Mapping* Komponente. Diese wandelt Anfragen an den Choreographiecontainer in Anfragen an den Datenserver um oder lädt die Daten aus dem internen Speicher des Choreographiecontainers. Außerdem enthält diese Schicht die *Parser* Funktionalitäten, die verwendet werden um die Konfigurationsdatei einzulesen. Die Konfiguration wird anschließend in der Datenhaltungsschicht gespeichert.

Die *Datenhaltungsschicht* enthält eine *interne Speicherkomponente*, in der die Konfiguration des Choreographiecontainers gespeichert wird. Diese Komponente kann in nachfolgenden Arbeiten zur Verbesserung der Abfragegeschwindigkeit um einen *Cache* erweitert werden. Die Datenhaltungsschicht enthält zusätzlich die Client-Funktionalitäten, die benötigt werden, damit der Choreographiecontainer mit dem Datenserver kommunizieren kann.

4.9. Choreographie mit Choreographiecontainer Editor

Dieser Abschnitt beschreibt einen Graphischen Editor mit dem es möglich ist, eine Choreographie mit Choreographiecontainer zu erstellen. Der Editor orientiert sich an bereits bestehenden Editoren wie dem BPEL Designer [Foud] und dem BPEL4Chor Designer [Son13]. Abbildung 4.21 zeigt die Oberfläche des Editors. Am oberen Rand befindet sich eine Toolbar die gängige Editor Elemente,

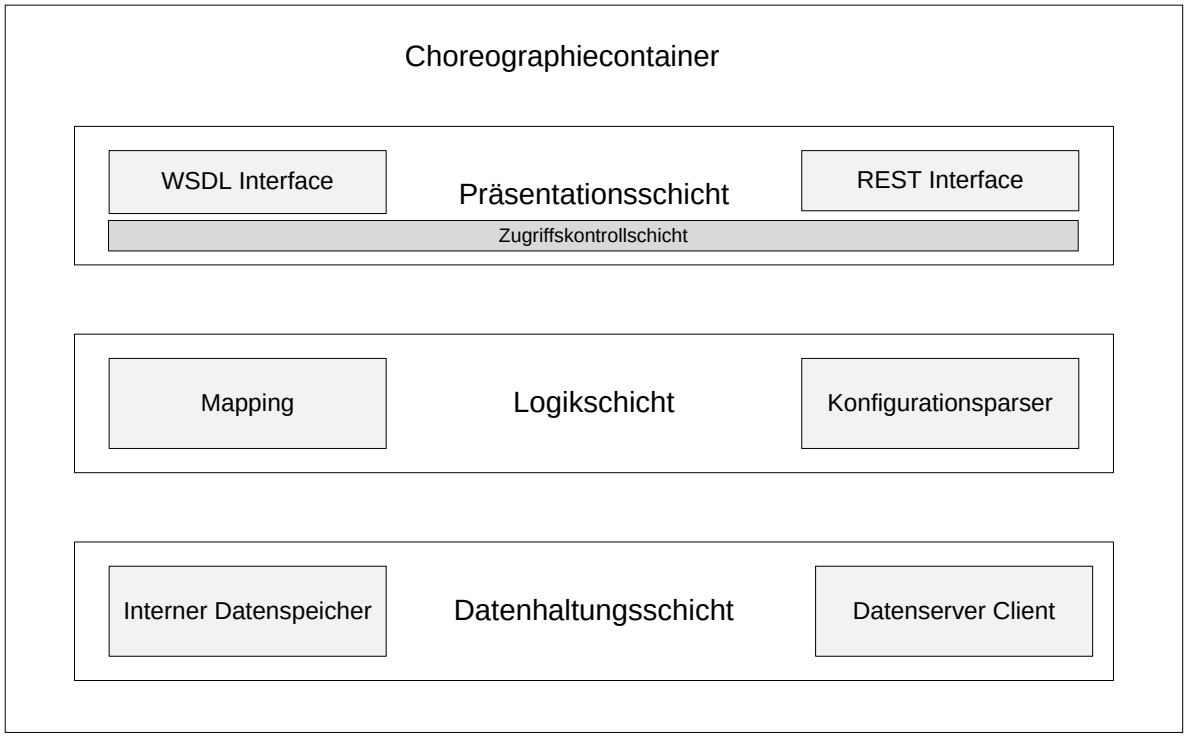


Abbildung 4.20.: Softwarearchitektur eines Choreographiecontainers

wie einen Speicher- und einen Ladebutton, beinhaltet. Speichern und laden beziehen sich auf die graphischen Elemente und deren darunter liegende Modelle. Es wird durch das Drücken von Speichern nur die Darstellung der Choreographie gespeichert und noch keine spezifischen Artefakte erzeugt. Die Toolbar kann um weitere Buttons wie: Bearbeiten, Ansicht, Optionen oder Suchen erweitert werden.

Unter dieser Toolbar, befindet sich eine weitere Toolbar die Buttons speziell für die Verwendung als Choreographie mit Choreographiecontainer Editor beinhaltet. Der Button mit der Aufschrift *Erzeuge Prozessartefakte*, serialisiert die für die verwendete Choreographiesprache spezifischen Prozessartefakte. Der Button *Erzeuge Container Descriptor*, serialisiert den Container Descriptor, in der in 4.4.2 beschriebenen Form, für die Choreographie. Der letzte Button in der Toolbar *Erzeuge External User Descriptor* serialisiert den External User Descriptor, in der in 4.4.3 beschriebenen Form, für die Choreographie.

Den zentralen Platz des Editors nimmt die, als solche benannte, Zeichenfläche ein. Auf dieser Fläche können die einzelnen Elemente einer Choreographie mit Choreographiecontainer platziert werden. Einzelne Elemente können aus der Palette am rechten Rand selektiert werden. Die Elemente sind in folgenden Gruppen angeordnet: Prozesse, externe Nutzer, Choreographiecontainer, Variablen und Verbindungen. Die Elemente, die den Gruppennamen tragen und farblich hervorgehoben sind, können nicht auf der Zeichenfläche platziert werden. Wenn diese angeklickt werden, wird die Gruppe aus- oder eingeklappt. Wie bereits in 4.3.2 beschrieben, können nur die Elemente Prozess, externer Nutzer und Choreographiecontainer direkt auf der Zeichenfläche platziert werden. Aktivitäten lassen sich

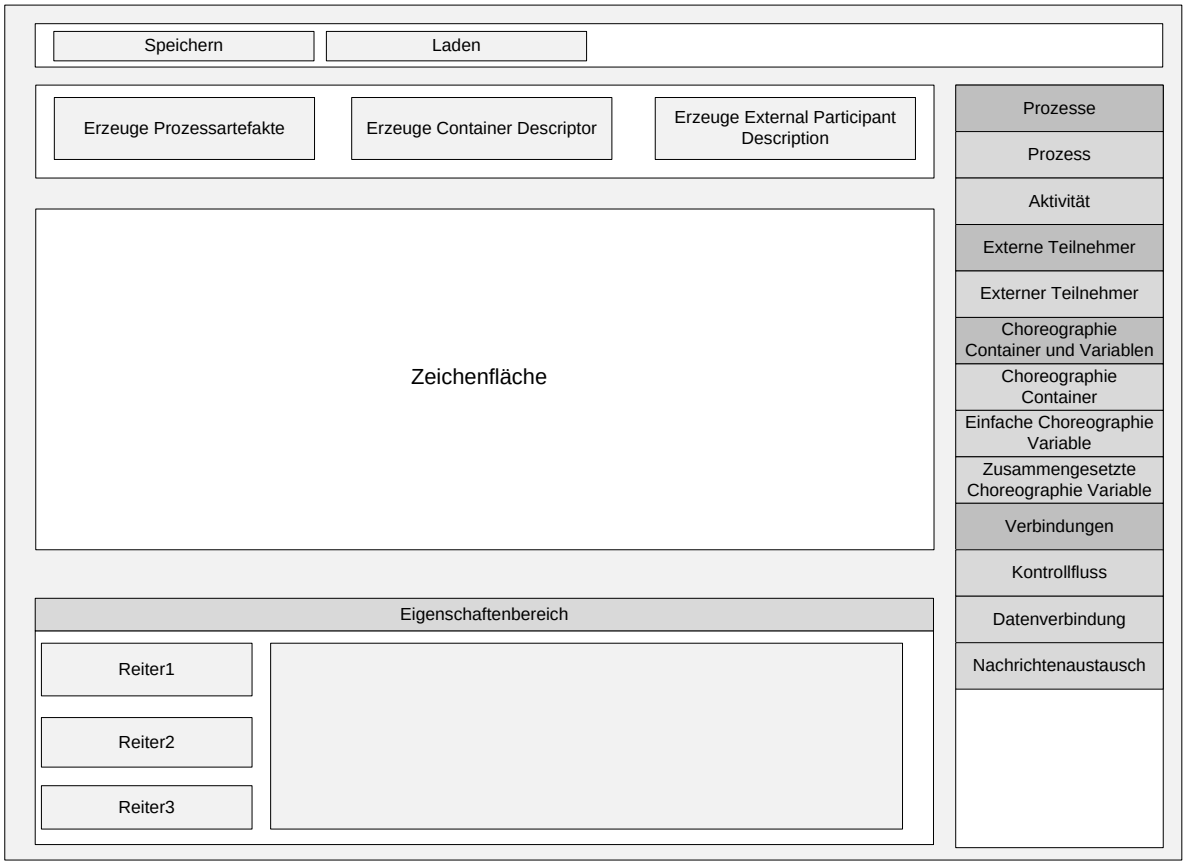


Abbildung 4.21.: Oberfläche eines Designers für eine Choreographie mit Choreographiecontainer

nur in einem bereits platzierten Prozess platzieren und die beiden Variablen Typen nur in einem Choreographiecontainer. Verbindungen lassen sich nur zwischen den entsprechenden Elementen ziehen. Der Editor ist so konstruiert, dass es nur möglich ist Elemente nur ihren Definitionen gemäß zu platzieren. Würde z. B. versucht eine Aktivität auf der Zeichenfläche zu platzieren, verändert sich das Maussymbol und bei einem Klick passiert nichts.

Den unterem Teil des Editors nimmt ein Eigenschaftsbereich ein. In der Leiste, in der in Abbildung 4.21 Eigenschaftsbereich steht, wird der Typ des aktuell auf der Zeichenfläche selektierten Elements angezeigt. Auf der linken Seite des Bereichs sind mehrere Reiter, die die verschiedenen Einstellungsmöglichkeiten eines Elements gruppieren. Der aktuell selektierte Reiter wird, wie in Abbildung 4.26 gezeigt, durch einen dickeren Rand hervor gehoben. Die rechte Seite der Ansicht zeigt die Einstellungsmöglichkeiten an, die der aktuelle Reiter bereitstellt. Diese Reiter stellen ein Graphische Oberfläche dar, über die die Eigenschaften eines Modellelements verändert werden können.

Die Abbildung 4.22 zeigt die verschiedenen Inhalte der Eigenschaftsbereiche der Elemente einer Choreographie mit Choreographiecontainer. Alle Elemente außer Verbindungen, auf welche später näher eingegangen wird, haben den Reiter *Eigenschaften*. Jedes Element hat in diesem Reiter ein Label mit dem Inhalt Name und ein leeres Textfeld. In das leere Textfeld wird der, für diese Choreographie, einzigartige Name des Elements eingetragen. Da der Name vom Typ *NCName* ist, darf er keinen

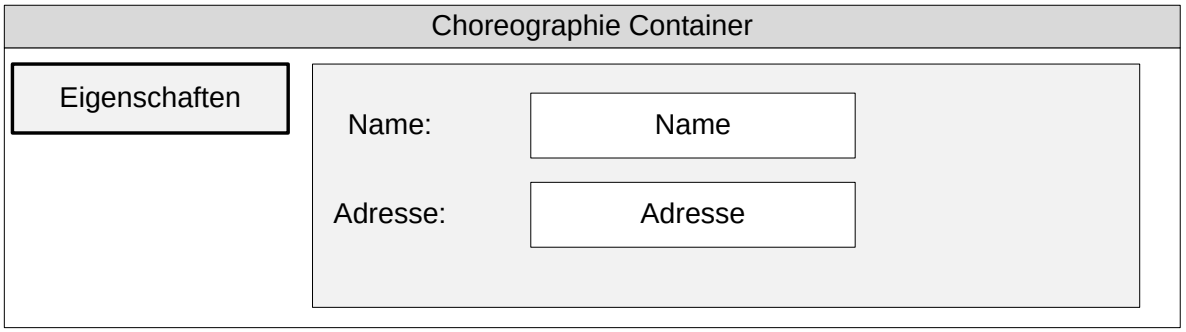


Abbildung 4.22.: Eigenschaftsreiter des Choreographiecontainers

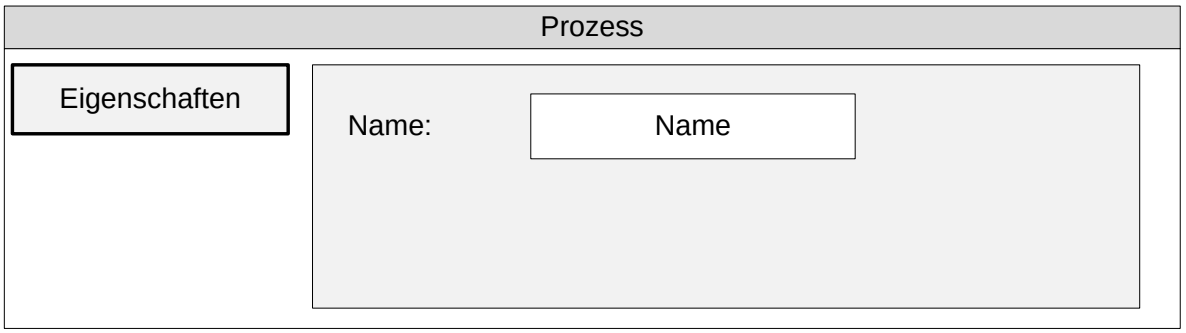


Abbildung 4.23.: Eigenschaftsreiter von Prozessen und Aktivitäten

Doppelpunkt enthalten. Außerdem enthält der Reiter ein leeres Textfeld, in welches die Adresse des Choreographiecontainer eingetragen wird.

Abbildung 4.23 zeigt die Eigenschaftsreiter von Prozessen und Aktivitäten. Da die Details dieser Elemente zum Zeitpunkt der Erstellung des Choreographiecontainers noch nicht im Vordergrund stehen, verfügen die Elemente nur über ein Textfeld für den Namen des jeweiligen Elements.

Externe Nutzer haben, wie in Abbildung, 4.24 dargestellt, eine Checkbox mit dem Label Schreiber. Diese Checkbox stellt einen zusätzlichen Sicherheitsmechanismus dar, da es generell Sicherheitsrisiken birgt, Teilnehmer von außerhalb der Choreographie Daten in den Choreographiecontainer schreiben zu lassen. Aus diesem Grund muss zusätzlich zu einer Datenverbindung von externem Nutzer zu dem Container oder einer Variablen die Checkbox aktiviert werden. Sollte diese nicht aktiviert sein wenn Choreographieartefakte erzeugt werden sollen, wird die entsprechende Datenverbindung ignoriert.

Der, in Abbildung 4.25, gezeigte Eigenschaftsbereich für das Element zusammengesetzte Variable hat eine Checkbox mit dem Label Permanent. Das Aktivieren dieser Checkbox setzt das *Permanent* Attribut der Variablen auf wahr. Bei allen Variablen die sich innerhalb dieser Variablen befinden wird das Attribut ebenfalls auf wahr gesetzt. Dadurch wird der Choreographieumgebung signalisiert, dass diese Variablen persistent gespeichert werden müssen.

Atomare Variablen haben, wie in Abbildung 4.26, zusätzlich zu der Permanent Checkbox, eine Checkbox mit dem Label Referenz und eine mit dem Label Konstante. Die Checkbox mit dem Label Referenz setzt das *Referenz* Attribut auf den Wert wahr und die Checkbox mit dem Label Konstante setzt das

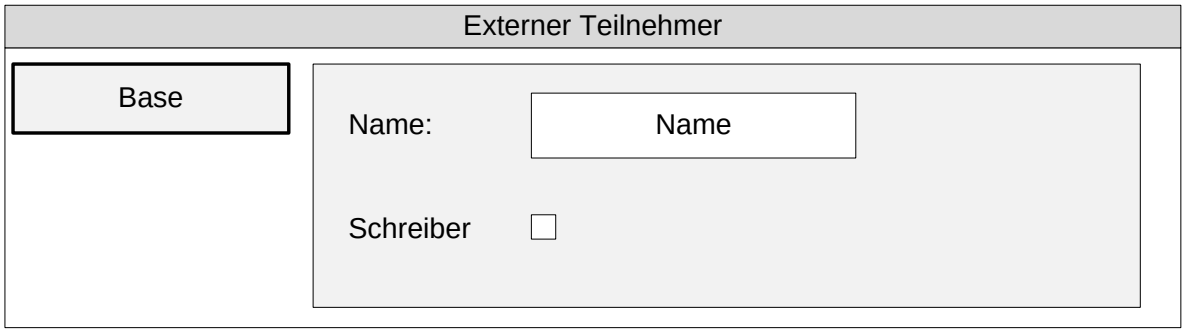


Abbildung 4.24.: Eigenschaftsreiter eines externen Nutzers

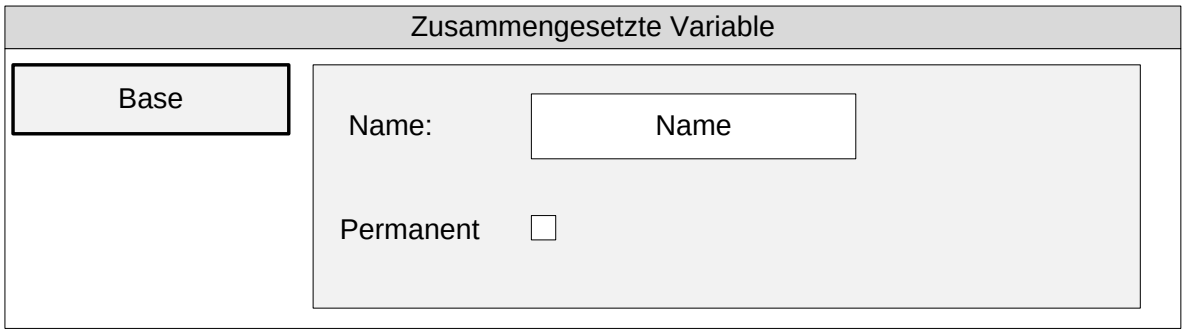


Abbildung 4.25.: Eigenschaftsreiter einer zusammengesetzten Variable

Konstante Attribut auf den Wert wahr. Zusätzlich zu dem Feld mit dem Namen der Variablen, hat eine atomare Variable ein leeres Feld mit dem Label Datentyp. Der Datentyp muss angegeben werden. Der Datentyp der Variablen wird in Form einer Zeichenkette eingetragen. Falls dieses Eintragen nicht vorgenommen wird, wird beim Versuch Prozessartefakte zu erzeugen ein Fehler angezeigt. Die beiden Reiter *Schreiber* und *Leser* haben eine leere Liste mit dem Label Teilnehmer. Wird der Button *Hinzufügen* gedrückt, öffnet sich ein Fenster in dem sämtliche Teilnehmer einer Choreographie, welche die Möglichkeit haben auf Choreographievariablen zuzugreifen, angezeigt werden. Durch einen Klick kann der entsprechende Teilnehmer selektiert werden und durch das drücken des Hinzufügen *Buttons* wird der Teilnehmer in die Liste hinzugefügt. Die Verwendung der Reiter Schreiber und Leser ist optional. Das Verbinden durch Datenflusspfeile ist verpflichtend. Die Pfeile können für eine bessere Übersicht ausgeblendet werden.

Die Eigenschaftsbereiche der Verbindungen unterscheiden sich von denen der anderen Elemente. Eine Verbindung vom Typ Kontrollfluss verfügt im Eigenschaftsbereich, wie in Abbildung 4.27 gezeigt, über ein leeres Textfeld mit dem Label Bedingung. Die Verwendung dieses Feldes ist optional und gibt Bedingungen an, die erfüllt sein müssen, damit der Kontrollfluss ausgeführt wird.

Der Eigenschaftsbereich einer Verbindung vom Typ Nachrichtenaustausch hat, wie in Abbildung 4.28 zwei leere Textfelder, eines mit dem Label Nachricht und eine mit dem Label Datentyp. Das Nachrichtenfeld ist optional zu befüllen. Wenn ein Name eingetragen wird, wird er, wie in 4.8 gezeigt, in der Linie des Pfeils angezeigt. Der Datentyp ist, in Form einer Zeichenkette einzutragen. Die Angabe des Datentyps ist verpflichtend.

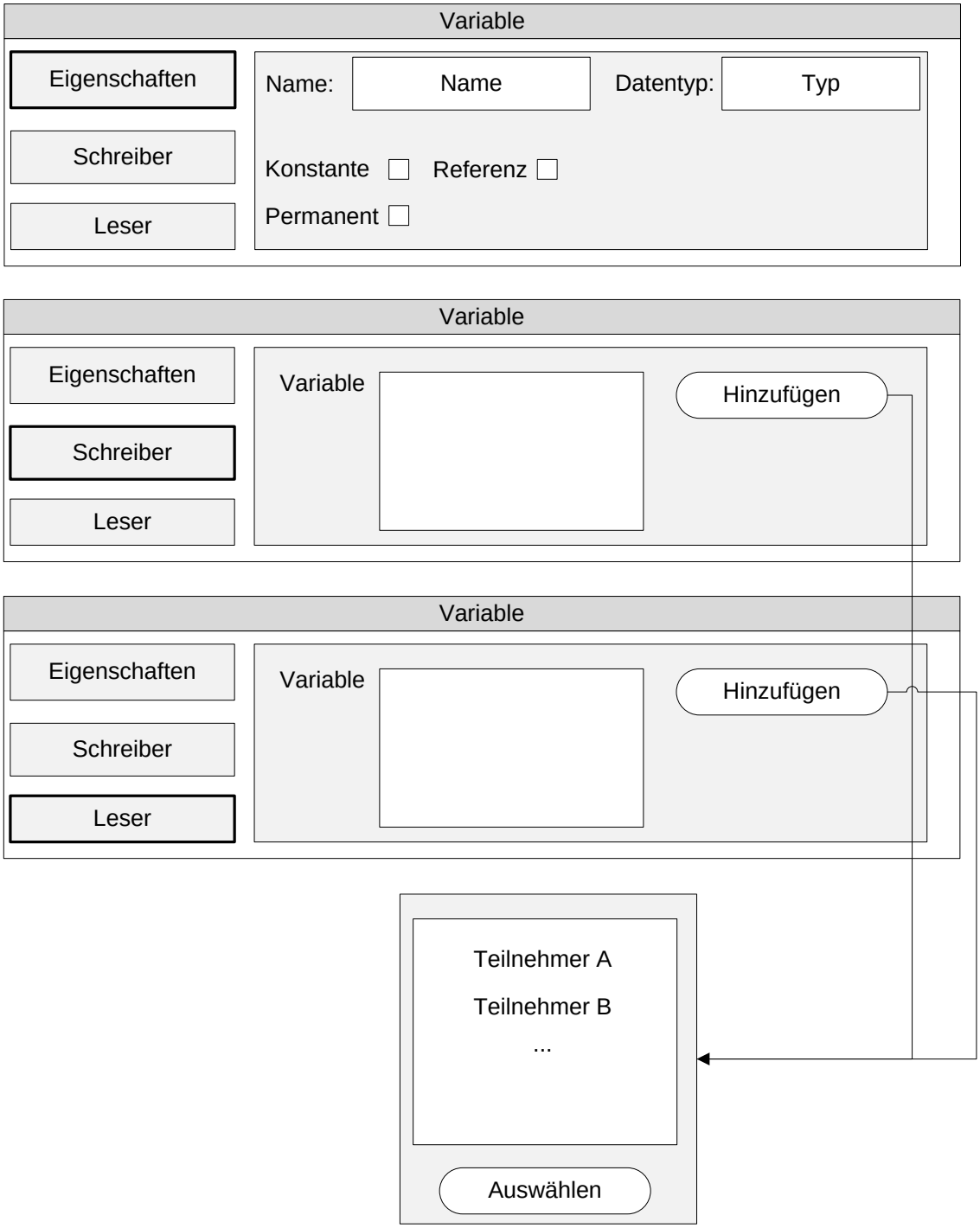


Abbildung 4.26.: Eigenschaftsreiter einer atomaren Variablen

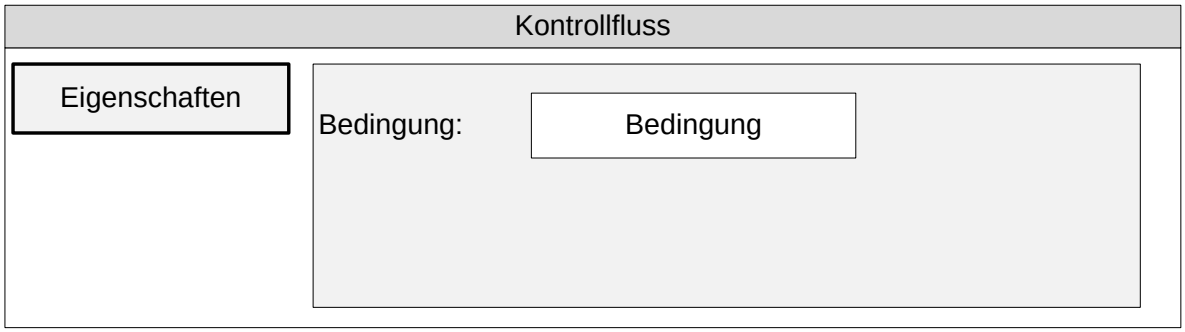


Abbildung 4.27.: Eigenschaftsreiter eines Kontrollflusses

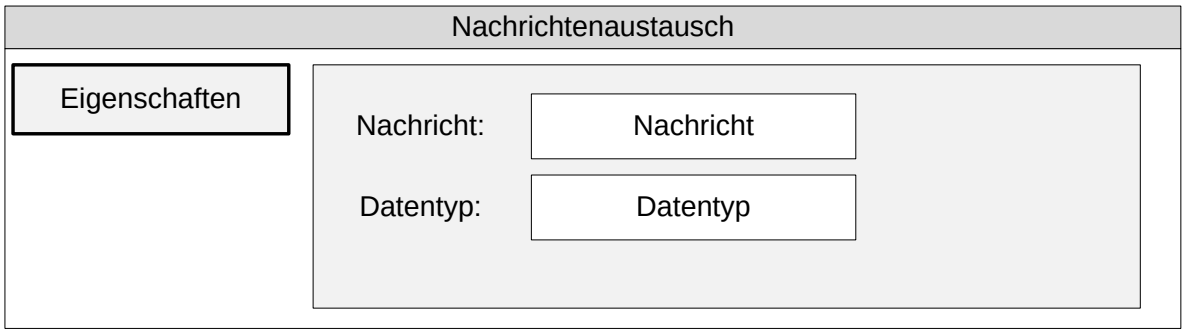


Abbildung 4.28.: Eigenschaftsreiter einer Nachrichtenverbindung

Eine Verbindung vom Typ Datenfluss hat, wie in Abbildung 4.29 zu sehen ist, ein leeres Textfeld mit dem Label Name. In dieses Textfeld kann man optional die Art des Zugriffs, z. B. lesen oder schreiben eintragen. Wird eine Name eingetragen, wird diese in der Linie des Verbinders angezeigt. Das Eintragen eines Namens dient nur der Übersicht und wird in kein Choreographieartefakt übernommen.

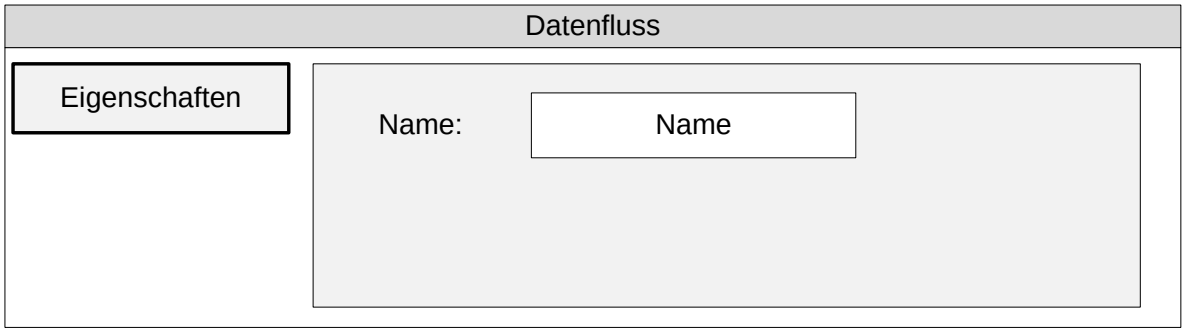


Abbildung 4.29.: Eigenschaftsreiter eines Datenflusses

5. Realisierung

In diesem Abschnitt werden die Erweiterungen des BPEL4Chor [Son13] beschrieben, die für die Realisierung, des in dieser Arbeit beschriebenen, graphischen Konzepts einer Choreographie mit Choreographiecontainer, durchgeführt wurden. Der bereits existierende BPEL4Chor Editor wurde als Grundlage gewählt, da dieser zum Einen, den Teil der zur Erstellung von Prozessartefakten bereits implementiert hat und zum Anderen als Eclipse Plugin realisiert wurde und somit vergleichsweise einfach erweiterbar ist.

5.1. Übersicht des vorhandenen Editors

Abbildung 5.1 zeigt ein Bildschirmfoto des bereits existierenden Editors. Am oberen Ende des Editors befindet sich, gelb umrandet, eine Toolbar mit der, die wichtigsten Elemente zur Steuerung eines Editors wie z. B. ein File Abschnitt mit dem Choreographien gespeichert und geladen werden können. Der rot umrandete Teil des Editors ist die Zeichenfläche, auf der die graphischen Elemente platziert werden können. Diese Elemente können auf der rechten Seite, aus der rot umrandeten, Palette selektiert werden. Im unteren Teil des Editors befindet sich grün umrandet ein Eigenschaftenfenster, in dem die Eigenschaften eines selektierten, graphischen Elements, auf verschiedene Reiter aufgeteilt, angezeigt werden.

Der bereits bestehende Editor, erfüllt somit, von Seiten der Nutzeroberfläche bereits alle Bedingungen die in 4.9 beschrieben wurden. Jedoch müssen noch die für die Choreographie mit Choreographiecontainer nötigen, zusätzlichen graphischen Elemente hinzugefügt werden. Wie diese modelliert wurden, wird in den folgenden Abschnitten beschrieben.

Die in den nächsten Abschnitten gezeigten Modelle wurden in [Son13] beschrieben und auch erstellt. Sie mussten jedoch erweitert werden um die Möglichkeit zu bieten Choreographien mit Choreographiecontainer darstellen zu können. Aus diesem Grund wird nur auf die entsprechenden Erweiterungen dieser Modelle eingegangen, die benötigt werden um eine Choreographie mit Choreographiecontainer darstellen und bearbeiten zu können.

5.2. EMF Modelle

Abbildung 5.2 zeigt das Ecore Modell der Participant behavior description. Dieses Modell enthält im wesentlichen die Elemente die im BPEL Standard beschrieben wurden. Von besonderem Interesse für das Konzept dieser Arbeit sind die beiden Elemente Process und Scope. Diese beiden Elemente

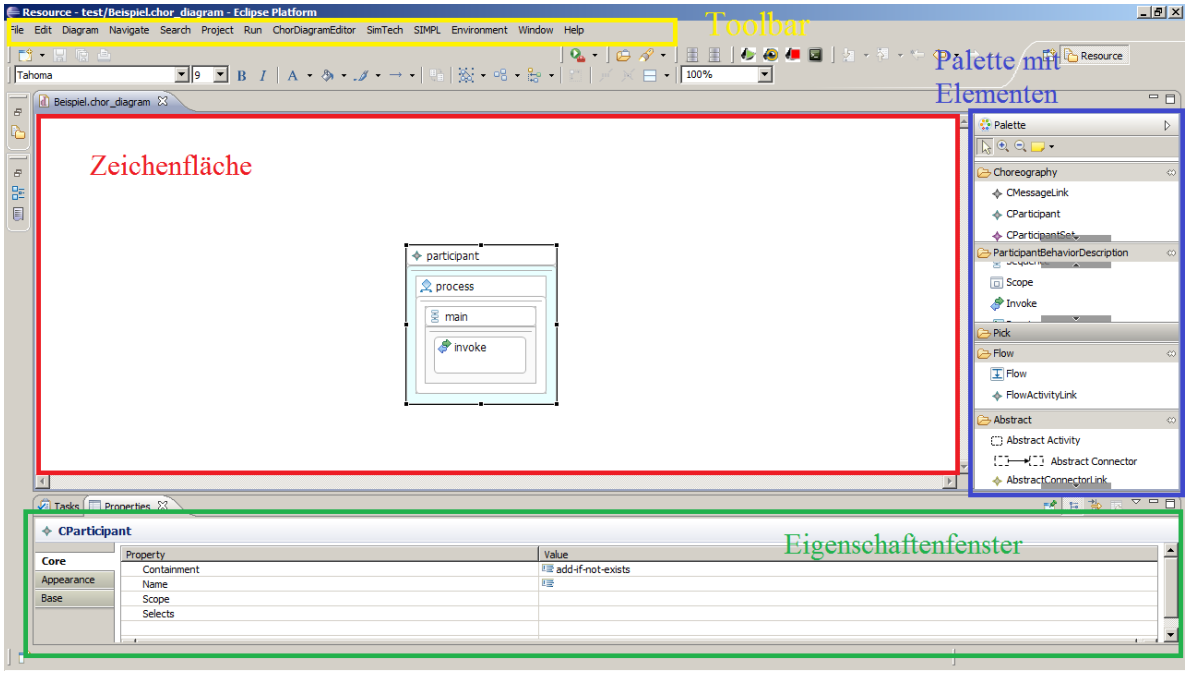


Abbildung 5.1.: Übersicht des Editors

enthalten bereits gemäß BPEL Standard Variablen. Aus diesem Grund werden sie genutzt um die Datenmodell Elemente, Prozess und Aktivität aus 4.3.2 darzustellen. An der graphischen Darstellung und der standardmäßigen Verwendung der Elemente wurde nichts verändert. Sie wurden nur dahingehend verändert, dass es möglich ist eine Verbindung zwischen ihnen und einem Choreographiecontainer bzw. einer Choreographievariablen zu erstellen.

Um eine Verbindung zwischen verschiedenen Elementen zu ermöglichen wurde das Element *CDataLinkable* eingeführt, dessen Eigenschaftenfenster in 5.3 dargestellt wird. Die Eigenschaften *Abstract* und *Interface* wurden auf *true* gesetzt, um eine Verwendung als Interface zu ermöglichen. Jedes Element, dass von diesem Element erbt, kann potentiell mit dem anderen verbunden werden. Das Interface *CDataLinkable* wurde im PBD Modell hinzugefügt, da die beiden Elemente *Process* und *Scope* in der Lage sein müssen es zu nutzen. Das hinzufügen des Interfaces *CDataLinkable* ist die einzige Änderung die am Datenmodell der Elemente *Process* und *Scope* durchgeführt wurde. Das Einfügen des Interfaces in dieses Modell ist notwendig, da zwar die Elemente aus dem PBD Modell im Chor Modell sichtbar sind, aber nicht die Elemente aus dem Chor Modell im PBD Modell. Es wäre möglich dies zu ändern, dies würde dann allerdings zu einer gegenseitigen Abhängigkeit führen.

Das in Abbildung 5.4 dargestellte Ecore Modell der Choreographie hat als Wurzelelement *Choreography*. Dieses Element beschreibt die Choreographie als ganzes, das bedeutet, dass alle Elemente die direkt mit der Choreographie in Verbindung stehen als Referenz unter dieses Element eingefügt werden müssen. In diesem Fall sind es die Elemente *ccontainer*, *cexternalparticipant* und *cdatalinks*. Diese Elemente entsprechen dem Choreographiecontainer, den externen Nutzern und den Datenverbindungen im Datenmodell 4.3.2. Durch die entsprechende Einstellungen kann jede *Choreography*

Abbildung 5.2.: Ecore Modell der PBD

nur eine Instanz vom Typ *ccontainer* erstellen und eine unbegrenzte Menge von *cexternalparticipant* und *cdatalinks* Instanzen.

Die Elemente *CContainer*, *CContainerSimpleVariable*, *CContainerComplexVariable* und *CExternalParticipant* erben von dem Interface *CDataLinkable* und enthalten die in 4.3.2 beschriebenen Attribute. Diese Elemente entsprechen: dem Choreographiecontainer, der atomaren Variablen, der zusammengesetzten Variablen und dem externen Nutzer im Datenmodell 4.3.2. Die Namen sind hierbei vom Typ *EString* und die binären Attribute vom Typ *EBoolean*. Da sowohl der Choreographiecontainer als auch zusammengesetzte Variablen, Choreographievariablen enthalten können, haben diese Referenzen auf die beiden Choreographievariablen Typen *CContainerSimpleVariable* und *CContainerComplexVariable*.

Das Element *CDataLink* ermöglicht es durch dessen beide Attribute *target* und *source* vom Typ *CDataLinkable*, die Elemente, die von diesem Interface erben, miteinander zu verbinden.

5.3. Tooling Definition Model

Abbildung 5.5 zeigt das gesamte Tooling Definition Model des BPEL4Chor Editors. Jedes graphische Element, welches auf der Zeichenfläche platziert werden soll, muss in dieses Modell eingetragen werden. Das Modell ist in mehrere *Tool Groups* aufgeteilt. Die Einträge für den Choreographiecontainer, die Choreographievariablen, die externen Nutzer und die Datenverbindung, wurden zu der Gruppe *Choreography* hinzugefügt.

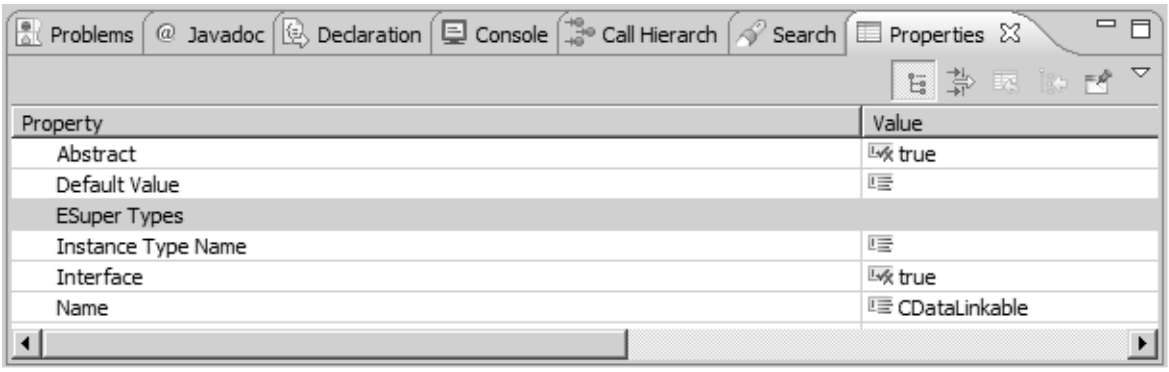


Abbildung 5.3.: Eigenschatfsfenster des CDataLinkable Elements

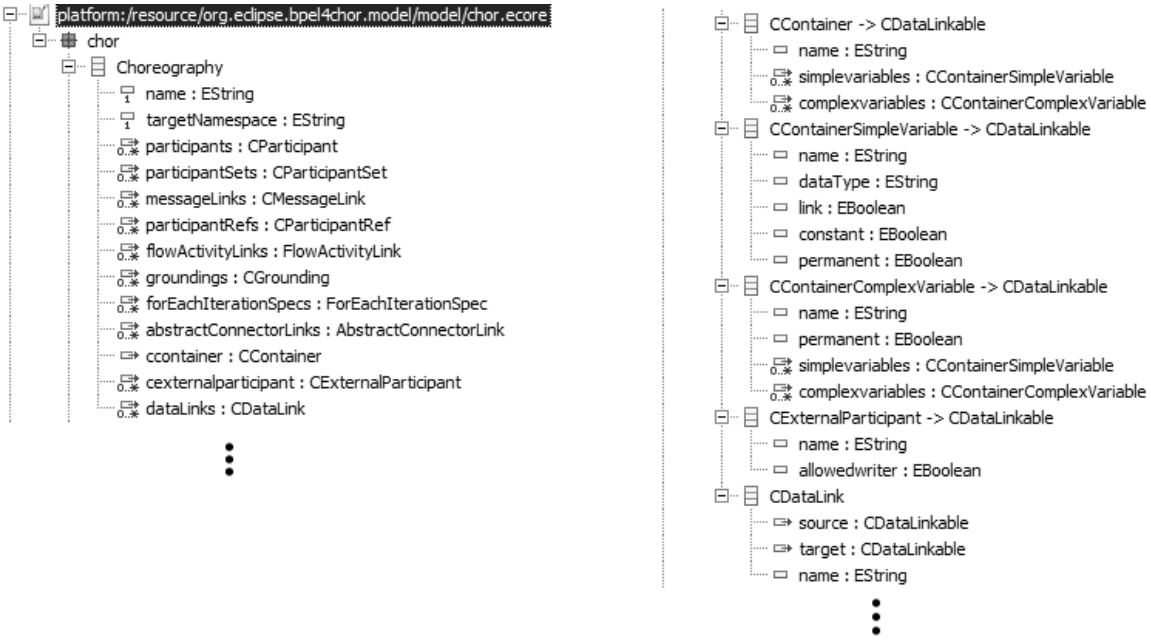


Abbildung 5.4.: Ecore Modell der Choreographie

5.4. Graphical Definition Model

Abbildung 5.6 zeigt die *Figure Descriptors* des Graphical Definition Model. Die ausgeklappten Elemente sind jene, welche neu hinzugefügt wurden. Die Figure Descriptors bestimmen das Aussehen der graphischen Elemente auf der Zeichenfläche. Die Zeichenfläche wird durch das *Canvas Chor* Element definiert. Das *Canvas Chor* Element enthält alle Elemente die, graphische Elemente definieren.

Das Aussehen aller, in dieser Arbeit erstellten, graphischen Elemente wird in der *Figure Gallery choreography* festgelegt. Das Aussehen des Choreographiecontainers wird durch die *CContainerFigure* festgelegt. Das erste Element gibt die äußere Form des Elements an. Diese wurde gemäß der graphischen Darstellung in 4.5 als *Rectangle*, zu deutsch Rechteck festgelegt. Zusätzlich wird ein *Layout*,

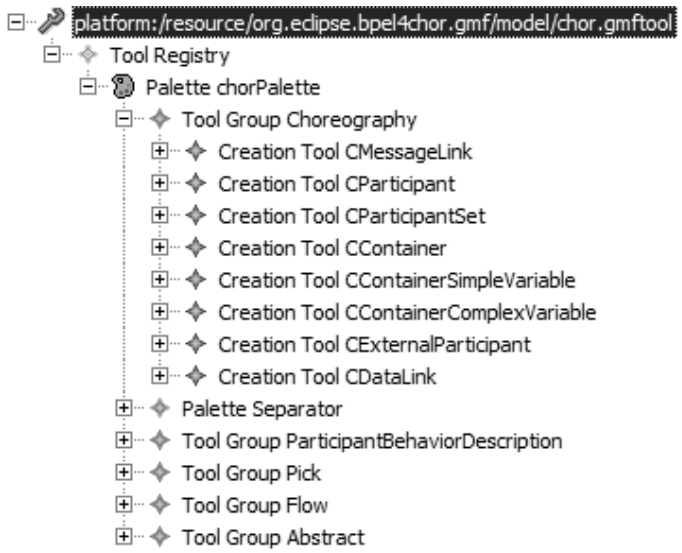


Abbildung 5.5.: Das Tooling Definition Model

ein *Label*, welches in Abbildung 5.8 dargestellt wird und ein *Compartment*, welches in Abbildung 5.7 dargestellt wird, festgelegt. Das *Compartment* gibt die Form des Teils von dem Element an, in welcher andere Elemente platziert werden können. Diese Form wurde auch als rechteckig festgelegt.

Die atomare Variable ist als *CContainerSimpleVariableFigure* und die zusammengesetzte Variable als *CContainerComplexVariableFigure* definiert. Beide Sorten werden als abgerundete Rechtecke dargestellt. Diese *Figures* beinhalten zusätzlich ein *Background* Element, welches es ermöglicht eine Hintergrundfarbe für das Element festzulegen. Die Farbe der atomaren Variablen ist weiß und der zusammengesetzten Variablen ist hellgrau. Die zusammengesetzte Variable enthält, wie auch der *Choreographiecontainer* ein *Compartment*, da in diesem Element weitere atomare und zusammengesetzte Variablen platziert werden können.

Die externen Nutzer werden als *CExternalParticipantFigure* definiert. Dieses Element ist vom Prinzip gleich aufgebaut wie der *Choreographiecontainer*, es enthält jedoch kein *Compartment*.

Die Datenverbindung wird als *CDataLinkFigure* definiert. Diese enthält als Element eine *Polyline Connection*. Diese lässt das Element als eine Verbindungslinie erkennen, da es sich bei der *Polyline* um keine geometrische Figur sondern um eine Linie handelt.

Abbildung 5.9 zeigt die *Nodes* für geometrische Formen und *Connections* für Verbindungen, des Modells. Diese repräsentieren die Elemente welche später graphisch dargestellt werden sollen. Zu diesem Zweck beinhalten sie die Information, welcher *Figure Descriptor* zu ihnen gehört.

5.5. Mapping Definition Model

Das Mapping Definition Model verbindet die Ecore, Tooling Definition und Graphical Definition Modelle miteinander. Abbildung 5.10 zeigt das Mapping Definition Model des Editors. Die ausgeklappten

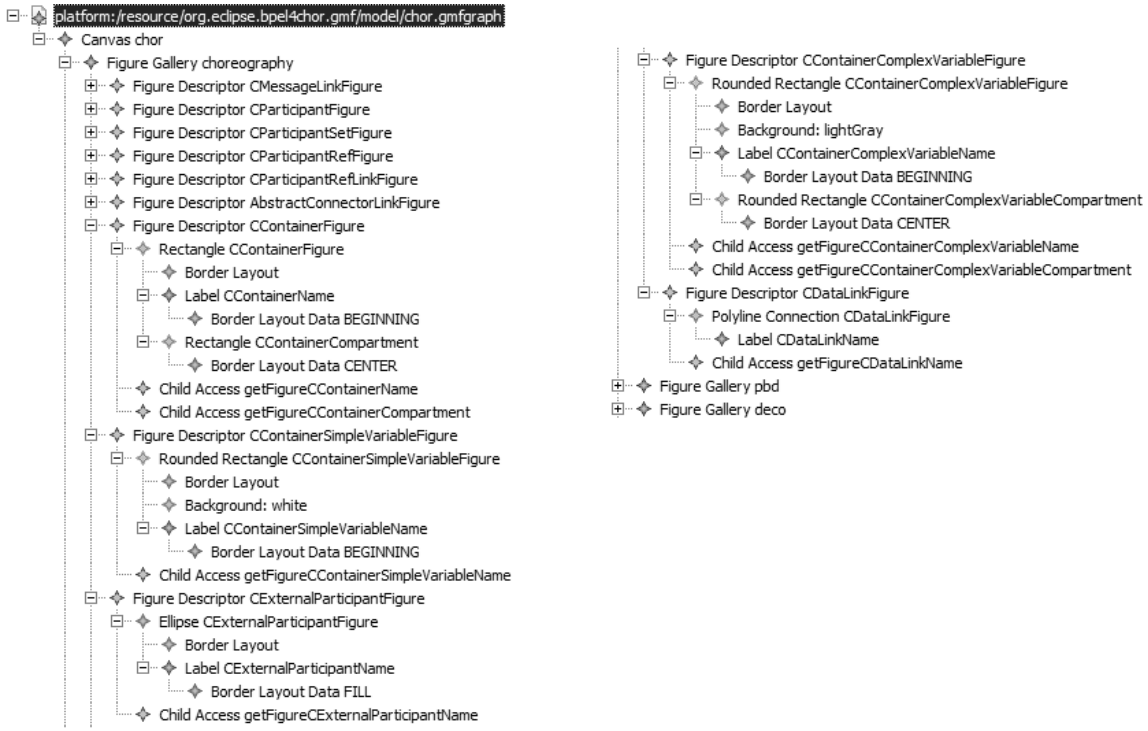


Abbildung 5.6.: Figure Descriptor Abschnitt im Graphical Definition Model

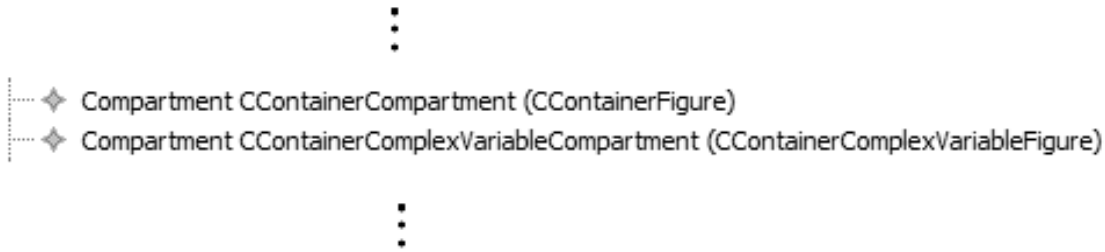


Abbildung 5.7.: Compartment Abschnitt im Graphical Definition Model

Elemente wurde neu hinzugefügt. Die Elemente, die direkt auf der Zeichenfläche platziert werden können, werden entweder als Top Node Reference oder als Link Mapping definiert.

Der *Choreographiecontainer* entspricht hier dem Element *Top Node Reference CContainer*. Dieses Element gibt an, welches Element aus dem Ecore Modell dargestellt werden soll. Die *Top Node Reference* enthält unter anderem das *Node Mapping*. Abbildung 5.11 zeigt das Eigenschaftenfenster des *Node Mappings* für einen Choreographiecontainer. Das Element *Attribut* gibt an, welches Element aus dem Ecore Modell dargestellt werden soll. Das *Diagram Node* Attribut gibt an welche *Node* und über diese welchen *Figure Descriptor* bzw. welche Form das Element haben soll. Zusätzlich enthält das *Node Mapping*, über die *Child References*, die Information welche Kindelemente ein Element

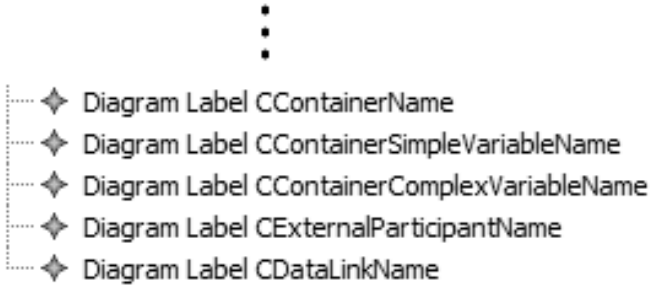


Abbildung 5.8.: Labels Abschnitt im Graphical Definition Model

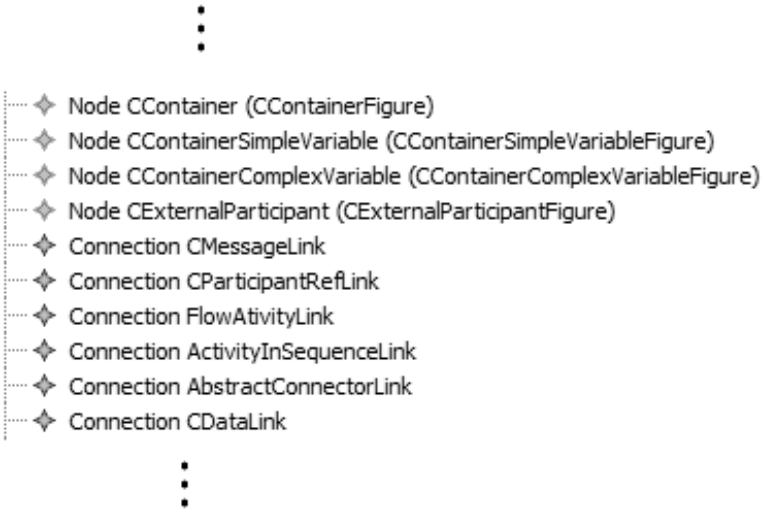


Abbildung 5.9.: Nodes und Connections Abschnitt im Graphical Definition Model

beinhalten kann. Das Eigenschaftenfenster der *Child Reference*, der *CContainerComplexVariable* wird in Abbildung 5.12 dargestellt. Die Attribute *Child* und *Referenced Child* geben an welches *Node Mapping* die *Child Reference* beinhaltet. In diesem Fall das *Node Mapping* der zusammengesetzten Variablen *CContainerComplexVariable*. Das Attribut *Compartment* gibt an, in welchem *Compartment* dieses Element später platziert werden soll. Das *Compartment* entspricht in diesem Fall dem *Compartment Mapping CContainerCompartment*, welches bedeutet, dass das Element im *Choreographiecontainer* Element platziert werden kann. Das Attribut *Containment Feature* gibt an über welche Referenz auf die Daten im Ecore Modell zugegriffen werden können. In diesem Fall kann auf die Daten über die Referenz des *Choreographiecontainers* auf die Daten im Ecore Modell zugegriffen werden. Das Element *CExternalParticipant* folgt einem ähnlichen Aufbau, enthält jedoch keine *Child References* und *Compartments*.

Abbildung 5.13 zeigt das Eigenschaftenfenster der *Child Reference* der zusammengesetzten Variablen, die sich innerhalb der *Child Reference* der zusammengesetzten Variablen befindet. Dies stellt somit eine

Abbildung 5.10.: Mapping Definition Model

Rekursion dar, da zusammengesetzte Variablen sich selbst enthalten können. Die Attribute entsprechen den selben wie in der *Child Reference* eine Stufe darüber, jedoch ändert sich das *Compartment*, da als *Compartment* nicht der *Choreographiecontainer* sondern das *Compartment CContainerComplexVariableCompartment* der zusammengesetzten Variablen *CContainerComplexVariable* verwendet wird. Eine weitere wichtige Änderung ist, dass nicht mehr über die Referenz des *Choreographiecontainer*, *CContainer.complexvariables*, sondern über die Referenz der zusammengesetzten Variablen *CContainerComplexVariable.complexvariables* auf die Ecore Daten der beinhalteten zusammengesetzten Variablen zugegriffen wird.

Verbindungen werden nicht als *Nodes* sondern als *Link Mapping* realisiert. Abbildung 5.14 zeigt das Eigenschaftenfenster der Datenverbindung *CDataLink*. Das *Containment Feature* gibt an, über welche Ecore Referenz auf die Daten zugegriffen wird. Das Element Attribut gibt an, welches Element aus dem Chor Ecore Modell verwendet wird. Die Attribute *Source Feature und Target Feature* geben an, welche Elemente als Ursprung und Ziel der Verbindung verwendet werden können. In diesem Fall sind die die *source* und *target* Attribute des *CDataLink* Elements im Ecore Modell. Das Attribut *Diagram Link* gibt an, welche graphische Repräsentation, aus dem Graphical Definition Model, für die Verbindung verwendet wird. Das letzte Attribut *Tooling*, gibt an welches Element aus dem Tooling Definition Model verwendet wird, um die Verbindung zu erzeugen.

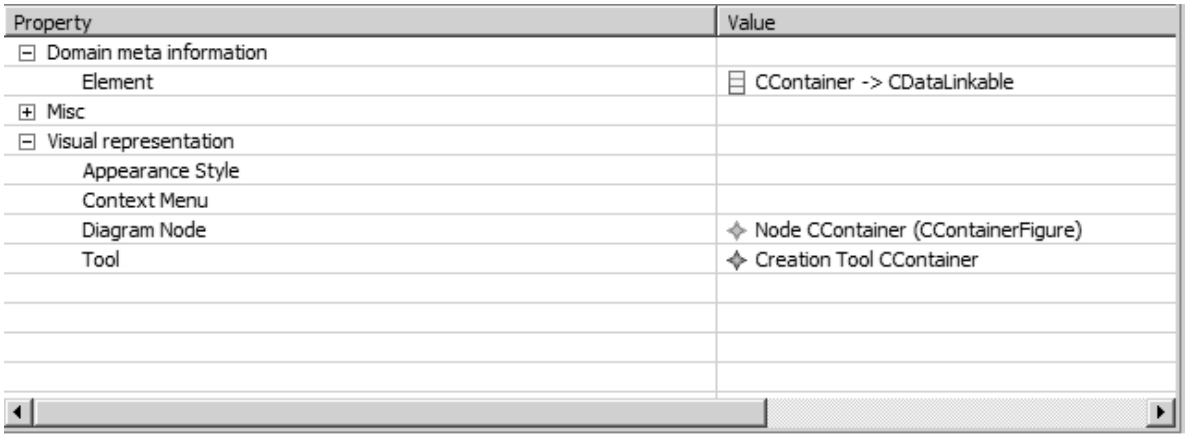


Abbildung 5.11.: Eigenschaftenfenster des CContainers

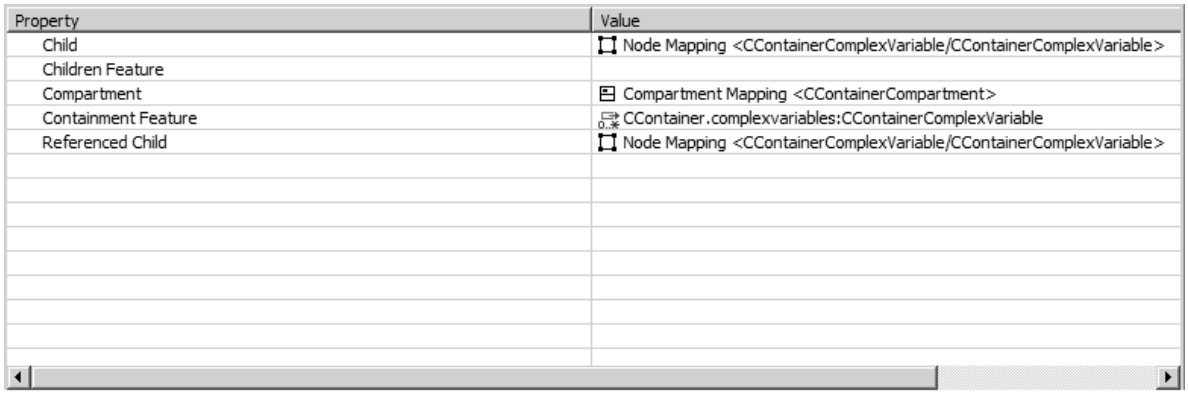


Abbildung 5.12.: Eigenschaftenfenster der Child Reference einer CComplexVariable

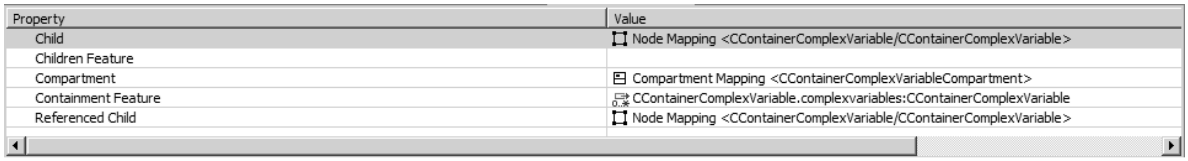


Abbildung 5.13.: Eigenschaftenfenster der Child Reference einer CComplexVariable innerhalb einer CComplexVariable zur Selbstbeinhaltumg

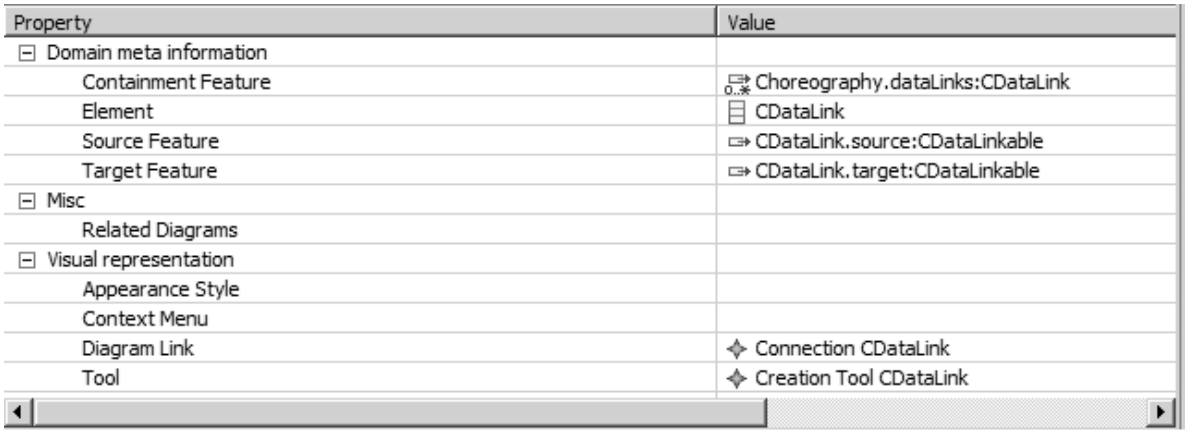


Abbildung 5.14.: Eigenschaftenfenster eines CDataLink

5.6. Ergebnis

Abbildung 5.15 zeigt das Ergebnis, des aus den vorherigen Abschnitten beschriebenen Editor. Im oberen Teil der Choreographie sind zwei externe Nutzer zu sehen, die einmal schreibend und einmal lesen, auf den Choreographiecontainer darunter zugreifen. Der Choreographiecontainer enthält eine atomare Variable und eine zusammengesetzte Variable. Die zusammengesetzte Variable enthält selbst eine atomare und eine zusammengesetzte Variable, welche wiederum zwei atomare Variablen enthält.

Die verschiedenen Teilnehmer sind über lesende und schreibende Datenverbindungen mit den Choreographievariablen im Choreographiecontainer und auch dem Choreographiecontainer selbst verbunden. Es werden auch die Möglichkeiten gezeigt, dass die Choreographievariablen sowohl mit einem *Process* als auch mit einem *Scope* verbunden werden können.

Der untere Teil der Abbildung 5.15 zeigt das Eigenschaftenfenster einer atomaren Variablen. Die Darstellung der verschiedenen Attribute werden von Eclipse automatisch in dieser Form generiert. In die beiden Felder *Name* und *DataType* können die entsprechenden Daten in Form einer Zeichenkette eingetragen werden. Die Attribute *Constant*, *Link* und *Permanent* sind als *Dropdown–Listen* realisiert und die Werte *true* oder *false* können ausgewählt werden.

5.7. Geplante Umsetzung für das Einfügen von Variablen

Abbildung 5.16 zeigt ein stark vereinfachtes Beispiel einer Choreographie mit zwei Teilnehmern und einem Choreographiecontainer der eine atomare und eine zusammengesetzte Variable enthält, welche wiederum zwei weitere atomare Variablen enthalten. Die Prozesse beider Teilnehmer haben Zugriff auf die atomare Variable *SimulationID* und die *Scopes* der Prozesse auf die atomare Variable *Teil1* bzw. *Teil2*. Listing 5.1 zeigt den BPEL Code der automatisch für *Prozess1* aus der graphischen Darstellung generiert werden soll. In dem bisherigen Editor wurden keine Variablen erzeugt sondern nur *Prozess1*, *main* und *Berechnung1*.

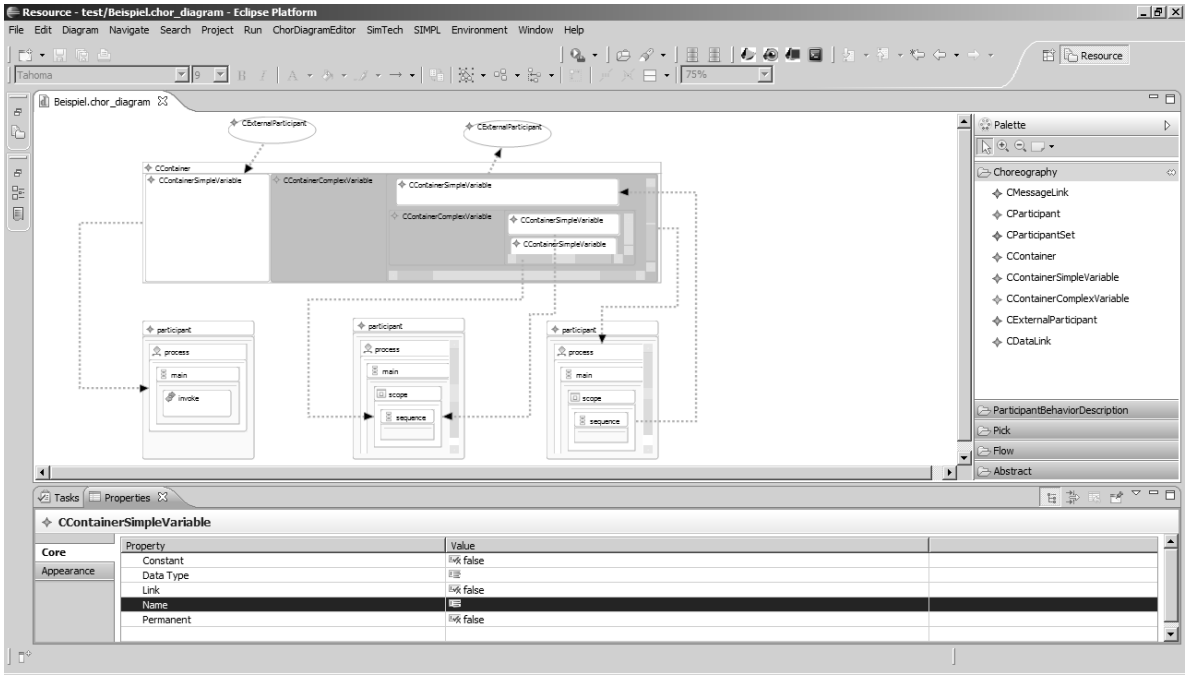


Abbildung 5.15.: Beispiel des Editors mit Choreographiecontainer

Die Variablen werden, je nachdem welche Prozesse und welche Scopes mit den Variablen verbunden sind, in die BPEL Serialisierung eingefügt. Die eingefügten Variablen sind alle vom Typ String, da in diesem Typ alle anderen Datentypen enthalten sein können und die Größe der enthalten Daten keiner direkten Größenbeschränkung, außer der mit welcher Größe von Strings die Orchestrierungsengine effektiv umgehen kann, unterliegt. Der erzeugt BPEL Code ist standardkonform.

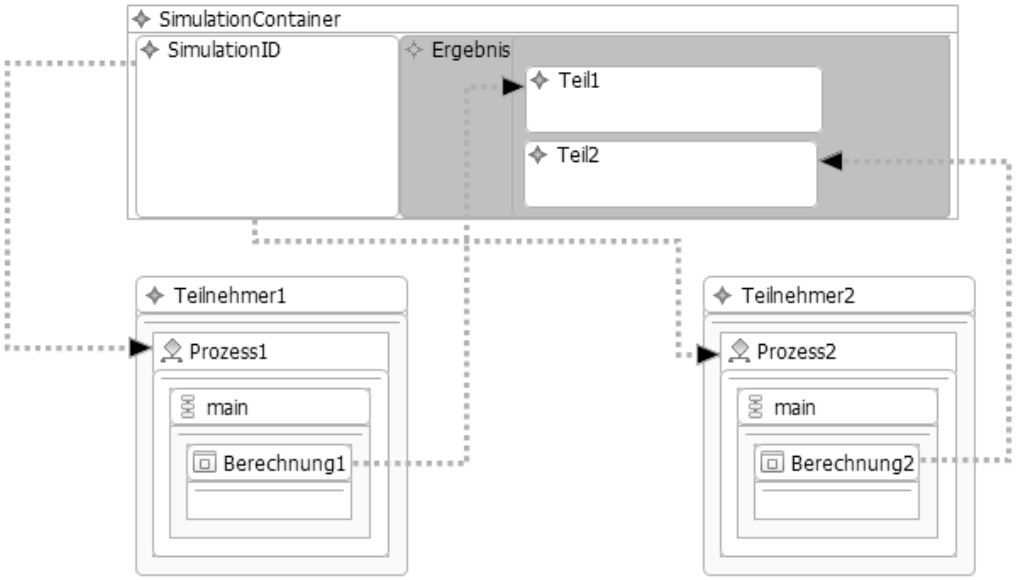


Abbildung 5.16.: Beispiel einer Darstellung auf der Zeichenfläche

Listing 5.1 BPEL Serialisierung von Prozess1

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<bpel:process
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    name="Prozess1" xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/abstract">

      <variables>
            <variable   name="SimulationID" type="xsd:String" />
      </variables>

      <bpel:sequence name="main">

            <bpel:scope name="Berechnung1">

                  <variables>
                    <variable  name="Teil1" type="xsd:String" />
                  </variables>

            </bpel:scope name="Berechnung1">

      </bpel:sequence>

</bpel:process>
```

6. Zusammenfassung und Ausblick

Um dem Problem dass Daten nur statisch in den Prozessmodellen einer Choreographie modelliert werden können und dass unnötige viele Daten über Teilnehmer einer Choreographie geleitet werden, zu begegnen, wird das Konzept des Choreographiecontainers eingeführt. Das Konzept der Choreographiecontainer wird in dieser Arbeit erläutert und es wird ein bestehender graphischer Editor [Son13] dahingehend erweitert, dass Choreographien mit Choreographiecontainer graphisch modelliert werden können.

Die Grundlagen deren Verständnis wichtig sind werden in Kapitel 2 erläutert.

In Kapitel 4 werden Arbeiten, deren Ansätze sich ebenfalls mit dem Datenfluss in Choreographien befassen, vorgestellt.

In Kapitel 4 wird zunächst das grobe Konzept eines Choreographiecontainers vorgestellt. Zu diesem Zweck werden die benötigten Komponenten, die ein System welches Choreographiecontainer unterstützt, beschrieben und ein Datenmodell für die Choreographie selbst erläutert. Im Anschluss wird der Zusammenhang, der zwischen den einzelnen Choreographieartefakten und zu dem Datenmodell besteht, erläutert. Außerdem werden die beiden, für die Choreographie mit Choreographiecontainer benötigten Choreographieartefakte: Container Descriptor und External User Description beschrieben und anhand eines konkreten Beispiels erläutert. Darauf folgend wird eine graphische Notation eingeführt, mit der es möglich ist eine Choreographie mit Choreographiecontainer darzustellen. Es werden auch Anwendungsfälle dargestellt und beschrieben, bei denen der Einsatz von Choreographiecontainern besonders sinnvoll ist. Im weiteren Verlauf werden die Softwarearchitektur eines Choreographiecontainers und die graphische Oberfläche eines Editors der für die Modellierung einer Choreographie mit Choreographiecontainer verwendet werden kann erläutert.

Für die Implementierung des graphischen Editors, wird in Kapitel 5 ein bereits bestehender Editor [Son13], erweitert um eine Choreographie mit Choreographiecontainer darstellen zu können. Zu diesem Zweck werden die GMF– und EMF–Modelle angepasst und aus diesen entsprechender Code generiert. Aufgrund des engen Zeitrahmens konnten keine entsprechenden Komponenten zur Serialisierung der Choreographieartefakte erstellt werden.

Abschließend kann man sagen, dass die Ziele dieser Arbeit größtenteils erreicht werden. Es existiert nun ein umfassendes Konzept, auf das in weiteren Arbeiten aufgebaut werden kann. Außerdem können mittels des BPEL4Chor–Editors Choreographien mit Choreographiecontainer graphisch dargestellt werden.

Ausblick

Weitere Arbeiten könnten sich mit der Erstellung der eigentlichen Choreographiecontainer Komponente beschäftigen. Es existiert zwar bereits eine grobe Softwarearchitektur für diese Komponente jedoch müsste diese noch implementiert werden. Es wäre besonders sinnvoll, diese als einfachen Webservice mit Anschluss an einen Datenserver zu realisieren, da Orchestrierungsengines bereits über die Möglichkeiten verfügen, Webservices aufzurufen. Besondere Teilgebiete dieser Arbeit könnte die Möglichkeit zur internen Speicherung von Daten innerhalb des Choreographiecontainers, zwecks schnellerer Antwortzeiten, beschäftigen. Außerdem könnte die Möglichkeit analysiert werden, eine Garbage Collection einzuführen, die es ermöglicht nicht mehr genutzte Daten sowohl aus dem internen Speicher als auch aus dem Datenserver zu löschen. Des Weiteren könnte die Möglichkeit, verschiedene Datensysteme wie einen REST–Server und eine Datenbank gleichzeitig an den Choreographiecontainer anzuschließen und die Daten entsprechend ihrer Art und Verwendungshäufigkeit auf diese zu verteilen, analysiert und realisiert werden.

Eine weitere Arbeit könnte sich mit der Erweiterung der verwendeten Orchestrierungsengine befassen, um die Möglichkeit, Daten von außerhalb des Workflows von einem Choreographiecontainer abzufragen. Ein besonderes wichtiges und interessantes Themengebiet für diese Arbeit ist der Umgang mit verschiedenen Datentypen aus dem Choreographiecontainer und in der eigentlichen Choreographie.

Eine kleinere Arbeit könnte sich mit der Möglichkeit beschäftigen, aus dem erstellten External User Descriptor, gültige und standardisierte Zugriffskonfigurationen für verschiedene Arten von Datenservern zu erstellen.

A. Anhang

Listing A.1 Vollständiger Container Descriptor von Beispiel 4.6

```
<?xml version="1.0" encoding="utf-8"?>

<choreographyContainer
  xmlns="urn:IAAS:choreography:schemas:choreography:choreographycontainer:2014"
  name="Beispiel">

  <atomicVariable name="Konfiguration" dataType="KonfigurationTyp">
    <writer name="Administrator"/>
    <reader name="Prozess1"/>
  </atomicVariable>

  <complexVariable name="ZwischenergebnisGesamt">

    <atomicVariable name="TeilA" dataType="TeilATyp">
      <writer name="Prozess2"/>
    </atomicVariable>

    <complexVariable name="ZwischenergebnisProzess1">

      <atomicVariable name="TeilB" dataType="TeilBTyp">
        <writer name="C1"/>
      </atomicVariable>

      <atomicVariable name="TeilC" dataType="TeilCTyp">
        <writer name="E1"/>
      </atomicVariable>
    </complexVariable>

    <reader name="Forscher"/>
    <reader name="Prozess3"/>

  </complexVariable>

  <atomicVariable name="Endergebnis" dataType="EndergebnisTyp" permanent="true">
    <writer name="Prozess3"/>
    <reader name="Forscher"/>
    <reader name="Interessent"/>
  </atomicVariable>

</choreographyContainer>
```

Listing A.2 Vollständiger External User Descriptor von Beispiel 4.6

```
<?xml version="1.0" encoding="utf-8"?>
<externalUsersRoles
  xmlns="urn:IAAS:choreography:schemas:choreography:externalUsers:2014"
  name="Beispiel">

<role name="Administrator">
      <writesTo>
            <variable name="Konfiguration" dataType="KonfigurationTyp"/>
      </writesTo>

      <readsFrom>
            <complexVariable name="ChoreographieContainerBeispiel">
                  <variable name="Konfuguration" dataType="KonfugurationTyp">
                  <variable name="TeilA" dataType="TeilATyp">
                  <variable name="TeilB" dataType="TeilBTyp">
                  <variable name="TeilC" dataType="TeilCTyp">
                  <variable name="Endergebnis" dataType="EndegebnisTyp">
            </complexVariable>
      </readsFrom>
</role>

<role name="Forscher">
      <readsFrom>
            <variable name="Endergebnis" dataType="EndegebnisTyp">
            <complexVariable name="Zwischenergebnis">
                  <variable name="TeilA" dataType="TeilATyp">
                  <variable name="TeilB" dataType="TeilBTyp">
                  <variable name="TeilC" dataType="TeilCTyp">
            </complexVariable>
      </readsFrom>
</role>

<role name="Forscher">
      <readsFrom>
            <variable name="Endergebnis" dataType="EndergebnisTyp">
      </readsFrom>
</role>

</externalUsersRoles>
```

Listing A.3 XML-Schema des Container Descriptors

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns="urn:IAAS:choreography:schemas:choreography:choreographycontainer:2014"
      targetNamespace="urn:IAAS:choreography:schemas:choreography:choreographycontainer:2014"
      elementFormDefault="qualified">

      <xs:element name="choreographyContainer" type="ChoreographyContainerType"/>
      <xs:complexType name="ChoreographyContainerType">
            <xs:sequence>
                  <xs:choice maxOccurs="unbounded">
                        <xs:element ref="atomicVariable"/>
                        <xs:element ref="complexVariable"/>
                  </xs:choice>
                  <xs:element ref="reader" maxOccurs="unbounded"/>
            </xs:sequence>

            <xs:attribute name="name" type="xs:NCName" use="required"/>
      </xs:complexType>

      <xs:element name="atomicVariable" type="atomicVariableType"/>
  <xs:complexType name="atomicVariableType">
    <xs:choice maxOccurs="unbounded">
      <xs:element ref="reader"/>
      <xs:element ref="writer"/>
    </xs:choice>

    <xs:attribute name="name" type="xs:NCName" use="required"/>
    <xs:attribute name="dataType" type="xs:string" use="required"/>
    <xs:attribute name="constant" type="xs:boolean"/>
    <xs:attribute name="reference" type="xs:boolean"/>
    <xs:attribute name="permanent" type="xs:boolean"/>
  </xs:complexType>

      <xs:element name="complexVariable" type="complexVariableType"/>
      <xs:complexType name="complexVariableType">
            <xs:sequence>
                  <xs:choice maxOccurs="unbounded">
                        <xs:element ref="atomicVariable"/>
                        <xs:element ref="complexVariable"/>
                  </xs:choice>
                  <xs:element ref="reader" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="name" type="xs:NCName" use="required"/>
            <xs:attribute name="permanent" type="xs:boolean"/>
      </xs:complexType>

      <xs:element name="reader" type="readerType"/>
      <xs:complexType name="readerType">
            <xs:attribute name="name" type="xs:NCName" use="required"/>
      </xs:complexType>


      <xs:element name="writer" type="writerType"/>
  <xs:complexType name="writerType">
    <xs:attribute name="name" type="xs:NCName" use="required"/>
  </xs:complexType>

</xs:schema>
```

Listing A.4 XML-Schema des External User Descriptors

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns="urn:IAAS:choreography:schemas:choreography:externalUsers:2014"
      targetNamespace="urn:IAAS:choreography:schemas:choreography:externalUsers:2014"
      elementFormDefault="qualified">

      <xs:element name="externalUsersRoles" type="externalUsersRolesType"/>
      <xs:complexType name="externalUsersRolesType">
   <xs:sequence>
                  <xs:element ref="role" maxOccurs="unbounded"/>
   </xs:sequence>
            <xs:attribute name="name" type="xs:NCName" use="required"/>
      </xs:complexType>

      <xs:element name="role" type="roleType"/>
  <xs:complexType name="roleType">
    <xs:choice maxOccurs="unbounded">
     <xs:element ref="readsFrom"/>
     <xs:element ref="writesTo"/>
    </xs:choice>

    <xs:attribute name="name" type="xs:NCName" use="required"/>
  </xs:complexType>

      <xs:element name="readsFrom" type="readsFromType"/>
      <xs:complexType name="readsFromType">
            <xs:choice maxOccurs="unbounded">
              <xs:element ref="variable"/>
     <xs:element ref="complexVariable"/>
            </xs:choice>
      </xs:complexType>

      <xs:element name="writesTo" type="writesToType"/>
            <xs:sequence maxOccurs="unbounded">
              <xs:element ref="variable"/>
            </xs:sequence>
  </xs:complexType>

      <xs:element name="variable" type="variableType"/>
      <xs:complexType name="variableType">
            <xs:attribute name="name" type="xs:NCName" use="required"/>
            <xs:attribute name="dataType" type="xs:string" use="required"/>
      </xs:complexType>

      <xs:element name="complexVariable" type="complexVariableType"/>
      <xs:complexType name="complexVariableType">
            <xs:sequence maxOccurs="unbounded">
              <xs:element ref="variable"/>
                   <xs:element ref="complexVariable"/>
            </xs:sequence>

            <xs:attribute name="name" type="xs:NCName" use="required"/>
      </xs:complexType>

</xs:schema>
```

Literaturverzeichnis

[BWH08a] A. Barker, J. Weissman, J. van Hemert. Orchestrating Data-Centric Workflows. In *The 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, S. 210–217. IEEE Computer Society, 2008. (Zitiert auf Seite 19)

[BWH08b] A. Barker, J. B. Weissman, J. van Hemert. Eliminating the Middleman: Peer-to-peer Dataflow. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, S. 55–64. ACM, New York, NY, USA, 2008. (Zitiert auf den Seiten 19, 43, 45 und 46)

[DK14] G. Decker, O. Kopp. Topologgy.xsd, 2014. URL https://github.com/IAAS/BPEL4Chor-model/blob/master/doc/BPEL4Chor20schema/topology.xsd. (Zitiert auf den Seiten 32 und 36)

[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 6, 16 und 17)

[Fie00] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Dissertation, 2000. (Zitiert auf den Seiten 24 und 29)

[Foua] A. S. Foundation. Apache ODE. URL http://ode.apache.org/. (Zitiert auf den Seiten 11 und 20)

[Foub] A. S. Foundation. External Variable: JDBC Mapping. URL http://ode.apache.org/extensions/external-variables-jdbc-mapping.html. (Zitiert auf Seite 20)

[Fouc] A. S. Foundation. External Variables. URL http://ode.apache.org/extensions/external-variables.html. (Zitiert auf Seite 20)

[Foud] E. Foundation. BPEL Designer Project. URL https://eclipse.org/bpel/. (Zitiert auf Seite 52)

[Foue] E. Foundation. Eclipse. URL https://eclipse.org/home/index.php. (Zitiert auf Seite 16)

[Fow] M. Fowler. GUI Architectures. URL http://martinfowler.com/eaaDev/uiArchs.html. (Zitiert auf Seite 17)

[Gro09] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 1 Auflage, 2009. (Zitiert auf den Seiten 17 und 28)

[JKP+04] J. C. Jacob, D. S. Katz, T. Prince, B. G. Berriman, J. C. Good, A. C. Laity, E. Deelman, G. Singh, M.-H. Su. The montage architecture for grid-enabled science processing of large, distributed datasets. 2004. (Zitiert auf Seite 43)

[KL08] O. Kopp, F. Leymann. Choreography Design Using WS-BPEL. *IEEE Data Eng. Bull.*, 31(3):31–34, 2008. (Zitiert auf Seite 11)

[Maj] B. Majewski. A Shape Diagram Editor. URL http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html. (Zitiert auf Seite 17)

[Mic] Microsoft. Layered Application. URL http://msdn.microsoft.com/en-us/library/ff650258.aspx. (Zitiert auf Seite 52)

[OAS07] 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.html. (Zitiert auf den Seiten 7, 15, 20 und 39)

[Ora] Oracle. Java Garbage Collection Basics. URL http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html. (Zitiert auf Seite 39)

[RHEA05] N. Russell, A. H. M. ter Hofstede, D. Edmond, W. M. P. van der Aalst. Workflow Data Patterns: Identification, Representation and Tool Support. In *Proceedings of the 24th International Conference on Conceptual Modeling*, ER'05, S. 353–368. Springer-Verlag, 2005. (Zitiert auf Seite 20)

[SBPM09] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd Auflage, 2009. (Zitiert auf Seite 17)

[SK13] M. Sonntag, D. Karastoyanova. Model-as-you-go: An Approach for an Advanced Infrastructure for Scientific Workflows. *Journal of Grid Computing*, 11(3):553–583, 2013. (Zitiert auf Seite 44)

[Son13] O. Sonnauer. *Modellierung von Scientific Workflows mit Choreographien.* Diploma thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2013. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3429&engl=1. (Zitiert auf den Seiten 9, 16, 52, 59 und 71)

[Ull10] C. Ullenboom. *Java ist auch eine Insel: Das umfassende Handbuch (Galileo Computing).* Galileo Computing, 2010. (Zitiert auf den Seiten 16 und 17)

[W3Ca] W3C. XML Schema Tutorial. URL http://www.w3.org/TR/2008/REC-xml-20081126/. (Zitiert auf Seite 12)

[W3Cb] W3C. XML Schema Tutorial. URL http://www.w3schools.com/schema/default.asp. (Zitiert auf Seite 13)

[W3C05] W3C. Web Services Choreography Description Language Version 1.0, 2005. URL http://www.w3.org/TR/ws-cdl-10/. (Zitiert auf den Seiten 19 und 39)

[WGSL09] M. Wieland, K. Görlach, D. Schumm, F. Leymann. Towards Reference Passing in Web Service and Workflow-based Applications. In *Proceedings of the 13th IEEE Enterprise Distributed Object Conference (EDOC 2009)*, S. 109–118. IEEE, 2009. (Zitiert auf den Seiten 19, 26 und 46)

[WK14] A. Weiß, D. Karastoyanova. Enabling coupled multi-scale, multi-field experiments through choreographies of data-driven scientific simulations. *Computing*, S. 1–29, 2014. (Zitiert auf den Seiten 6, 13 und 14)

Alle URLs wurden zuletzt am 11. 01. 2015 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, Unterschrift