Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3633

Specification and Development of Choreography Fragments for a Choreography Designer

Joas Schilling

Course of Study: Informatik

Examiner: Jun.-Prof. Dr.-Ing. Dimka Karastoyanova

Supervisor: M.Sc. Wirt.-Inf. Andreas Weiß

Commenced: March 4, 2014

Completed: September 4, 2014

CR-Classification: D.1.7, D.2.11, H.4.1, I.3.4

Abstract

This thesis specifies choreography fragments. Also the process of extracting them from an existing choreography as well as importing them into another choreography is defined. Then these choreography fragments are implemented for a choreography designer, that was written by Oliver Sonnauer. The implementation also connects the choreography designer with a repository for fragments called Fragmento, which can be used to version, share and reuse fragments easily.

Contents

List of Figures

List of Listings

List of Acronym

API – Application Programming Interface

APP – Application

BPEL – Web Service Business Process Execution Language, also named WS-BPEL

BPEL4Chor – BPEL for Choreographies

EMF – Eclipse Modeling Framework

GEF – Graphical Editing Framework

GMF – Graphical Modeling Framework

ID – Identifier

IDE – Integrated Development Environment

MVC – Model-View-Control

PBD – Participant Behaviour Description

RCP – Rich Client Platform

SWT – Standard Widget Toolkit

UI – User Interface

UML – Unified Modeling Language

WS – Web Service(s)

WSDL – Web Services Description Language

XMI – XML Metadata Interchange

XML – Extensible Markup Language

XSD – XML Schema Definition

1 Introduction

Today smartphone apps can be used, to order pizza and other fast food quite easily. The user makes his selection and the information together with his current location is sent to the server of the pizza delivery service. The server receives the order and notifies the kitchen about the order. Then the delivery cars are checked for availability and the price of the order and the approximated waiting time is sent back to the costumer.

A *workflow* like this can be modeled as a *choreography*. Choreographies can be seen as interaction of *business processes*, in the example above four such processes interact with each others:

- smartphone app
- server of the delivery company
- the kitchen
- the delivery cars

Now if the choreography for a similar scenario, e.g. a taxi company, is created, the complete choreography needs to be modeled from scratch again, although some parts of the choreography could be reused. Choreography fragments try to provide exactly this functionality. Parts of an existing choreography can be extracted and shared between users, so they can be reused, while modeling another choreography.

Two example parts that could be reused are listed below and can be seen in the following figures 1.1 and 1.2:

- Communication between the app and the company's server (blue section)

  In both cases, the app sends a request to the server, which then later responds with the price of the service and the approximated waiting time.

- Communication between the company's server and the delivery cars or taxis (red section)

  The server requests the status of all cars, selects the one with the closest location and shortest waiting time, and finally sends the details about the trip to the car.

The use of fragments can also help to structure choreographies and to implement patterns, e.g. for approvals of an action. But fragments also allow, to easily replace the implementation of a pattern, with another fragment, which implements the same pattern in a different way.

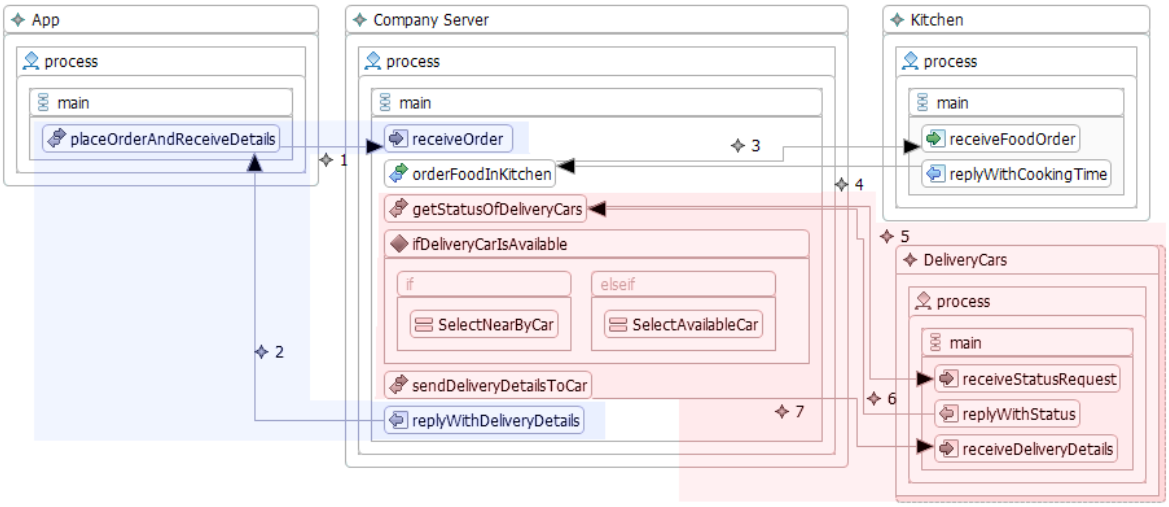

Figure 1.1: Pizza order scenario with the "app communicates with the server" part (blue) and the "get free delivery car" part (red)

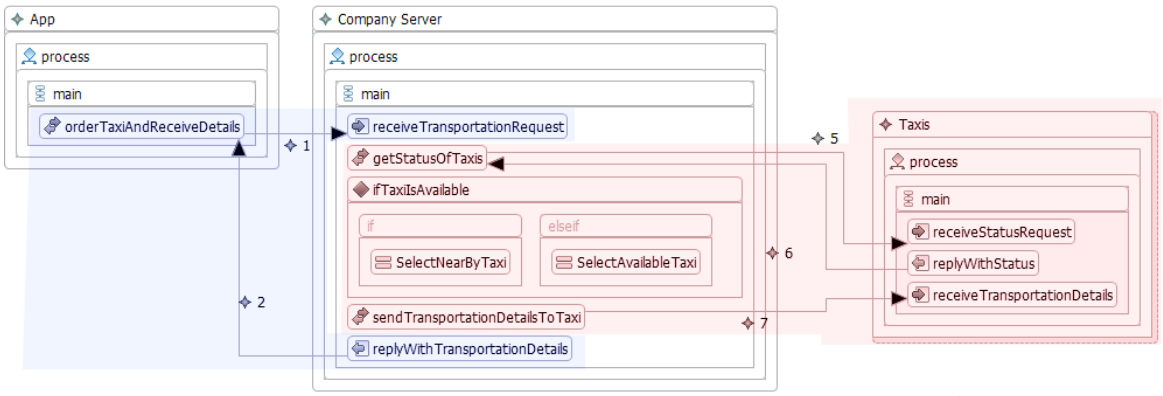

Figure 1.2: Taxi order scenario with the "app communicates with the server" part (blue) and the "get free taxi" part (red)

Goal of this Thesis

The goal of this thesis is, to define, what choreography fragments in general are and how they can be extracted from an existing choreography and be imported into another choreography. Afterwards, these choreography fragments should be implemented as a new plug-in for an existing choreography designer, that was written by Oliver Sonnauer. With the help of the new plug-in fragments can be exported and imported in the choreography designer, while the fragments themselves should still be editable in the choreography designer, so they can be easily modifies before being reused. Additional the fragments should be stored in Fragmento, an existing repository for process fragments.

Table of Contents

Chapter 2 – Basics In the basics chapter, the basic technologies and used tools are described.

Chapter 3 – Related Work The related work chapter gives a short overview on papers that have been written in the past, about choreography fragments or related topics.

Chapter 4 – Concept Choreography fragments and of the export and import of fragments from and into an existing choreographies are defined in the concept chapter.

Chapter 5 – Design In the design chapter then describes the basic structure and workflow of the new Eclipse plug-in. It also connects the formal definition of choreography fragments with the choreography model of the choreography designer.

Chapter 6 – Implementation The implementation chapter summarises the implementation of the plug-in.

Chapter 7 – Evaluation and Conclusion In the last chapter some tests and scenarios for the implementation are done, to outline potential problems with the implementation or proof that it is working as intended.

2 Basics

2.1 BPEL - Web Service Business Process Execution Language

BPEL, also known as WS-BPEL, stands for *Web Service Business Process Execution Language*. The 2.0 version of the language, that is defined as a standard since April 2007, allows to describe the behaviour of *Web Services*. BPEL itself is a XML[1]-based language. [OAS07b]

A BPEL document consists of *Partner Links*, *Variables*, *Correlation Sets*, *Handlers* and *Activities*. [LK12] *Activities* can be split up into two main groups: [OAS07a]

1. **Basic Activities** – Basic activities are simple activities that do not have an activity as a children in the XML-tree. Examples for basic activities are the *Reply* and *Receive* activities, which are used to send and receive messages between processes:

Listing 2.1 Sample of a *Receive* activity based on [OAS07a]

```
<receive partnerLink="NCName" operation="NCName" />
```

2. **Structured Activities** – Structured activities on the other hand are activities that have an activity in their descendants. An example for a structured activity is the *While* activity. With a *While* activity the descendant activitiy is repeated, until a given condition is not matched anymore.

Listing 2.2 Sample of a *While* activity based on [OAS07a]

```
<while>
       <condition>$receivedMessages < 100</condition>
       <scope>...</scope>
</while>
```

These activities are part of a process model which represents the process structure of one web service of a business.

[1]Extensible Markup Language – http://www.w3.org/XML/

2.2 BPEL4Chor - BPEL for choreographies and choreographies

BPEL4Chor[2] is an extension for BPEL, which allows defining and specifying choreographies. Choreographies are a collection of so called orchestrations or compositions, whereby an orchestration is the invocation of a web service inside of a BPEL process. Choreographies thereby focus on the interaction between multiple processes.

Additional to the activities from BPEL, BPEL4Chor also knows *Participants*, *Participant Sets* and *Message Links*. A participant represents a single process, whereby a participant set represents a set of equal participants that all implement the same process. An easy example would be a taxi company: the company's control system would be a participant and the taxis would be a participant set. The message links are the messages, that are sent between the different participants and participant sets.

BPEL4Chor itself is composed of 3 main artefacts [DKLW07]:

- **Participant Topology** – The topology describes which participants exist and how they communicate with each others.
- **Participant Behaviour Description** – The participant behaviour description (PBD) defines the control flow between the activities inside a single participant and between the different participants.
- **Participant Grounding** – The grounding holds all implementation specific data. This data is kept split from the other two models, so the details of the communication is not linked with the details of the model and choreography.

The choreography fragments that are implemented by this diploma thesis mainly cover the topology and behaviour description part of choreographies.

2.3 Eclipse

In the past couple of years the Eclipse[3] IDE[4] has been used to develop applications and tools in Java[5]. In version 3.0 Eclipse was changed to only contain the core features in Eclipse itself. Other features were loaded as plug-ins.

[2] BPEL for choreographies – http://www.bpel4chor.org/
[3] http://www.eclipse.org/
[4] integrated development environment
[5] https://www.java.com/

2.3.1 Choreography Designer

One of the many plug-ins for Eclipse is a Choreography Designer, that has been written by Oliver Sonnauer for his diploma thesis. [Son13] The plug-in allows creating a choreography scenario in a graphical way. The different activities and items are made available via a tool palette and can be added to the diagram editor with a simple mouse click. A basic screenshot of the designer can be seen in figure 2.1.

The choreography designer is based on the Graphical Modeling Framework (GMF).

2.3.2 GMF, GEF and EMF

The Graphical Modeling Framework is a framework itself is based on two other frameworks:

- **Eclipse Modeling Framework** (EMF) – EMF is a framework that provides a simple way to build tools and applications, that are based on a structured data model. The model can either be imported from an existing data model, e.g. a UML [6] file, or it can be generated using the UI[7] of EMF. The UI provides a tree view onto the model, whereby a property section can be used, to modify the details of the models. After the model has been imported or created, it is stored in a `.ecore`-file, which contains a XML Metadata Interchange (XMI) version of the model. [BSM+04]
- **Graphical Editing Framework** (GEF) – GEF, on the other hand, provides the user with an easy to use Model-View-Control (MVC) architecture.

GMF was developed, to combine the MVC architecture with the model from the EMF framework. Many projects tried to achieve this before. [BSM+04] GMF consists of 2 main components: the first component is a *tooling* component, which allows to define graphical elements and links them to the underlying model. The second component is a *runtime* component. The *runtime* component provides an API[8] for developing a graphical editor and also takes care of the connection between the EMF and GEF frameworks.

2.4 Fragmento

Fragmento[9] is a repository, which allows storing and retrieving of artefacts. Artefacts, as defined by Fragmento, can be e.g. process fragments for web service processes, deployment descriptors and other meta data. The different artefacts can be linked by relations, so a meta data artefact can be linked to the respective process fragment artefact. A set of linked artefacts and relations is called a bundle.

[6] Unified Modeling Language – http://www.uml.org/
[7] User Interface
[8] Application Programming Interface
[9] http://www.iaas.uni-stuttgart.de/forschung/projects/fragmento/start.htm

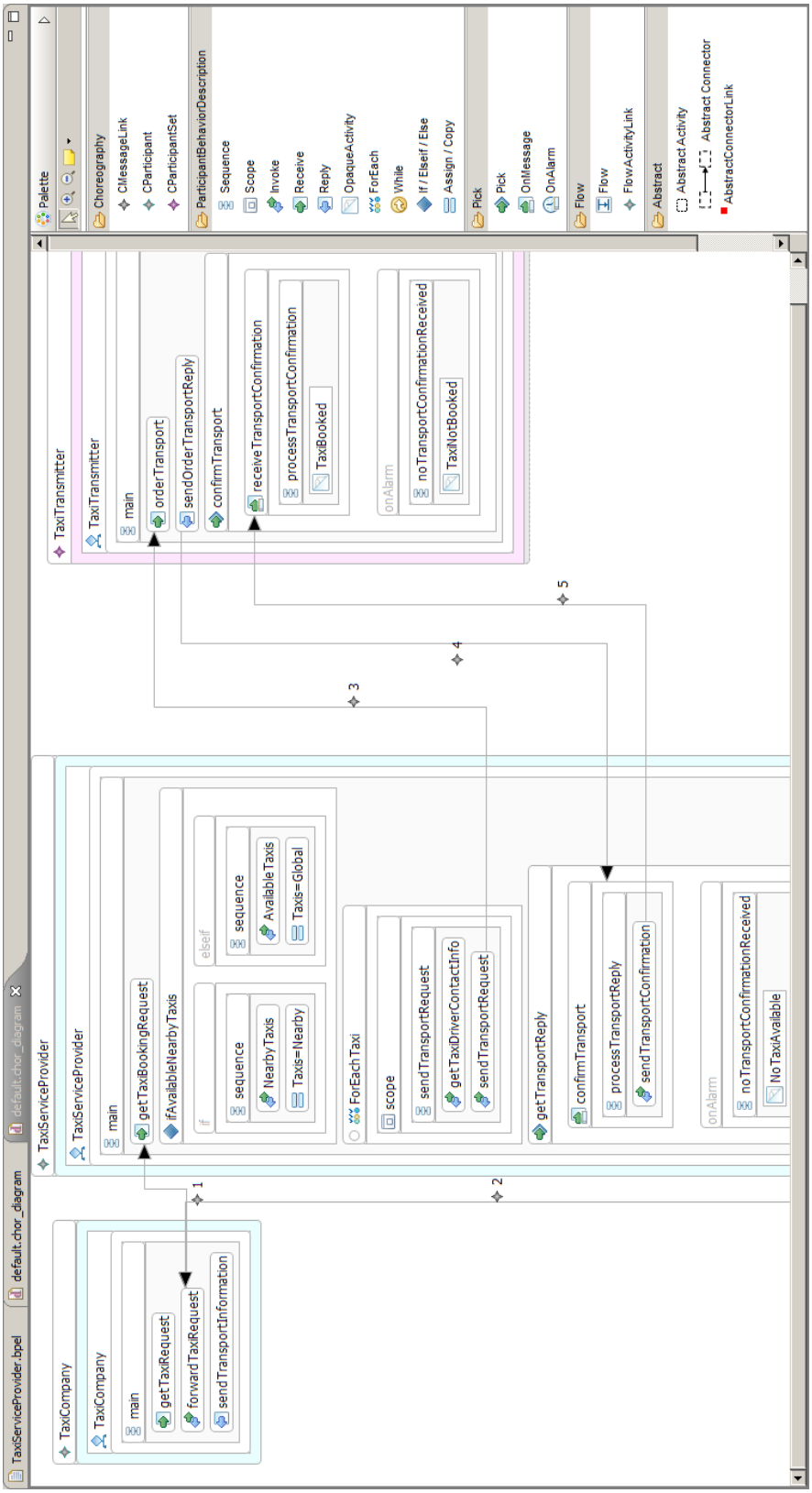

Figure 2.1: Choreography Designer plug-in for Eclipse with a diagram editor (left) and a tool palette (right) with activities, participants, etc.

Fragmento provides multiple web services, to interact with the repository, so artefacts can be added, modified and linked to each others. Fragmento itself is independent from Eclipse and the BPEL Designer[10], an eclipse-plug-in, that can be used to model web service processes. [EF14a]

The Eclipse integration of Fragmento was done in another Eclipse plug-in, by Dimitrios Dentsas in his student thesis [Den11] and is described in a more detailed way in section 3.

2.4.1 Extracting Process Fragments using the FragmentoRCP Plug-in

After an activity has been selected in the BPEL Designer, the FragmentoRCP plug-in provides two options to extract the fragment:

- **Option 1 – "Extract activity as fragment"**

  This option extracts the fragment into a new "*ProcessFragments*" project in Eclipse. If the project already exists, it's being overwritten. Inside this project the user can modify, extend or simplify the fragment.

  After all modifications have been made, the fragment can be pushed to Fragmento using the "Publish fragment to Fragmento" option in the context menu of the project. Now a wizard appears and collects the meta data for the fragment.

- **Option 2 – "Extract activity to fragment palette"**

  When this second option is used, the fragment, is stored in an "*export*" folder, which can be configured in the settings of the FragmentoRCP, using a wizard to collect the meta-data. The fragments stored in this folder are then displayed in the palette tool. From the palette the fragment can be simply inserted into a process.

  This allows reusing the fragment in the same or another process, while it is not being sent to and stored in Fragmento. But the user is also able to "Publish [the fragment] to Fragmento". This will send the fragment to the web service of Fragmento, similar to the second step of *option 1*.

Both of the options mention a wizard that is used to get the necessary meta data. The wizard consists of three or five steps, depending on the extraction option that is being used.

1. First the meta data about the fragment itself can be entered. The available fields are: name, author, description and file.
2. The second step of the wizard allows selecting an icon for the fragment. This icon is then displayed next to the name in the tool palette.
3. *Option 1 only* (see previous list) – In this step additional WSDL files can be added to or removed from the fragment.

[10] http://www.eclipse.org/bpel/index.php

4. *Option 1 only* (see previous list) – Similar to the third step in this fourth step additional XSD files for the fragment can be managed.
5. In the final step the deployment descriptor of the fragment can be selected and described.

In order to be able to import a fragment into an existing process, it needs to be in the export folder of the FragmentoRCP plug-in. This can either be achieved by using *Option 2* for extracting the fragment or using the "Export selection to file system" option from the "Repository View" of the FragmentoRCP plug-in. When the fragment is then dragged into the process, a second wizards helps inserting the fragment into the process.

This second wizard helps resolving conflicts between existing elements and new elements that are being inserted by the fragment. A list of possible conflicting items can be found in the section about the Fragmento Integration in the related work chapter 3.

3 Related Work

Process Fragment Libraries for Easier and Faster Development of Process-based Applications

The article "Process Fragment Libraries for Easier and Faster Development of Process-based Applications" [SKK+11] by Schumm et al. focuses on process fragments, their annotations and what the advantages of process fragments are.

A process fragment is defined as a connected sub-graph of a process. Thereby two ways to generate such process fragments are described. The first way is the "top-down" approach, where a sub-graph is extracted from a given process graph. The second approach is called "bottom-up". In this approach the fragments are created from scratch.

These fragments however are not always directly executable. There may be undefined activities (placeholders), the context (variables) for activities might be missing as well as the start and end point of the process. It is also possible that the fragment contains incoming or outgoing control edges. They are called fragment entry for the incoming edges and fragment exit for the outgoing edges. While a single entry and single exit (SESE) structure is also possible, the definition is not limited to this structure and can have multiple entries and exits.

The main advantages of process fragments are:

- Reuse of the process fragment itself.
- Can be used as an annotation to a process to explain how a service or process can be used or interacted with.
- Reuse of the fragment and its fragment counterpart as a collaboration. A fragment counterpart is a fragment that interacts with another fragment.

The article also introduces a "process fragment library". In this process fragment library fragments can be stored, even as a version controlled item with history. Other users of the same library can then search for fragments which fit their needs, retrieve them and update them aswell. This collaboration on the same fragments can help improving their quality of the fragments and also the quality of the design of the process itself.

Generating a counterpart for the interaction with an existing service or process can be helpful to explain how the service or process can be used. This is especially helpful when the internal structure of the service or process is considered a business secret and should not be exposed to business partners. The article defines three different visibilities on a process by a business partner. "White Box" describes a visibility where all details are visible to the business

partner. “Gray Box” hides the most details and only makes the basic behavior as well as the communication relevant information visible to the business partner. The last visibility is called “Black Box”. In a Black Box all details of the process are hidden and only the communication relevant information is visible to the business partner. However a fragment of this most basic and most simple visibility of the process aswell as it’s counterpart can be very helpful for business partners, when modeling their own processes.

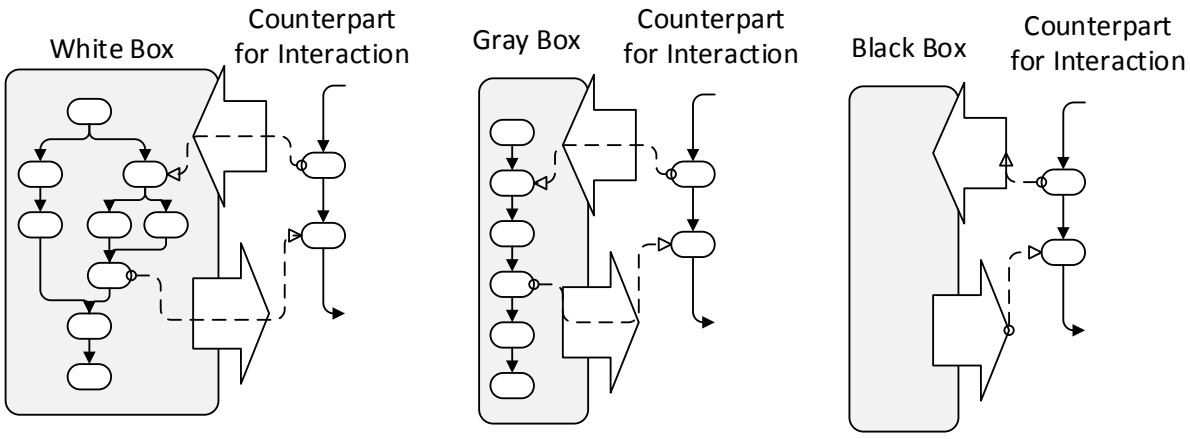

Figure 3.1: White Box (left), Gray Box (center) and Black Box (right) samples of a service or process and its interacting fragment counterpart, based on Fig. 11 of [SKK+11]

In section 4.3, the article then extends the concept of annotations to allow reusing process fragments and process fragment counterparts as “process fragment choreography”. These “process fragment choreographies”, however, focus on the interconnection model of the processes. So fragments that are extracted from the processes only contain the parts relevant for the collaboration between the different processes.

However, these choreographies and the involved fragments can be used to build a replacement for a process. The new process only needs to implement the corresponding fragment counterpart. So these fragments and counterparts can be seen as a basic structure, similar to interfaces for classes in several programming languages. Figure 3.2 shows a quick example of a choreography with 3 participants `A`, `B` and `C`. The participants are then extracted into `Role A`, `Role B` and `Role C`. Any participant that can be extracted into the same role, can be used to replace the respective participant in the choreography.

The definition of “process fragments” is used as a base for “choreography fragments” in section 4.1. Also “process fragment choreographies”, as described by this work, are the first part of choreography fragments, so the definition and description of choreography fragments will be factored into the final definition later.

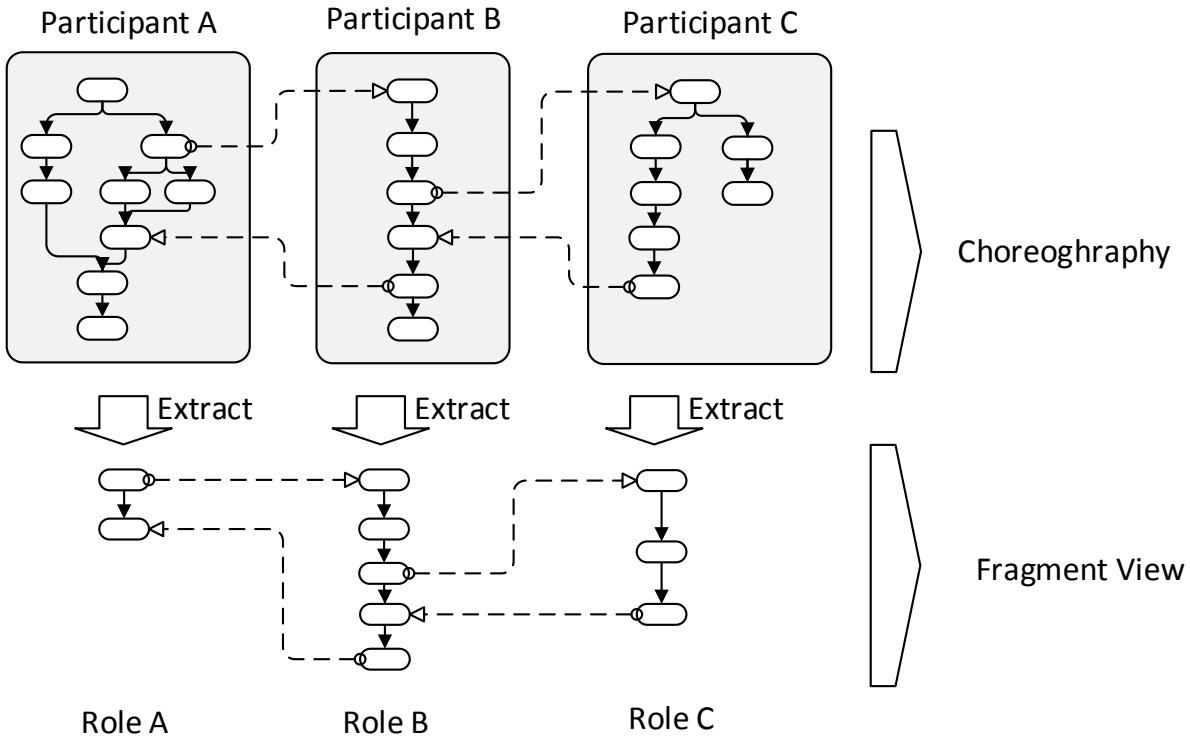

Figure 3.2: Process fragment choreographies describing a collaboration scenario, based on Fig. 12 of [SKK+11]

Choreographies as Federation of Choreographies and Orchestrations

The article "Choreographies as Federation of Choreographies and Orchestrations" [ELT06] by Johann Eder, Marek Lehmann, and Amirreza Tahamtan brings up another use case and view on choreography fragments. Choreographies are seen as federations of process models. A simple purchasing choreography serves as a good example to point out the details of the federation concept.

The buyer:

- orders something at a seller,
- pays by credit card
- and receives the details from a shipper.

The seller:

- receives the order
- checks availability
- organizes the shipping

The shipper:

- receives the shipping order
- ships the ordered good
- sends the shipping cost details to the sender

The seller and credit card provider of the buyer realize their own communication protocol. The same applies for the seller, shipper and the banks of those two. Both of these choreographies are independent from the buyer. The following figure 3.3 shows the idea of federated choreographies:

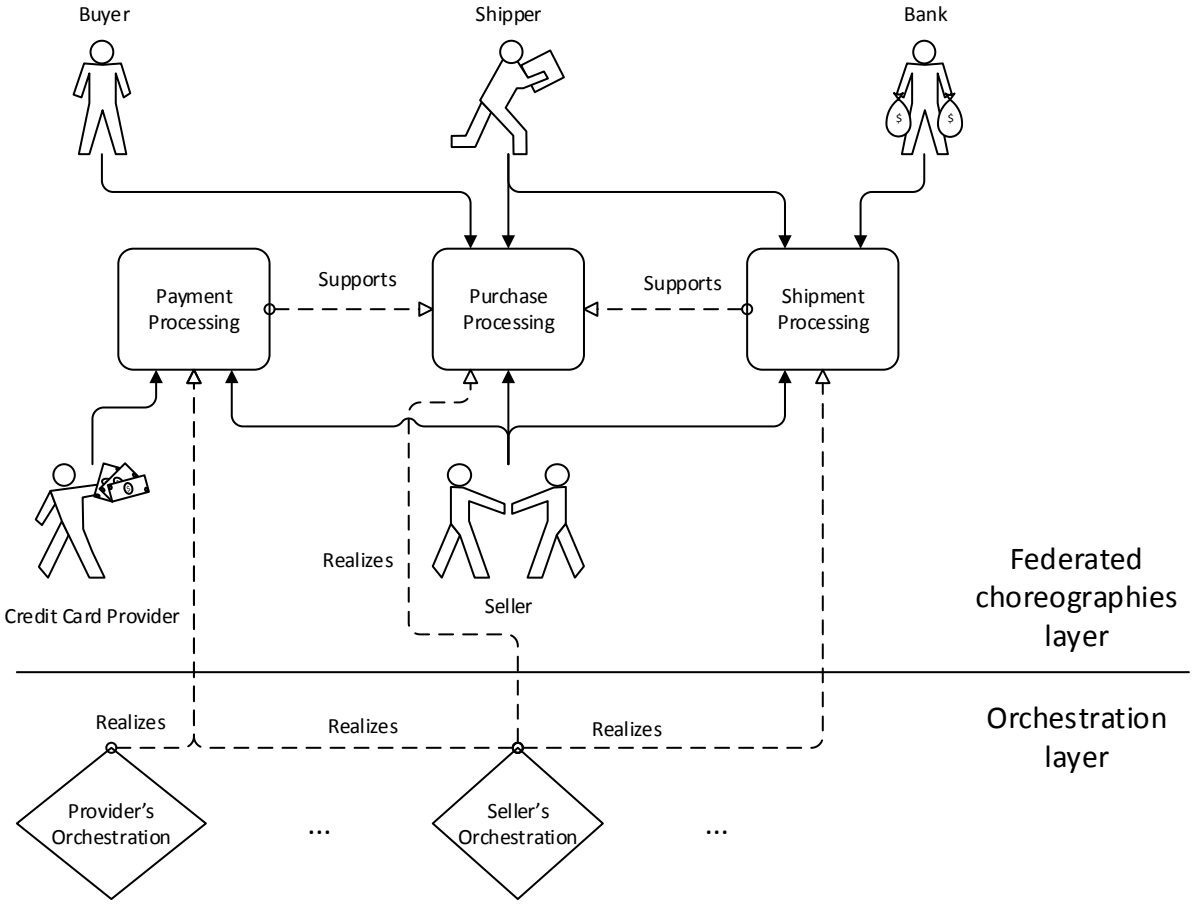

Figure 3.3: Federated choreographies, based on Fig. 1 of [ELT06]

The figure also shows that on the partner's end all choreographies a partner takes part in, are part of one orchestration. Splitting up a big choreography and federating them for the final choreography, allows partners to hide details from a third partner, that does not take part in the smaller choreography. However, these choreographies "support" the main choreography, which means that they either contribute to or elaborate it.

With this method multiple choreographies can be federated again and again into more complex choreographies, were each subsequent result choreography can be stored and reused in another federated choreography later.

Each of the partners taking part in a choreography has the same model of the choreography, but its own orchestration to implement the details. To stay compatible with each others, the choreographies and orchestrations must maintain and retain all assumptions made by choreographies they support and realize.

At the end two ways to help modularizing choreographies are described. The first one has already been mentioned before:

- **bottom-up** – Bottom-up means that choreographies are created from scratch. Then these choreographies are combined into more complex choreographies, maybe even multiple times.
- **top-down** – The second approach is top-down, like the well-known "divide and conquer" paradigm. In this approach an existing huge choreography is split up into multiple smaller choreographies. Similar to the first approach the resulting choreographies can be split up again and again.

Both of these ways will be implemented by two different components for the choreography designer. The top-down approach is covered by the extractor component, see sections 4.3 (concept) and 5.3 (design), while the bottom-up approach is implemented by the importer component, in sections 4.4 (concept) and 5.4 (design).

Compositional Choreographies

While the article "Compositional Choreographies" [MY13] by Fabrizio Montesi and Nobuko Yoshida defines a formal model for choreographies themselves, it also mentions choreography fragments as "partial choreographies".

The main problem with existing formal models is, that choreographies do not allow developing libraries that only implement a subset of roles required by a protocol, whereby a role can be a complete participant or a subset of a participant. This is a required step, in order to be able to reuse such existing libraries at a later point in a different choreography or protocol. Therefor they developed their own formal model with "partial choreographies". Partial choreographies allow defining a subset of the roles in an existing choreography, instead of requiring all roles to be defined and implemented.

The model allows omitting some roles. The number of participants in partial choreographies can span from one to unlimited in this model, so it is possible to model communication scenarios as well as single endpoint scenarios. These partial choreographies can then be later composed into the fully functional "compositional choreography".

For the authors these compositional choreographies have mostly two advantages, that are not covered by previous choreography models, which are explained on a short buyer-seller example:

- The buyer's and seller's web service choreographies and systems can not be developed independent from each others. They always need to define the roles of the seller and buyer respectively.
- As a result of the coupling between the buyer and the seller, the buyer can not select the most suitable seller company at runtime.

With the compositional choreographies the buyer can omit the *seller* role in his partial choreography. At runtime the choreography can then be composed with the most suitable seller's partial choreography. If a composed choreography does not implement the behavior of all available roles, it is just another partial choreography step on the way to the complete choreography.

While the definition of "partial choreographies" is pretty close to the implementation of "choreography fragments" that has been done in this diploma thesis, they differ at one point. Partial choreographies are considered incomplete and partial, until all roles and parts are defined, choreography fragments however need to be valid choreographies at every time, so they can still be opened and modified by the choreography designer.

Integration of Fragmento into a Rich Client Platform

In his student thesis "Integration of Fragmento into a Rich Client Platform" (original German title: "Integration von Fragmento in eine Rich Client Plattform") [Den11] Dimitrios Dentsas describes his steps to integrate the existing Fragmento[1] tool into the Eclipse[2] IDE.

While the thesis in general focuses on process fragments, similar to [SKK+11], it can still be compared to choreography fragments and quite a lot of points from the paper apply to both topics. Fragmento, which has been developed by the University of Stuttgart, is a repository that is being used to manage the process fragments. But generally speaking Fragmento is independent from process fragments and web service processes. It just stores artefacts and their relations, so it should also be possible to store choreography fragments in Fragmento.

The artefacts are basically just a XML document with a unique identifier, a artefact type, some additional meta data (like name, description, keywords, icon, ...) and a list of relations. At the moment the following artefact types are known to Fragmento: [Den11]

- A web service **process or process fragment** model in standard BPEL version or an extended BPEL Version
- A **WSDL** document

[1] http://www.iaas.uni-stuttgart.de/forschung/projects/fragmento/start.htm
[2] https://www.eclipse.org/

- A **deployment descriptor** for a process
- A WS-Policy **Annotations** document
- **View transformation rules**
- **Modeller Information**: Additional information for process modeling tools (e.g. graphical information like x/y coordinates for children of a parent activity)

The relations which are used to link two artefacts together consist of: [Den11]

- A **source activity**
- A **target activity**
- A **relation type**: e.g. annotation
- A **description**

A third item type that is known to Fragmento is *Bundles*. A bundle contains a process(-fragment) artefact and all its relations and related artefacts. These bundles can be stored and retrieved by Fragmento, so the user does not have to take care of the relations himself.

The integration into Eclipse FragmentoRCP consists of three main parts:

- A connector, called "Fragment Service", which interacts with the Fragmento installation and synchronises the artefacts and relations
- A view component, which contains the control options for the connector and also gives a tree view on the local stored artefacts and relations
- A BPEL Designer component, which allows adding fragments into the BPEL designer and extracting them from existing processes

The following figure 3.4 gives a good overview, how the Fragment Service is the connection component between the FragmentoRCP plugin and the Fragmento Repository.

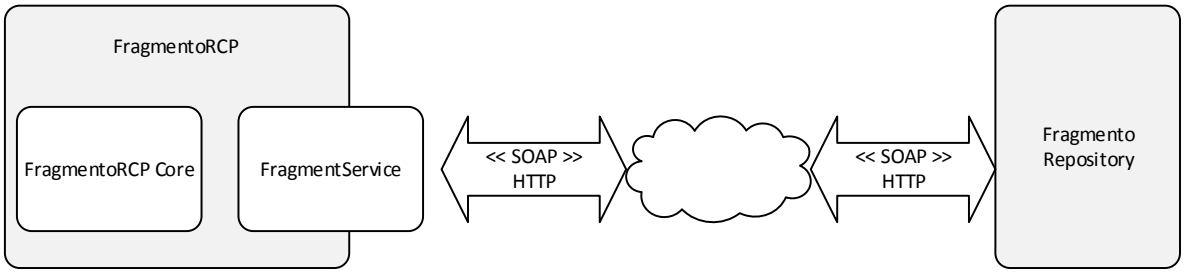

Figure 3.4: FragmentoRCP service component, based on Fig. 3.8 of [Den11]

The Fragment Service component is based on Apache Axis2 [3], a web services / SOAP / WSDL engine which is available in Java and C [ASF12]. Fragmento itself offers a web service for all operations on the artefacts and relations, e.g.: creating and receiving artefacts, browsing through the history of a artefact or the list of available artefacts, checkout and checkin of artefacts to update them. A full list of the available operations can be found on Fragmento website[4].

The options for interaction are also displayed in the view component, as shown in figure 3.5:

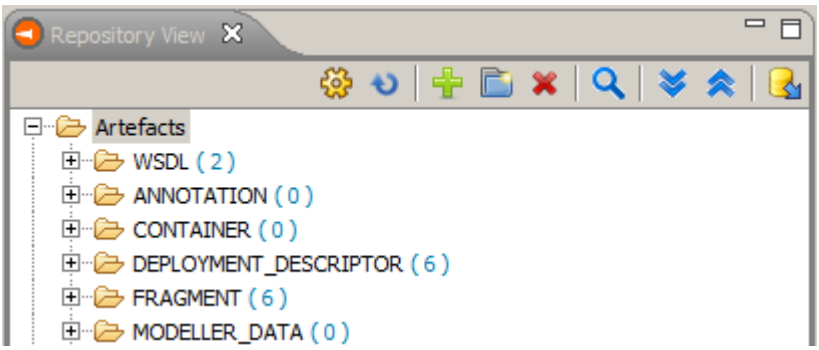

Figure 3.5: FragmentoRCP core plugin control options

- **Options** – Allows setting the Fragmento Repository URL as well as the selecting the local folders for the check-out of the repository and the target folder for exporting fragments
- **Reload** – Reloads the the content from the Fragmento Repository
- **Create new item** – Allows creating a new artefact and/or relation
- **Create new bundle** – Allows creating a new bundle
- **Delete from tree** – Deletes the selected items from the local tree view
- **Search** – Allows searching for specific artefacts, based on the author name, meta data and relations
- **Expand/Collapse all** – Expands or collapses all categories in the tree view
- **Export selection to file system** – Exports the selected artefacts or bundles to the filesystem, so they can be reused later

The artefacts retained from the services, by either a reload or a search request, are then represented in the view component. If an artefact is then extracted to the file system, using the "Export selection to file system"-option from the list above, it is stored in the selected

[3] http://axis.apache.org/axis2/java/core/
[4] http://www.iaas.uni-stuttgart.de/forschung/projects/fragmento/interfaces.htm and http://www.iaas.uni-stuttgart.de/forschung/projects/fragmento/wsdl.htm

folder. The content of the selected folder is then displayed in the tool palette option of the BPEL Designer.

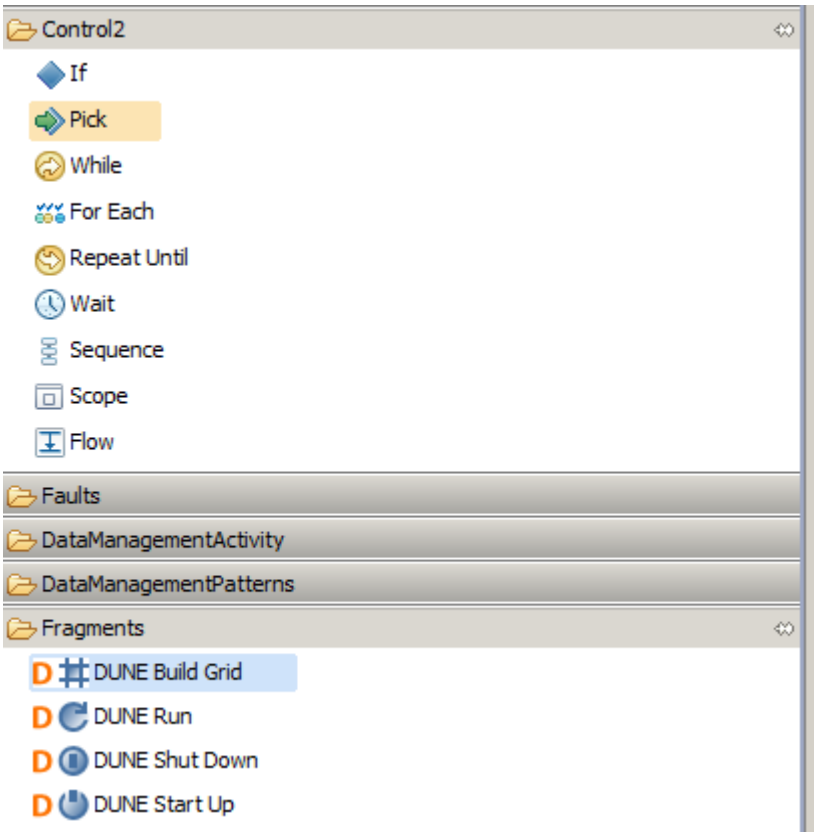

Figure 3.6: FragmentoRCP plugin integration into the tool palette of the BPEL Designer

From there the fragment can be drag-n-dropped into the process diagram, just like the normal process activities. When the fragment is dropped, a wizard opens up, which helps the user to:

- Resolve conflicts between existing items and items of the fragment:
  - Variables
  - PartnerLinks
  - CorrelationSets
  - MessageExchange
  - Extensions
- Importing WSDL and XSD documents

- Resolving conflicts of the imported WSDL and XSD documents and already existing ones

In order to extract a fragment from the existing process, a simple right click onto the desired activity/component is enough. A simple wizard retrieves the necessary meta data from the user, before the fragment is then stored onto the hard drive.

The implementation of the choreography fragment extraction and import will be developed similar to the implementation of this plug-in. This helps users getting used to the workflow. Also, if possible, the tree view with the artefacts should be reused, so the user can see all his artefacts, stored in the same Fragmento installation, in one view.

4 Concept

4.1 Definition of Choreography Fragments

In this section the definition of choreography fragments is being build, based on a definition for process fragments.

4.1.1 Definition of Process Fragments

The following definition of *process fragments* is based on the definitions from [SKLS10] and [SKK+11]:

Definition – A process fragment is a connected graph `G = (V, E)`. *Activities*, process beginning, process end and placeholders, so called *regions*, represent the *nodes*. The *control-flow* between the nodes represent the *edges* of the graph.

A process fragment must contain at least 1 activity or placeholder, the process beginning and end are optional. The number of edges between the nodes, as well as incoming and outgoing edges can also be 0. Additional process fragments may also have a defined context (e.g., variables).

4.1.2 Enhancement for Choreography Fragments

This definition can be enhanced to allow defining *choreography fragments*.

Definition – A choreography fragment is a graph `G = (V, E)`, which can be a connected graph, but doesn't have to. *Activities* and activity placeholders represent the *nodes*. *Edges* represent the control-flow between the activities and *MessageLinks* between the activities alike:

1. Edges type `I` (control-flow edges): These edges must only appear between nodes of the same participant and represent the control-flow between the defined activities. As these activities all have the same context, there is an *implicit data-flow* between the members of the same participant.
2. Edges type `II` (message-link edges): These edges must only appear between nodes of two different participants and represent the message-links that are sent between different activities/participants. As these messages contain information of any type, they represent an *explicit data-flow*.

3. Edges type `III` (data flow edges): These edges can appear between nodes of different participants. The data flow between nodes of different participants are directly linked to the edges of type `II`. Between nodes of the same participant the data flow represents the variables and values that are available in the participant.

Participants and *ParticipantSets* are not represented by a node or edge element in the graph. Instead they are represented by sets of nodes that are connected by edges of type `I` only. But not every set of nodes is also a Participant.

There has to be at least 1 activity, while the number of participants and edges of each type may also be 0.

This definition allows to represent activities as well as participants. Figure 4.1 shows the available elements while figure 4.2 shows a simple example with 2 participants.

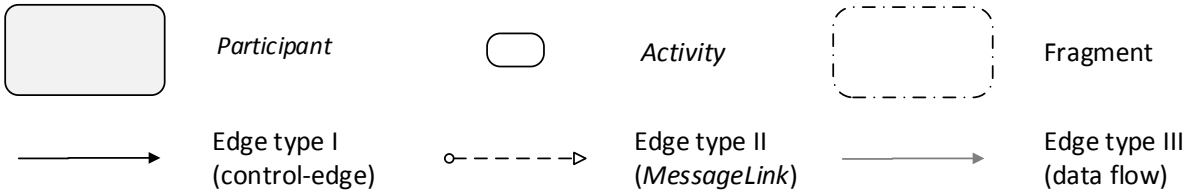

Figure 4.1: Choreography fragment constructs

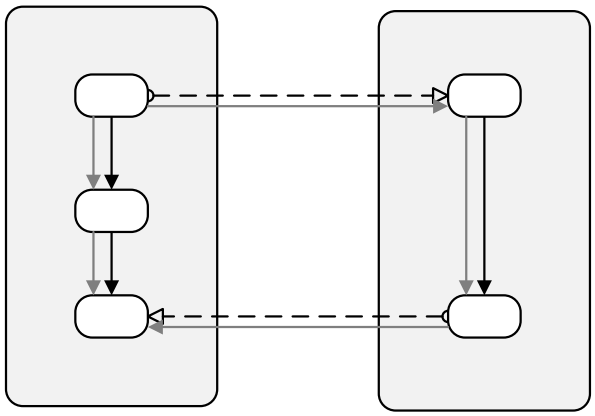

Figure 4.2: Simple example of a choreography fragment: two participants, with three and two activities, as well as message links at the beginning and end of the participants

Note: In the following figures the edges of type `III` are not shown, to keep the figures clearer and less crowded. There should be an edges of type `III` next to each edge of type `I` and type `II`.

Another point that is different in the definition of choreography fragments compared to the one of process fragments in section 4.1.1, is the missing restriction to a "connected graph". While, from a theoretical point of view, it would also be perfectly fine to restrict choreography

fragments to a set of connected activities and participants only, it also makes sense to allow non-connected parts in some cases, especially since message links without a start- or end-point are not allowed. The following example shows a simple scenario that results in a graph with two independent subgraphs:

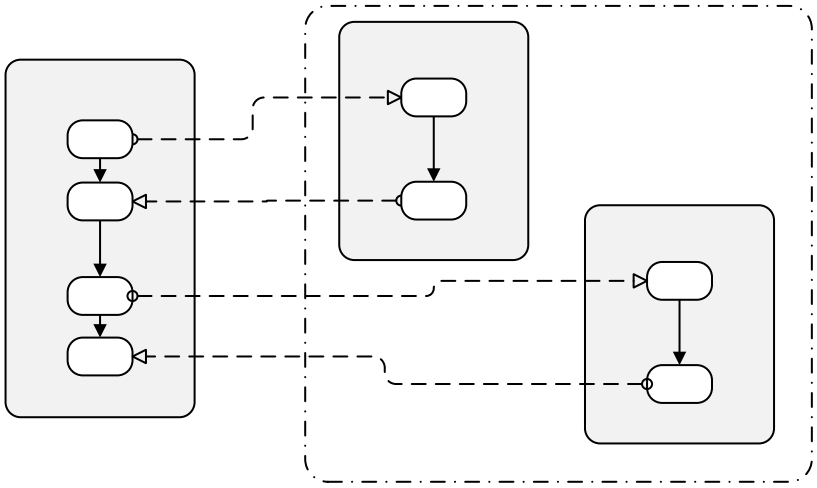

Figure 4.3: Participant of `Partner I` on the left, participants of `Partner II` on the right

- Two business partners, `Partner I` and `Partner II`, participate in one choreography.
- `Partner I` only has one participant, `Partner II` has two participants.
- Both participants of `Partner II` only communicate with the participant of `Partner I`.
- For the choreography fragment only the two participants of `Partner II` are extracted from the choreography.
- The resulting graph of the choreography fragment contains two sets of nodes that are not connected with each others, as can be seen in figure 4.4.

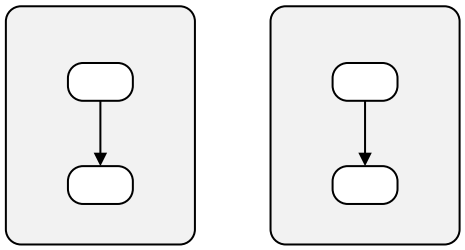

Figure 4.4: Resulting choreography fragment: two participants of `Partner II`, which are not connected with each others

4.2 Approval Sequence as an Example for Process and Choreography Fragments

In this section a simple real world example will be used to explain process and choreography fragments.

4.2.1 Prozess Fragments

Figure 4.5 shows a simple example of an approval sequence. It involves one node and has one incoming and two outgoing edges.

First the decision, whether the action requires approval, is made. In case the action has to be approved, the check is performed and the result propagated. Otherwise the check is skipped and the result is the same, as if the test would have been passed.

The process fragment does not mention how the check is performed, or which conditions have to be met, to avoid the check, if it is possible at all. These details have to be added, when the fragment is reused in a new process. It may also involve multiple activities and control-flow edges.

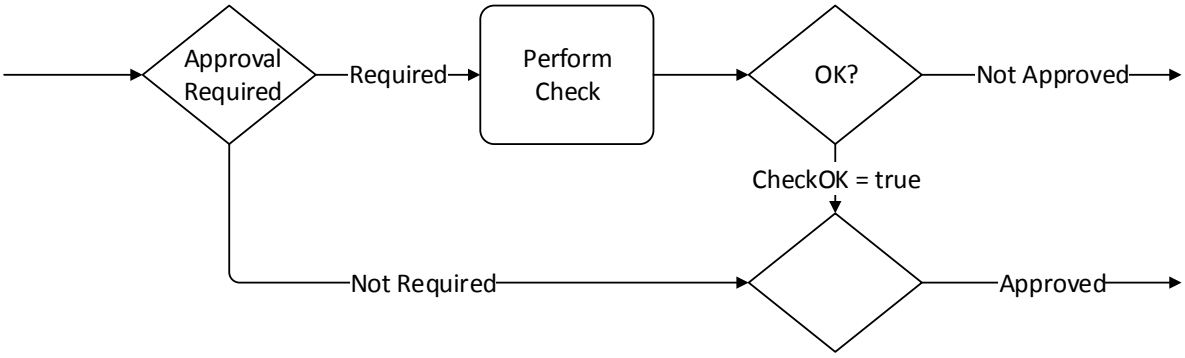

Figure 4.5: Process fragment for performing an approval [ART-2011-02-...]

4.2.2 Choreography Fragments

When implementing the same scenario as a choreography fragment, the "*Approval Requesting*"-Component and the "*Request Processing*"-Component might also be implemented by different *Participants*. One possible representation can be found in the following figure 4.6.

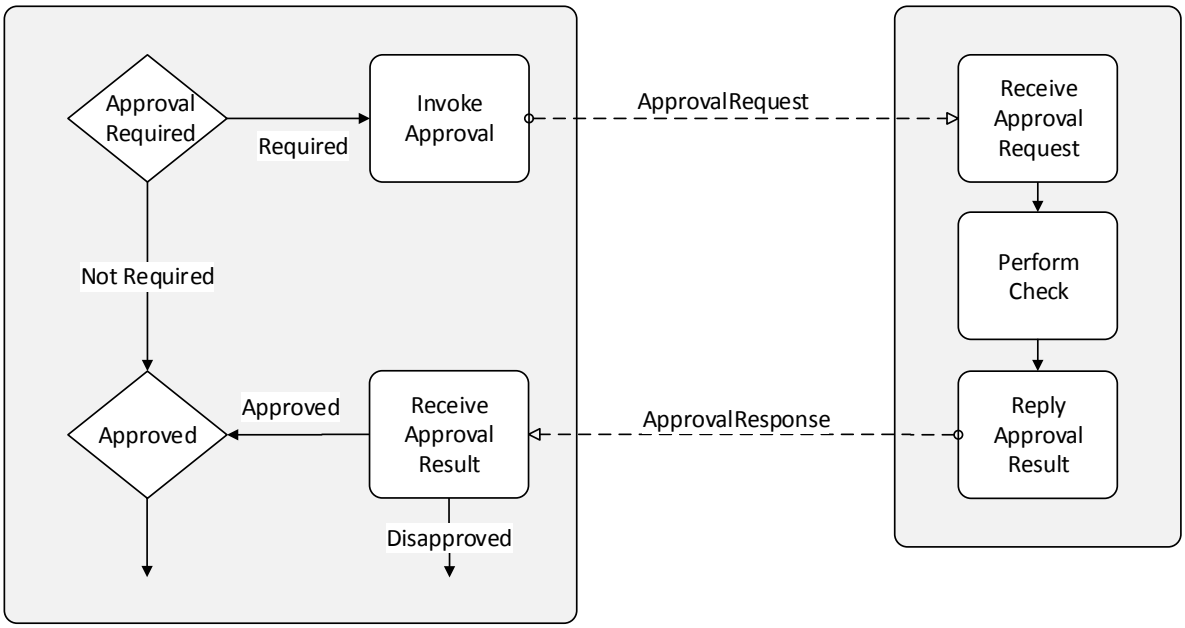

Figure 4.6: Approval sequence as part of a choreography with two processes

1. Complete Scenario as a Choreography Fragment

The choreography scenario from figure 4.6 itself is already a choreography fragment. The *Approval Requesting*-Part (left) and the *Request Processing*-Part (right) are implemented as participants. Additionally to the control-flow inside the two participants, two *MessageLinks*, representing the communication between the two participants, are included.

However, as the scenario contains multiple activities, it can also be split into smaller fragments. Some of the possible *sub-fragments* are listed below. But even these fragments can be split into multiple fragments, as a minimal fragment requires just one activity.

2. Execution of the Request and Processing of the Result

In figure 4.7 the whole logic of the approval requesting participant has been extracted into a fragment. This includes the decision whether approval is required at all, as well as the processing of the approval-request's result. As this fragment only contains activities and edges of type `I`, it also fulfills the requirements of a *process fragment*.

3. Sending and Receiving of the Request / Result

In figure 4.8 and figure 4.9 the sending and receiving activity, as well as the connecting message link have been marked as a fragment. These *connector* fragments are the lightest version of

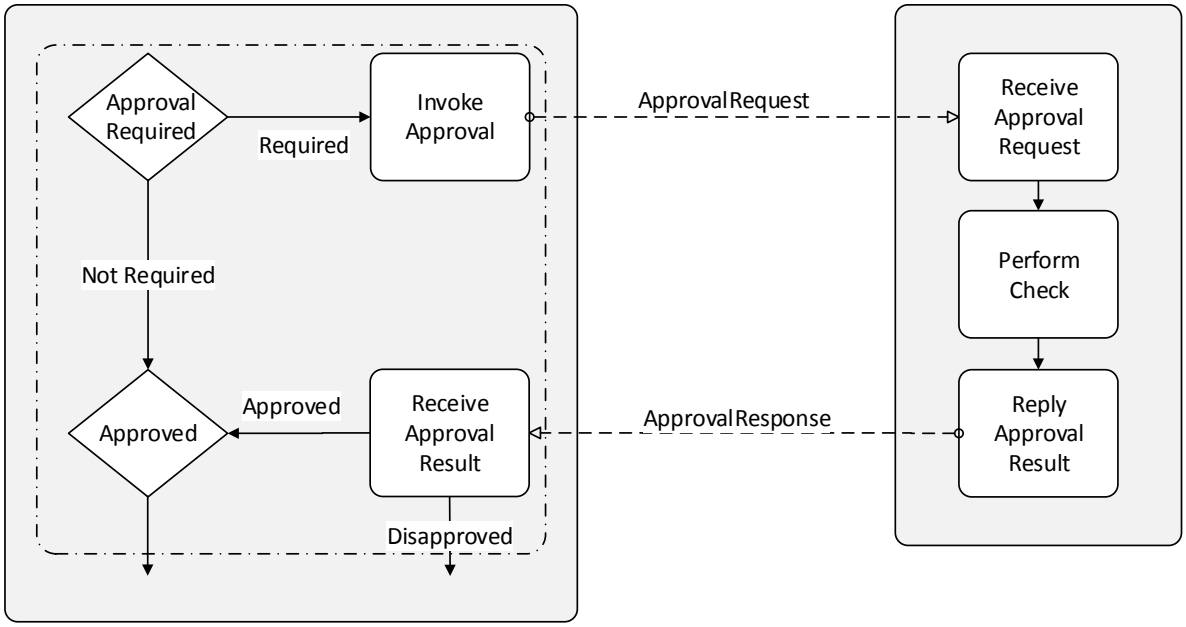

Figure 4.7: *Choreography fragment* for executing a request and processing of the response

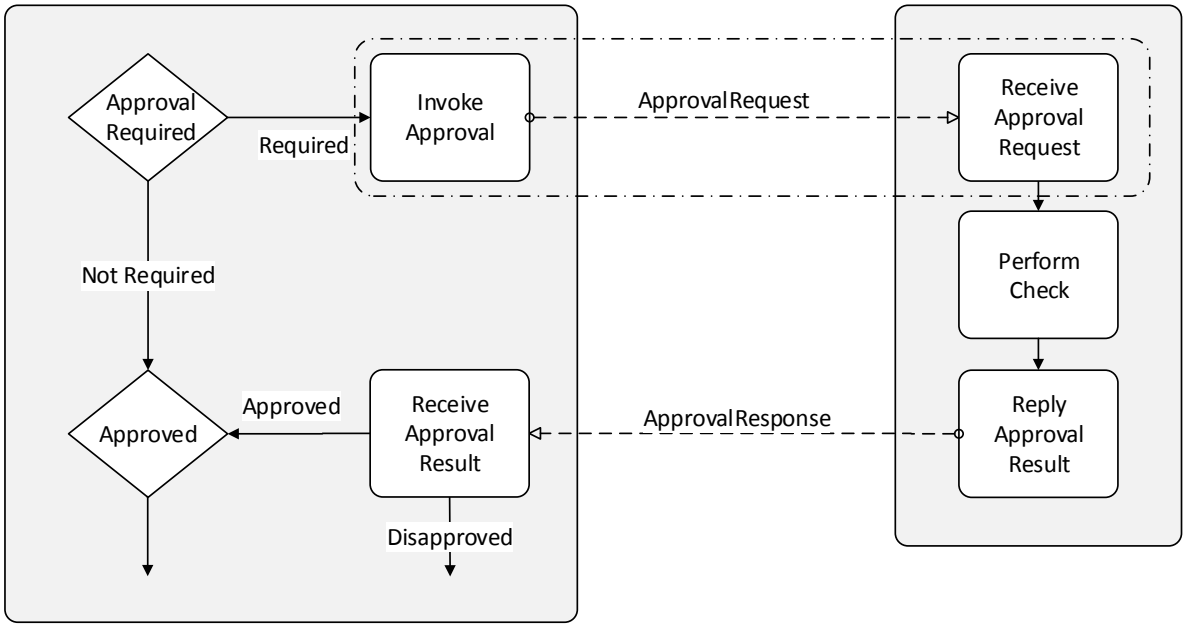

Figure 4.8: *Choreography fragment* for sending and receiving an approval request

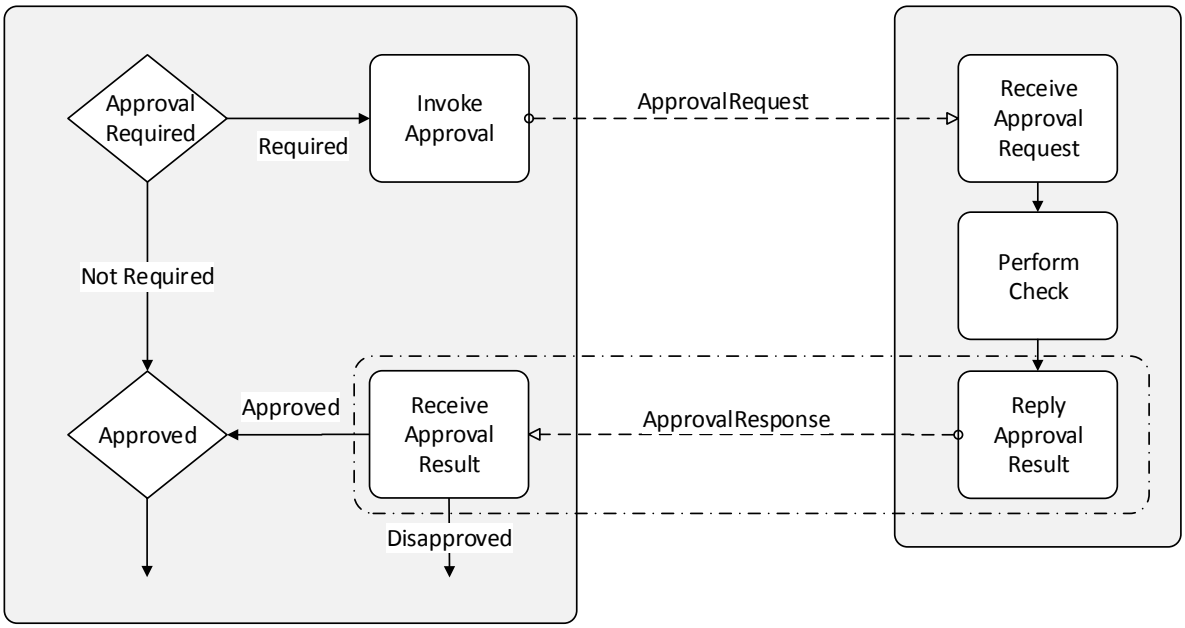

Figure 4.9: *Choreography fragment* for sending and receiving the response

a choreography fragment, that includes multiple participants. They only contain nodes and edges relevant for the connection of the two participants.

4. Processing Logic and Participant

Similar to the fragment of first example, figure 4.10 shows a fragment that contains all logic from the approving participant. This fragment also matches the process fragment requirements.

However, the participant itself can also be extracted into a choreography fragment, as seen in figure 4.11. This fragment does not match the process fragment definition, just like the connector examples form the previous section. The reason is, that process fragments may not contain participants (used here) or edges of type `II` (used in the connector example).

4.3 Extracting a Fragment from a Choreography Graph

Creating a choreography fragment from an existing choreography can be compared to copying the subgraph out of the original graph, as it has already been shown on the figures of the previous scenario and in section 4.1.2.

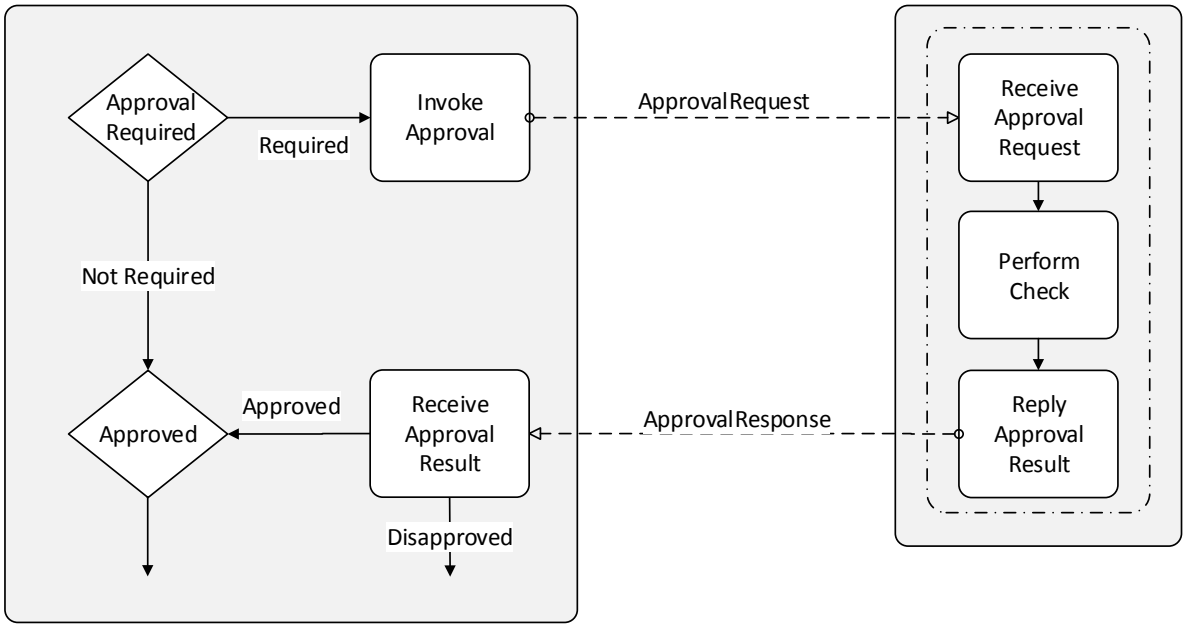

Figure 4.10: *Choreography fragment* for processing of the approval request

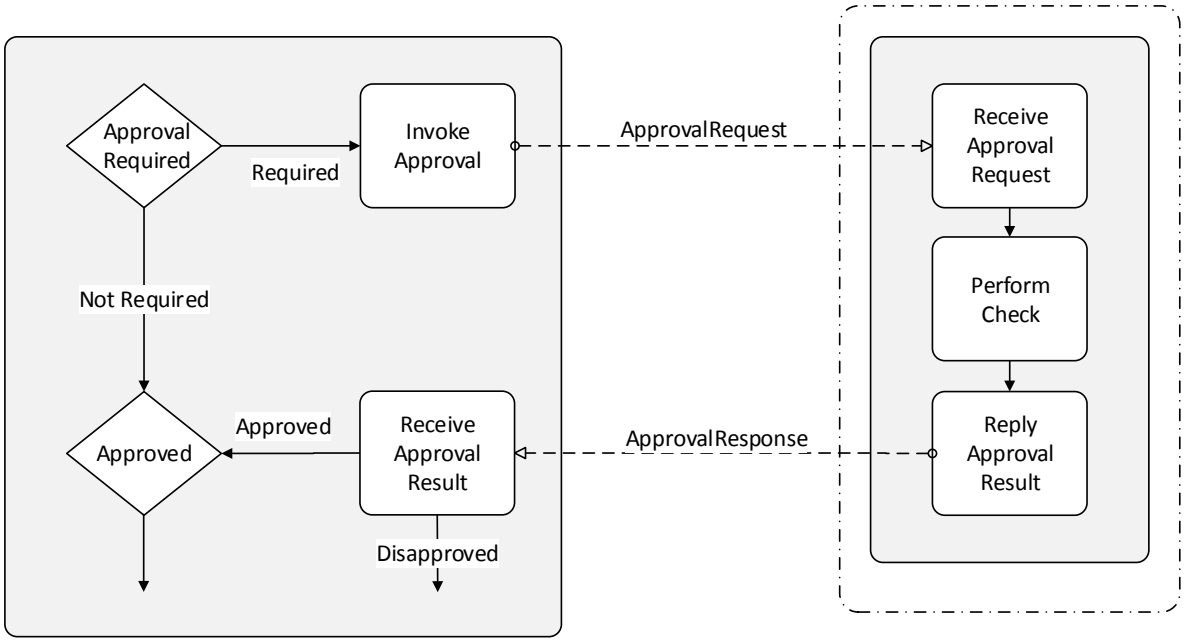

Figure 4.11: Complete approval *participant* as a *choreography fragment*

4.3.1 Extracting a Fragment from one Participant only

The first sample fragment shows a simple choreography fragment that only contains nodes from one participant. Therefore, this sample fragment also matches the definition of a process fragment. Any other participants can be ignored when extracting the fragment.

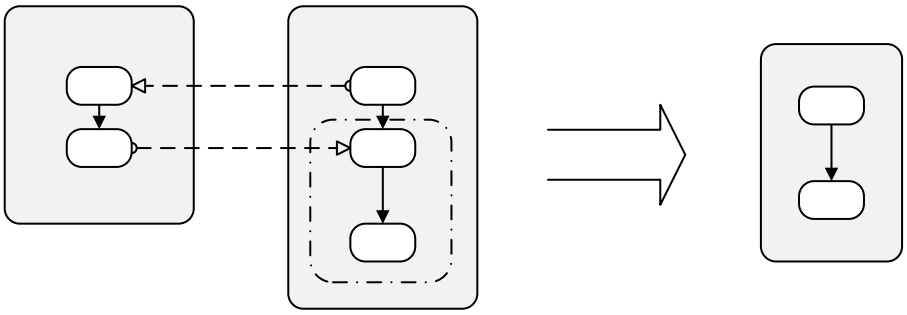

Figure 4.12: Extracting a choreography fragment from only one participant in a choreography

4.3.2 Extracting a connected Fragment from multiple Participants

The second sample fragment shows the extraction of two nodes from two different participants. The resulting choreography fragment still contains the edge of type `II`, which connected the two nodes in the original choreography.

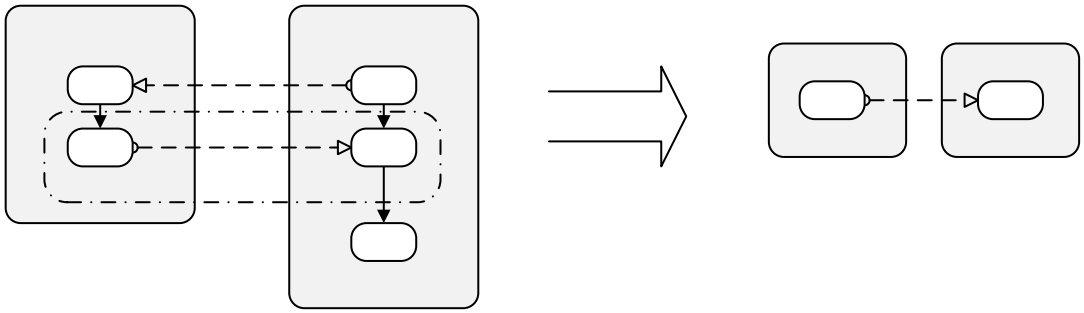

Figure 4.13: Extracting a choreography fragment from two participants in a choreography

4.3.3 Extracting a split Fragment from multiple Participants

The last sample fragment shows an extraction of multiple nodes from multiple participants. But since some of the nodes are not connected with edges, the fragment has two independent subgraphs.

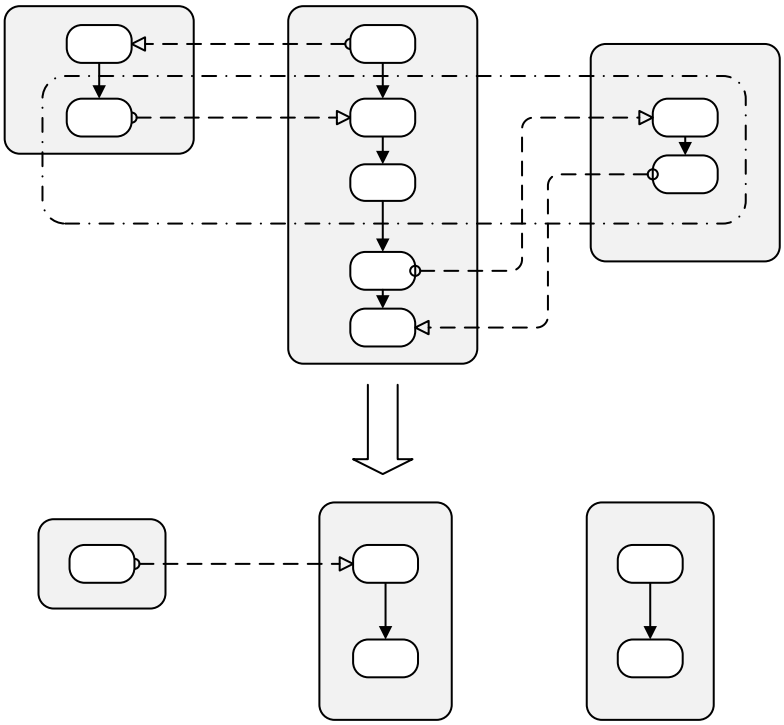

Figure 4.14: Extracting a disconnected choreography fragment from multiple participants, with two independent subgraphs

4.4 Importing a Fragment into a Choreography Graph

When importing a fragment into a choreography, the different fragments from above need to be inserted in different ways.

4.4.1 Importing a Fragment with one Participant only

The choreography fragment from the first sample in figure 4.12 can be easily imported and only needs one additional edge of type `I`, i.e., a control flow edge. It just requires the selection of the target participant. Then the existing edge of type `I` from the first to the second node is reused to connect the fragment to the connected subgraph of the participant. The additional edge of type `I` connects the end of the fragment with the currently unconnected, previously second node. Afterwards all nodes are connected again, as shown in figure 4.15.

4.4.2 Importing a connected Fragment with multiple Participants

Importing the second fragment, the connection example from figure 4.13, requires an additional edge of type `I` for each of the participants. Figure 4.16 shows that the edge of type `II`, that

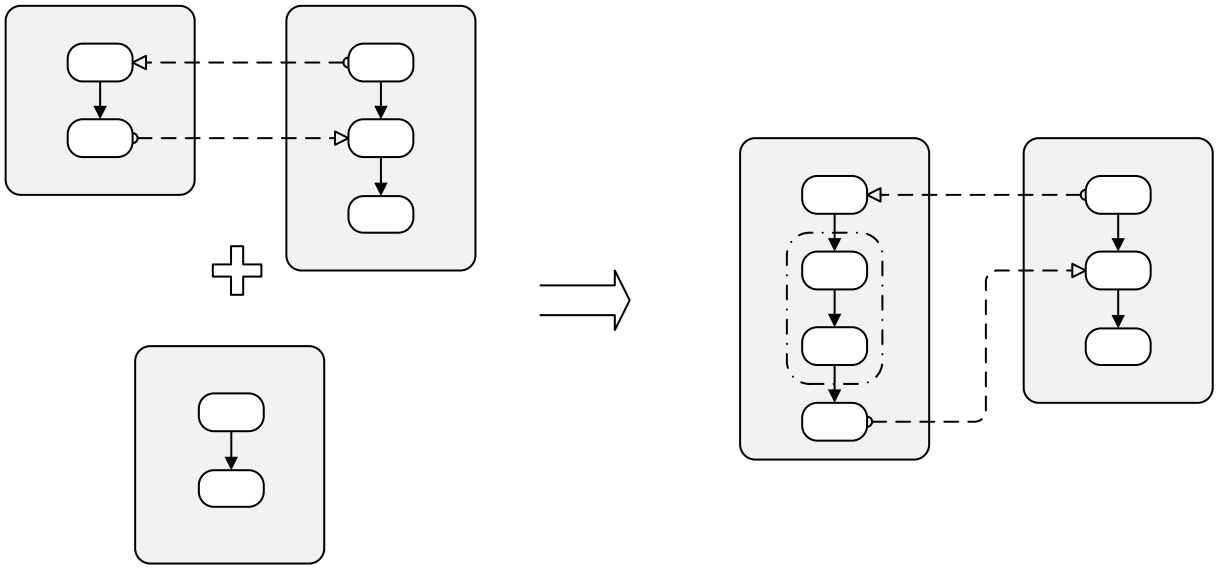

Figure 4.15: Importing a choreography fragment into one participant only

has been part of the fragment before, is also imported and still connects the two nodes that it has been connecting previously.

4.4.3 Importing a split Fragment with multiple Participants

The last extraction fragment from 4.14, containing the two disconnected subgraphs, can be imported in two steps. The left part is imported similar to the connection fragment from the second sample. However, after importing the right part of the fragment, it is not yet connected to the choreography. One way to connect the third participant of the fragment to the choreography is, to add edges of type `II` between nodes of one of the other participants and nodes of this one, as shown in figure 4.17. But, it is also possible to leave the participant as is, since the definition does not require the graph to be a connected graph. This has to be decided by the user.

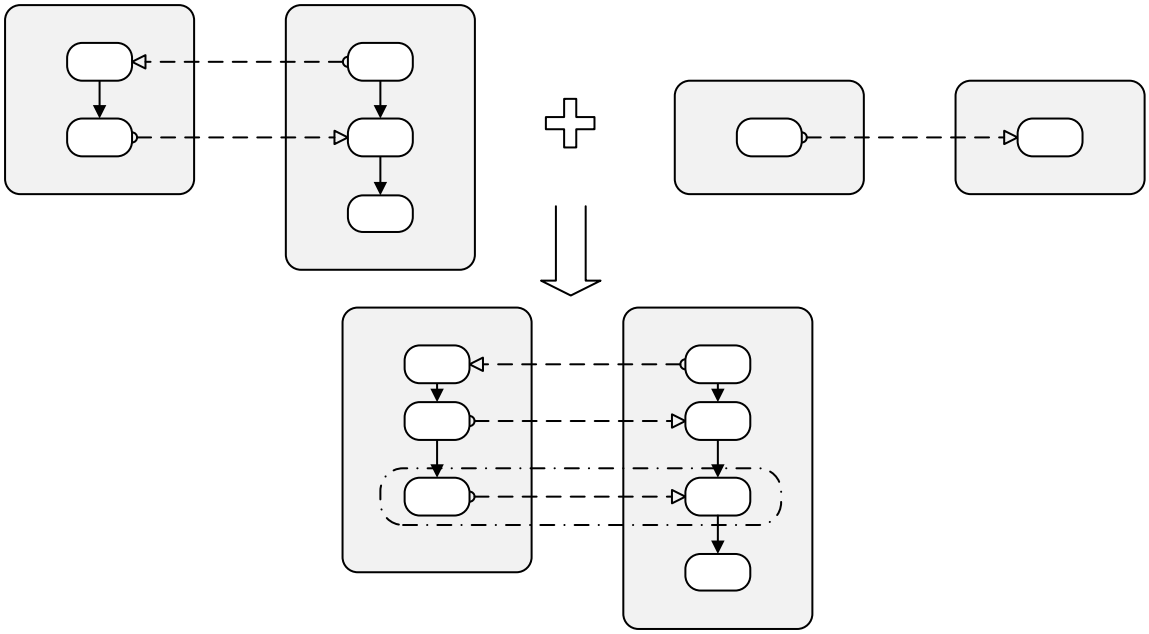

Figure 4.16: Importing a choreography fragment into two different participants

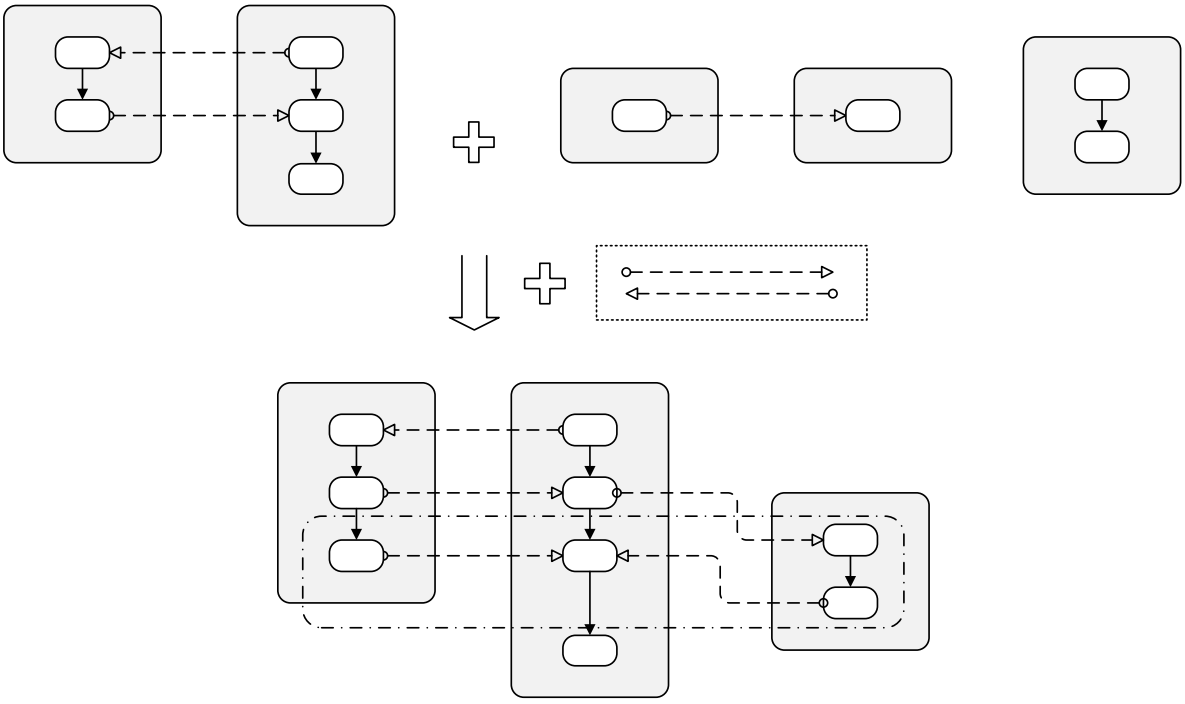

Figure 4.17: Importing a disconnected choreography fragment into two different participants and creating a new participant

5 Design

5.1 Choreography Fragment Requirements

Additional to the definition from section 4.1.2, another requirement is introduced. In order to be able to modify the choreography fragments with the existing choreography editor, after they have been extracted from a choreography, the fragments need to be stored as valid choreographies again.

This limitation breaks the compatibility with process fragments:

- Parts of processes that are not part of choreographies can not be represented:
  - Process beginning and process end
  - Variables
- Edges that are missing their start-point (incoming edges) or the end-point (out-going edges) can not exist. Instead a placeholder activity has to be added which represents the communication partner and completes the *connector*
- Also activities and processes themselves are not allowed anymore. They must not appear in a choreography fragment directly, but have to be wrapped by a process and a participant.
- And of course participants and participant sets should be allowed to inside a choreography fragment. But they also need a wrapping choreography around them, so they do not appear in the root of a choreography fragment.

These activities, processes, participants, participant sets and the choreography should be removed automatically, when the fragment is reused in a different choreography. So the user can pick the target activity, process, participant and participant set respectively.

There is also another problem for business partner with choreography fragments, if they would be restricted to a connected graph, that should be pointed out.

Whether the implementation of *Partner2* uses multiple participants or just one, is and should be of no interest for *Partner1*.

If the fragment would be required to be a connected graph, the participant of *Partner2* must be added to the fragment, so the two participants from *Partner1* are connected by (multiple) message links, although they do not communicate directly. However, when two different participants are used to implement the business logic of *Partner2* they need to be connected again, making the choreography and choreography fragment way more complex.

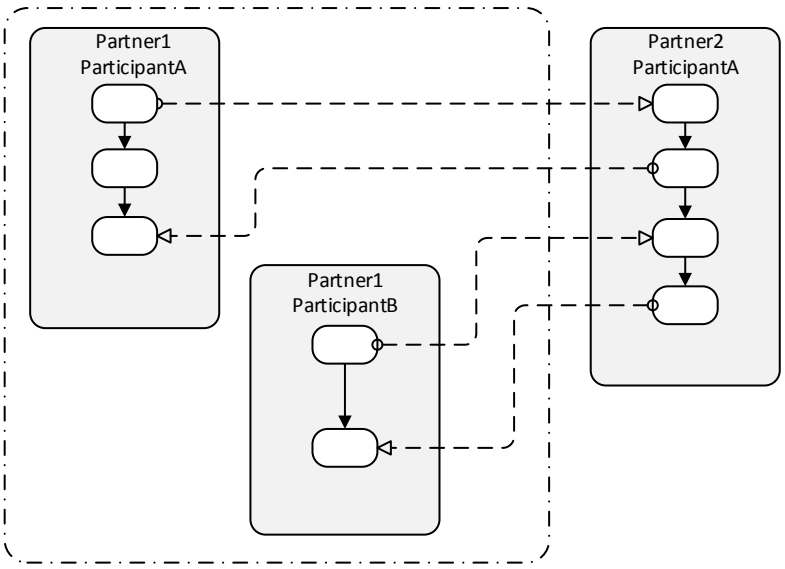

Figure 5.1: Example of a choreography with a disconnected choreography fragment

5.2 Structure of the Fragment Component

As seen on figure 5.2 the fragment component consists of three main components:

- **Extractor** – The extractor component is responsible for the extraction of a selection from an existing choreography. First the selection is retrieved from the editor. Afterwards the selection is verified, so no invalid items are selected. Then the fragment is created and stored in the fragment storage.
- **Storage** – The storage component manages the fragments themselves. New fragments can be added to the storage and existing fragments can be retrieved from it. The easiest storage component is the file system.

  However also other storages like Fragmento or even version control system, e.g. Git[1] or SVN[2], could be used to implement the storage component.
- **Importer** – The importer component retrieves a choreography fragment from the storage and finds the participants and activities that had been selected for extraction. For each of these fragment items a valid parent needs to be selected or created, before the item itself can be inserted into the existing choreography.

The following sections will describe the behaviour of the components more detailed.

[1] http://git-scm.com/
[2] Subversion - https://subversion.apache.org/

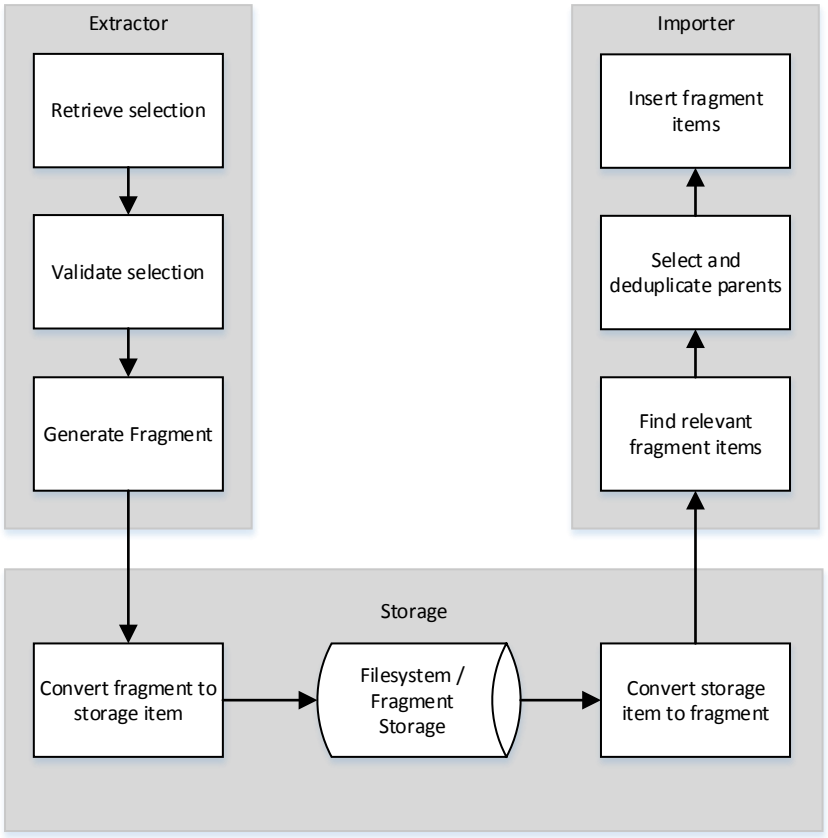

Figure 5.2: General structure of the fragment component

5.3 Extractor Component – Extracting a Choreography Fragment from an existing Choreography

When extracting a part of the choreography as a fragment, all details such as names, children, attributes and other settings should be kept. The easiest way to achieve this, is to clone the complete choreography diagram and model. However since the user can also select sub-parts of the choreography, activities and participants that are not in the user's selection need to be removed.

1. As the first step, items that are not needed for the fragment, are marked as "*To Be Deleted*". This applies to items, which neither are themselves part of the selection nor have a child item, that is part of the selection.
2. If an item itself has neither been selected nor is a child of a selected item nor is marked as "*To Be Deleted*", it is marked as "*Generated*". These generated items are the placeholder activities and participants that have been introduced in section 5.1, which help retaining the correct nesting of the choreography.

By marking the *unselected* parent activities and participants, they can be identified, when reinserting the fragment later into another choreography. Then the user can be asked, whether the parent should be created aswell, or whether a parent is already available.

3. After the items have been marked, *MessageLinks* which have an item that is marked as "*To Be Deleted*" as their startpoint or endpoint, need to be removed.
4. Then the "*To Be Deleted*" items can actually be deleted, as they are not referenced anymore.

Now the fragments choreography and model can be saved and only contain items, that are required to model the selected fragment or to retain its validity.

If the user selected an invalid item for extraction, the fragment creation should be denied. Invalid items are one of the following:

- **MessageLinks** – MessageLinks must not be selected directly. If both end points of a message link are selected, the message link itself is also kept and extracted. If one of the end points is missing from the selection, the message link is invalid and can not be kept.
- **Sub-items** – Items that are not activities themselves, but must always appear as a child of a certain activity. Examples for such sub-items are:
  - `Else` and `Elseif` which must always have a parent `If`
  - `OnMessage` and `OnAlarm` which must always have a parent `Pick`
  - `Copy` which must always have a parent `Assign`,

    accordingly `From` and `To` must always have a parent `Copy`
- **Empty selection** – At least one valid item has to be selected, in order to extract the fragment.

When this extraction feature is available, the "top-down" approach to generate choreographies, as described in [ELT06] and section 3 is completed.

A huge and existing choreography can be split up into smaller parts for every business partner. The choreography fragments, which are still valid choreographies, only contain the participants and activities relevant for their implementation of the choreography.

5.4 Importer Component – Importing a Choreography Fragment into an existing Choreography

When importing a fragment into an existing choreography, all details should be retained again, just like when extracting them.

1. In the beginning the items that are going to be imported need to be found. All items that do not have the "*Generated*" prefix from the exporter themselves, but have a direct ancestor with the "*Generated*" prefix should be imported into the *Choreography*, including their children.

2. For each of the fragment items, found by step 1, the user selects whether the parent of the item already exists in the *Choreography* or whether the "*Generated*" placeholder parent should be created in the *Choreography* aswell.

   The following options are only available for *Activities* of the fragment. Imported *Participants* and *ParticipantSets* are always inserted into the *Choreography* directly.

   Creating the parent – As the exporter retains all parents instead of only the necessary parents (*Participant(Set)*, *Process* and most outer *Sequence*) some of the parents are skipped while creating the parent path. Only the necessary *Participant(Set)*, *Process* and *Sequence* activity are going to be created and the *Activity* is added as a child of the *Sequence*.

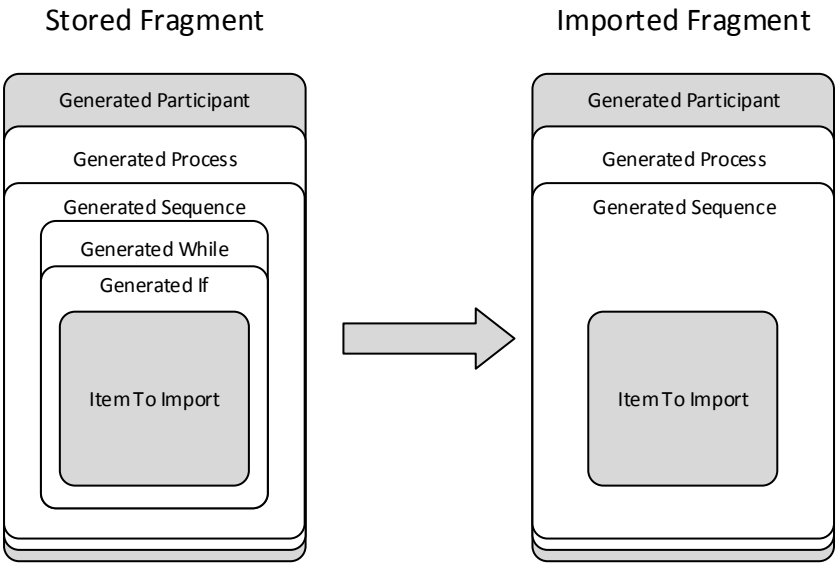

Figure 5.3: Difference between a stored and imported fragment. Parent activities that are marked as "*Generated*" are not being added, when the parent is created.

   Select the parent – The other option to insert an *Activity* is to select an already existing parent. In order to give the user all possible parents the current *Choreography* needs to be checked for the following *Activities*:

   - **Sequence** – Not only the *Process* –> *Sequence* [3] is allowed, but any *Sequence*.
   - **If** – *If* activities can be a parent as long as they do not have a child activity yet.

[3] A –> B means that B is a direct child of A

The same applies to the sub-items *Else* and *Elseif*.

- **While** and **RepeatUntil** – Similar to *If* a *While* and *RepeatUntil* can be parent, as long as they do not already have a child activity.
- *Pick* –> **OnMessage** and *Pick* –> **OnAlarm** – The *OnMessage* and **OnAlarm** child of the *Pick* activity can be a parent as well, if it is currently empty.
- **Flow** – As *Flow* activities can have multiple children, it is always a valid parent.
- **ForEach** – *ForEach* activities can have one *Scope* activity as a child. So if the item we want to insert is a *Scope* activity, *ForEach* activities need to be listed too.
- **Scope** – The last possible parent is a *Scope* activity, but these activities can only have one child. This means the *Scope* must not have a child at the moment, similar to the *If* and loop activities.

In order to make the activities better distinguishable in the selection, the name of the *Participant(Set)* should be displayed next to the *Activities* name, as shown on the following figure:

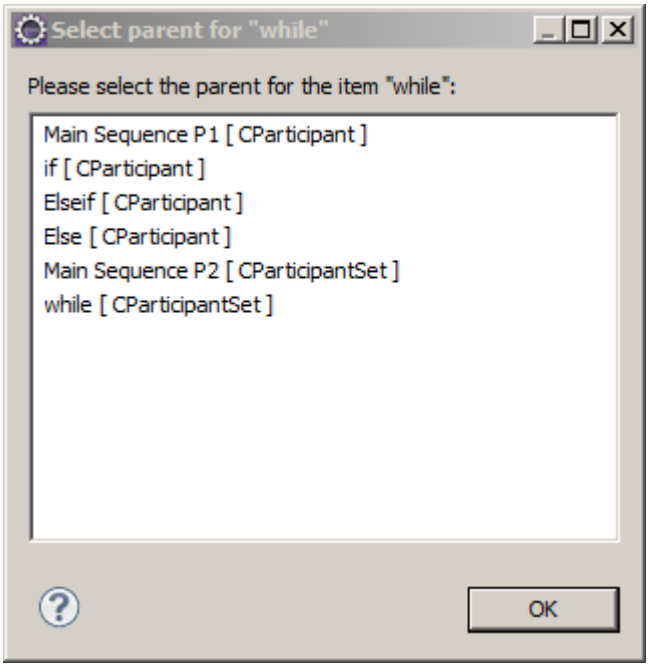

Figure 5.4: Selection dialog, asking the user for the parent of the item “while” that is currently being imported.

Note: when there is only one option available it should be selected automatically, instead of forcing the user to select it manually.

3. After the parent has been created or selected for each item, the item should be added to the choreography instantly. Otherwise possible parents, that can only have one child, could be selected multiple times.

4. In the last step, `CMessageLinks` between items that have been imported, are restored from the choreography fragment.

   Message links can only appear between *Reply*, *Receive*, *Invoke* and *OnMessage* element. *Reply*, *Receive* and *Invoke* activities can not have a child *Activity*. Therefore, only *OnMessage* activities could be skipped in step 2.

   So when importing the *MessageLinks* the availability of the receiving *Activity* in the target choreography needs to be verified. Otherwise the user may end up with a broken choreography.

Now the "bottom-up" approach for federated choreographies is done. Multiple choreographies can be combined to a more complex choreography as described in [ELT06] and section 3.

Later the importer could also be extended, so it is also possible for the user to select that the item itself already exists, rather then the parent. This allows an easier combination of two choreographies, that have been split from the same choreography before. Currently manual work is required for this. The user would select the same parent for item and after the importer has finished successfully, he would move all related links and items from the new item to the original item.

But this rises another issue: Related message links and children activities might already exist in the original choreography and therefor need to be checked for duplicates again. Also if a child is duplicate and contains children itself, their duplication needs to be investigated, which is why only parents are deduplicated up on import for now.

So for each of the items, the user would be prompted with another dialog, asking whether the item already exists or whether it should be created, until all items have being dealt with or a parent item has been created.

5.5 Storage Component – Storing and Retrieving of Choreography Fragments

The storage engine as described in the overview in section 5.2 basically has three tasks:

1. Convert a given diagram and choreography model into a storage item,
2. Convert a storage item into a well formed diagram and choreography model,
3. And storing storage items.

The basic solution is to store the diagram and choreography model separately in two different files in the file system of the current project. From here the files can be exported to Fragmento or to any other storage solution as already mentioned in section 5.2.

Additional meta data, e.g. author names, a short description or a icon for better recognition, that is either required to export the fragment into another storage solution or used in another way, can be collected, while exporting the fragment to the file system. The collected data should be saved as a third file in the file system.

Though the importer should not rely on this file. Otherwise two existing choreographies can not be easily combined anymore, as the user would need to generate the meta data file for at least one of the choreographies, before being able to import them.

5.6 Fragmento as an additional Storage Component

When using Fragmento as a storage component, the behaviour should be similar to the one of the FragmentoRCP plug-in described in [Den11]. This will help users getting used to the workflow more easily. See section 2.4.1.

5.6.1 Choreography Fragments

When implementing the Fragmento storage component for choreography fragments, both options, a persistent "export" folder and a temporary "Fragment" project, should be implemented. The fragments should be extracted to the same or a similar export folder. The second option should use a "*ChoreographyFragment*" project, so process and choreography fragments do not overwrite themselves unnecessarily.

The extraction wizard should consists of two steps only, asking the user for the name and description of the fragment and the icon respectively.

The import wizard is completely different from the process import wizard and is described in section 5.4.

5.7 Implementing Choreography Fragments into the Choreoghraphy Editor

In order to separate the fragment feature from the already existing choreography editor, a new Eclipse plug-in is created. This new plug-in depends on the existing choreography editor and will use *Extensions Points* to interact with the editor. [EF14b]

5.7.1 Choreography Fragment Plug-in Structure

The basic structure of an Eclipse plug-in is as follows:

- `bin/` – Folder, containing the compiled java *.class files
- `META-INF/MANIFEST.MF` – Manifest file containing the plug-in's (bundle) name, version and dependencies
- `src/` – Folder, containing all java source files of the plug-in
- `.classpath` – Hidden file, containing information where the source files are and which Java version is required to compile them
- `.project` – Hidden file, containing information about the Eclipse project of the plug-in
- `build.properties` – File, containing information how the plug-in is build from the project source
- `plugin.xml` – File, containing all information about the functionality of plug-in itself (available extension points, extension points that are being used, ...)

The files named above can be easily edited using Eclipse' plug-in manifest editor. It provides a user-interface and validates the given input. E.g. in order to be able to extend another plug-in using an extension point, the dependency needs to be added first.

5.7.2 Required Extension Points

Two *Extensions Points* hare required in the choreography editor in order to integrate the fragment plug-in:

- **Extending the context menu** – The first *Extensions Point* is required in order to be able to extract an existing structure into a choreography fragment. The fragment plug-in needs to add that option to the context menu, so when a right click is performed on a valid scenario selection, the selected activities, participants and message links can be saved as a fragment by the new plug-in.

  Note: This extension point is already provided by the GMF framework, which the choreography editor is based on. In order to extend a certain editors context menu the ID of the editor has to be specified in the extending plug-in.

- **Extending the tool palette** – The second *Extensions Point* allows including a fragment from a list of available fragments into the choreography diagram. Therefor a tool group is added to the tools palette and each fragment is added as a separate tool. In order to include a fragment into the diagram, it needs to be selected in the list and then a click into the editor starts a wizard, which then allows to include the fragment as desired.

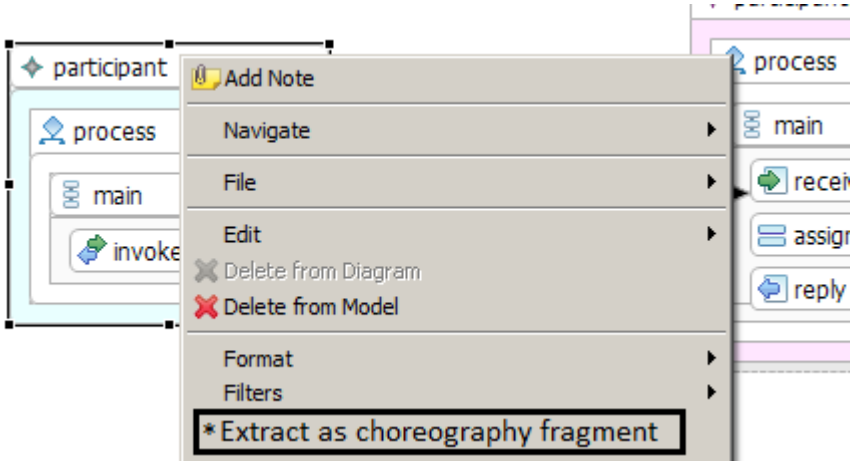

Figure 5.5: Mock-Up of the context menu extension point being used by the fragment plug-in

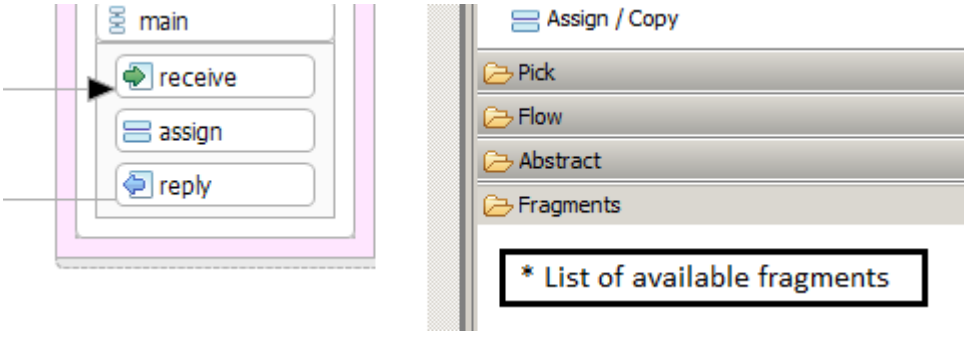

Figure 5.6: Mock-Up of the palette extension point being used by the fragment plug-in

6 Implementation

As already mentioned in last section of the design chapter, the new choreography fragment feature should be a new Eclipse plug-in[1], separated from the existing choreography designer. The designer is split into multiple plug-ins/projects already:

- `org.eclipse.bpel4chor.diagram`
- `org.eclipse.bpel4chor.diagram.extensions`
- `org.eclipse.bpel4chor.gmf`
- `org.eclipse.bpel4chor.model`
- `org.eclipse.bpel4chor.model.edit`
- `org.eclipse.bpel4chor.transform`

6.1 Extensions and Extension Points

In this section the implementation of the extensions and extension points is described. Extension points are the places in the code or system, where extensions can extend the functionality.

6.1.1 Extension for the "Context Menu" Extension Point

In order to be able to extract a selection as a choreography fragment, a new entry should be added to the context menu of the choreography editor, as shown in figure 5.5. The responsible extension point is already provided by the GMF framework itself.

In the `plugin.xml`, see the file explanation list in section 5.7.1, the fragments plug-in subscribes to this extension point called `org.eclipse.ui.menus` . Thereby the fragments plug-in registers a command in the menu contribution section, see listing 6.1.

The command `org.eclipse.bpel4chor.fragments.ExtractFragmentAction` needs to be registered using a second extension point provided by GMF: `org.eclipse.ui.commands` . When defining the command, see listing 6.2, the `categoryId` needs to match the ID of the Diagram Editor, in which the option should be displayed, in the case of the fragments plug-in this is the ID property of the ChorDiagramEditor.java class in the `org.eclipse.bpel4chor.diagram`

[1]The plug-in will be called "fragments plug-in" from here on

Listing 6.1 Context menu registration

```
<extension point="org.eclipse.ui.menus" id="context-menus">
    <menuContribution
        locationURI="popup:org.eclipse.gmf.runtime.diagram.ui.DiagramEditorContextMenu">
       <command commandId="org.eclipse.bpel4chor.fragments.ExtractFragmentAction"/>
    </menuContribution>
</extension>
```

plug-in. The visibility can be restricted further, in this case the active part that is selected needs to be a part of the ChorDiagramEditor as well and the selection must not be empty, but each item must be a instance of a `ChoreographyEditPart`. All activities, participants and message links are defined as such `ChoreographyEditPart`s. The `defaultHandler` finally points to the an instance of `org.eclipse.core.commands.AbstractHandler` and when the command is selected, the `execute()` method of that class is called. In this method the exporter component from section 6.2 will do its work.

Listing 6.2 Extract fragment command definition

```
<extension point="org.eclipse.ui.commands" id="menu-commands">
    <command id="org.eclipse.bpel4chor.fragments.ExtractFragmentAction"
        name="Extract selection as fragment"
        categoryId="org.eclipse.bpel4chor.model.chor.diagram.part.ChorDiagramEditorID"
        defaultHandler="org.eclipse.bpel4chor.fragments.ExtractFragmentAction">
            <visibleWhen>
                <and>
                    <with variable="activePartId">
                        <equals
        value="org.eclipse.bpel4chor.model.chor.diagram.part.ChorDiagramEditorID"/>
                    </with>
                    <with variable="selection"><iterate ifEmpty="false">
                        <instanceof
        value="org.eclipse.bpel4chor.model.chor.diagram.edit.parts.ChoreographyEditPart"/>
                    </iterate></with>
                </and>
            </visibleWhen>
    </command>
</extension>
```

6.1.2 New Extension Point: “Palette Factory”

The second extension point, required by the fragments plug-in, needs to be created from scratch. Therefor a simple entry, including the name, a unique identifier and the path to the schema definition of the extension point, is added in the `plugin.xml` of the `org.eclipse.bpel4chor.diagram` plug-in, as shown in listing 6.3.

The `schema/palettefactory.exsd` file defines the extension point and also keeps some annotation data like a description. Luckily a good user interface helps developers easily define the

Listing 6.3 Registration of the extension point

```
<extension-point id="paletteFactory" name="PaletteFactory"
    schema="schema/palettefactory.exsd"/>
```

extension point without having to write the XML directly. The most important part in the definition is, that other plug-ins that want to use this extension point, can be forced to implement a given class or interface. In the case of the tool palette extension point the clients need to implement the `org.eclipse.bpel4chor.model.chor.diagram.part.IChorPaletteExtension` interface.

The `paletteFactory` extension point that is being added here, is used to extend the Tool Palette of the Choreography Diagram Editor, that can be seen in figure 5.6. Currently a factory class for the Tool Palette is generated from the `chor.gmftool` model in the `org.eclipse.bpel4chor.gmf` project. The `fillPalette()` method of this factory receives a reference to the palette's root and then adds the necessary tool groups and tool elements. After the basic tools have been added, the new extension point is added, so plug-ins can add new entries and also delete already added entries.

First all extensions, which are registered to the extension point, are grabbed from the platform. Then each of the registered elements is checked, whether they implement the necessary interface, before the `extendPalette()` method, which is defined in the interface, is called, as shown in listing 6.4.

Listing 6.4 Execution of the extensions in the code of the Palette Factory

```
IExtensionRegistry reg = Platform.getExtensionRegistry();
IConfigurationElement[] extensions = reg.getConfigurationElementsFor(ExtensionPoint_ID);

try {
    for (IConfigurationElement ext : extensions) {
        final Object o = ext.createExecutableExtension("class");

        // Load extend the palette, if the extension implements the interface
        if (o instanceof IChorPaletteExtension) {
            paletteRoot = ((IChorPaletteExtension) o).extendPalette(paletteRoot);
        }
    }
} catch (Exception ex) { ... }
```

After all extensions have been called, the final `paletteRoot` is used to display the palette in the diagram editor.

6.1.3 Extension for the "Palette Factory" Extension Point

Now, since the extension point is defined correctly, the fragments plug-in can register a client to it. Therefore only the class name needs to be set in the client's `class` attribute.

Listing 6.5 Registration of the extension for the "Palette Factory" extension point

```
<extension point="org.eclipse.bpel4chor.diagram.paletteFactory">
    <client class="org.eclipse.bpel4chor.fragments.ChorPaletteExtension"/>
</extension>
```

In the class that has been given here, a new group section "Fragments" is created and added to the palette root. Then an entry for every fragment, that can be imported into the choreography and has been returned by the `createFragmentsImportToolList()` method, is added. Afterwards the palette container is added to the palette root.

Listing 6.6 Creation of the "Fragments" group, which is used as a parent for all available fragments

```
PaletteDrawer paletteContainer = new PaletteDrawer("Fragments");
paletteContainer.setId("org.eclipse.bpel4chor.fragments.ChorPaletteExtension");
paletteContainer.setDescription("");
paletteContainer.addAll(createFragmentsImportToolList());
paletteRoot.add(paletteContainer);
```

When one of the fragments in the tool palette is selected and inserted into the choreography, the importer component from section 6.4 takes over and assists the user in the process.

6.2 Implementation of the Exporter Component

As already mentioned in section 6.1.1 the exporter component can be accessed via the context menu, when a selection has been made in the choreography diagram.

Before being able to extract the selection, it needs to be validated and verified. The definition of choreography fragments does not allow stray *MessageLinks* and elements that are neither an *Activity*, a *Participant* nor a *ParticipantSet*. So the extractor component iterates over the selected elements and verifies their types. If one of the selected elements is not allowed, an error message is displayed to the user.

While validating the types, the extractor also creates a list with the unique IDs[2] of the selected *Activities*, so they can be identified later again.

Participants and *ParticipantSets* do not have an ID. In order to be able to identify them later again, we store the IDs of their *Process*es' *Sequence* activity in a second list. Since there is always only one *Process* added as a direct child to a *Participant(Set)* and one *Sequence* as a direct child to the *Process*, this path is unique and therefor identifies a *Participant*. See figure 6.1.

[2]Identifiers – Java code: `((Activity) element).getId()`

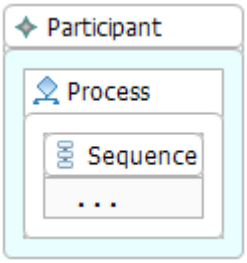

Figure 6.1: Simple example of a *Participant* –> *Process* –> *Sequence* sequence, where the *Sequence* activity can be used to identify the *Participant*

Now a copy of the diagram and it's model is created, so elements that are not required for the fragment can be removed, without removing them from the current choreography.

In the second step of the extractor component, as described in the design chapter in section 5.3, elements that are going to be deleted and elements that are just placeholders, are marked as such, by prefixing the name attribute with "*ToBeDeleted*" or "*Generated*".

In order to retain the validity of the choreography, *MessageLinks* that have a sending or receiving activity that is prefixed with "*ToBeDeleted*" need to be deleted, before the activity is being deleted.

Listing 6.7 Deleting *MessageLinks* that link to *Activities* that are going to be deleted

```
CMessageLink messageLink = (CMessageLink) element;

// If one of the activities is marked as "to be deleted", we need to delete the message link
if (
       ((Activity) messageLink.getReceiveActivity()).getName().indexOf("ToBeDeleted") == 0 ||
       ((Activity) messageLink.getSendActivity()).getName().indexOf("ToBeDeleted") == 0
) {
    EcoreUtil.delete(eo, true);
}
```

Afterwards the *Activities*, *Participants* and other elements, that are marked as "*ToBeDeleted*" can finally be deleted, without breaking the choreography. Thereby it is important, to delete the elements at the bottom first, so, while iterating over the list, elements are not skipped accidentally because their index is now lower then the current cursor position.

Then the choreography fragment is in its final shape and can be passed to the storage component, so it can be stored to the file system.

6.3 Implementation of the Storage Component

The implementation of the storage component can be split into two features. A save feature that is used to save the fragment after it has been extracted and a load feature, for loading a fragment so it can be imported.

6.3.1 Saving a Fragment in the Storage Component

The first option to store the fragment, as described in section 5.6, is a new *ChoreographyFragment* project. Similar to the ProcessFragment project in the FragmentoRCP plug-in, all files that exist in the project are deleted. The second storing option works pretty similar, the files are just stored in another folder in the file system.

Then two resources are created, one for the model and one for the diagram. When the model and diagram are added to the respective resource and the resources are saved to the project, two problems arise:

- The first problem is, that the diagram file still references the old model file. Each of the shape elements has a reference pointing to the file and path that identifies the element in the model file. While the path inside the document is still correct, the file name and path of the file is not. The following example is extracted from the `extraction_bug.chor_diagram` and stored as `fragment.chor_diagram`. The `href`-attribute of the `element`-element however still references the old model file, see line 5 in listing 6.8.

Listing 6.8 Excerpt of a fragment that has been extracted from a choreography

```
...
<children xmi:type="notation:Shape" xmi:id="_0Ju7YBGPEeSYgvobE-4MwA" type="3074">
    <children xmi:type="notation:DecorationNode" xmi:id="_0Ju7YhGPEeSYgvobE-4MwA"
        type="5084"/>
    <element xmi:type="pbd:Reply"
        href="platform:/resource/ChoreographyFragment/extraction_bug.chor#//
        @participantSets.0/@process/@activity/@activity.0/@activity"/>
    <layoutConstraint xmi:type="notation:Bounds" xmi:id="_0Ju7YRGPEeSYgvobE-4MwA"/>
</children>
...
```

  A workaround for this problem is to load the resource as a string and then replace "`href="platform:/resource/ChoreographyFragment/extraction_bug.chor#`" with "`href="fragment.chor#`".

- The second problem appears when opening the fragments diagram. When opening the files with a plain text editor, one can easily see, that the model file `fragment.chor` only contains elements that are part of the fragment, see listing 6.9.

Listing 6.9 Model file `fragment.chor` of a fragment that only contains one *Invoke* activity

```
<?xml version="1.0" encoding="UTF-8"?>
<chor:Choreography xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:chor="urn:HPI_IAAS:choreography:schemas:choreography:2006/12"
    xmlns:pbd="http://docs.oasis-open.org/wsbpel/2.0/process/abstract">
  <participants name="GeneratedParticipant">
    <process name="GeneratedProcess"
        abstractProcessProfile="urn:HPI_IAAS:choreography:profile:2006/12">
      <activity xsi:type="pbd:Sequence" name="GeneratedSequence"
          id="ab6bb58f-5717-4fe9-9a61-e52f1f5c4d6d">
        <activity xsi:type="pbd:Invoke" name="InvokeMe"
            id="06b22057-8c38-4496-af1a-bb7f7f0bfa91"/>
      </activity>
    </process>
  </participants>
</chor:Choreography>
```

Listing 6.10 Excerpt of the diagram file `fragment.chor_diagram` of a fragment that only contains one *Invoke* activity

```
<?xml version="1.0" encoding="UTF-8"?>
<notation:Diagram ...>
    <children xmi:type="notation:Shape" ... type="2011">
        <children xmi:type="notation:DecorationNode" ... type="7042">
            ...
              <children xmi:type="notation:Shape" ... type="3046">
                ...
                <element xmi:type="pbd:Invoke"
                    href="fragment.chor#//@participants.0/@process/@activity/@activity.0"/>
                ...
            </children>
            <children xmi:type="notation:Shape" ... type="3047">
                ...
                <element xmi:type="pbd:Receive"
                    href="fragment.chor#//@participants.0/@process/@activity/@activity.1"/>
                ...
            </children>
            ...
    </children>
    <element xmi:type="chor:CParticipant" href="fragment.chor#//@participants.0"/>
    <layoutConstraint xmi:type="notation:Bounds" ... x="40" y="135"/>
  </children>
    ...
    <element xmi:type="chor:Choreography" href="/ChoreographyFragment/fragment.chor#/"/>
    <edges xmi:type="notation:Connector" type="4001" ...>
        ...
        <element xmi:type="chor:CMessageLink" href="fragment.chor#//@messageLinks.0"/>
        ...
  </edges>
    ...
</notation:Diagram>
```

The diagram file `fragment.chor_diagram` however still contains all data from the shapes of the old *Activities*, *Participants* and *ParticipantSets*, that have been deleted by the extractor. Also the *MessageLinks* that have been deleted still exists, as it can be seen in listing 6.10.

If the diagram is opened with the chor diagram editor, it looks like it is supposed to look, but the tab is prefixed with a asterisk, see figure 6.2. This means that the diagram has been modified, without being saved later on. Now the user can simply save the diagram manually to update the file in the file system.

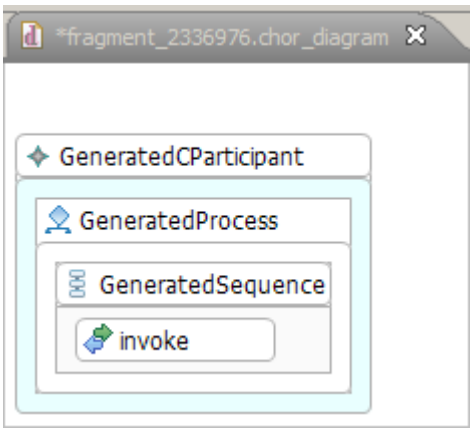

Figure 6.2: Choreography fragment diagram is marked as modified

When the file is reopened a second time in both editors, everything is as intended: the content of the `fragment.chor_diagram` file no longer contains the data of the old elements, neither is the diagram marked as modified.

6.3.2 Loading a Fragment from the Storage Component

Loading a fragment from the storage is way simpler, then writing it to the storage. After selecting the desired choreography, the two resources for the diagram and the model are loaded from the Eclipse project. The only thing that needs to be done, before using the diagram for the import is, to link the *Choreography* model to the *Diagram*, as shown in listing 6.11. No other adjustments have to be made.

Listing 6.11 Linking the loaded *Choreography* to the *Diagram*

```
Choreography chor = (Choreography) loadEObjectFromResource(fragmentFileName + ".chor");
Diagram diagram = (Diagram) loadEObjectFromResource(fragmentFileName + ".chor_diagram");
diagram.setElement(chor);
```

6.4 Implementation of the Importer Component

Now the importer recursively iterates over the elements and finds the items, which have been originally extracted. Therefore, the name of the items is checked. If the name starts with the "*Generated*" prefix and hence is a placeholder item, the item is not imported. Only if the current item is not prefixed with "*Generated*", but the parent is, the item is part of the fragment and the importing process continues. When an item is being imported, the recursion is stopped, since all descendant items are already imported in the same step.

In order to import the item as desired, the user now gets involved in the process. The first thing the user needs to decide is, whether or not, the parent item of the item that is currently imported, already exists in the *Choreography*. If the parent does exist, a list of possible parent items is generated.

6.4.1 Selecting a valid Parent for an Item

For *Participants*, *ParticipantSets* and *MessageLinks* the *Choreography* itself is the only valid parent. In this case the item can be imported directly without further user interaction, as there is only one *Choreography* that can be the parent for these items.

When the imported item is an *Activity* a couple of possible parents might be found in the choreography. The list of possible parents can be found in the design chapter of the importer, described in section 5.4. The possible parents are then presented to the user, so he can select the parent for the item, as shown in figure 5.4.

Some other items that may exist in a *Choreography* are ignored when importing them directly into the choreography:

- *MessageLinks* where not both of the end-points have been imported from the fragment before
- Items that are also ignored when they are selected for extraction:
  - *Process*,
  - *Elseif* and *Else*,
  - *OnMessage* and *OnAlarm*
- And some other items, that can not be selected in the diagram:
  - *Condition* as a immediate child of *If*, *While* and *RepeatUntil*,
  - *Copy*, *From* and *To*,
  - and some more.

6.4.2 Generating a Parent for an Item

This case can only appear for *Activities*, as the parent for *MessageLinks*, *Participants* and *ParticipantSets*, the *Choreography* always exists.

If no valid parent was found, or the user chose that the parent does not yet exist in the choreography, the parent is generated. As already mentioned in the design chapter, see figure 5.3, only some of the parents should exist in the new choreography.

First the parent *Participant* or *ParticipantSet* of the activity needs to be found. Therefore, iterative the `eContainer()` method is called, until the item is a *Participant*, or the *Choreography* was found or the container is `null`, which means that no parent *Participant* exists.

Listing 6.12 Finding the parent *Participant* or *ParticipantSet* for a given item

```
public EObject getParticipant(EObject item) {
	if (ChorPackage.Literals.CPARTICIPANT.isSuperTypeOf(item.eClass())
			|| ChorPackage.Literals.CPARTICIPANT_SET.isSuperTypeOf(item.eClass())) {
		return item;
	}

	if (ChorPackage.Literals.CHOREOGRAPHY.isSuperTypeOf(item.eClass())
			|| item.eContainer() == null) {
		return null;
	}

	return getParticipant(item.eContainer());
}
```

When the parent item was found, it is imported into the choreography as described in section 6.4.1. Then all *Activities* that are children of the *Sequence* of the imported parent, are removed, so only the *Participant*, *Process* and *Sequence* activity are generated.

So the "in between" parent *Activities* are removed while importing rather then exporting. The reason for this is, that with the current implementation they are also removed from fragments, which have been created from scratch with the choreography designer, and not only from fragments, which have been extracted by the extractor component.

6.4.3 Inserting an Item into the Choreography

Now, that the parent exists, the item itself can be inserted into the choreography.

But before the items can be inserted into the choreography, it needs to be ensured, that no ID of an *Activity* is used multiple times. Therefor a new random ID is created for each of the *Activities* in the tree that is going to be added:

This also allows the user to import the same fragment multiple times.

Listing 6.13 Generating new unique IDs for *Activities*

```
protected void refreshActivityIDs(EObject item) {
	if (ChorPackage.Literals.CMESSAGE_LINK.isSuperTypeOf(item.eClass())) {
		return;
	}

	if (PbdPackage.Literals.ACTIVITY.isSuperTypeOf(item.eClass())) {
		String newActivityID = java.util.UUID.randomUUID().toString();
		((Activity) item).setId(newActivityID);
	}

	List<EObject> references = ChoreographyHelper.getReferences(item);
	for (int i = 0; i < references.size(); i++) {
		refreshActivityIDs((EObject) references.get(i));
	}
}
```

When finally the item is imported into the parent item, two different transaction commands have to be used. The first command is an `AddCommand`, which is used, when the item is added to the parent item, that can have multiple immediate children. If the parent can only have one *Activity*, the `SetCommand` is used. Modifying items in the active choreography, requires a transaction, so the displayed choreography is always in a well-formed state, otherwise an exception is thrown by the GMF framework.

Now, at the end of the import process, all *MessageLinks*, where both, the sending and receiving, *Activity* have been imported, can be imported as well.

6.4.4 Plug-in Overview

In the end, the main code of the plug-in is split into three packages:

- `org.eclipse.bpel4chor.fragments.components`

  The `components` package contains the three components that have been outlined in 5.2:

  - `FragmentExtractor`
  - `FragmentImporter`
  - `StorageEngine`

- `org.eclipse.bpel4chor.fragments.handlers`

  In the `handler` package the different extensions for the extension points are stored:

  - `ContextMenuExtractToPalette` – Extension for the “Extract selection to fragment palette” option in the context menu of the diagram editor
  - `ContextMenuExtractToProject` – Extension for the “Extract selection as fragment” option in the context menu of the diagram editor

  - `PaletteContextMenuPublishToFragmento` – Extension for the "Publish to Fragmento" option in the context menu of the palette
  - `PaletteImportFragment` – Extension for importing a fragment from the palette into the diagram editor

- `org.eclipse.bpel4chor.fragments.util`
  - `ChoreographyHelper` – A helper class, which provides various methods to work with `EObjects` (*Activities*, *Participants*, etc.), like getting their name attribute, their parent *Participant* and some more.
  - `ChoreographyLabelProvider` – A label provider, that displays the name of an item together with the name of the parent *Participant*, as shown in figure 5.4.

7 Evaluation and Conclusion

In this final chapter, the implementation from chapter 6 is tested with the scenarios described in section 4.3 and section 4.4.

7.1 Evaluation of the Extraction Component

7.1.1 Evaluation Sample 1 – Extracting a Fragment from only one Participant

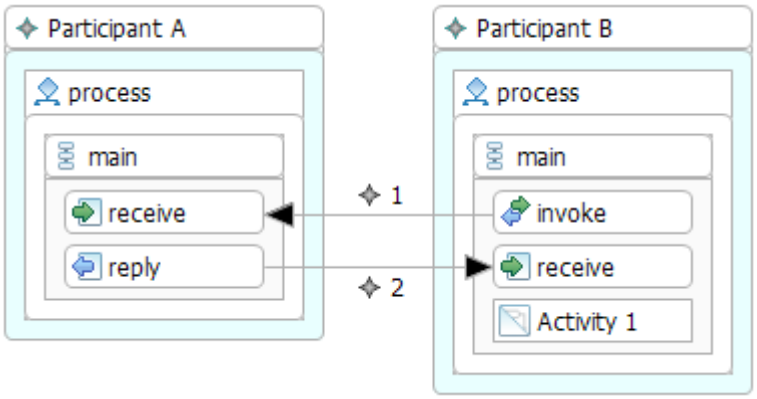

Figure 7.1: Simple choreography with two *Participants* "A" and "B"

The sample in figure 7.1 models a simple choreography with two participant.

Selection: Only the *Receive* activity "receive" and the *Opaque Activity* "Activity 1" from "Participant B" are selected as a fragment.

Expected: When the extracted choreography is opened, only the *Participant* "Generated#Participant B" should be present. Inside the *Participant* there should be one *Process* "Generated#Process" enclosing a *Sequence* "Generated#main". The two selected activities should be inside this *Sequence* activity.

Actual: However, as mentioned in the implementation the diagram shows data that should be deleted and is marked as changed, see figure 7.2. After the diagram is saved and reopened, the diagram looks as expected, as shown in figure 7.3, and only contains the items, that have been listed in the "Expected" list.

This problem has already been mentioned in the implementation section of the storage component, see section 6.3.1. So the additional "saving and reopening" of the diagram will not be documented in the following samples and only the final result will be compared to the "Expected" list.

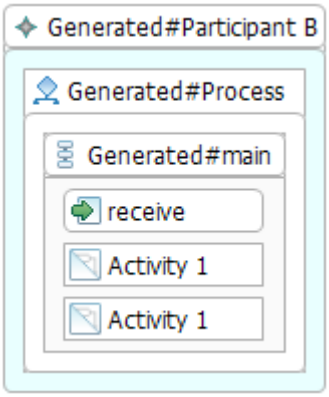

Figure 7.2: Broken fragment immediate after the extraction

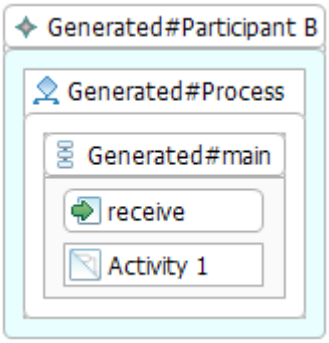

Figure 7.3: Final fragment after the necessary "saving" and "reopening" of the diagram

7.1.2 Evaluation Sample 2 – Extracting a Fragment with a Connector

The sample uses the same choreography as the first sample, so see figure 7.1 again.

Selection: This time the *Reply* activity from "Participant A" and the *Receive* activity "receive" from "Participant B" are selected as a fragment.

Expected: The resulting fragment should contain two participants "Generated#Participant A" and "Generated#Participant B", including a generated *Process* and *Sequence* each. The *Sequence* of the "Generated#Participant A" should contain a the *Reply* activity "reply", while the one of "Generated#Participant B" contains the "receive" *Receive* activity. Both activities

should be linked with a *Message Link*, named "2", pointing from the *Reply* to the *Receive* activity.

Actual: As shown in figure 7.4 the extracted fragment matches the expected fragment.

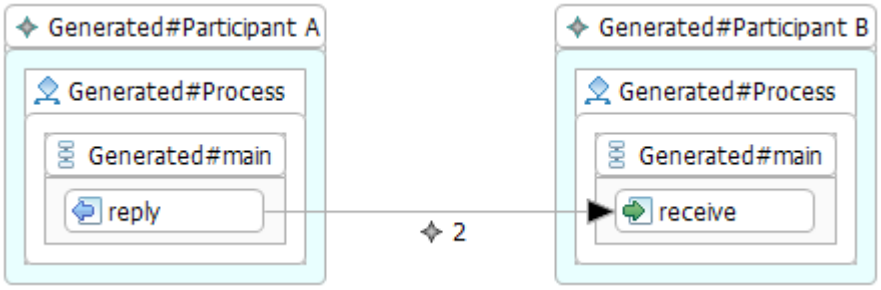

Figure 7.4: Final fragment with a connector sample

7.1.3 Evaluation Sample 3 – Extracting a Connector and loose Activities/Participant

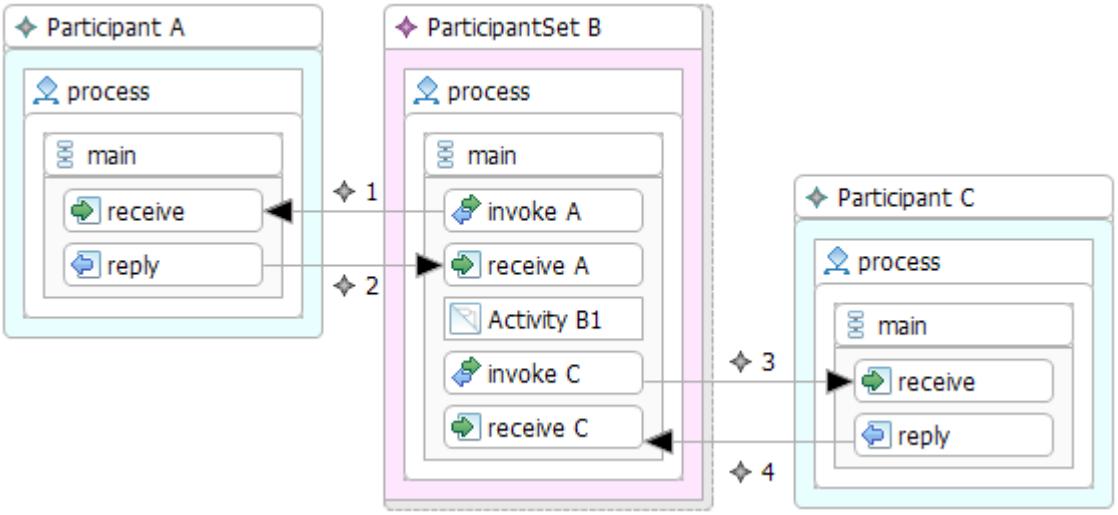

Figure 7.5: Choreography with two *Participants* A and C which both communicate with a *ParticipantSet* B

In this sample two *Participants* "Participant A" and "Participant C" both communicate with a *ParticipantSet* "ParticipantSet B". Then the connector from "Participant A" to "ParticipantSet B" is selected, the *Activity* "Activity B1", as well as "Participant C" or a subset of "Participant C".

Evaluation Sample 3.1 – Loose Content of “Participant C”

Selection:

1. As already mentioned in the general section, the *Reply* activity from “Participant A” and the *Receive* activity “receive A” from “ParticipantSet B” and “Activity B1” from “ParticipantSet B” are selected as a fragment.
2. Additionally the *Activities* “receive” and “reply” of “Participant C” are selected.

Expected:

1. The expected fragment should contain a generated *Participants* “Generated#Participant A” and a generated *ParticipantSet* “Generated#ParticipantSet B”. Both of them should contain a generated *Process* and *Sequence*. The *Sequence* of “Generated#Participant A” should contain a *Reply* activity that is linked with a *Message Link* “2” to the “receive A” *Receive* activity of “Generated#ParticipantSet B”. “Generated#ParticipantSet B” should contain the *OpaqueActivity* “Activity B1”.
2. The fragment should also contain another generated participant “Generated#Participant C”. This participant also contains the generated *Process* and *Sequence*, in which the selected *Receive* and *Reply* activity from “Participant C” can be found.

Actual: As shown in figure 7.6 the extracted fragment matches the expected fragment.

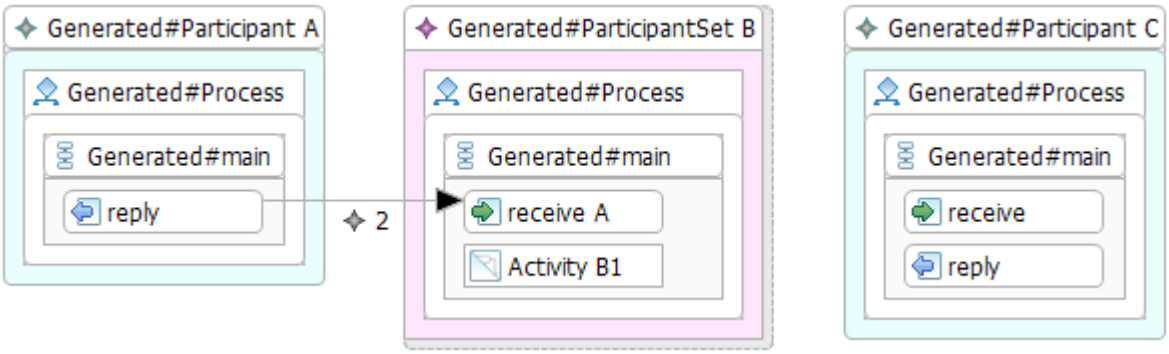

Figure 7.6: Fragment with a connector and *Activities* of an additional *Participant*

Evaluation Sample 3.2 – Loose Sequence of “Participant C”

Selection:

1. See **Selection 1** of section 7.1.3.
2. Additionally the *Sequence* activity of “Participant C” is selected.

Expected:

1. See **Expected 1** of section 7.1.3.
2. The fragment should also contain another generated participant “Generated#Participant C”, which itself contains a generated *Process* “Generated#Process”. Inside this *Process* the *Sequence* “main” from “Participant C”, including its activities *Receive* and *Reply*, should exist.

Actual: As shown in figure 7.7 the extracted fragment matches the expected fragment.

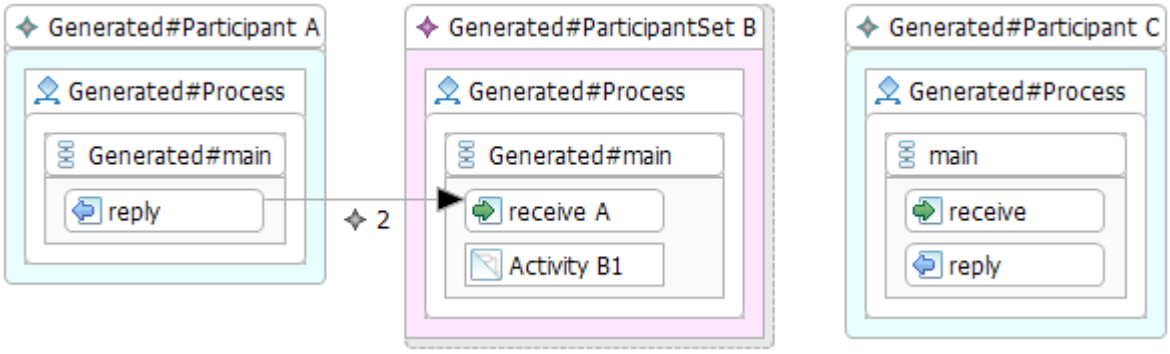

Figure 7.7: Fragment with a connector and the *Sequence* of an additional *Participant*

Evaluation Sample 3.3 – Loose Participant “Participant C”

Selection:

1. See **Selection 1** of section 7.1.3.
2. In this third version of the sample, the *Participant* “Participant C” itself is selected.

Expected:

1. See **Expected 1** of section 7.1.3.
2. The fragment should now contain “Participant C”, including all its child items, just like it has been modeled in figure 7.5.

Actual: As shown in figure 7.8 the extracted fragment matches the expected fragment.

7.1.4 Extractor Result

The samples listed in this section cover the export of *Participants*, *Activities* and *Message Links*, which is everything that has been considered valid as an extraction item. All tests have been successful, after the workaround from section 6.3.1 has been applied, therefore the extractor is considered fully functional.

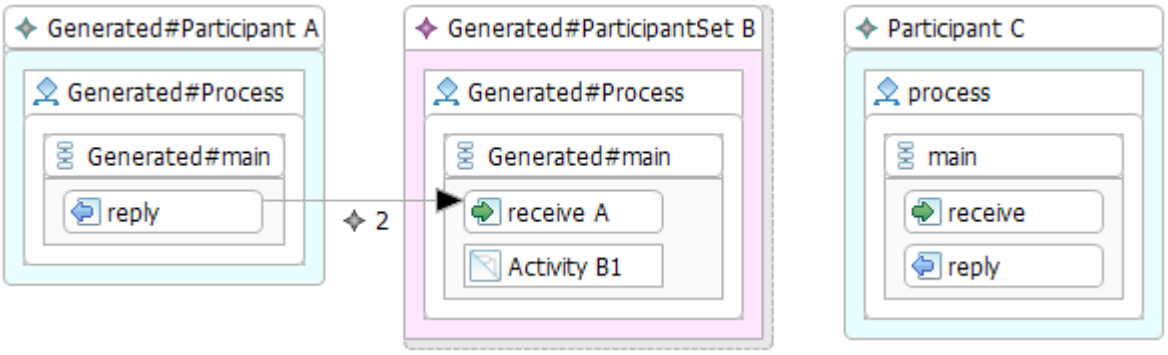

Figure 7.8: Fragment with a connector and the *Sequence* of an additional *Participant*

7.2 Evaluation of the Importer Component

Note: All evaluation samples for the importer component start with the choreography that has been shown in figure 7.1. Therefor each sample will only have one figure with the fragment, which is being imported, and the final result.

7.2.1 Evaluation Sample 4 – Import of a Fragment into one Participant

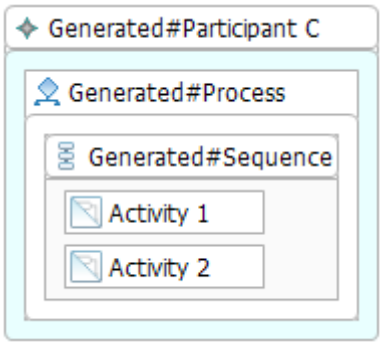

Figure 7.9: Simple fragment with two *Activities* "Activity 1" and "Activity 2" inside a fully generated *Participant*

The imported choreography from figure 7.9 only contains one fully generated *Participant*[1]. In this *Participant* "Generated#Participant C" there are two *OpaqueActivities* "Activity 1" and "Activity 2".

Selected parent: For both *Activities* the *Sequence* of "Participant A" is selected as a parent.

[1] A *Participant* is considered "fully generated", if the *Participant*, as well as the associated *Process* and *Sequence* are generated.

Expected: The final choreography should still contain the two *Participants* from the original choreography. Only the "main" *Sequence* of "Participant A" has two new *Activities*.

Actual: As shown in figure 7.10 the combined choreography matches the expected result.

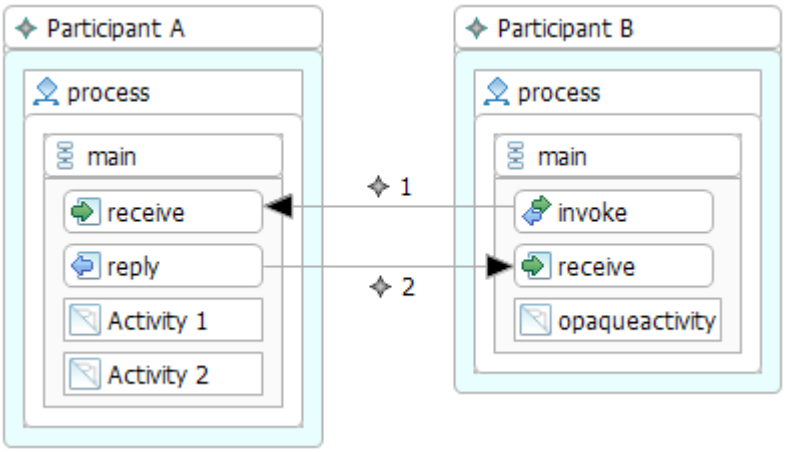

Figure 7.10: Resulting choreography with two new *Activities* "Activity 1" and "Activity 2" inside the *Sequence* of "Participant A"

7.2.2 Evaluation Sample 5 – Import of a Connector Fragment into multiple Participant

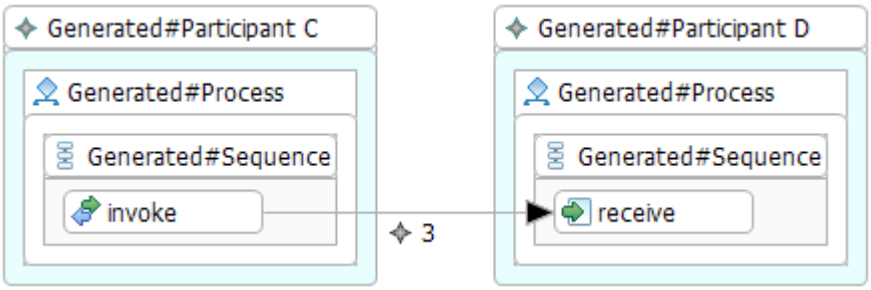

Figure 7.11: Simple connector with a *Message Link* "3", connecting two *Activities* "invoke" and "receive" inside two fully generated *Participants*

The imported choreography from figure 7.11 contains a *Message Link* "3". The *Message Link* connects an *Invoke* activity from a fully generated *Participant* "Generated#Participant C" with the *Receive* activity of a fully generated *Participant* "Generated#Participant D".

Selected parent: For the *Invoke* of "Generated#Participant C" *Participant* "Participant A" is selected as a parent, while the *Receive* of "Generated#Participant D" is imported to "Participant B".

Expected: The final choreography should only contain the two *Participants* from the original choreography. Additionally the two imported *Activities* and the *Message Link* should be found in the choreography.

Actual: As shown in figure 7.12 the combined choreography matches the expected result.

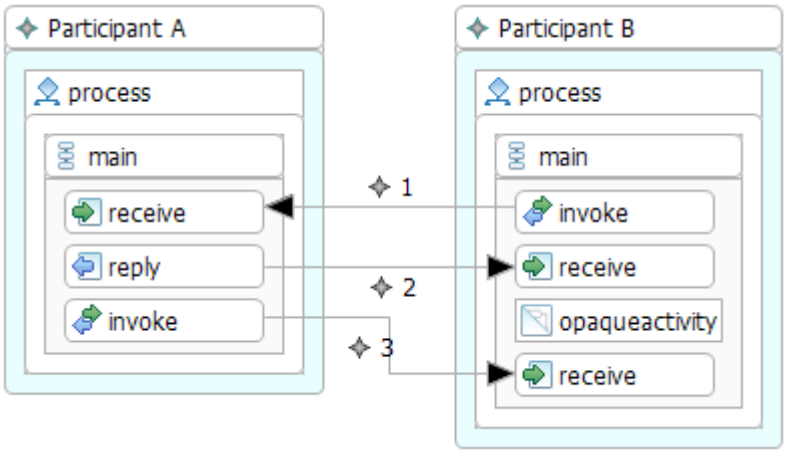

Figure 7.12: Resulting choreography with the new connector between "Participant A" and "Participant B"

7.2.3 Evaluation Sample 6 – Import of a loose Participant and Generation of a Participant

Comparable to sample 3 from section 7.1.3, now the import of a connector and loose *Activities* or *Participants* is tested.

Evaluation Sample 6.1 – Loose Content of "Generated#ParticipantSet E"

In the first version of the sample the loose *Activities*, are members of a generated *Sequence* that is inside a fully generated *ParticipantSet* "Generated#ParticipantSet E".

Selection:

1. For the connector, the parents have been selected as in section 7.2.2.
2. For the loose *Activities* the parent *Participant* is generated, excluding the unnecessary *Sequence*.

Expected:

1. The result of the connector import is expected to be like the result of section 7.2.2.

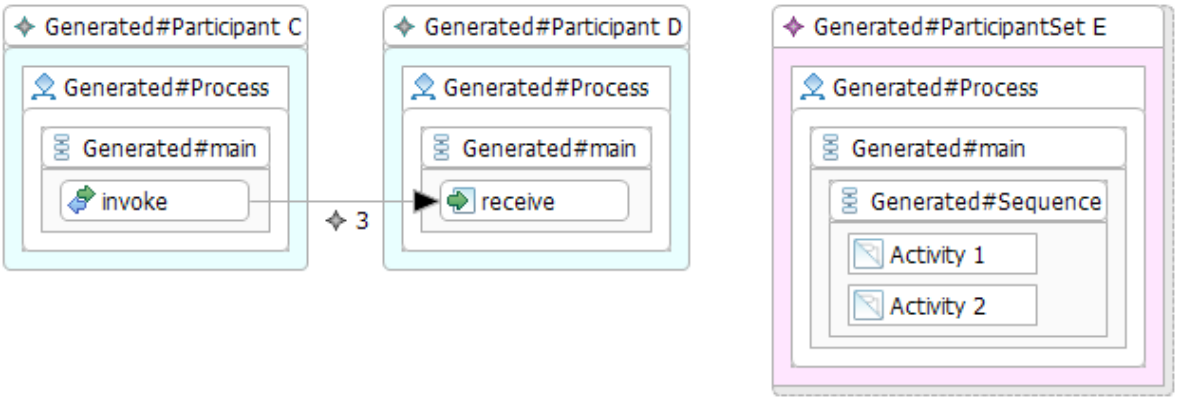

Figure 7.13: Fragment with a connector and *Activities* of a generated *Activity*

2. Additional there should be a fully generated *ParticipantSet* "Generated#ParticipantSet E" directly containing the two *Activities*, without the additional *Sequence* "Generated#Sequence" from figure 7.13.

Actual: As shown in figure 7.14 the combined choreography matches the expected result.

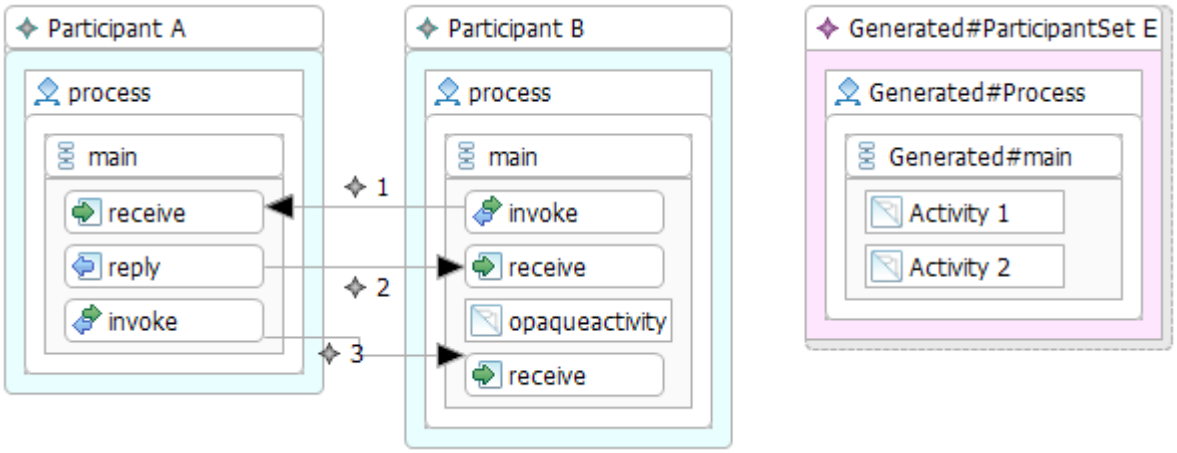

Figure 7.14: Import of a fragment with a connector and *Activities* of a generated *Activity*

Evaluation Sample 6.2 – Loose child of "Generated#ParticipantSet E"

In the second version of the sample the member of the fully generated *ParticipantSet* "Generated#ParticipantSet E", including its subsequent *Activities* is imported.

Selection:

1. For the connector, the parents have been selected as in section 7.2.2.

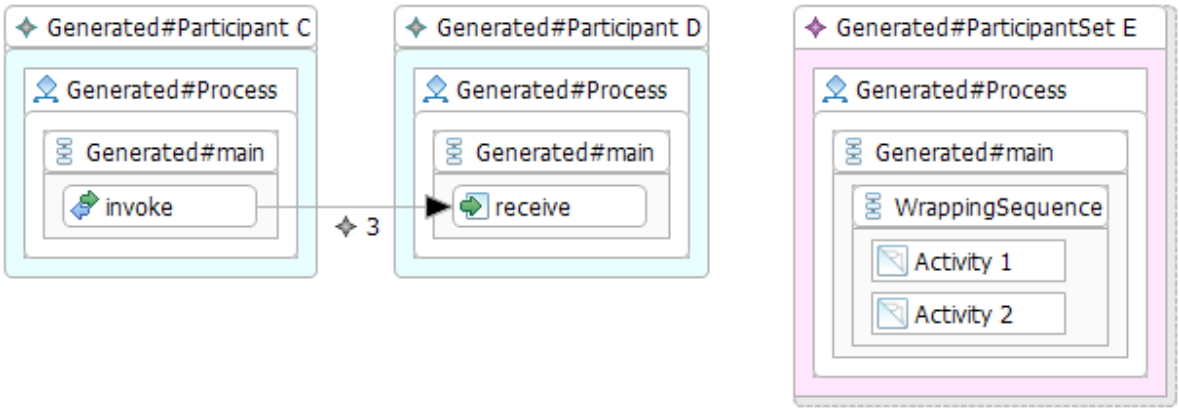

Figure 7.15: Fragment with a connector and an *Activity* of a fully generated *Participant*

2. For the loose *Activity* "WrappingSequence" the parent *Participant* is generated.

Expected:

1. The result of the connector import is expected to be like the result of section 7.2.2.
2. Additional there should be a fully generated *ParticipantSet* "Generated#ParticipantSet E" directly containing the "WrappingSequence" *Activities*, including its children from figure 7.15.

Actual: As shown in figure 7.16 the combined choreography matches the expected result.

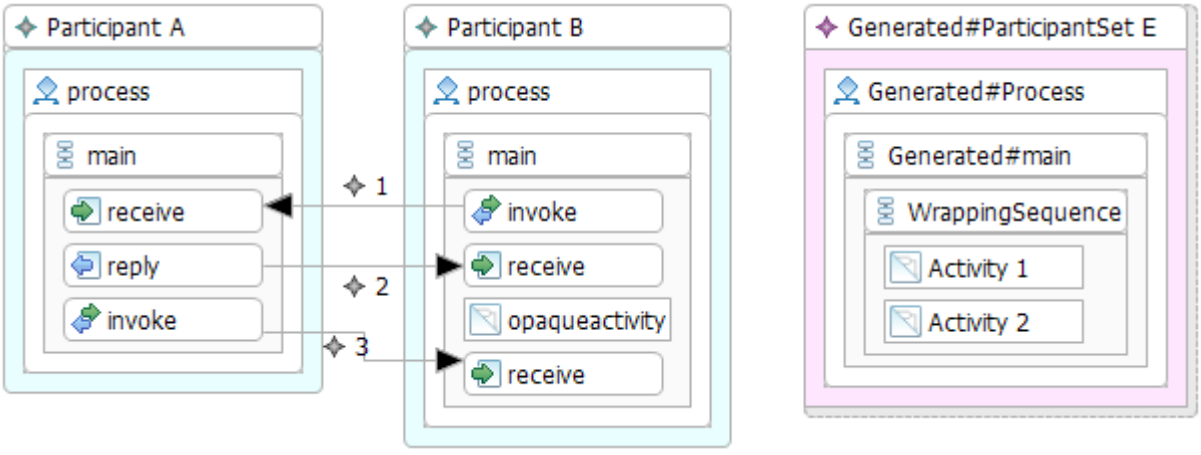

Figure 7.16: Import of a fragment with a connector and an *Activity* of a fully generated *Participant*

Evaluation Sample 6.3 – Loose Participant “ParticipantSet E”

Now in the last version of the sample the *ParticipantSet* is imported directly.

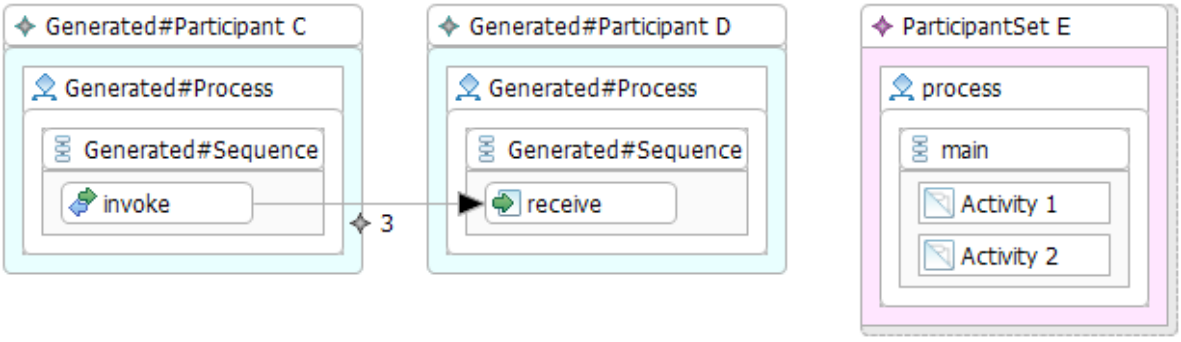

Figure 7.17: Fragment with a connector and a *Participant* “ParticipantSet E”

Selection:

1. For the connector, the parents have been selected as in section 7.2.2.
2. For the *Participant* the only valid parent is the *Choreography* itself.

Expected:

1. The result of the connector import is expected to be like the result of section 7.2.2.
2. The *Participant* “ParticipantSet E” should be added to the *Choreography* without any parents.

Actual: As shown in figure 7.18 the combined choreography matches the expected result.

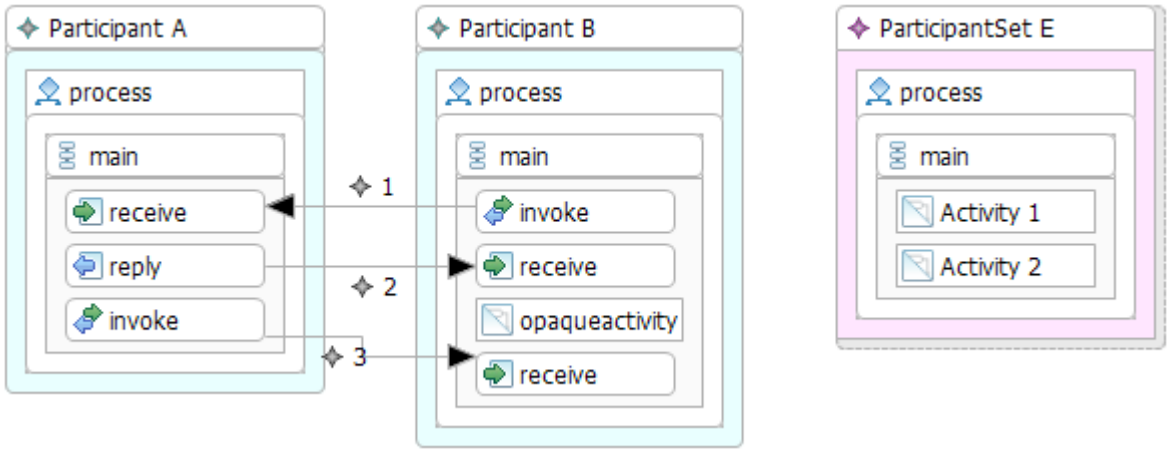

Figure 7.18: Import of a fragment with a connector and a new *Participant*

7.2.4 Importer Result

The samples listed in this section cover the import of *Participants*, *Activities* and *Message Links*, as well as the generation of required parent *Participants*, *Processes* and *Sequences*. All tests have been successful, therefore the importer is considered fully functional.

7.3 Evaluation of the Storage Component

Currently only the storage component does not fully work as expected:

- **Missing implementation of "Publish to Fragmento"** – At the moment the fragments can only be exported to the file system and the Eclipse project. A future task should finish the integration and add the missing functionality to publish the Eclipse project and/or export folder to Fragmento.
- **Save-Reopen Workaround** – The currently required workaround for the second problem described in section 6.3.1 should be investigated. In the final implementation the opening and saving should be done without any user interaction, so fragments can be easily generated, e.g. by an API, and reused.

7.4 Conclusion

As the Related Work section shows, multiple different models for choreography fragments have been developed in the past. Each of them fits the needs of the respective authors.

Similar to these definitions, the one from the chapter 4, fits best for the usecase with the choreography designer. Having a valid choreography at any time, instead of only parts of a valid choreography, is the main reason why the other definitions could not be used as is.

Being able to open the fragments in the existing choreography designer, so they can be further modified and adjusted is an important requirement. The restrictions that have been outlined in section 5.1 help to ensure, that the extracted choreography fragments are still valid.

Bibliography

[ASF12] T. Apache-Software-Foundation. Apache Axis2/Java, 2012. URL http://axis.apache.org/axis2/java/core/. (Cited on page 28)

[BSM+04] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2004. (Cited on page 17)

[Den11] D. Dentsas. Integration von Fragmento in eine Rich Client Plattform, 2011. (Cited on pages 7, 19, 26, 27 and 50)

[DKLW07] G. Decker, O. Kopp, F. Leymann, M. Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. In *Proceedings of the IEEE 2007 International Conference on Web Services (ICWS)*. 2007. URL http://bpt.hpi.uni-potsdam.de/pub/Public/GeroDecker/icws2007-BPEL4Chor.pdf. (Cited on page 16)

[EF14a] T. Eclipse-Foundation. Eclipse - BPEL Designer Project, 2014. URL http://www.eclipse.org/bpel/. (Cited on page 19)

[EF14b] T. Eclipse-Foundation. Eclipse Platform Help: Extensions and Extension Points, 2014. URL http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.pde.doc.user/concepts/extension.htm. (Cited on page 50)

[ELT06] J. Eder, M. Lehmann, A. Tahamtan. Choreographies as Federations of Choreographies and Orchestrations. 2006. (Cited on pages 7, 23, 24, 46 and 49)

[LK12] F. Leymann, D. Karastoyanova. Chapter 12: BPEL. Vorlesungsunterlagen von Services and Service Composition, 2012. (Cited on page 15)

[MY13] F. Montesi, N. Yoshida. Compositional Choreographies. 2013. (Cited on page 25)

[OAS07a] 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. (Cited on pages 9 and 15)

[OAS07b] OASIS. Web Services Business Process Execution Language Version 2.0 - Specification, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/process/abstract/ws-bpel_abstract_common_base.xsd. (Cited on page 15)

[SKK+11] D. Schumm, D. Karastoyanova, O. Kopp, F. Leymann, M. Sonntag, S. Strauch. Process Fragment Libraries for Easier and Faster Development of Process-based Applications. *Journal of Systems Integration*, 2(1):39–55, 2011. (Cited on pages 7, 21, 22, 23, 26 and 31)

[SKLS10] D. Schumm, D. Karastoyanova, F. Leymann, S. Strauch. Fragmento: Advanced Process Fragment Library. In *Proceedings of the 19th International Conference on Information Systems Development (ISD 2010), 25 August 2010, Prague, Czech Republic*. Springer-Verlag, 2010. (Cited on page 31)

[Son13] O. Sonnauer. *Modellierung von Scientific Workflows mit Choreographien*. Diplomarbeit, Universitaet Stuttgart, 2013. URL http://elib.uni-stuttgart.de/opus/volltexte/2013/8504/. (Cited on page 17)

All links were last followed on September 02, 2014.

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature